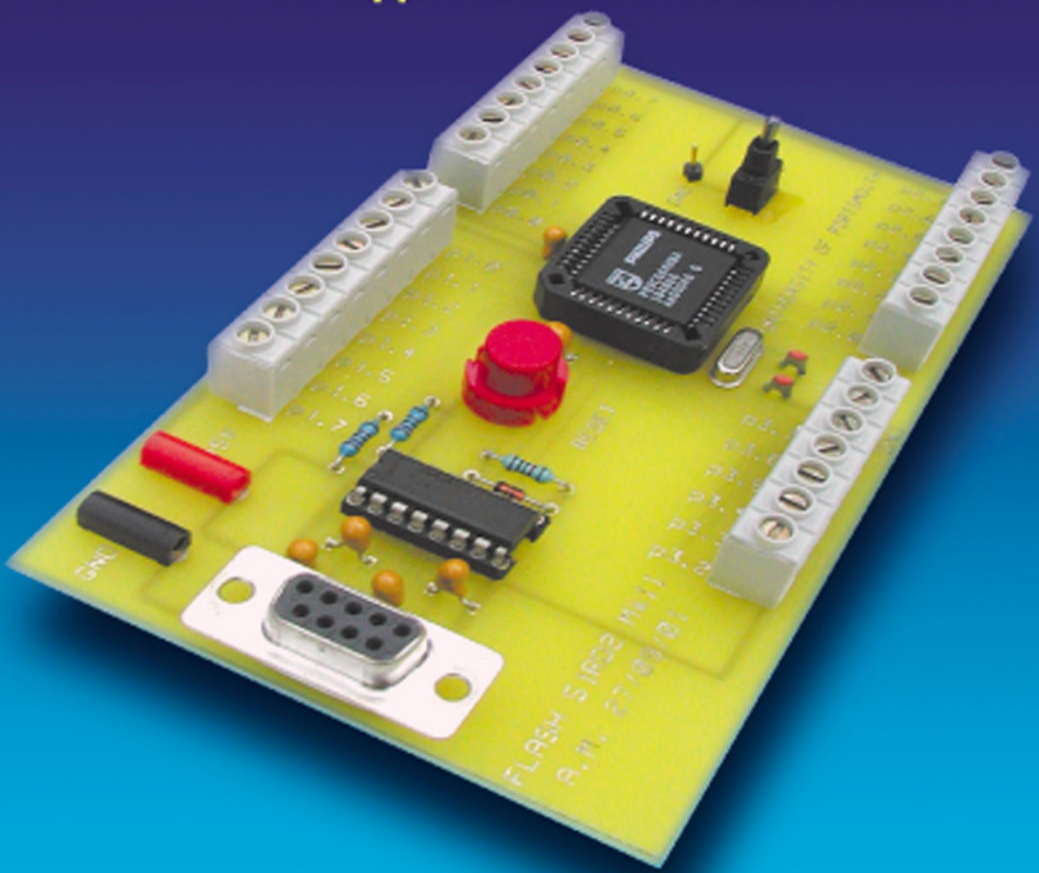
ELSEVIER
BUTTERWORTH HEINEMANN

8051 MICROCONTROLLER

An Applications Based Introduction



Chris Braithwaite • Fred Cowan
Hassan Parchizadeh

8051 Microcontrollers

An Applications-Based Introduction

David Calcutt
Fred Cowan
Hassan Parchizadeh

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Newnes is an imprint of Elsevier

Newnes
An imprint of Elsevier
Linacre House, Jordan Hill, Oxford OX2 8DP
200 Wheeler Road, Burlington, MA 01803

First published 2004

British Library Cataloguing in Publication Data
Calcutt, D.
8051 microcontrollers : an applications based introduction
1. INTEL 8051 (Computer) 2. Digital control systems
I. Title II. Cowan, Frederick J. III. Parchizadeh, G. Hassan 004.1′65

Library of Congress Cataloguing in Publication Data
Calcutt, D. M.
8051 microcontrollers : an applications-based introduction / David Calcutt, Fred Cowan, Hassan Parchizadeh.
p. cm.
1. Intel 8051 (Computer) 2. Digital control systems. I. Cowan, Frederick J. II. Parchizadeh, G. Hassan. III. Title.

QA76.8.I27C35 2003
004.165—dc22

2003066606

ISBN 0 7506 5759 6 (alk. paper)

Typeset by Integra Software Services Pvt. Ltd, Pondicherry, India
www.integra-india.com
Printed and bound in Meppel, The Netherlands by Krips bv.

Contents

Preface

A potential reader of this text may be forgiven for initially viewing this book as yet another text on the ubiquitous 80C51 microcontroller, a topic on which many books have already been published. However, the authors believe their application-based coverage using only Flash memory devices will bring home to the reader a depth of coverage and an understanding of the versatility of the various members of the 80C51 family, including the 16-bit devices, that have not been seen before. Three devices in particular are described in the text with their own chapters and relevant appendices. The devices are those available from Philips Semiconductors although the applications, both hardware and software, have a broader scope and could apply to other manufacturer's devices.

The text includes a chapter on simulation, using evaluation software that can be downloaded on to a computer. Such software allows the user to compile their program and simply run it to achieve an objective or single step through their program to establish how the program affects registers, timers, ports, etc., as the program develops. It is hoped that the reader will wish to go beyond simulation and interface with external inputs/outputs via an actual microcontroller. Artwork is included, in an appendix, for a single-sided pcb that could be used for the construction of a development board. Two different boards are described; one board is designed for an 8-bit device while the other is for a 16-bit device, both devices being covered in the text. Information relating to the microcontroller boards can be found in Chapter 2. The use of a Flash microcontroller board and in-system programming techniques allows the user to simulate and debug his/her program and refine it before downloading it to the microcontroller. Source code could then be removed, if required, and replaced with a new program to serve a different purpose. The use of a microcontroller board allows an interface to the outside world and the effect of the program stored in the microcontroller can be observed in real time i.e. to light LEDs, cause a motor shaft to rotate, etc. For those not wishing to have their own microprocessor board the text still offers the opportunity to simulate programs and much can be learnt about the devices by its use.

Three members of the Philip's 80C51 family have been utilised in the text to explain circuit action and used as the basis for specific applications. Several

appendices complete the story with details on the 8-bit and 16-bit microcontroller instruction set and manufacturers' data on the devices.

The book is intended to be read on a chapter by chapter basis for those new to the subject and in this format would be suitable for those on degree, including postgraduate, courses. The text would also be suitable for any reader familiar with the devices but requiring information that takes them somewhat deeper into the detail and applications. For such readers some of the chapters could be omitted and particular chapters studied in more depth. Practising engineers could find the text helpful as an aid to the development of prototype systems prior to full-scale commercial application. Chapters dealing with specific devices have numerous examples to help reinforce key points, and there are also numerous exercises for the reader to attempt if they so wish; answers to these exercises can be found at the end of the book. Relevant appendices can be used for reference where necessary.

The text assumes that readers have some experience of programming although some information on assembly language programming can be found in Chapter 1. Programming examples have been implemented using assembly language and C.

The authors have attempted to show throughout the text programming applications where relevant, and the final chapter is devoted to practical applications that the authors have found to work. Notwithstanding this, the authors can accept no responsibility for any program that a reader might attempt and find unsatisfactory.

It should perhaps be mentioned at this point that the Department of Electronic and Computer Engineering at the University of Portsmouth is a Philips Accredited Product Expert Centre, one of only five in England. Short courses relevant to industry are run at the Centre on a regular basis and the Centre is kept up to date on new developments relating to Philips Semiconductors Ltd devices.

The authors hope that all readers of this text will find the information therein of some use in their studies and/or as a reference text.

David M Calcutt, Frederick J Cowan and G Hassan Parchizadeh

Acknowledgements

The authors would like to take the opportunity to thank all those individuals and/or companies who have contributed or helped in some way in the preparation of this text. Particular thanks must go to Philips Semiconductors Ltd* for their encouragement and for permission to use so much information from Philips' sources. Thanks also are due to Keil** for the use of their evaluation software for the 8-bit microcontrollers; to Raisonance† for the use of their evaluation software for both 8-bit and XA 16-bit microcontrollers and to Maxim‡ for the use of some items from their range of devices. Our thanks also to Andy Mondey for his assistance in the production of the Flash microcontroller pcb artwork.

Also the authors would like to thank their respective wives for their understanding and forbearance shown when the preparation of the book took time that could have been spent with the family. Our thanks then to David's wife Daphne, Fred's wife Sheila and Hassan's wife Hoory.

Additionally Hassan would like to dedicate his contribution to this text to the memory of his mother and would also like to express his gratitude to both his mother and father for their encouragement and support over the years.

* Philips Semiconductors Ltd. Head Office. PO Box 218, 5600 MD Eindhoven, The Netherlands. www.semiconductors.philips.com

** Keil. Head Office. Keil Elektronik GmbH, Bretonischer Ring 15, D-85630, Grasbrunn, Germany. www.keil.com

† Raisonance. Head Office. RAISONANCE, 17 Avenue Jean Kuntzmann, 38330 Montbonnot, France. www.raisonance.com

‡ Maxim Integrated Products. Head Office. 120 San Gabriel Drive, Sunnyvale, CA 94086, USA. www.maxim-ic.com

This page intentionally left blank

1

Introduction to Microcontrollers

1.1 Introduction

A microcontroller is a computer with most of the necessary support chips onboard. All computers have several things in common, namely:

- A central processing unit (CPU) that 'executes' programs.
- Some random-access memory (RAM) where it can store data that is variable.
- Some read only memory (ROM) where programs to be executed can be stored.
- Input and output (I/O) devices that enable communication to be established with the outside world i.e. connection to devices such as keyboard, mouse, monitors and other peripherals.

There are a number of other common characteristics that define microcontrollers. If a computer matches a majority of these characteristics, then it can be classified as a 'microcontroller'. Microcontrollers may be:

- 'Embedded' inside some other device (often a consumer product) so that they can control the features or actions of the product. Another name for a microcontroller is therefore an 'embedded controller'.
- Dedicated to one task and run one specific program. The program is stored in ROM and generally does not change.
- A low-power device. A battery-operated microcontroller might consume as little as 50 milliwatts.

A microcontroller may take an input from the device it is controlling and controls the device by sending signals to different components in the device. A microcontroller is often small and low cost. The components may be chosen to minimise size and to be as inexpensive as possible.

The actual processor used to implement a microcontroller can vary widely. In many products, such as microwave ovens, the demand on the CPU is fairly low

and price is an important consideration. In these cases, manufacturers turn to dedicated microcontroller chips – devices that were originally designed to be low-cost, small, low-power, embedded CPUs. The Motorola 6811 and Intel 8051 are both good examples of such chips.

A typical low-end microcontroller chip might have 1000 bytes of ROM and 20 bytes of RAM on the chip, along with eight I/O pins. In large quantities, the cost of these chips can sometimes be just a few pence.

In this book the authors will introduce the reader to some of the Philips' 8051 family of microcontrollers, and show their working, with applications, throughout the book. The programming of these devices is the same and, depending on type of device chosen, functionality of each device is determined by the hardware devices onboard the chosen device.

1.2 Microcontroller types

The predominant family of microcontrollers are 8-bit types since this word size has proved popular for the vast majority of tasks the devices have been required to perform. The single byte word is regarded as sufficient for most purposes and has the advantage of easily interfacing with the variety of IC memories and logic circuitry currently available. The serial ASCII data is also byte sized making data communications easily compatible with the microcontroller devices. Because the type of application for the microcontroller may vary enormously most manufacturers provide a family of devices, each member of the family capable of fitting neatly into the manufacturer's requirements. This avoids the use of a common device for all applications where some elements of the device would not be used; such a device would be complex and hence expensive. The microcontroller family would have a common instruction subset but family members differ in the amount, and type, of memory, timer facility, port options, etc. possessed, thus producing cost-effective devices suitable for particular manufacturing requirements. Memory expansion is possible with off-chip RAM and/or ROM; for some family members there is no on-chip ROM, or the ROM is either electrically programmable ROM (EPROM) or electrically erasable PROM (EEPROM) known as flash EEPROM which allows for the program to be erased and rewritten many times. Additional on-chip facilities could include analogue-to-digital conversion (ADC), digital-to-analogue conversion (DAC) and analogue comparators. Some family members include versions with lower pin count for more basic applications to minimise costs. Many microcontroller manufacturers are competing in the market place and rather than attempting to list all types the authors have restricted the text to devices manufactured by one maker. This does not preclude the book from being useful for applications involing other manufacturer's devices; there is a commonality among devices from various sources, and descriptions within the text can, in most cases, be applied generally. The chapters that follow will deal with microcontroller family members available from Philips Semiconductors, and acknowledgement is due to the considerable assistance given by that

company in the production of this text. The Philips products are identified by the numbering system:

8XCXXX

where in general, since there are exceptions, the digit following the 8 is:

0 for a ROMless device
3 for a device with ROM
7 for a device with EPROM/OTP (one time programmable)
9 for a device with FEEPROM (flash).

Following the C there may be 2 or 3 digits. Additional digits, not shown above, would include such factors as clock speed, pin count, package code and temperature range. Philips also produces a family of 16-bit microcontrollers in the eXtended Architecture (XA) range. For these devices Philips claims compatibility with the 80C51 at source code level with full support for the internal registers, operating modes and 8051 instructions. Also claimed is a much higher speed of operation than the 8051 devices. The XA products are identified by the numbering system:

PXAG3XXXX

where:

PXA is Philips 80C51 XA
G3 is the derivative name
next digit is memory option:
  0 = ROM less
  3 = ROM
  5 = Bond-out (emulation)
  7 = EPROM/OTP
  9 = FEEPROM (flash)
next digit is speed:
  J = 25 MHz
  K = 30 MHZ
next digit is temperature:
  B = 0°C to + 70°C
  F = −40°C to + 85°C
final digit is package code:
  A = Plastic Leaded Chip Carrier (PLCC)
  B = Quad Flat Pack (QFP)
etc.

The XA architecture supports:

- 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;

- 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 multiplies and 32/16 divide;
- instruction set tailored for high-level language support;
- multitasking and real-time executives that include up to 32 vectored interrupts, 16 software traps, segmented data memory and banked registers to support context switching.

The next section of this chapter will look at a member of the Philips 80C51 family in more detail.

1.3 P89C66x microcontroller

Figure 1.1 shows a P89C664 microcontroller in a PLCC package.

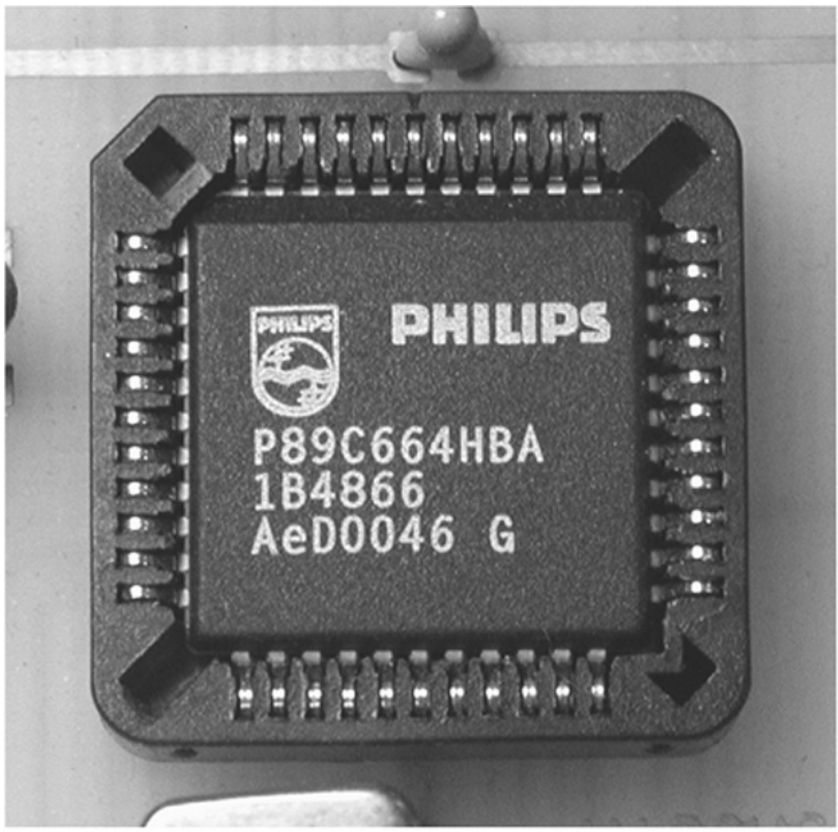


Figure 1.1 Philips P89C664 PLCC package microcontroller

P means the device is manufactured by Philips Semiconductors
8 means the micro belongs to the 8-bit 8051 family
9 means Flash code (program) memory
C means CMOS technology and
664 belongs to the 66x family

where:

x = 0 16 KB Flash code memory, 512 bytes onboard RAM
x = 2 32 KB Flash code memory, 1 KB onboard RAM
x = 4 64 KB Flash code memory, 2 KB onboard RAM
x = 8 64 KB Flash code memory, 8 KB onboard RAM

All devices belonging to this family of devices have a universal asynchronous receive transmit (UART), which is a serial interface similar to the COM interface on a PC. Figure 1.2 shows the logic symbol for the device and illustrates the pin functions.

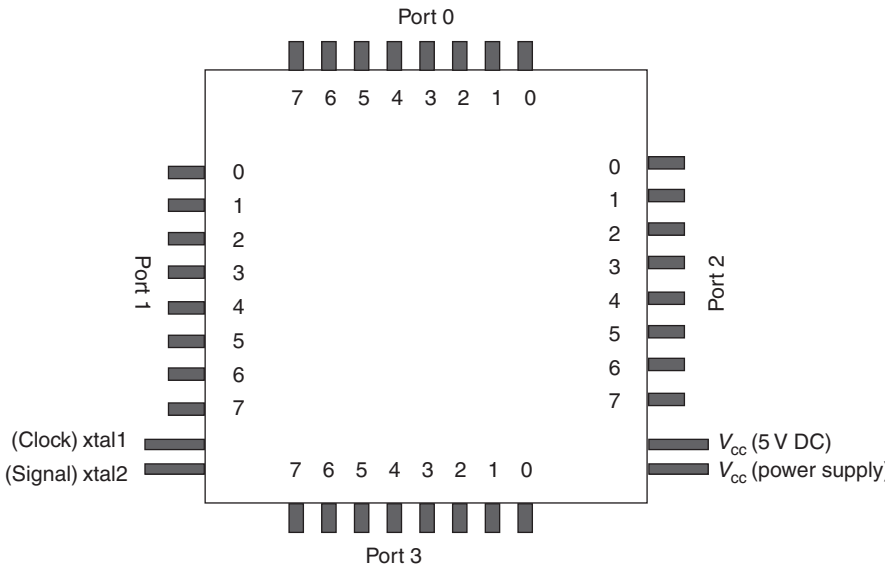


Figure 1.2 Logic symbol for the P89C66x family

The P89C66x family of microcontrollers have four 8-bit ports: port 0, port 1, port 2 and port 3.

Traditionally in the 80C51 family of microcontrollers the function of port 0 and port 2 is primarily to allow for connection to an external PROM (code memory chip). Port 0 provides both the 8-bit data and the lower 8 bits of the address bus, A0 to A7. Port 2 provides the upper 8 address bits, A8 to A15. All of the flash microcontrollers referred to in this text have onboard code memory, which can be as much as 64 KB.

Port 0 pins are all from open-drain transistors and the port pins should have pull-up resistors (e.g. 2.7 kΩ from pin to 5 V DC supply) if the port is to be used as a general-purpose interface.

Port 3 has some special function pins, e.g. pins 0 and 1 of port 3 may be used as receive and transmit for the UART. Functions of other pins will be covered in later chapters.

In the 80C51 family of microcontrollers the RAM is organised into 8-bit locations.

MSB							LSB
7	6	5	4	3	2	1	0

The bits are numbered 7, 6, 5, 4, 3, 2, 1, 0 where bit 7 is the most significant bit (MSB) and bit 0 the least significant bit (LSB).

A bit (binary digit) has two values, logic 0 or logic 1. Electrically logic 0 is 0 V whereas logic 1 is the value of the microcontroller IC positive supply voltage. Logic 1 is usually 5 V but nowadays with increasing use of batteries for power supplies logic 1 could be 3 V or 1.8 V.

Power depends on the square of the voltage and there is a significant saving in power (i.e. battery lasts longer) if the microcontroller is powered by 3 V or 1.8 V power supplies.

The maximum number that can be stored in an 8-bit memory location is $2^8 - 1$, which equals 255. This would occur when all the bits are equal to 1 i.e.:

MSB							LSB
1	1	1	1	1	1	1	1

Binary is a base 2 number system and the electronic devices in the microcontroller's logic circuits can be set to logic 0 and logic 1.

The value of each bit is:

MSB							LSB
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Example 1.1
Show that if an 8-bit register contains all logic 1s then the value stored is 255.

Solution
With all bits of the register set to logic 1 the total value stored is given by:

$$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

Remember the sequence by recalling that the LSB is 1 and the other numbers are successively doubled.

Exercise 1.1
What is the maximum number that can be stored in a 10-bit wide register?

1.4 Bits, nibbles, bytes and number conversions

BITS, BYTES AND NIBBLES

A bit is a single binary digit of value logic 1 or logic 0. A nibble is a group of 4 bits, e.g. 1010 is a nibble. A byte is a group of 8 bits e.g. 10100111 is a byte and the byte is made up of two nibbles 1010 and 0111.

DECIMAL TO BINARY CONVERSION

A decimal number may be converted to binary by dividing the number by 2, giving a quotient with a remainder of 0 or 1. The process repeats until the final quotient is 0. The remainders with the first remainder being the least significant digit determine the binary value. The process is best explained with an example.

Example 1.2
Express the decimal number 54 as a binary number.

Solution

$54/2 = 27,$ remainder 0
$27/2 = 13,$ remainder 1
$13/2 = 6,$ remainder 1
$6/2 = 3,$ remainder 0
$3/2 = 1,$ remainder 1
$1/2 = 0,$ remainder 1

Thus 54 decimal = 110110 and in an 8-bit register the value would be 00110110. A binary value is often expressed with a letter B following the value i.e. 00110110B.

It may be easier to use the weighted values of an 8-bit register to determine the binary equivalent of a decimal number i.e. to break the decimal number down to those weighted elements, which have a logic 1 level.

Example 1.3
Express the decimal number 54 as a binary number using weighted values.

Solution

$$54 = 32 + 16 + 6$$

$0 \times 128 \quad + 0 \times 64 \quad + 1 \times 32 \quad + 1 \times 16 \quad + 0 \times 8 \quad + 1 \times 4 \quad + 1 \times 2 \quad + 0 \times 1$

0	0	1	1	0	1	1	0

Hence 54 decimal = 00110110B.

Example 1.4
Express decimal 167 as a binary number.

Solution
Using the technique of Example 1.3:

$$167 = 128 + 32 + 4 + 2 + 1 = 10100111\text{B}$$

Exercise 1.2
Represent decimal numbers 15 and 250 in binary format.

It follows that to convert binary to decimal the reverse procedure applies i.e. to convert the binary number 00110110 to decimal is achieved by simply adding the weighted values of the logic 1 states. This is shown in the answer to Example 1.3 where 00110110B = 54 decimal.

BINARY, HEXADECIMAL (HEX) AND DECIMAL

When working out values at the port pins, the tendency is to think in binary, e.g. which LED to turn on, the logic level on a switch, etc.

The assembly language software tends to use hexadecimal, a base 16 number system useful for grouping nibbles. Since childhood we have been taught to become familiar with the base 10 decimal system. It is useful to be able to work between the three number systems:

Binary	Hex	Decimal
0 0 0 0	00	00
0 0 0 1	01	01
0 0 1 0	02	02
0 0 1 1	03	03
0 1 0 0	04	04
0 1 0 1	05	05
0 1 1 0	06	06
0 1 1 1	07	07
1 0 0 0	08	08
1 0 0 1	09	09
1 0 1 0	0A	10
1 0 1 1	0B	11
1 1 0 0	0C	12
1 1 0 1	0D	13
1 1 1 0	0E	14
1 1 1 1	0F	15

Consider the following examples:

Example 1.5
Express ABCD as binary.

Solution

$$\text{ABCD} = 1010\,1011\,1100\,1101$$

Example 1.6
Express 101111000001 as a hexadecimal value.

Solution

$$1011\ 1100\ 0001 = \text{BC1}$$

Example 1.7
Express 01110011110 as a hexadecimal value.

Solution

$$0011\ 1001\ 1110 = \text{39E}$$

Because in this last example the number of bits does not subdivide into groups of four bits, the method used is to group into nibbles from the right, filling the spaces at the front with zeros.

Example 1.8
Express decimal 71 as a hex number.

Solution

$$71/16 = 4 \text{ remainder } 7 = 47 \text{ Hex, usually written as } 47\text{H}$$

Example 1.9
Express decimal 143 as a hex number.

Solution

$$143/16 = 8 \text{ remainder } 15 = 8\text{FH}$$

Conversion from binary to decimal can be achieved quickly by first converting the binary number to hex and then converting the hex number to decimal. An example illustrates the process.

Example 1.10
Express 11000101B in decimal form.

Solution
Converting to hex:

$$11000101\text{B} = \text{C5H}$$

The hex number represents a nibble of binary data and each value is to a power of 16 with the least significant nibble equal to $16^0(=1)$ and the next significant nibble equal to $16^1(=16)$. Hence the decimal number is:

$$(C \times 16) + (5 \times 1) = (12 \times 16) + (5 \times 1) = 197 \text{ decimal}$$

Check:

$$11000101 = (1 \times 128) + (1 \times 64) + (1 \times 4) + (1 \times 1) = 197 \text{ decimal}$$

Exercise 1.3
Express decimal 200 as a hex number and then as a binary number.

Exercise 1.4
Express the following binary numbers as hex and then decimal numbers.

1. 10000110
2. 10011000011

1.5 Inside microcontrollers

Microcontrollers normally contain RAM, ROM (EEPROM, EPROM, PROM), logic circuits designed to do specific tasks (UART, I^2C, SPI) and square-wave oscillator (clock).

Built from the logic circuitry the microcontroller has two parts, the processor core and the onboard peripherals. See Figure 1.3.

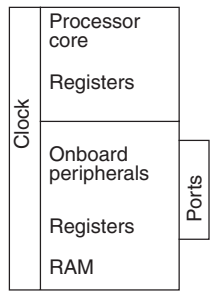


Figure 1.3 Constituent parts of a microcontroller

RAM locations that have special functions and support the processor core and onboard peripheral circuitry are called special function registers (SFRs) and are reserved areas.

The program instructions provide the primary inputs to the processor core circuitry. See Figure 1.4.

The microcontroller program resides in the PROM (programmable ROM), which, in the microcontrollers we are considering, uses Flash technology and is located in the microcontroller IC.

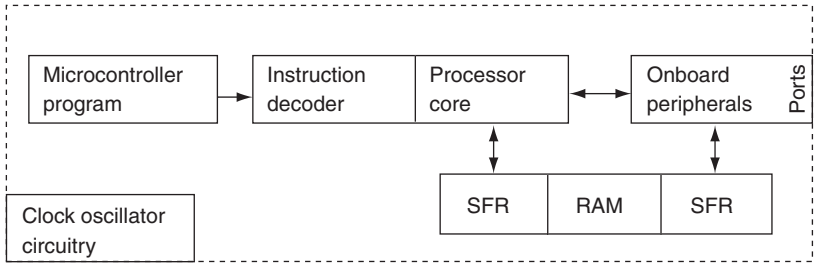


Figure 1.4 Block diagram of a microcontroller

1.6 Microcontroller programming

The microcontroller program comprises a set of instructions written by the program designer. There are four classes of instructions:

1. Arithmetic operations
2. Logic operations
3. Data transfer operations
4. Branch operations.

ARITHMETIC OPERATIONS

Arithmetic instructions operate on whole numbers only and support addition, subtraction, multiplication and division.

Addition

```
ADD   A,#66H   ; add the hex number 66 to the accumulator A
```

This is an example of immediate addressing.

The # sign is important, if it were omitted the operation would have a different meaning.

```
ADD   A,66H    ; add to accumulator A the contents of RAM address
               ; 0066H
```

This is an example of direct addressing.

Accumulator A is an SFR; it is an 8-bit register and its RAM address is 00E0H. A large number of instructions use accumulator A, but not all.

```
INC   66H      ; increment (add 1) the contents of address 0066H
```

Exercise 1.5
Is there any difference between the following two instructions?
A) INC A B) ADD A,#1

Subtraction

```
SUBB   A, #66H   ; subtract hex66 from the contents of A
```

The extra B in the instruction implies Borrow. If the contents of A are less than the number being subtracted then bit 7 of the program status word (PSW) SFR will be set. (For details of the PSW and other SFRs, see Appendix C.)

```
DEC   A   ; decrement A by 1, put result into A
```

Exercise 1.6
Is there any difference between the following two instructions?

(1) DEC A (2) SUBB A,#1

Multiplication

```
MUL   AB   ; multiply the contents of A and B, put the answer in AB
```

A is the accumulator and B is another 8-bit SFR provided for use with the instructions multiply and divide. A and B are both 8-bit registers. The product of the multiplication process could be a 16-bit answer.

Example 1.11
A = 135 decimal, B = 36 decimal. What would be the value in each register after executing the instruction MUL AB?

Solution

$A \times B = 4860 = 0001\ 0010\ 1111\ 1100B = 12FCH$
0001 0010 or 12H would be placed in A, 1111 1100 or FCH in B

Exercise 1.7
If A = 2FH and B = 02H, what would each register contain after execution of the instruction MUL AB?

Division

```
DIV   AB   ; divide A by B, put quotient in A and remainder in B
```

Example 1.12
A = 135, B = 36. What would be the value in each register after execution of the instruction DIV AB?

Solution
Decimal values are assumed if the value quoted is not followed by an H

A/B = 3, remainder 27 (27 = 1BH). Hence 03H in A, 1BH in B

If multiplication or division is not being used then register B, which is bit addressable, can be used as a general-purpose register.

Exercise 1.8
If A = 2FH and B = 02H, what would each register contain after the execution of the instruction DIV AB?

LOGIC OPERATIONS

The set of logic functions include:

ANL	AND Logic
ORL	OR Logic
XRL	exclusive OR Logic
CPL	Complement (i.e. switch to the opposite logic level)
RL	Rotate Left (i.e. shift byte left)
RR	Rotate Right (i.e. shift byte right)
SETB	Set bit to logic 1
CLR	Clear bit to logic 0

AND operation

The ANL instruction is useful in forcing a particular bit in a register to logic 0 whilst not altering other bits. The technique is called masking.

Suppose register 1 (R1) contains EDH (1110 1101B),

1	1	1	0	1	1	0	1

bit 1 and bit 4 are at logic 0, the rest at logic 1.

```
ANL   R1, #7FH   ; 7FH = 0111 1111B, forces bit 7 to zero
```

1	1	1	0	1	1	0	1

AND

0	1	1	1	1	1	1	1

=

0	1	1	0	1	1	0	1

Exercise 1.9
If A = 2D, what would be the accumulator contents after execution of the instruction ANL A, #3BH?

ORL operation

Another aspect of masking is to use the ORL instruction to force a particular bit to logic 1, whilst not altering other bits.

The power control (PCON) SFR in the 8051 family, is not bit addressable and yet has a couple of bits that can send the microcontroller into idle mode or power down mode, useful when the power source is a battery.

The contents of the PCON SFR are:

PCON

SMOD1	SMOD2		POF	GPF1	GPF2	PD	IDL

SMOD1 and 2 are used when setting the baud rate of the serial onboard peripheral. POF, GPF1 and GPF2 are indictor flag bits. IDL is the idle bit; when set to 1 the microcontroller core goes to sleep and becomes inactive. The on-chip RAM and SFRs retain their values. PD is the Power Down bit, which also retains the on-chip RAM and SFR values but saves the most power by stopping the oscillator clock.

```
ORL   PCON,#02H   ; enables Power Down
ORL   PCON,#01H   ; enables Idle mode
```

Either mode can be terminated by an external interrupt signal. Details of all device SFRs are to be found in Appendix C.

Exercise 1.10

If the contents of register 0 (R0) = 38H, what would the contents of that register be after execution of the following instruction?

```
ORL   R0,#9AH
```

CPL complement operation

The instructions described so far have operated on bytes (8 bits) but some instructions operate on bits and CPL is an example.

```
CPL   P1.7   ; complement bit 7 on Port 1
```

Port 1 is one of the microcontroller's ports with 8 pins.

Port 1

P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0

MSB LSB

Complement has the action of the inverter logic gate as shown in Figure 1.5.

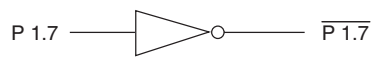


Figure 1.5 Production of the complement of pin function

Exercise 1.11
If the contents of port 0 (P0) = 125, what would be the port contents after execution of the following instruction?

```
CPL   P0
```

RL, rotate left one bit, RR, rotate right one bit operations

Suppose the accumulator A contents are 0000 0001B; this is 01H.

```
RL A   ; contents of A become 0000 0010B or 02H
RL A   ; 0000 0100B or 04H
RL A   ; 0000 1000B or 08H
```

RL three times has the effect of multiplying A by 2^3 i.e. by 8.
Suppose the accumulator A contents are 1000 0000B, or 128 decimal, then:

```
RR A   ; contents of A become 0100 0000B which is 64 decimal
RR A   ; A becomes 0010 0000B = 32 decimal
RR A   ; A becomes 0001 0000B = 16 decimal
RR A   ; A becomes 0000 1000B = 8 decimal
```

RR four times has the same effect as dividing A by 2^4 i.e. 16.

$$\frac{128}{16} = 8$$

Exercise 1.12
If the content of A is 128 and B is 2, what would the register contents be after execution of the following instructions?

```
RR A
RL B
RR A
RR A
RL B?
```

SETB set bit, CLR clear bit operations

This instruction operates on a bit, setting it to logic 1.

```
SETB   P1.7   ; set bit 7 on Port 1 to logic 1
```

Consider Figure 1.6 where pin 7 of port 1 is connected as shown.
SETB P1.7 puts logic 1 (e.g. 5 V) onto the inverter input and therefore its output, the LED cathode, is at 0 V causing current to flow through the LED. The LED has a particular forward voltage V_f (refer to component specification e.g. www.farnell.com).

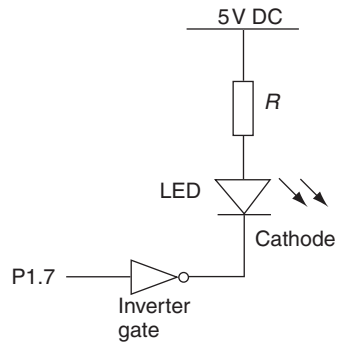


Figure 1.6 Use of an LED to indicate the state of port 1, pin 7

Typically $V_f = 2.2\,V$ and forward current $I_f = 8\,mA$ so that:

$$R = \frac{5\,V - V_f}{I_f} = \frac{5 - 2.2}{8 \times 10^{-3}} = \frac{2.8 \times 1000}{8} = 350\,\Omega = 330\,\Omega \text{ (preferred value)}$$

```
CLR    P1.7    ; clears bit 7 on port 1 to zero
```

CLR P1.7 puts logic 0 on the inverter gate input and therefore its output, the LED cathode, becomes logic 1 which is 5 V. This gives a voltage difference (5 V DC – cathode voltage) of 0 V and the LED turns off.

The inverter gate in the above circuit provides a good current buffer protecting the microcontroller port pin from unnecessary current loading. In the above circuit the current flow is between the inverter gate and the 5 V DC supply.

If an inverter gate is not used to drive a LED then the control may be directly from the port pin but this will demand a current in milliamps from the port pin.

Generally a microcontroller port pin can sink current better than it can source current. See Figure 1.7.

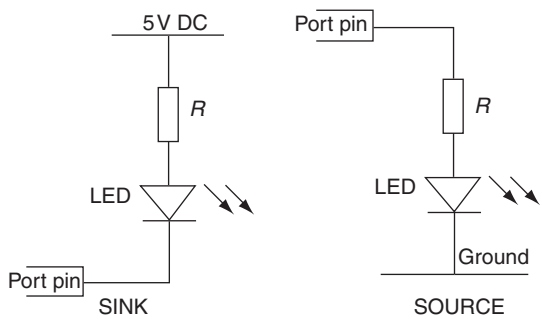


Figure 1.7 Arrangements to allow a port pin to SINK or SOURCE current

CLR port_pin; will ground the LED cathode in the SINK circuit and turn it on. This will turn the LED off in the SOURCE circuit.

SETB port_pin; will put logic 1 on the LED cathode in the SINK circuit and turn it off. This will turn the LED on in the SOURCE circuit.

Exercise 1.13
If $V_{cc} = 5\,V$ and for an LED, $V_f = 0.7\,V$ and the pin P0.0 of the microcontroller port can sink 10 mA and source 50 μA.

1. How you connect the LED to the microcontroller and
2. Calculate the value of series resistor R.

Data transfer operations

This is mainly concerned with transfer of data bytes (8 bits). SETB and CLR have just been covered; they operate on bits.

MOV operation

MOV moves bytes of data. Consider driving a seven-segment display (decimal point dp included) where each LED is driven by the sink method. See Figure 1.8.

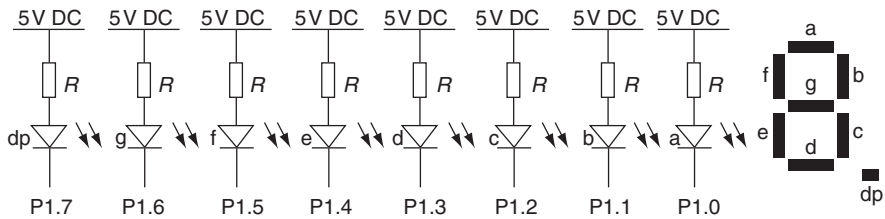


Figure 1.8 Arrangement for a seven-segment LED display

Each LED illuminates a segment. The seven-segment display is shown to the right with its standard segment identification letters.

Example 1.13
Write two program lines, one to display 3, the second to display 4. In both cases turn the decimal point off.

Solution

	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0	
	dp	g	f	e	d	c	b	a	
3	1	0	1	1	0	0	0	0	B0H
4	1	0	0	1	1	0	0	1	99H

```
MOV   P1,#B0H   ; display 3
MOV   P1,#99H   ; display 4
```

Note: MOV P1,#B0H would give a syntax error. In common with a number of cross assemblers the software would see B0H as a label because it starts with a hex symbol; 99H would be acceptable since it starts with a number. The correct program line should be MOV P1,#0B0H i.e. a zero must be placed in front of the hex symbol.

The instruction MOV is used to move RAM data that is onboard the microcontroller.

Examples

```
MOV   0400H,#33H   ; move the number 33 hex to RAM address 0400 hex
MOV   A,P1         ; move the contents of port 1 to accumulator A
MOV   R0,P3        ; move the contents of port 3 into register R0
```

Note: As well as the accumulator A the microcontroller has 32 registers in four banks of eight in the processor core. These 32 bytes are fast RAM and should be used in preference to standard onboard RAM.

Each of the banks contain 8 registers R7, R6, R5, R4, R3, R2, R1, R0. There are four banks: 0, 1, 2 and 3.

Bank 0 is the default bank; the other banks can be selected by two bits (RS1,RS0) in the program status word (PSW) SFR

PSW

CY	AC	F0	RS1	RS0	OV	F1	P

0	0	Register bank 0 (default)
0	1	Register bank 1
1	0	Register bank 2
1	1	Register bank 3

Other PSW bits are indicator flags:

CY (carry flag)
AC (auxiliary carry flag)
OV (overflow flag)
P (parity flag)
F0, F1 (general-purpose user-defined flags)

More information on the register banks and the SFRs can be found in Appendix C. MOVX is used to move data between the microcontroller and the external RAM. MOVC is used to move data (e.g. table data) from PROM (also called code memory) to RAM.

Exercise 1.14
Write an instruction to select the register bank 2 of the microcontroller.

Branch operations

There are two types, unconditional and conditional branching. Unconditional branch operations are

ACALL absolute call
LCALL long call

ACALL calls up a subroutine, the subroutine must always have RET as its last operation. ACALL range is limited to +127 places forward or −128 places backward. If your program jumps further than ACALL the compiler will report that the program is jumping out of bounds and replacement by LCALL will solve the problem.

ACALL is two bytes long, LCALL is three bytes long.

AJMP absolute jump
LJMP long jump
SJMP short jump

Similar to ACALL and LCALL, AJMP and LJMP jump to addresses whereas SJMP, which has a similar range to ACALL and AJMP, jumps a number of places.

The difference could be seen in the machine code. Consider the program:

```
$INCLUDE (REG66X.INC)     ; lists all sfr addresses
         ORG    0         ; sets start address to 0
         SJMP   START     ; short jump to START label
         ORG    0040H     ; puts next program line at address 0040H
START:   SETB   P1.7      ; set pin 7 on port 1 to logic 1
         CLR    P1.7      ; clear pin 7 on port 1 to logic 0
         AJMP   START     ; jump back to START label
         END               ; no more assembly language
```

The machine code can be viewed in the list file, progname.lst:

SJMP START
Shows as 803E 80 is the hex for instruction SJMP
3E is the relative jump to reach 0040H where START is; it jumps from address 0002, the address after SJMP START, 0002 + 3E = 0040H

AJMP START
Shows as 0140 01 is the hex for instruction AJMP
40 is short for address 0040

If LJMP had been used instead of AJMP then,
LJMP START
Shows as 020040 02 is the hex for instruction LJMP
0040 is the full address

Exercise 1.15
In your own words describe the difference between ACALL and AJMP instructions.
Conditional branch operations:

JZ Jump if zero
JNZ Jump if not zero
DJNZ Decrement and jump if not zero

Consider an example (a subroutine called by ACALL):

```
DELAY:   MOV    R0,#34      ; move decimal 34 into register R0
TAKE:    DJNZ   R0,TAKE     ; keep subtracting 1 from R0 until zero
RET                         ; return from subroutine
```

CJNE Compare and jump if not equal
Consider:

```
DELAY:   MOV    R0,#34          ; move decimal 34 into register R0
TAKE:    DEC    R0              ; decrement R0
         CJNE   R0,#12,TAKE     ; compare R0 with 12 jump to TAKE if not
         RET                    ; return when R0 equals 12
```

Other instructions are:

JC jump if carry is 1
JNC jump if carry is 0

JB jump if bit = 1
JNB jump if bit = 0

Consider a practical example of testing switched logic levels. Refer to Figure 1.9. If the switch is not pressed the voltage on the port pin is 0 V. When the switch is pressed and held, then the port pin is connected directly to 5 V. To test for switch being pressed, the following program could be used:

```
$INCLUDE (REG66X.INC)       ; lists all sfr addresses
         ORG    0           ; sets start address to 0
         SJMP   START       ; short jump to START label
         ORG    0040H       ; puts next program line at address 0040H
START:   JB     P1.0,PULSE  ; jump to PULSE if pin 0 port 1 is logic 1
         CLR    P1.7        ; otherwise clear pin 7 port 1 to zero
         SJMP   START       ; go to START check switch
```

```
PULSE:   SETB   P1.7          ; set pin 7 on port 1 to logic 1
         CLR    P1.7          ; clear pin 7 on port 1 to logic 0
         AJMP   START         ; go to START check switch
         END                  ; no more assembly language
```

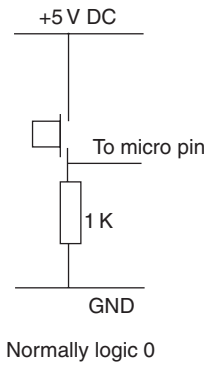


Figure 1.9 Circuit to produce logic levels 0 or 1 at a port pin. Circuit normally producing logic 0

Also, consider the case when pressing the switch generates a logic '0', as shown in Figure 1.10.

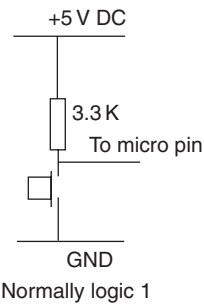


Figure 1.10 Circuit to produce logic levels 0 or 1 at a port pin. Circuit normally producing logic 1

If the switch is not pressed the voltage on the port pin is 5 V. When the switch is pressed and held, the port pin is directly connected to ground or 0 V. The test instruction could be:

```
CHECK:   JNB    P1.0,PULSE   ; jump to PULSE if pin 0 port 1 is logic 0
         SJMP   CHECK
PULSE:
```

Exercise 1.16
In your own words describe the difference between JNB and JNC instructions.

1.7 Commonly used instructions of the 8051 microcontroller

The P89C664 is a member of the 8051 family; it is a CISC device having well over 100 instructions. The instructions used in this text could be the first set to become familiar with.

MOV	move a byte
SETB CLR	set or clear bits
ACALL RET	call up a subroutine
SJMP AJMP	unconditional jump
JB JNB	bit test, conditional jump
DJNZ CJNE	byte test, conditional jump
ORL	OR logic, useful for forcing bits to logic 1
ANL	AND logic, useful for forcing bits to logic 0

The full 8051 instruction set is shown in Appendix A.

COMMONLY USED ASSEMBLER DIRECTIVES

ORG	define address
DB	define bytes, useful for table data
END	all assembly language programs must end with this.

1.8 Microcontroller clock

The microcontroller may be likened to a logic circuit whose logic states change in synchronism with the microcontroller clock signal. This is a square-wave signal as shown in Figure 1.11.

Knowledge of the microcontroller clock cycle time is useful in defining timing events used in applications.

Example 1.14
A P89C664 microcontroller has a clock frequency of 11.0592 MHz. What is the time for each cycle?

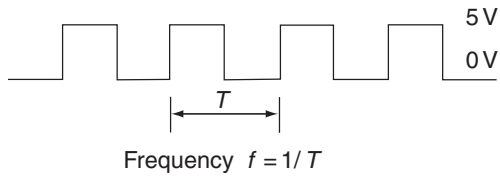


Figure 1.11 Square-wave signal at a frequency *f* Hz

Solution

$$\text{Cyclic time } (T) = \frac{1}{11.0592 \times 10^6} = 90.423\,\text{ns} \qquad (n = 10^{-9})$$

Now let us look at the previous program:

```
$INCLUDE (REG66X.INC)       ; lists all sfr addresses
         ORG    0           ; sets start address to 0
         SJMP   START       ; short jump to START label
         ORG    0040H       ; puts next program line at address 0040H
START:   JB     P1.0,PULSE  ; jump to PULSE if pin 0, port 1 is logic 1
         CLR    P1.7        ; otherwise clear pin 7 port 1 to zero
         SJMP   START       ; go to START check switch
PULSE:   SETB   P1.7        ; set pin 7 on port 1 to logic 1
         CLR    P1.7        ; clear pin 7 on port 1 to logic 0
         AJMP   START       ; go to START check switch
         END                ; no more assembly language
```

Initial inspection might lead to the conclusion that the output signal on port 1, pin 7 is a square wave being turned on by SETB and off by CLR. Closer inspection reveals that CLR is held for the extra duration of AJMP and JB. Reference to Appendix A shows that:

SETB	takes 6 microcontroller clock cycles
CLR	takes 6 microcontroller clock cycles
AJMP	takes 12 microcontroller clock cycles
JB	takes 12 microcontroller clock cycles

SETB is held for 6 clock cycles and CLR is held for 30 clock cycles, not an equal on/off waveform as Figure 1.12 shows.

If an equal on/equal off waveform is required then the NOP (No OPeration) can be used. The NOP operation takes 6 clock cycles. The program could be modified:

```
$INCLUDE (REG66X.INC)       ; lists all sfr addresses
         ORG    0           ; sets start address to 0
         SJMP   START       ; short jump to START label
         ORG    0040H       ; puts next program line at address 0040H
START:   JB     P1.0,PULSE  ; jump to PULSE if pin 0, port 1 is logic 1
         CLR    P1.7        ; otherwise clear pin 7, port 1 to zero
         SJMP   START       ; go to START check switch
```

```
PULSE:   SETB   P1.7          ; set pin 7 on port 1 to logic 1
         NOP                  ; hold logic 1 on pin 7, port 1
         NOP
         NOP
         NOP
         CLR    P1.7          ; clear pin 7 on port 1 to logic 0
         AJMP   START         ; go to START check switch
         END                  ; no more assembly language
```

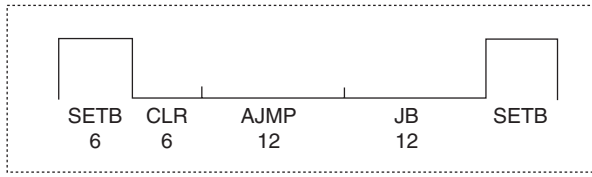


Figure 1.12 Waveform produced using specified instructions. Note that the waveform is not a square wave (i.e. there are unequal ON and OFF periods)

The modified waveform is shown in Figure 1.13.

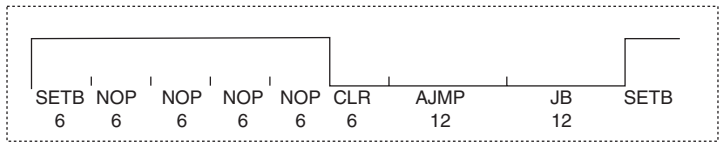


Figure 1.13 Modification to the waveform of Figure 1.12 using NOP instructions to produce a square-wave output

The cycle time of the equal on/off waveform = 60 microcontroller clock cycles. If the microcontroller had a clock frequency of 11.0592 MHz then a clock cycle period T is the reciprocal of this frequency, $T = 90.423$ ns. Therefore the cycle time of the equal on/off signal is $60 \times 90.423\,\text{ns} = 5.43\,\mu\text{s}$. The frequency of this signal is $1/5.43\,\mu\text{s} = 184\,\text{kHz}$. The maximum signal frequency would depend on the maximum microcontroller clock frequency; for the P89C664 microcontroller the maximum clock frequency is 20 MHz. Quite often there is a requirement to generate accurate lower frequency signals and for these the basic signal must be slowed down using a time delay.

1.9 Time delays

The NOP instruction is a simple time delay but apart from this there are two methods of creating time delays:

- register decrement
- onboard timers.

The use of onboard timers will be described in a later chapter; here the register decrement method will be described.

The basic single loop program lines are:

```
DELAY:   MOV     R0,#number      ; move a number into an 8-bit
                                 ; register R0
TAKE:    DJNZ    R0,TAKE         ; keep decrementing R0 until it is
                                 ; zero
         RET                     ; return from DELAY subroutine
```

MOV takes 6 clock cycles, DJNZ and RET each take 12 clock cycles. The delay is called up from the main program using ACALL, which takes 12 clock cycles. The delay time is $(12 + 6 + (\text{number} \times 12) + 12)$ clock cycles. When the number is small the NOPs (total 24 cycles) should be included,

$$\text{Delay time} = (24 + 12 + 6 + (\text{number} \times 12) + 12) \text{ clock cycles}$$
$$\text{Delay time} = (54 + (12 \times \text{number})) \text{ clock cycles}$$

Example 1.15

A P89C664 microcontroller has an 11.0592 MHz crystal-controlled clock oscillator. Write an assembly language program that will generate a 5 kHz square-wave signal on pin 7 of port 1 when a switch causes pin 0 on the same port to go to logic 1.

Solution

Clock frequency = 11.0592 MHz
Thus period of clock cycle = (1/11.0592 MHz) = 90.423 ns
Signal frequency = 5 kHz
Therefore period of signal cycle = (1/5 kHz) = 200 μs

The delay required is half of this value since the square wave has an equal logic 1/logic 0 time. See Figure 1.14.

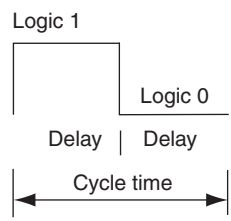


Figure 1.14 Delay period determination for a square-wave signal

$$\text{Delay} = 100\,\mu\text{s} = (54 + (12 \times \text{number})) \times 90.423\,\text{ns}$$

Hence number = ((100 μs/90.423 ns)−54)/12 = 88 decimal (to the nearest whole number).

```
$INCLUDE (REG66X.INC)        ; lists all sfr addresses
         ORG    0            ; sets start address to 0
         SJMP   START        ; short jump to START label
         ORG    0040H        ; puts next program line at address 0040H
START:   JB     P1.0,PULSE   ; jump to PULSE if pin 0, port 1 is logic 1
         CLR    P1.7         ; otherwise clear pin 7, port 1 to zero
         SJMP   START        ; go to START check switch
PULSE:   SETB   P1.7         ; set pin 7 on port 1 to logic 1
         ACALL  DELAY
         NOP                 ; hold logic 1 on pin 7 port 1
         NOP
         NOP
         NOP
         CLR    P1.7         ; clear pin 7 on port 1 to logic 0
         ACALL  DELAY
         AJMP   START        ; go to START check switch
DELAY:   MOV    R0,#88
TAKE:    DJNZ   R0,TAKE
         RET
         END                 ; no more assembly language
```

The delay depended on the chosen microcontroller clock frequency and in the example this was 11.0592 MHz. This apparently unusual number gives standard baud rate values, which will be useful later. For microcontroller clock frequencies in this region the single loop register decrement method gives delays in the region of microseconds. Generally a double loop gives delays in the region of milliseconds and a triple loop delay gives delays in the region of seconds.

Exercise 1.17
Using the techniques above, assuming the clock frequency is 11.0592 MHz, write a program to generate a pulse of 20 kHz on pin 7 of port 1 of the microcontroller.

DOUBLE LOOP DELAY

```
DELAY:  MOV   R1,#number1
INNER:  MOV   R0,#number2
TAKE:   DJNZ  R0,TAKE
        DJNZ  R1,INNER
        RET
```

Approximately the time delay = (number 1) × (number 2) × 12 clock cycle periods. For example, suppose number 1 = 200 and number 2 = 240 and 1 clock cycle = 90.423 ns.

$$\text{Time delay} = 200 \times 240 \times 12 \times 90.423\,\text{ns} = 52.1\,\text{ms}$$

The bigger the values of number 1 and number 2, the better the approximation. The software used has simulation and the values of number 1 and number 2 can be fine tuned to give the accurate delay during simulation.

TRIPLE LOOP DELAY

```
DELAY:  MOV   R2,#number1
OUTER:  MOV   R1,#number2
INNER:  MOV   R0,#number3
TAKE:   DJNZ  R0,TAKE
        DJNZ  R1,INNER
        DJNZ  R2,OUTER
        RET
```

Approximately the delay = (number 1) × (number 2) × (number 3) × 12 clock cycle periods. Suppose number 1 = 40, number 2 = 200, number 3 = 240, 1 clock cycle period = 90.423 ns.

$$\text{Delay} = (40 \times 200 \times 240 \times 12 \times 90.423)\,\text{ns} \approx 2\,\text{s}$$

Long enough to see a LED going on and off.

In later chapters the use of the microcontroller's onboard timers will be used to describe an alternative method of producing time delays. The timer method will require the configuration of the timer SFRs.

The register decrement method described above is a valid alternative, easy to implement and does not require the configuration of SFRs.

Summary

- A microcontroller is a computer with most of the necessary support chips onboard. Microcontrollers can be embedded and are available in a variety of forms to suit practical applications.
- Number systems, such as binary and hexadecimal, are used in microcontroller applications. If decimal numbers are required they can be converted to binary and/or hexadecimal and vice versa.
- There are four classes of instructions namely: arithmetic, logical, data transfer and branch instructions.
- The microcontroller port pins may be required to sink and source currents.
- Time delays may be achieved by using register decrement instructions or by using onboard timer circuits.
- Using register decrement, longer delays can be achieved by the use of double or triple loops.

2

Flash Microcontroller Board

2.1 Introduction

There are three microcontroller families covered in this book, the 8-bit P89C66x, the 16-bit extended architecture (XA) and the low pin count (LPC) devices. The P89C66x devices are essentially flash 8051 microcontrollers with up-to-date features. The XA was publicised by Philips Semiconductors as the 16-bit upgrade of the 8051, and this book covers the XAG49 which is the flash version of the basic XA microcontroller. The LPC76x devices are one time programmable (OTP) EPROM microcontrollers; initially some ultra violet (UV) erasable types were available but this is no longer the case. The number 7 in the device description identifies the technology as EPROM whereas the number 9 identifies flash technology. Currently 28 pin flash LPC932 devices are becoming available. LPC devices belong to the 8051 family and will develop to include 8-pin dual-in-line (DIL) devices.

Philips Semiconductor engineers have produced application notes describing the in-circuit programming of the P89C66x and XAG49 devices and included suggested schematic circuits. Application note AN761_10 describes the technique for the P89C66x devices (_10 is the revision number); AN716_2 is the equivalent application note for the XAG49. The reader should search the www.semiconductors.philips.com web pages for the latest revision.

The authors adapted the schematic designs, and printed circuits boards (PCBs) were produced, one for the P89C664 and one for the XAG49. Each design was based on the 44 pin, plastic leaded chip carrier (PLCC) package. Full size single-sided artworks are provided in Appendix G.

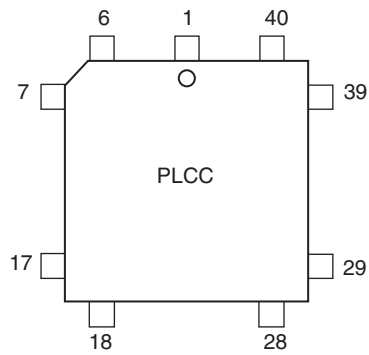
The PCB artwork in Appendix G can be used, together with the schematic circuit diagrams in this chapter to produce a microcontroller board (either for the P89C66x device, the XAG49 device or both). The board can then be used to verify the programs outlined in the relevant chapters as well as providing the user with an opportunity to develop their own programs that can be downloaded into the microcontroller. Components for the board are relatively inexpensive

and can be readily obtained from electronic distribution outlets. Simulation and debugging techniques (described in Chapter 3) can then be used to evaluate the programs prior to downloading into the microcontroller. However, should the reader not wish to make use of a microcontroller board, the applications as described in the relevant chapters can still be followed and understood.

At the time of writing, a similar application note is not available for the P89LPC932 although a low cost evaluation board, the MCB900, is available from Keil; their website is www.keil.com.

2.2 P89C66x microcontroller

The P89C66x is available in either PLCC or linear quad flat pack (LQFP) packages; Figure 2.1 shows the PLCC version.



Pin	Function	Pin	Function	Pin	Function
1	NIC*	16	P3.4/T0/CEX3	31	P2.7/A15
2	P1.0/T2	17	P3.5/T1/CEX4	32	$\overline{\text{PSEN}}$
3	P1.1/T2EX	18	P3.6/$\overline{\text{WR}}$	33	ALE
4	P1.2/ECI	19	P3.7/$\overline{\text{RD}}$	34	NIC*
5	P1.3/CEX0	20	XTAL2	35	$\overline{\text{EA}}/V_{PP}$
6	P1.4/CEX1	21	XTAL1	36	P0.7/AD7
7	P1.5/CEX2	22	V_{SS}	37	P0.6/AD6
8	P1.6/SCL	23	NIC*	38	P0.5/AD5
9	P1.7/SDA	24	P2.0/A8	39	P0.4/AD4
10	RST	25	P2.1/A9	40	P0.3/AD3
11	P3.0/RxD	26	P2.2/A10	41	P0.2/AD2
12	NIC*	27	P2.3/A11	42	P0.1/AD1
13	P3.1/TxD	28	P2.4/A12	43	P0.0/AD0
14	P3.2/$\overline{\text{INT0}}$	29	P2.5/A13	44	V_{CC}
15	P3.3/$\overline{\text{INT1}}$	30	P2.6/A14		

Figure 2.1 Pin functions for the 89C66x PLCC package microcontroller

With reference to Figure 2.1 the following pin details apply: NIC* means no internal connection; V_{SS} is ground and V_{CC} is the 5 V DC supply; RST is the

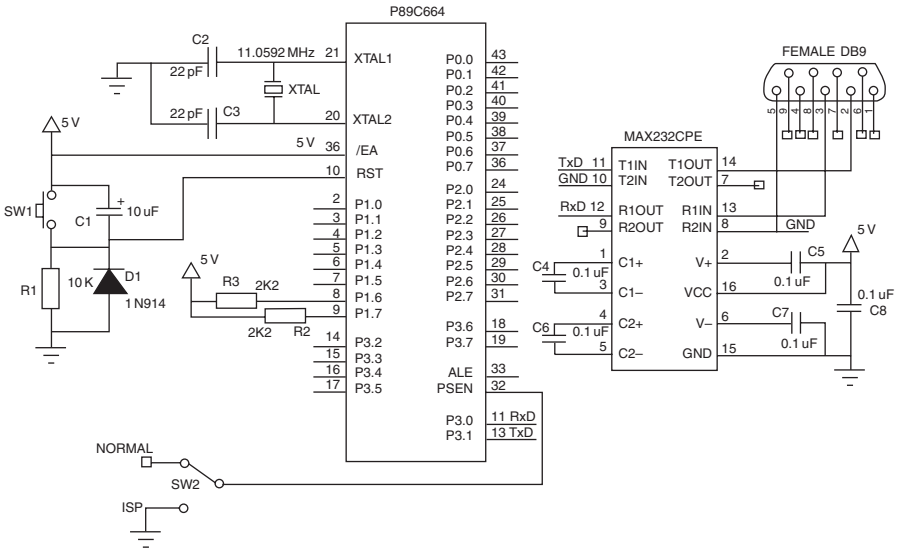


Figure 2.2 Schematic for the 89C66x microcontroller board

reset input, refer to Figure 2.2; XTAL2 and XTAL1 are the piezo crystal pins. 11.0592 MHz was used for the authors' board.

$\overline{\text{PSEN}}$ is program strobe enable. A switch was used to ground $\overline{\text{PSEN}}$ for in system programming (ISP); it was left floating when the normal program was running.

Address latch enable (ALE) was not used and was left unconnected. ALE generates a pulse and can be viewed as a source of local interference. Setting pin 0 in the auxiliary register (AUXR) to 1 disables ALE.

On the PCB all four ports are available for general use.

On port 3 pins 0 (RxD) and 1 (TxD) connect via a MAX232 chip to a 9-pin female D type socket. This connects to the PC for ISP but once the microcontroller is programmed this PC connection can be used for normal running PC microcontroller communication. Before using the serial connection for normal use the WinISP software, used to carry out the ISP, must be closed down.

On port 1, pins 6 (SCL) and 7 (SDA) may be used as serial connections for the onboard I^2C interface and as such are open-drain. Figure 2.2 shows 2k2 (2.2 kΩ) resistors connected from these pins to 5 V DC.

Port 0 is shown as an address (A0–A7) and data (D0–D7) port, and port 2 is shown as an address (A8–A15) port. These ports have been traditionally used for connection to external PROM, data memory and other peripherals in expanded systems.

The microcontroller board described in this book is not expanded; it uses the onboard flash code memory and the onboard static RAM (SRAM). Ports 0 and 2 are left for general use although port 0 pins are open-drain and if used should employ pull-up resistors.

The active low external access ($\overline{\text{EA}}$) pin was not required and was connected to the 5 V DC (V_{cc}).

2.3 Programming the device

The authors use WinISP software to program the flash P89C664 and XAG49 devices, the software being downloaded from Philips Semiconductors web pages. First the home page at www.semiconductors.philips.com should be accessed and then the Microcontrollers page by selecting Products and then Multimedia Semiconductors. Once on the Microcontrollers page Support Tools should be selected and then Software Downloads. WinISP.zip is on this page; it is approximately 2 MB. Selecting WinISP gives the window shown in Figure 2.3.

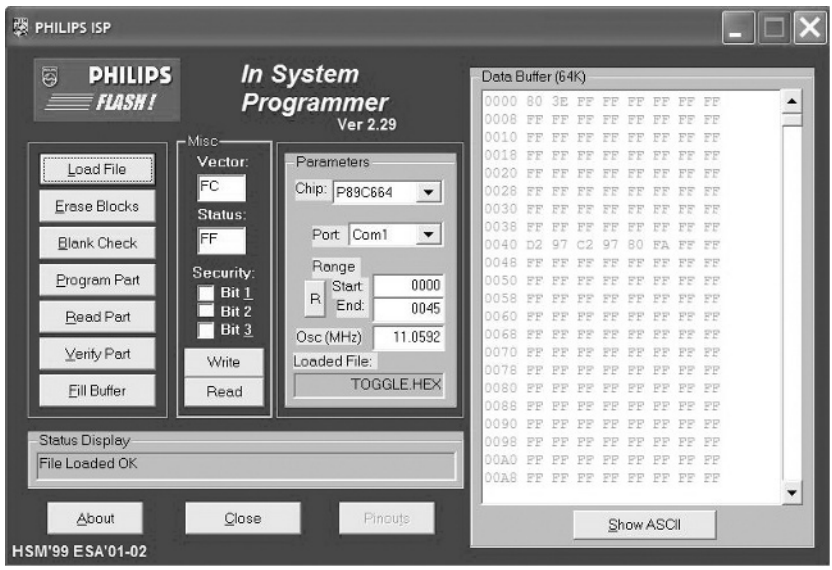


Figure 2.3 Philips WinISP window

As can be seen in the data buffer window a simple test program has been loaded; it toggles (turns on and off) repeatedly pin 7 on port 1. This can easily be checked on the microcontroller board by using an oscilloscope or logic probe. The program is:

```
        ORG     0           ; reset address
        SJMP    START       ; jump over reserved area
        ORG     40H         ; program start address
START:  SETB    P1.7        ; set pin 7 port 1 to logic 1
        CLR     P1.7        ; clear pin 7 to logic 0
        SJMP    START       ; repeat
        END                 ; end of assembly language
```

The hex version of the program is shown in the data buffer.

The Philips Semiconductors flash microcontrollers come with a small monitor routine already programmed into them at the top of code memory. In Figure 2.2, switch 2 (SW2) connects $\overline{\text{PSEN}}$ to ground while the reset push-to-make switch (SW1) is pressed and then released. This causes the microcontroller to communicate with the WinISP software on the PC. A board produced for the P89C664 device is shown in Figure 2.4(a); it should be noted that this board varies slightly from the design presented in Appendix G in that lettering was formed on the upper surface of the board. The design in Appendix G is simpler and utilises single-sided artwork with a plain upper surface. The board layout is shown diagrammatically in Figure 2.4(b) to illustrate the ports and show the connections to the 5 V DC power supply and to the computer.

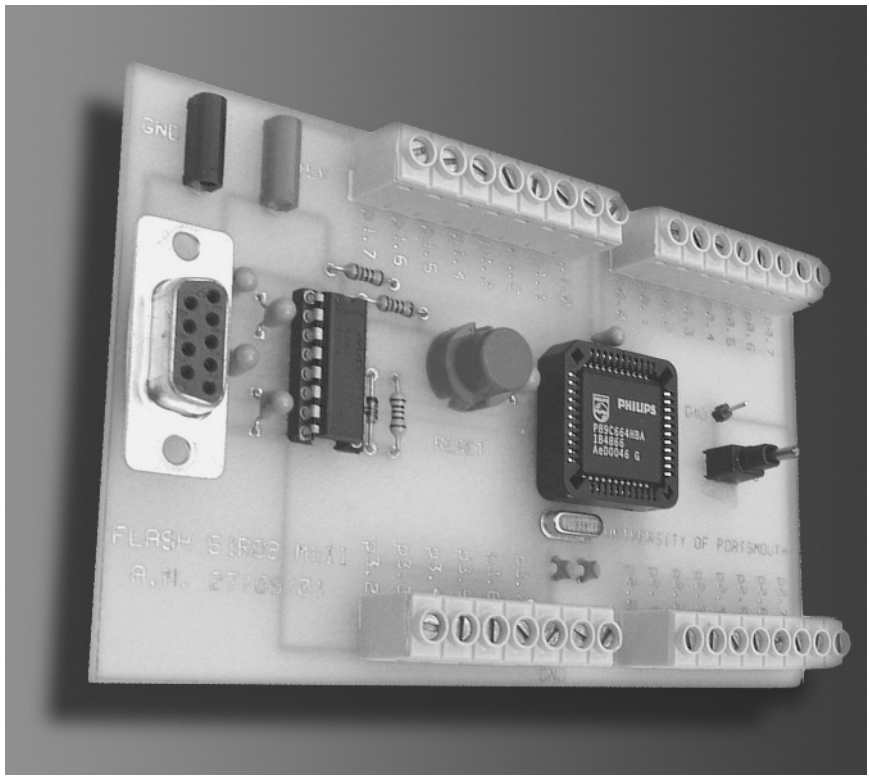


Figure 2.4(a) The 89C66x microcontroller board

Referring to Figure 2.3, the program hex code is in the WinISP data buffer. The reset address at 0000 can be seen. The hex code there must be for SJMP START. The program start address at 0040 can also be seen. The hex code that follows is that of the program.

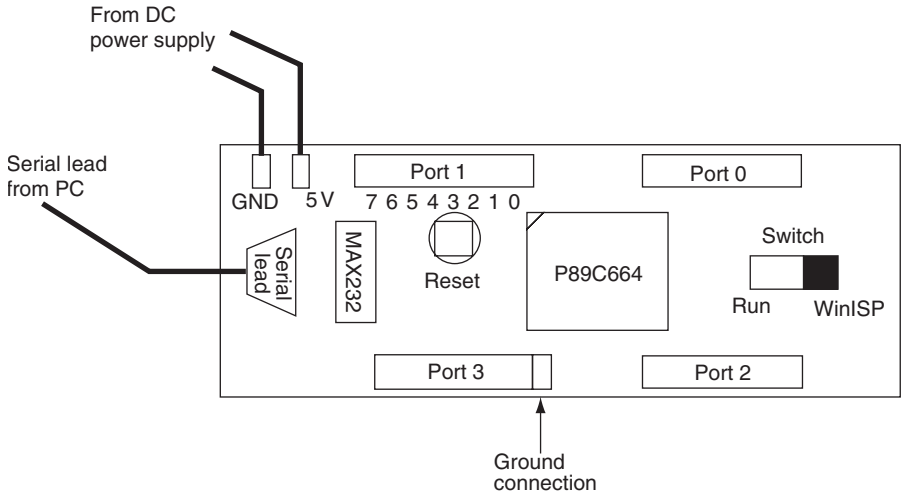


Figure 2.4(b) Diagram of the 89C66x microcontroller board showing ports, switches and external connections

Figure 2.5 Read button from the WinISP window

Test whether WinISP can communicate with the microcontroller board as follows:

1. Turn the microcontroller board DC power supply on.
2. Ensure the switch on the microcontroller board is set to the WinISP position ($\overline{\text{PSEN}}$ connected to ground).
3. Press and release the microcontroller board reset switch.
4. Back to the PC, CLM on the read button at the bottom of the middle Misc window in WinISP (see Figure 2.5).
5. If the status display reads boot vector read OK then that is good and you can proceed to the next step, if not then close the window and repeat. If there is still a problem then use an oscilloscope to check the signal from the PC, through the D connector, through the MAX232 to the RxD (see Figure 2.2) pin on the microcontroller. If this is satisfactory then check the signal back from the TxD pin on the microcontroller, through the MAX232 to the D connector. If using a logic probe then remember not to use it between the MAX232 and D type connector where the voltage levels are in the region of ± 10 V.

Before the microcontroller can be programmed it must first be erased.

Click left mouse button (CLM) on erase blocks and a window appears as shown in Figure 2.6. CLM on the 0 and the hatched pattern appears in the first

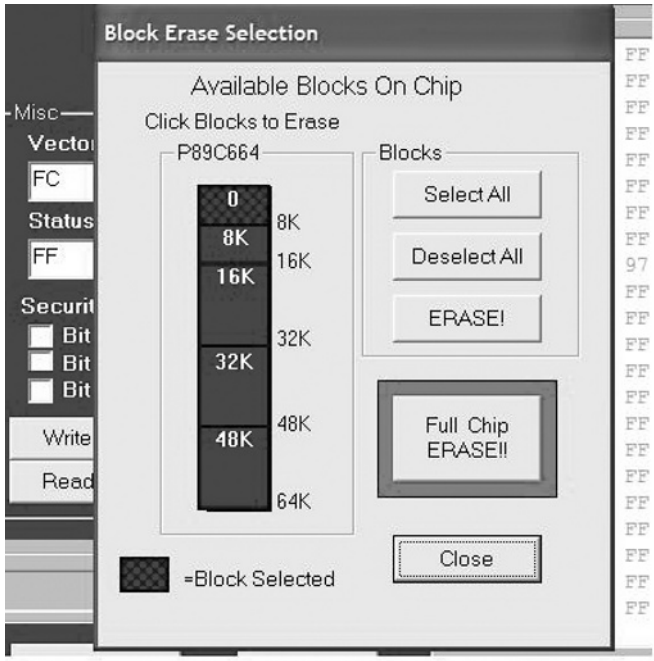


Figure 2.6 Block erase selection using WinISP

Figure 2.7 Program part button from the WinISP window

section. Now CLM on the ERASE! button and the hatched section should flash. When it finishes the micro is ready to be programmed. Now CLM on the program part button. See Figure 2.7. When there is a successful message in the status display, the micro is programmed.

HARDWARE CHECK

1. Ensure the microcontroller board DC supply is still on.
2. On the microcontroller board move the switch to the run mode ($\overline{\text{PSEN}}$ floating).
3. Press and release the microcontroller board reset switch, this makes the program run from 0000H.
4. Use a logic probe or oscilloscope to check that pin 7 on port 1 is pulsing.

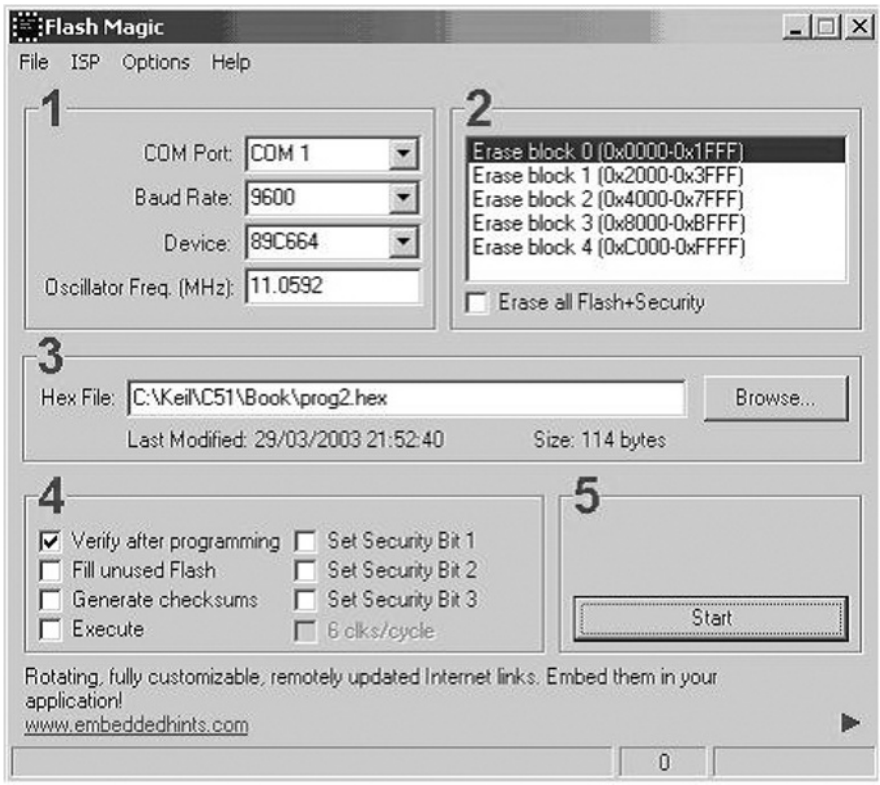


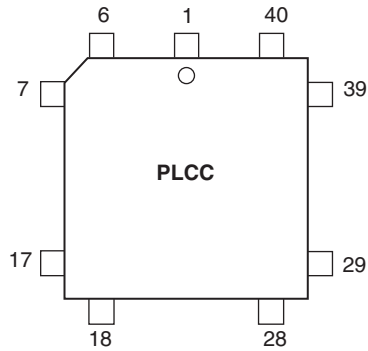
Figure 2.8 Flash Magic window

2.4 Flash magic

Flash Magic ISP software may be used in place of WinISP. It can be downloaded from the web page of Embedded Systems Academy (www.esacademy.com). The authors have not used it but there have been good reports on its ease of use. The window that is produced by the software is shown in Figure 2.8. Again the downloaded file size is just over 2 MB. Included in the installation is a manual.

2.5 XAG49 microcontroller

The PLCC package for the XAG49 device is shown in Figure 2.9. Comparison between the pin functions for this device and those of the P89C664 microcontroller (Figure 2.1) shows that they are almost pin compatible. One difference is pins 1 and 23, for the P89C664, they are not internally connected (NIC). On the XAG49 device, pin 1 (V_{SS}) is internally connected to pin 22 (V_{SS}) and pin 23 (V_{DD}) is internally connected to pin 44 (V_{DD}).



Pin	Function	Pin	Function	Pin	Function
1	V_{SS}	16	P3.4/T0	31	P2.7/A19D15
2	P1.0/A0/$\overline{\text{WRH}}$	17	P3.5/T1/BUSW	32	$\overline{\text{PSEN}}$
3	P1.1/A1	18	P3.6/$\overline{\text{WR}}$L	33	ALE
4	P1.2/A2	19	P3.7/$\overline{\text{RD}}$	34	NC
5	P1.3/A3	20	XTAL2	35	$\overline{\text{EA}}/V_{PP}$/WAIT
6	P1.4/R×D1	21	XTAL1	36	P0.7/A11D7
7	P1.5/T×D1	22	V_{SS}	37	P0.6/A10D6
8	P1.6/T2	23	V_{DD}	38	P0.5/A9D5
9	P1.7/T2EX	24	P2.0/A12D8	39	P0.4/A8D4
10	$\overline{\text{RST}}$	25	P2.1/A13D9	40	P0.3/A7D3
11	P3.0/RxD0	26	P2.2/A14D10	41	P0.2/A6D2
12	NC	27	P2.3/A15D11	42	P0.1/A5D1
13	P3.1/TxD0	28	P2.4/A16D12	43	P0.0/A4D0
14	P3.2/$\overline{\text{INT0}}$	29	P2.5/A17D13	44	V_{DD}
15	P3.3/$\overline{\text{INT1}}$	30	P2.6/A18D14		

Figure 2.9 Pin functions for the XAG49 PLCC package microcontroller

Another significant difference is the reset on pin 10. The reset of the P89C664 is active high, same as all the standard 8051 devices, whereas the XA reset is active low; check the schematic circuits. The XAG49 does not have an I^2C peripheral and so has no need for pull-up resistors on pins 6 and 7 of port 1.

The XAG49 has 20 address lines A0 to A19 and can be the 16-bit processor in a relatively large expanded system. Most of the expansion comes through ports 0 and 2 although 4 address lines A0 to A3 on port 1 are not multiplexed with data lines and provide fast (burst) memory addressing. Use of bus width (BUSW) on pin 17 in conjunction with the bus configuration register (BCR) can be set to make the data bus width 8 bits or 16 bits. $\overline{\text{WRH}}$ and $\overline{\text{WRL}}$ may be used to select 8-bit memory devices to work in 8-bit or 16-bit data transfer in a similar way to the use of upper data strobe (UDS) and lower data strobe (LDS) in a Motorola 68000 system.

The schematic for the XAG49 is shown in Figure 2.10. The circuit of Figure 2.10 is similar to that for the P89C664 (Figure 2.2), the difference being the reset. The full size single-sided artwork is available in Appendix G together with the component layout. Programming the XAG49 is carried out in the same way as for the P89C664. A test program to toggle pin 7 on port 1 could be:

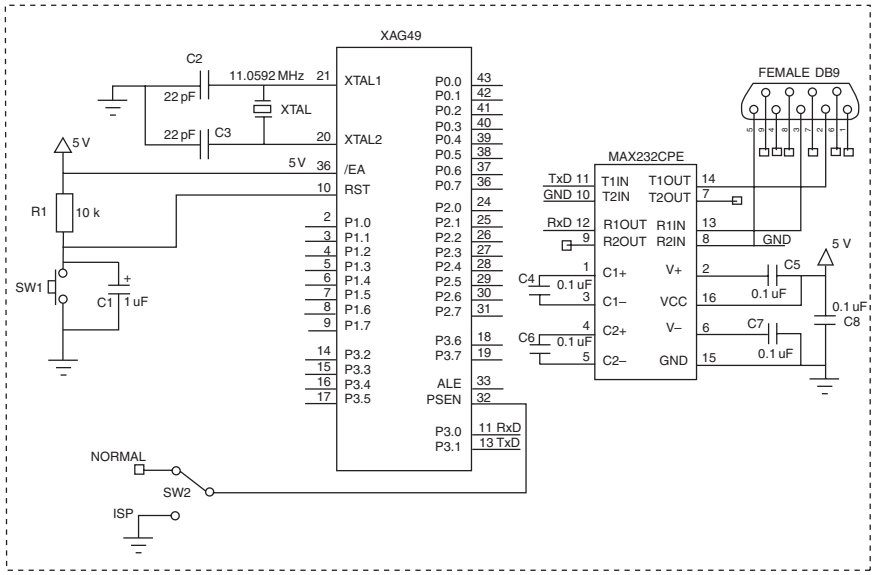


Figure 2.10 Schematic for the XAG49 microcontroller board

```
$INCLUDE(REGXAG49.INC)        ; list of register addresses
         ORG    0             ; reset address
         DW     8F00H,START   ; config registers, goto START
         ORG    120H          ; program start address
START:   SETB   P1.7          ; set port 1 pin 7 to logic 1
         CLR    P1.7          ; clear pin 7 to logic 0
         JMP    START         ; repeat
         END                  ; end of assembly language
```

Notice that the program start address is at 0120H, this means the program hex would not appear on the front page of the WinISP data buffer and the buffer would have to be scrolled to show the contents starting at hex address 0120H.

Summary

- A P89C66x microcontroller board can be constructed using the schematic of Figure 2.2 and the PCB artwork of Appendix G.
- An XAG49 microcontroller board can be constructed using the schematic of Figure 2.10 and the PCB artwork of Appendix G.
- The microcontroller device can be programmed using suitable ISP software such as Philips' WinISP.
- Suitable software and hardware checks can be carried out to establish that the boards operate satisfactorily.

3

Simulation Software

3.1 Introduction

The program examples and corresponding simulations shown in this book were carried out using evaluation software downloaded from:

www.keil.com

www.raisonance.com

The software from Keil supports the LPC and P89C66x microcontroller families, and that from Raisonance supports LPC, P89C66x and XA devices.

It should be pointed out that evaluation software from these companies support many other microcontroller devices. This book is based on P89C66x, XAG49 and LPC microcontrollers.

The software size is code limited. For Keil software it is limited to 2 KB while for Raisonance software it is limited to 4 KB for the 8051 and to 8 KB for the XA. Both manufacturer's evaluation software compile down to a reset hex address of 0000.

The software can be used for programming in both C and assembly language. Generally, if the first line of an assembly language program begins $INCLUDE then the Raisonance software is being used.

Both sets of software operate within an integrated development environment (IDE); the package from Keil is called μ Vision2 (micro Vision 2) and that from Raisonance is called Raisonance IDE (RIDE). Writing of the source program C or assembly language, syntax check, compilation to hex, program debugging/simulation all takes place within the integrated environment.

When the debugging/simulation is satisfactorily completed the microcontroller is programmed with the .hex version of the program.

It is assumed that if the reader is going to read this chapter then he/she will have downloaded the required software and will be following the text at the

same time the simulation software is being accessed. In this chapter, reference is often made to a colour used in the simulation window, i.e. a red breakpoint, a yellow arrow, etc. While it is appreciated that the diagrams shown in this chapter will be in various shades of grey the colour references are apposite if the reader is using the simulation package at the same time as reading the text. The diagrams in the text can then be cross-checked against those of the simulation package in glorious technicolour.

A flow diagram suitable for source program evaluation, and including the programming of the microcontroller using Philips WinISP, as described in Chapter 2, is shown in Figure 3.1.

3.2 Keil μ Vision2

Starting with this software a window is opened as indicated in Figure 3.2.

A start is made by selecting Project from the top menu bar and then choosing New Project. A folder can be created and a Project name chosen, in this example the Project is called Test1. See Figure 3.3.

Clicking left mouse button (CLM) on Save generates another window which prompts the user to select a chip vendor and then a particular microcontroller. See Figure 3.4.

For this example a Philips P89C664 is chosen. CLM on OK, another small window appears as shown in Figure 3.5.

The program is going to be in assembly language so CLM on No. With reference to Figure 3.2 it can be seen that in the top of the left hand window there is a small icon with a + sign to the left and the description Target 1 to the right. Moving the cursor over Target 1 and clicking right mouse button (CRM), another window appears as shown in Figure 3.6.

Selecting Options for Target 'Target 1', a new window appears as shown in Figure 3.7.

The oscillator frequency defaults to 33 MHz and requires changing to 11.0592 (it is already in MHz). Now CLM on Output, the tag to the right of Target, as shown in Figure 3.7, the window changes and the left mouse button should be clicked to select Create HEX File. See Figure 3.8.

CLM on OK. From the top menu bar File should be selected, followed by New to produce a text window. Going to File again and selecting Save As produces a small window defaulting to the Project folder. The program requires to be saved with the same name as the project but with the extension .a51. The program (e.g. test1.a51) and the project title should be in the same folder.

The simple assembly language program used to test the target board should be typed into this window. Refer to Chapter 2 for the program to test the P89C664 device. The program, shown in Figure 3.9, is one that causes pin 7 on port 1 to toggle. The program should now be saved. The program defaults to syntax colours, which is very helpful. This Editor characteristic and other attributes may be changed by selecting View, from the top menu bar, and then Options.

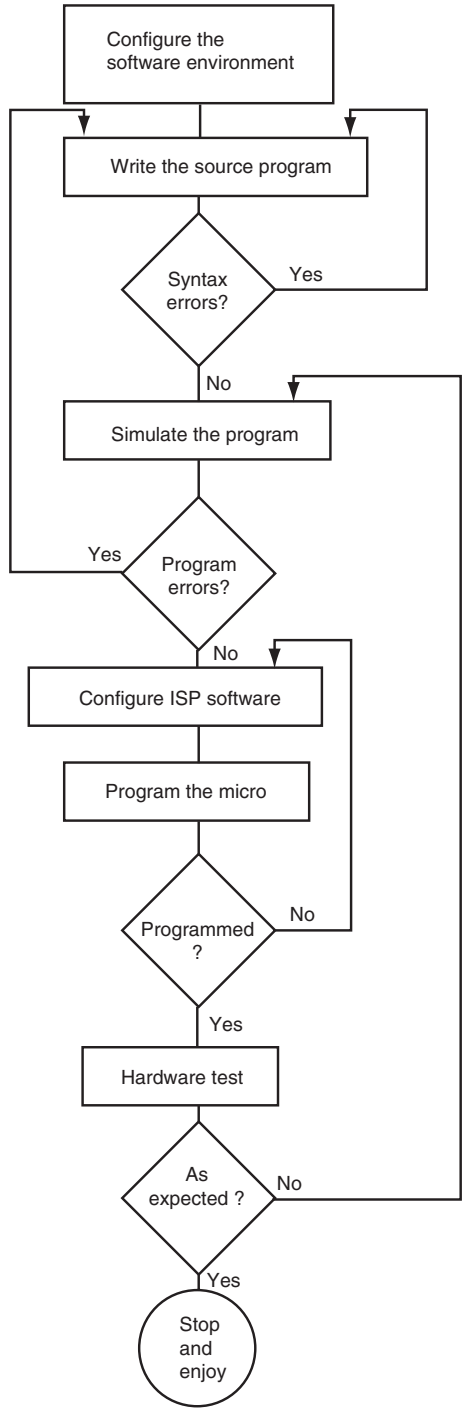


Figure 3.1 Design flow diagram

Figure 3.2 Keil μ Vision2 Simulation window

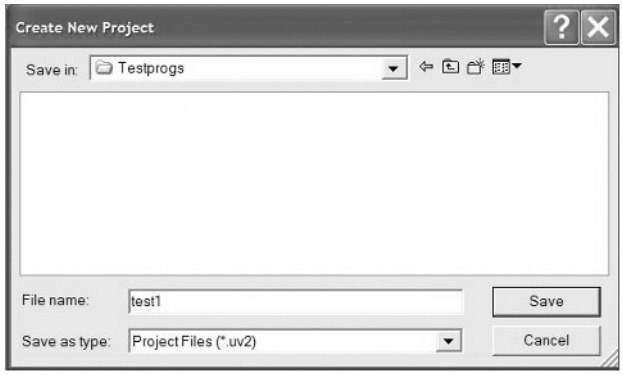


Figure 3.3 Window for the creation of a Project

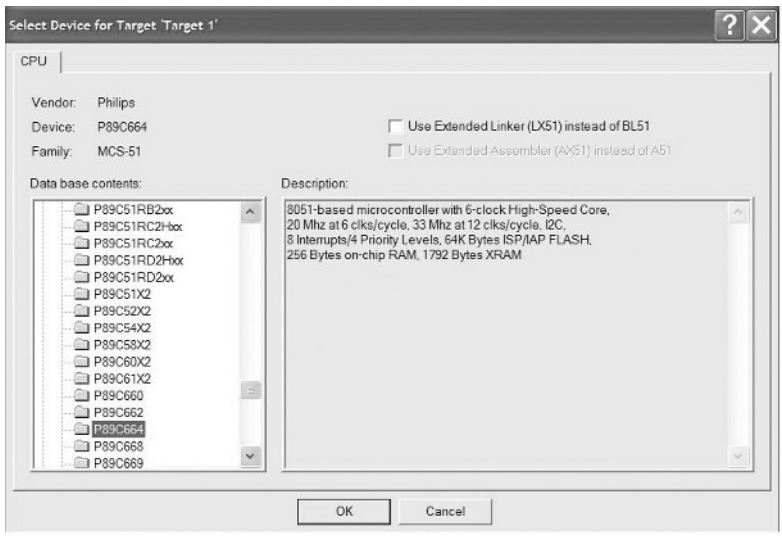


Figure 3.4 Window for the selection of a Vendor and particular microcontroller chip

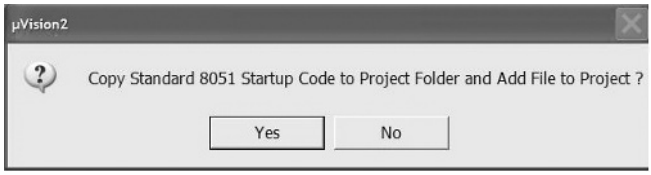


Figure 3.5 Window for possible use of 8051 Startup code

Figure 3.6 Window for Target operations

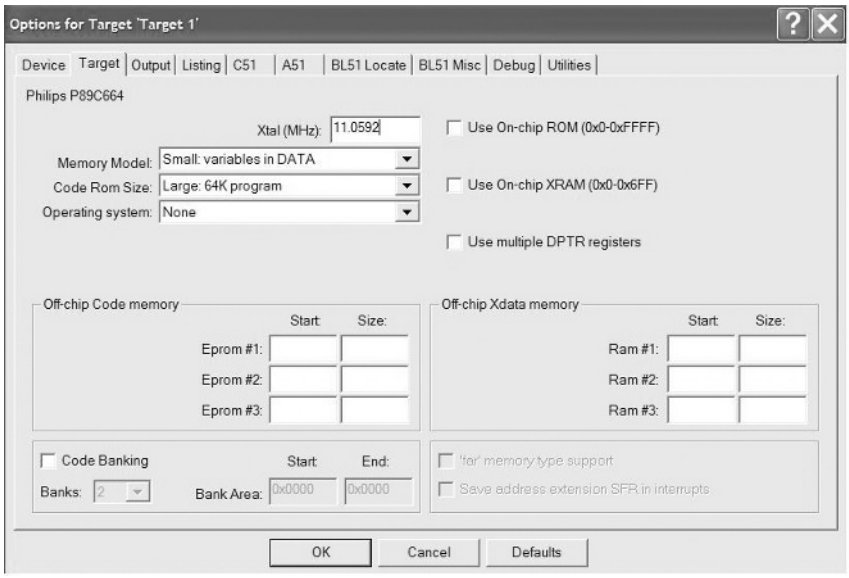


Figure 3.7 Window for Target options

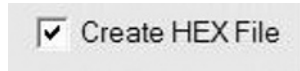


Figure 3.8 Creating a hex file

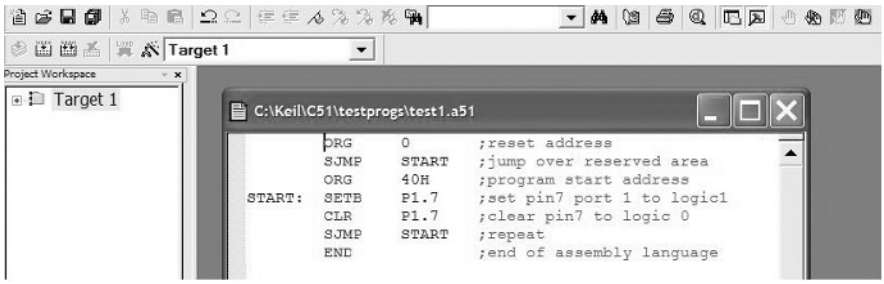


Figure 3.9 Text window showing test1.a51 source program

Now CLM on the + sign to the left of Target produces a subfolder Source Group 1, as shown in Figure 3.10. CRM on Source Group 1 generates another window shown in Figure 3.11. Choosing Add Files to Group 'Source Group 1' causes another window to appear which defaults to C programs. See Figure 3.12.

Figure 3.10 Target 1 subfolder Source Group 1

Figure 3.11 Window for Source Group 1 operations

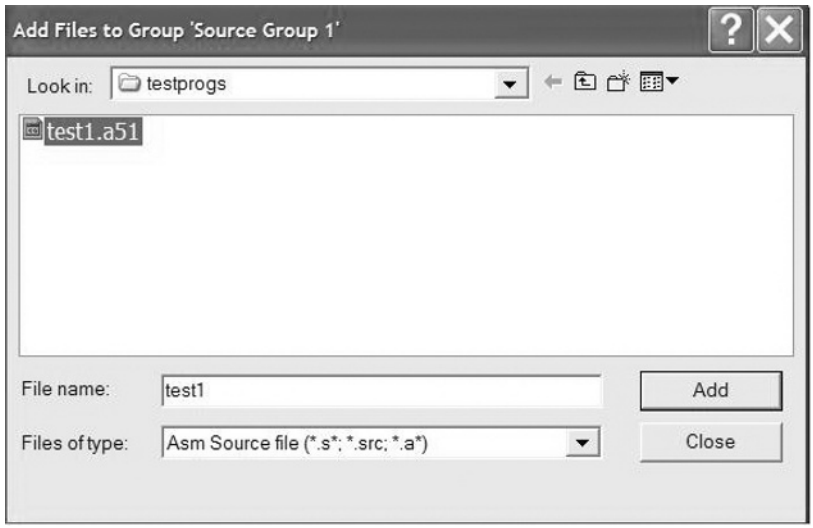


Figure 3.12 Window for adding files to Source Group 1

Changing to Asm and CLM on the program name, then CLM on Add and then on Close produces a + sign to the left of Source Group; CLM on this + sign the program file appears as shown in Figure 3.13.

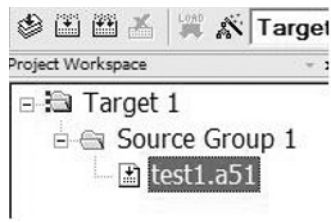


Figure 3.13 Program file test1.a51 added to Source Group 1

CLM on the top left hand icon will Translate the program and check the syntax. Alternatively from the top menu bar selecting Project and then Translate will achieve the same result.

A report will appear in the Build window at the bottom of the screen, as shown in Figure 3.14.

Any syntax errors would be reported at this stage. For example, putting SET instead of SETB would report a syntax error as shown in Figure 3.15. The syntax error is on line 4 of the program. Correcting the error and re-translating should result in an error-free program. If this is the case, then selecting the Build icon to the right of the Translate icon (alternatively selecting Project and then Build) will produce the hex file. This is shown in Figure 3.16.

Figure 3.14 Build window report showing errors and warnings as appropriate

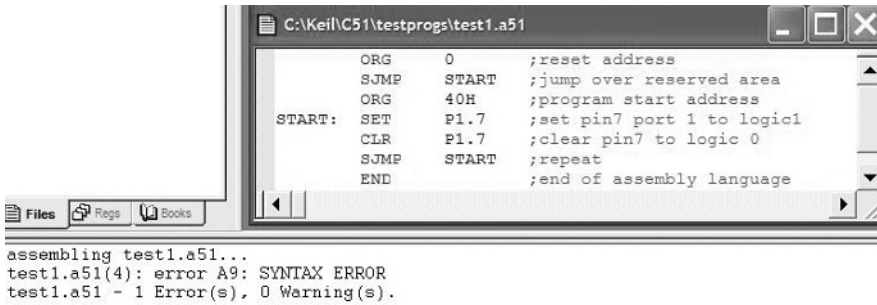


Figure 3.15 Syntax error in the Build window

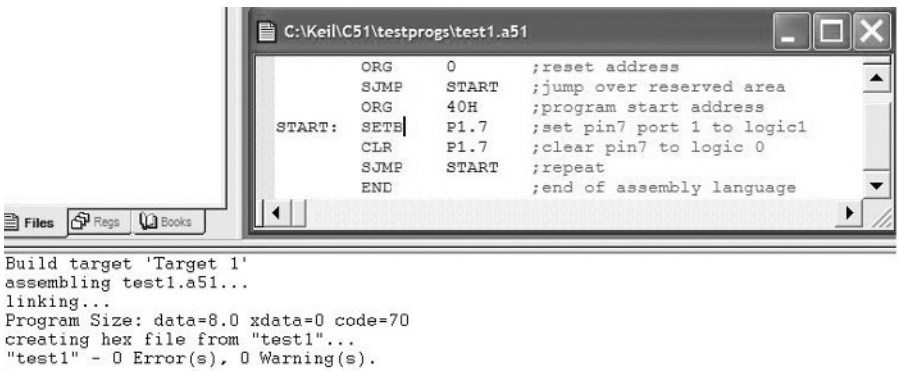


Figure 3.16 Creation of a hex file for test1.a51 program

DEBUGGING/SIMULATION

Debug may be accessed by either CLM on the red letter d or selecting Debug from the main menu bar and then selecting Start Debug Session. See Figure 3.17. A register window would show on the left and a command window at the bottom of the screen. This program toggles pin 7 on port 1 so a port 1 window would be useful to display. The window may be selected by choosing Peripherals on the main menu bar, then I/O Ports and then Port 1. See Figure 3.18.

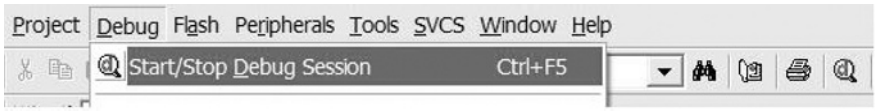


Figure 3.17 Using the Debug facility

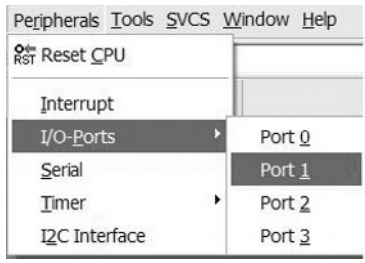


Figure 3.18 Selection of port 1 window

The port 1 window appears and there are two rows of eight ticks; the top row represents the logic level set by the microcontroller while the lower row represents the actual level on the port pins. Ideally they should always be the same but there are times when a bad interface might load a port pin and cause it to be logic 0 when the microcontroller is assigning logic 1.

When debugging complex programs it is useful to single step through complete loops; in this simple example the program is a simple loop. Selecting Debug and then Step, the function key F11 can be pressed to step or the brackets icon with the arrow into it can be used. See Figure 3.19. By repeatedly

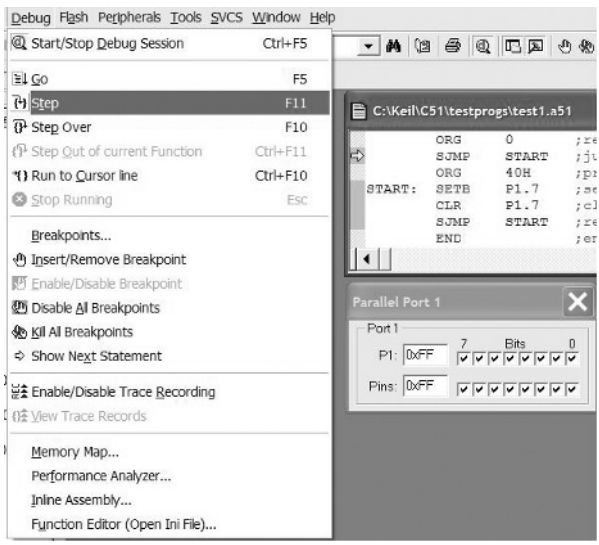


Figure 3.19 Selection of Step function

pressing and releasing key F11, the cursor and yellow arrow in the program window allows the program to be followed step by step. The corresponding logic levels of bit 7 in the port 1 window can be checked with the hex and corresponding binary values of port 1.

Time delays can be measured using sec in the Sys registers, between states and PSW. The reset value of sec is 0.00000000. See Figure 3.20. Consider a modification to the program that puts a delay after SETB and CLR. The data from the first Time Delay example in Chapter 1, where register R0 was loaded with decimal 88 to establish a 5 kHz square wave, could be used. The result is shown in Figure 3.21.

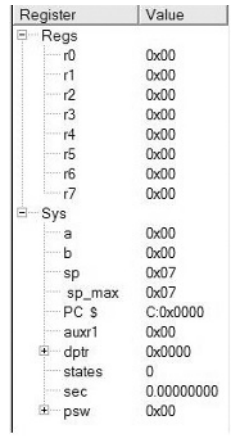


Figure 3.20 Register window display

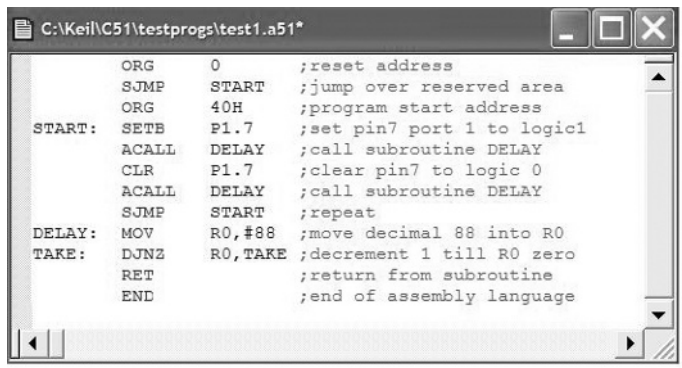


Figure 3.21 Modification to test1.a51 to include a time delay

The program should then be translated before returning to Debug. There are seven icon buttons above the Register space available for use and they are shown in Figure 3.22.

Figure 3.22 Icons used for program control purposes

With reference to Figure 3.22, the RST icon with the red arrow is the Reset. The icon to the right, a sheet with a blue arrow to the right, is the Run button. The cross icon next right is the Stop button; it changes to red when the simulated program is running. The next four sets of brackets to the right are:

1. Single stepping (alternative is key F11).
2. Single step avoiding subroutines (alternative is key F10).
3. Jumping out of subroutines when stuck in a loop.
4. Run simulation to stop at blinking cursor.

Breakpoints at suitable points in the program may be inserted by moving the PC mouse to the beginning of a line and positioning the blinking cursor at this point. Then moving the mouse cursor over the hand icon, shown on the top row in Figure 3.23, and CLM positions a red breakpoint block against that line. This has been done in Figure 3.23 alongside the line with the first ACALL DELAY and the next line CLR P1.7.

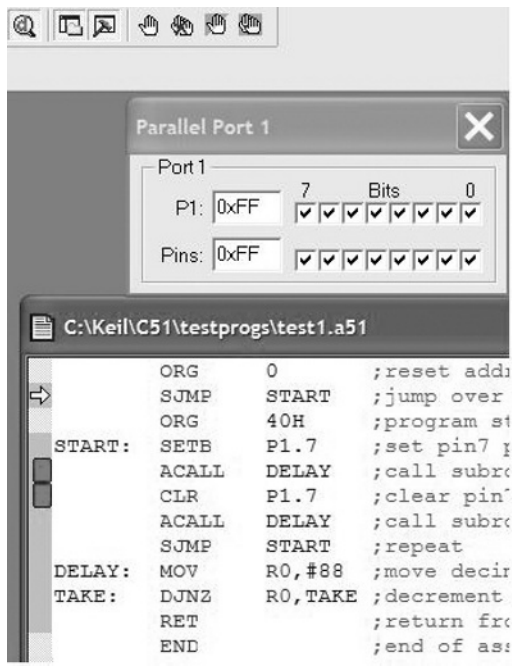


Figure 3.23 Inserting breakpoints in the program test1.a51

Resetting (RST icon) the simulation causes the yellow arrow to go to address 0000H where SJMP START is located. At reset the contents of sec are 0.00000000; see Figure 3.24(a). Running the simulation (sheet with blue arrow icon or pressing function key F5) causes the program to run to ACALL DELAY at the first breakpoint. The contents of sec will then be as shown in Figure 3.24(b). Running the simulation once more to the next breakpoint increases the contents of sec to the value shown in Figure 3.24(c).

The time difference can be determined from Figure 3.24(b) and (c) i.e.

$$(0.00009983 - 0.00000163)\,\text{s} = 0.0000982\,\text{s} = 98.20\,\mu\text{s}$$

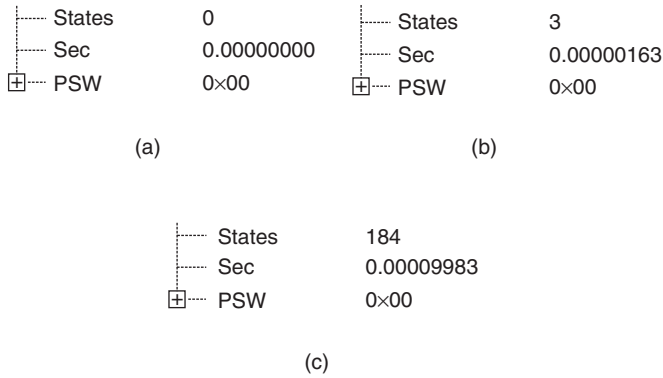


Figure 3.24 (a) Time at reset of program test1.a51. (b) Time at first breakpoint of program test1.a51. (c) Time at second breakpoint of program test1.a51

For the 5 kHz square wave it should be 100 μs. Increasing R0 to decimal 89 gives a delay of 99.28 μs and increasing R0 to decimal 90 gives a delay of 100.37 μs.

Assembly language programs are written using the microcontroller registers and the SFRs of the onboard peripherals. The microcontroller registers are shown on the Debug/Simulation screen. See Figure 3.20.

The onboard peripheral SFR windows may be selected via Peripherals on the top menu bar. An example is shown in Figure 3.25.

Programs written in C may also use onboard peripheral SFRs but they can also use defined variables that do not represent an SFR. The values of these defined variables may be checked using a Watches window; SFRs may also use a Watches window. From the top menu bar selecting View then choosing Watch & Call Stack window produces a window at the bottom of the PC screen as shown in Figure 3.26.

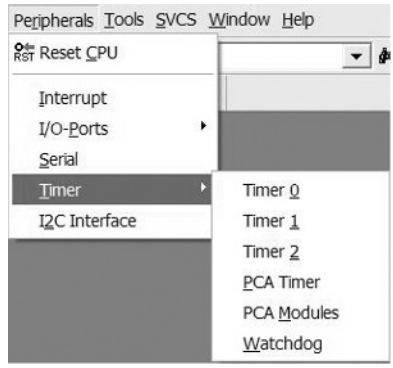


Figure 3.25 Selection of particular onboard peripherals

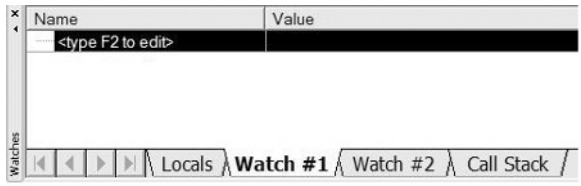


Figure 3.26 Production of a Watch window

The time delay program written earlier using assembly language can be written in C language. The C equivalent of the original program is shown in the program window of Figure 3.27.

The delay variable *j* and the main variable pin 7 are shown in the Watch window. To initialise *j* it is necessary to start by single stepping until the delay function is executed once. The variables are entered into the Watch window by pressing function key F2. Breakpoints have been set at the first delay ( ) and pin 7 = 0. The delay time can be measured as before by subtracting the two sec values.

3.3 Raisonance IDE (RIDE)

The Raisonance software can be downloaded as a separate 8051 package, a separate XA package or as a combined 8051 + XA package. Start by selecting Project from the top menu bar and then selecting New. See Figure 3.28.

If using the combined 8051 + XA package the relevant microcontroller family should be selected. Note that the LPC microcontrollers are part of the

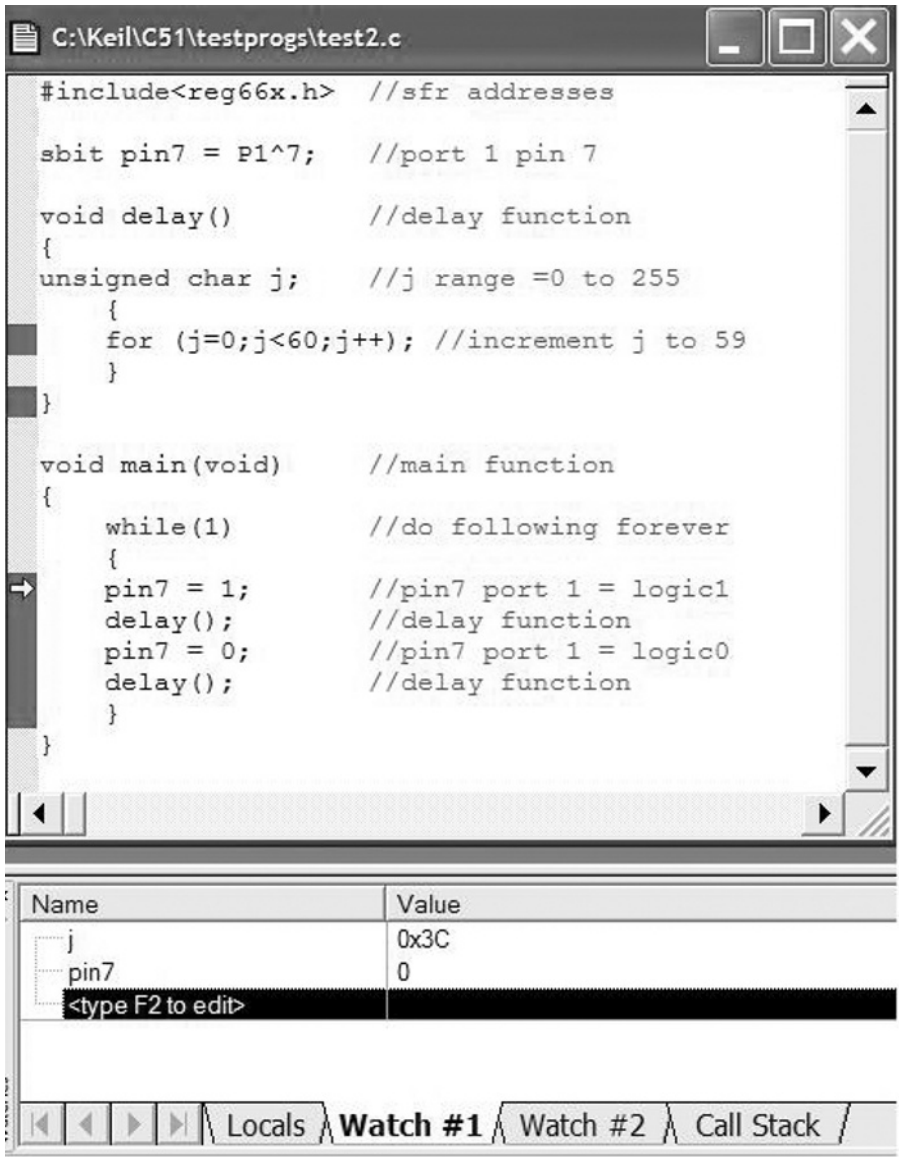


Figure 3.27 C language program to produce toggling on pin 7, port 1 with a time delay

8051 family. Choosing 80C51 and CLM on OK will produce the window shown in Figure 3.29, which is similar to the window obtained with the Keil software. The device manufacturer should then be selected (e.g. Philips) and the particular device selected by scrolling down the list until the required device (e.g. P89C664) is found.

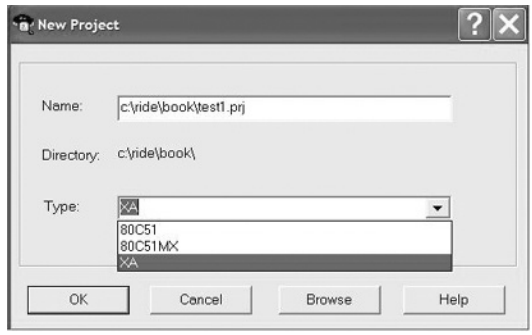


Figure 3.28 Raisonance window for the creation of a new project

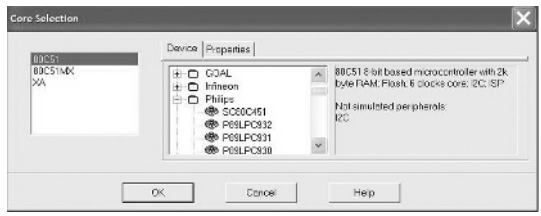


Figure 3.29 Window for Core Selection

For the explanation of the Raisonance software that follows, the XA family has been chosen. The XA family member (e.g. XAG49) is chosen at the point of entering the Debug/Simulation. CLM on OK will generate another window; XA should be selected to confirm the core, then CLM on OK. Selecting File from the top menu bar and then New results in a small window, as shown in Figure 3.30.

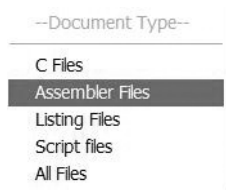


Figure 3.30 File selection window

Selecting Assembler Files and a blank text window appears, labelled untitled.axa. From the top menu bar selecting Options and then choosing Project result in the window shown in Figure 3.31.

CLM on the + sign to the left of Environment and choosing Editor will give an opportunity to change the values indicated. TabStop defaults to 3 and

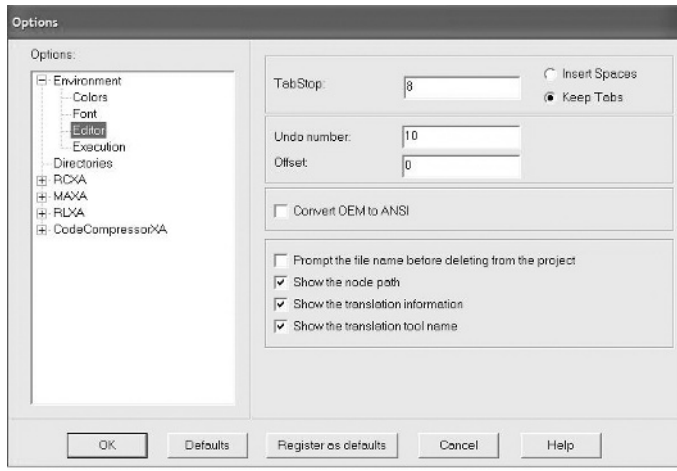


Figure 3.31 Options window

should be changed to 8; Offset defaults to 2 and should be changed to 0. The result of the changes is illustrated in Figure 3.31.

To illustrate the use of the simulation software a simple assembly language program, to toggle pin 7 of port 1, is to be used. Once the test program has been written in the text window, File may be selected from the top menu bar and the program saved by choosing Save As. The program is shown in the text window of Figure 3.32.

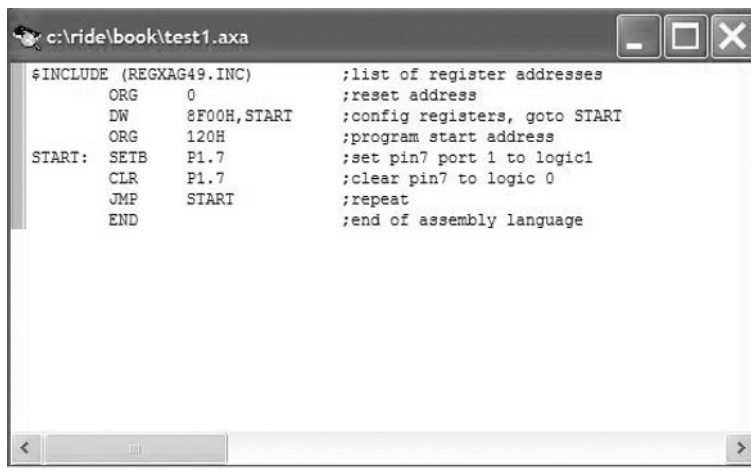


Figure 3.32 Test program to toggle pin 7, port 1 using the XA device

The XA assembly language program extension must be .axa (it is .a51 for 8051). The source program should be saved in the same directory as the Project.

The XA program has a start that is different from that of the 8051. The explanation for this is given in Chapter 6. Another difference is that the XA uses the instruction JMP.

When complete the Source program should be added to the Project by selecting Project in the top menu bar and then choosing Add node Source/Application. In the window that appears, add the program name and CLM on Open. See Figure 3.33.

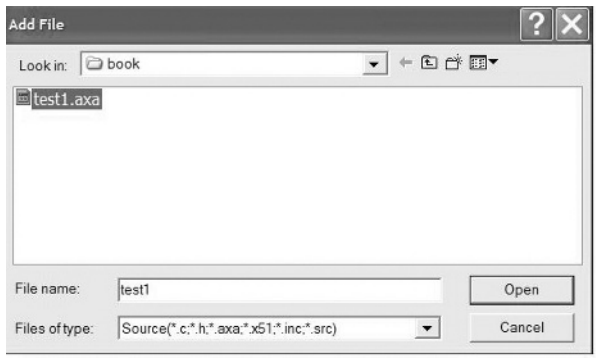


Figure 3.33 Adding file test1 to the project

From the window illustrated in Figure 3.28 there are icons which have frequent use. Some of these icons are shown in Figure 3.34.

The source program may be translated by CLM on the icon shown in Figure 3.34(a) or by selecting Project from the top menu bar, then choosing Translate; the end result can also be obtained by simultaneously pressing Alt F9. If there are any syntax errors the information will appear at the bottom of the screen. A typical window is shown in Figure 3.35.

Figure 3.34 (a) Icon used to Translate source program. (b) Icon used to 'Make all'. (c) Icon used to select Debug. (d) Animation button

To produce the Debug/Simulation file and the hex file, CLM on the icon Make all, shown in Figure 3.34(b), which is to the right of the Translate icon. Alternatively Project may be selected from the top menu bar, then choosing Make all or simply press function key F9.

DEBUGGING/SIMULATION

With the Raisonance software the particular device belonging to the microcontroller family is chosen at the Debug/Simulation stage. Debug/Simulate is entered

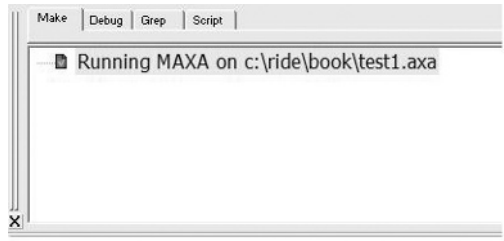


Figure 3.35 Window for display of syntax errors

by CLM on the icon shown in Figure 3.34(c) or by selecting Debug on the top menu bar, then choosing Start, when a window appears as shown in Figure 3.36.

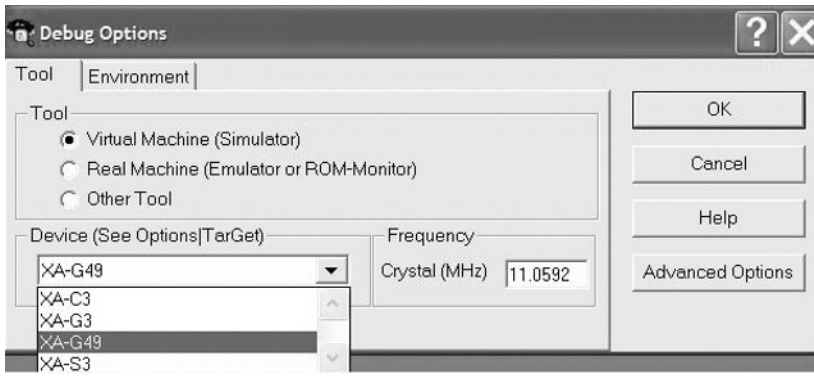


Figure 3.36 Debug options window

For our purposes, the selected device is the XAG49; three other XA family members are also shown. The XAG3 is the one time programmable (OTP) EPROM version of the Flash XAG49. The XAC3 is the XA version having an onboard controller area network (CAN). The XAS3 has 24 address lines, six 8-bit ports, I^2C, 8 channel 8-bit ADC and a programmable counter array (PCA).

The crystal frequency might default to 11.059; the value should be changed to 11.0592. The value chosen is in MHz. CLM on OK, another window appears, as shown in Figure 3.37. The values required have been changed to the values shown in Figure 3.37 and CLM on OK causes the software to go into Debug/Simulation mode.

A Watches window will appear at the bottom of the screen. If it appears too large then CLM on the two vertical bars on the left of the Watches window and dragging the window into the program window and then out again, should make it smaller.

More frequently used icons are shown in Figure 3.38. Reset is the icon with the finger pointing towards the red button.

The icon to the right of reset is Step into (fast key F7), used for single stepping through the program. The icon to the right of this is Step over (fast

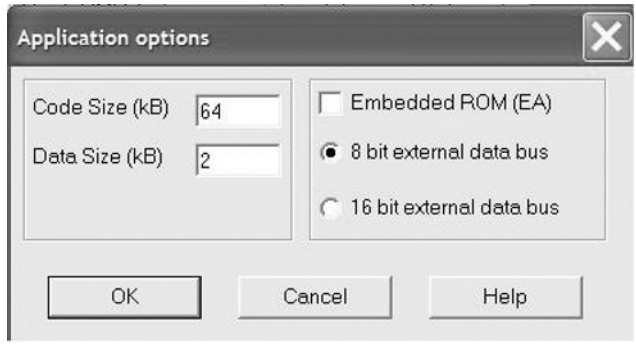


Figure 3.37 Applications options window

Figure 3.38 Frequently used icons

key F8), similar to Step into, but steps over (avoids) subroutines or functions. The spectacle with the + sign, to the right of the Step over icon, adds a variable to the Watches window. The letter s in the red circle with the + sign is used to add or remove breakpoints. All of these icon controls are accessible by selecting Debug in the top menu bar.

The Raisonance software has an animation icon shown in Figure 3.34(d). CLM on the animation icon, then CLM on GO in the green box should result in the blue horizontal cursor continually moving through the program.

The test program used toggles pin 7 on port 1. Port 1 can be accessed by selecting View on the top menu bar and then choosing Hardware Peripherals which causes a window to appear listing the XAG49 onboard peripherals as shown in Figure 3.39.

CLM on P1 l produces the window shown in Figure 3.40.

Single stepping (continually pressing F7) should show pin 7 toggling (red for logic 0, green for logic 1). The corresponding hex number in the LATCH window should also change (7F when pin 7 is logic 0, FF when pin 7 is logic 1). The LATCH window represents bit settings by the program, equivalent to the top row of eight ticks in the Keil software. The column numbered 0 down to 7 represents the port pins, equivalent to the bottom row of eight ticks in the Keil port window. The LATCH value can be changed by editing, using the mouse cursor. For example it could be changed to 7E to make pin 0 = logic 0. The port pins can also be changed. Moving the cursor arrow over the second pin down, pin 1, and CLM when the arrow cursor changes to a pointing finger will produce a small window as shown in Figure 3.41.

Selecting Ground should cause pin 1 to change to red, logic 0, but the LATCH value would not change.

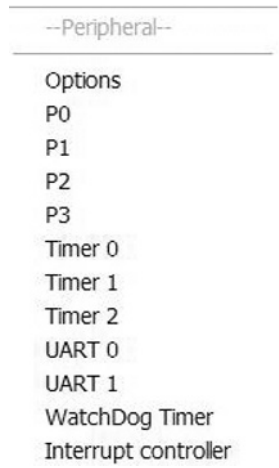


Figure 3.39 List of XAG49 onboard peripherals

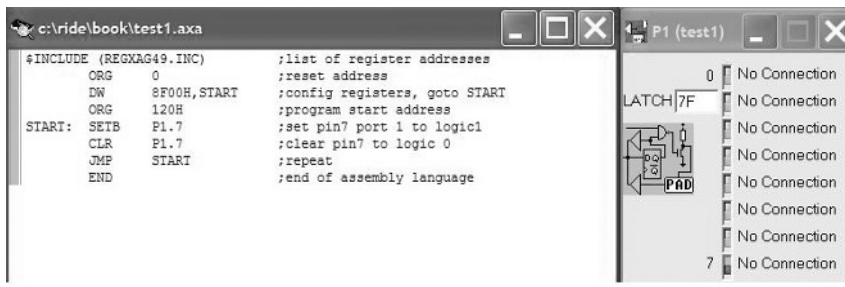


Figure 3.40 test1.axa program window with program and port 1 window showing pin logic levels

To detect a pin change would require the port to be read, e.g. MOV.B R0L, P1

MOV.B means move a byte(B)
R0L means Register 0 Low byte

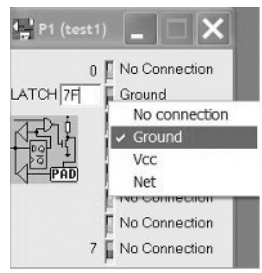


Figure 3.41 Window used to alter value of port pin

The XA is a 16-bit microcontroller able to move data in bytes (8 bits) or words (16 bits) and its registers are 16 bits (W), e.g. comprising a high byte R0H and low byte R0L. This is dealt with in Chapter 6.

If the simulation is reset and the animation made active, then by pressing GO the blue cursor will move through the program and pin 7 should toggle on and off. Coming out of the simulation by CLM on the icon of Figure 3.34(c), the program can be modified by changing the program details in the text window. For example, Figure 3.42 shows a variation in the original program caused by adding a time delay. This is achieved by putting a delay after SETB and CLR.

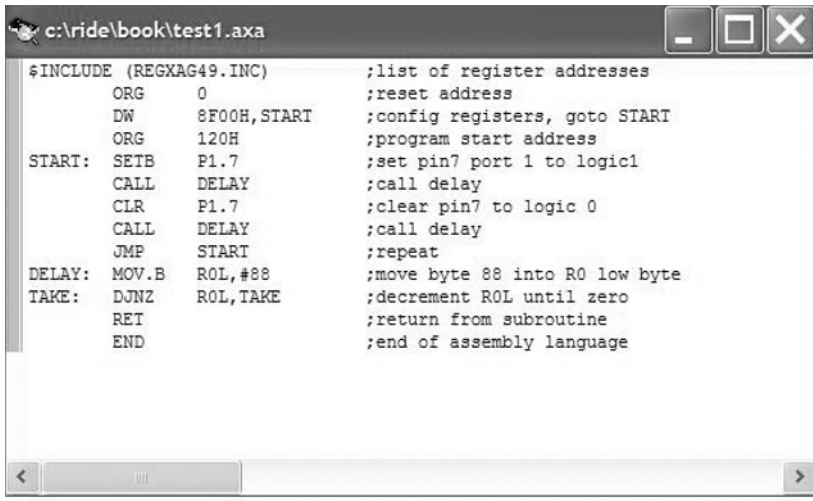


Figure 3.42 Program amendment to test1.axa introducing a time delay

If there are no syntax errors then pressing the icon of Figure 3.34(c) will save the program. Make all by pressing the icon of Figure 3.34(b) and returning to Debug/Simulation mode check for syntax errors. If there are syntax errors then they will be reported and the software will remain in Edit mode.

Moving the mouse cursor to the beginning of the first CALL DELAY line and CLM will set a blinking cursor at this point. Now move the mouse to the breakpoint icon, shown in Figure 3.38, and CLM will establish a red breakpoint line. Another breakpoint on the next line down (CLR P1.7) can be set using the same procedure. This is shown in Figure 3.43.

An alternative method of setting the breakpoint is to move the mouse cursor close to the green icons on the grey left column. As the cursor moves over the green icon, the cursor changes into the breakpoint icon. CLM when this change occurs sets a breakpoint. The same breakpoint can be removed by repeating the procedure over the breakpoint.

Running (GO) will take the program to the first breakpoint, where it will change colour to a light purple. The time in a panel at the bottom right of the screen should be noted; it will probably be zero on this first run, if not then it

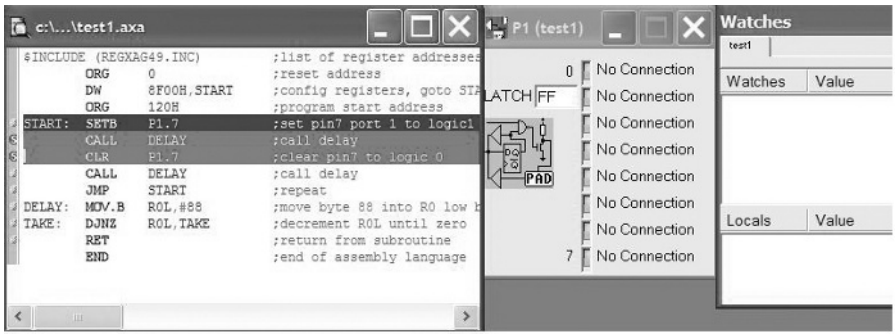


Figure 3.43 test1.axa program with breakpoint set

can be made zero with Ctrl T. See Figure 3.44(a). Running (GO) to the next breakpoint and noting the time in the panel should give the delay time which, for this test program, reads 0.089 ms or 89 μs. See Figure 3.44(b).

Using the P89C664, the time delay was 98.2 μs; the XAG49 microcontroller is faster. To establish a 5 kHz square wave, a half-cycle time delay of 100 μs is required. If decimal 88 sets a time delay of 89 μs with the XAG49, then an intuitive value of decimal 99 could be tried to establish a delay of 100 μs. To achieve this, the icon of Figure 3.34(a) should be pressed to come out of simulation; the value #88 in the program should be changed to #99 and the icon of Figure 3.34(a) pressed again to save the program. Pressing the icon Make all will then cause a return to simulation.

Running to the first breakpoint will give a time that, if necessary, can be set to zero by pressing Ctrl T. Running to the next breakpoint should give a time of 0.100 ms = 100 μs as shown in Figure 3.44(c).

Figure 3.44 (a) Time recorded at first program breakpoint. (b) Time interval between breakpoints. (c) Time interval between breakpoints with amended delay time

This could be checked since the Raisonance software has another, more visual, way of measuring the time. Whilst in the Debug/Simulation mode if View is selected from the top menu bar, then from the window that appears, choosing Trace and then View (see Figure 3.45), a Trace window appears.

The windows, including the Watches window, can be organised and the breakpoints removed (e.g. by moving the cursor to grey column till cursor changes, then CLM). The result is shown in Figure 3.46.

To see the effect of variations in P1.7 logic levels with time, it is necessary to add P1.7 into the Watches window; CLM on the Watches icon, shown in Figure 3.38, results in a blank Expression window appearing. P1.7 can be typed in, as shown in Figure 3.47, and the OK button clicked using the left mouse button.

Figure 3.45 Selecting Trace View

Note: CLM on the radio button to the right of the Expression window, shown in Figure 3.47, drops down a list of all valid expressions for the current program.

P1.7 will appear in the Watches window with its default state True (i.e. logic 1). Now moving the mouse cursor onto the light grey line coming from P of P1.7 and clicking the Right mouse button (CRM), a small window appears as shown in Figure 3.48. Choosing Add/Delete from Trace List, as shown in

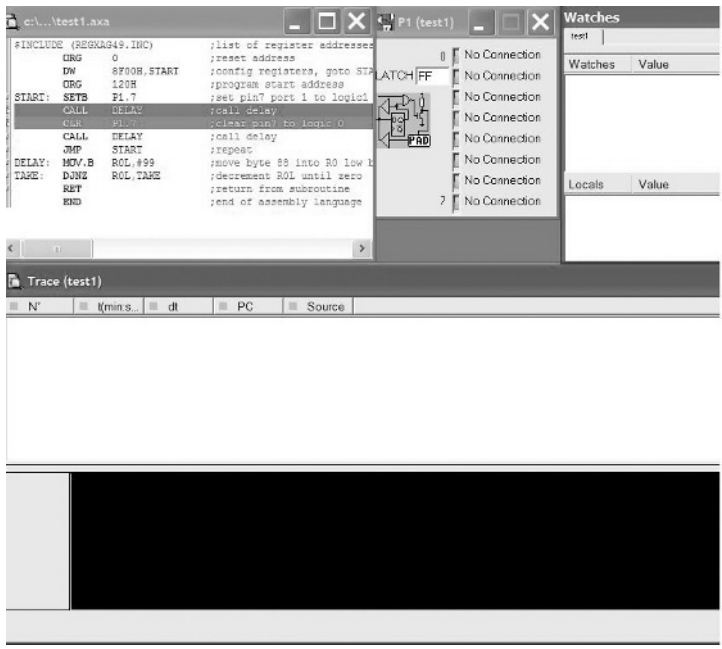


Figure 3.46 Trace window for test1.axa program

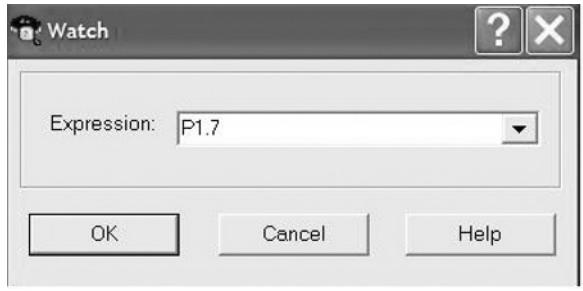


Figure 3.47 Entering an expression in a Watch window

Figure 3.48, causes a small blue T in a circle to appear to the left of P1.7. See Figure 3.49. Also a P1.7 button will appear as the last on a row in the Trace window. CLM on this P1.7 button will cause P1.7 to also appear to the left of the black space. See Figure 3.50.

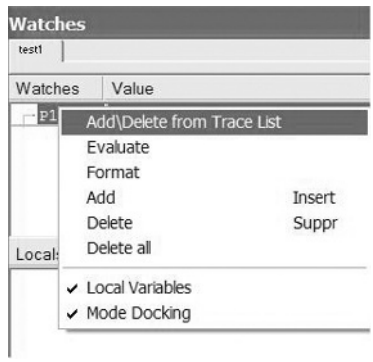


Figure 3.48 Window with option to add/delete from Trace list

Figure 3.49 Indication that P1.7 is added to Trace list

Moving the mouse cursor into white space, above the black space, below the row of buttons, and CRM will generate a small window: only Options is active and this should be chosen. The window of Figure 3.51 should now appear. Mode should be set to Continual, the Rolling trace checked to ensure it is ticked and the maximum number of records set to 2000. CLM on OK and the black space P1.7 disappears, CLM on the Trace P1.7 button will retrieve it.

Making sure animation is off and CLM on GO, the program should be allowed to run for about 3 s before CLM on Stop; the Trace window should be as shown in Figure 3.52. The time column shown in Figure 3.52 can be

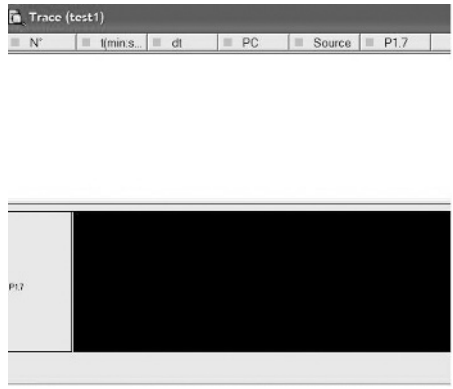


Figure 3.50 Trace window for program test1.axa indicating P1.7 is on the Trace list

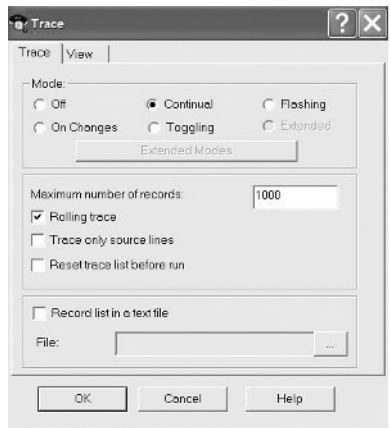


Figure 3.51 Trace Options window

widened by positioning the cursor on the line joining the t button to the dt button, CLM, hold and dragging right. The trace shown in Figure 3.52 starts off as logic 0 and so under P1.7 appears as FALSE. It is a binary signal, hence it will start TRUE (logic 1) or FALSE (logic 0).

There are five sets of numbers under the time t column; their units are minutes, seconds, ms (10^{-3}), μs (10^{-6}), ns (10^{-9}). It is possible to CLM on the Trace scroll button until P1.7 changes from FALSE to TRUE (or TRUE to FALSE). If the cursor is positioned next to the corresponding record number under the N column (see Figure 3.53) (161 in this example) and CLM, a blue horizontal marker appears in the Trace text and a white vertical cursor line appears in the Trace screen on the signal edge which changes from FALSE to TRUE. The time reads 140 ms 428 μs 440 ns.

Scrolling the Trace to the next FALSE and again marking the position, as shown in Figure 3.54, will give the time interval between transitions. From Figure 3.54 the time is seen to be 140 ms 528 μs 700 ns.

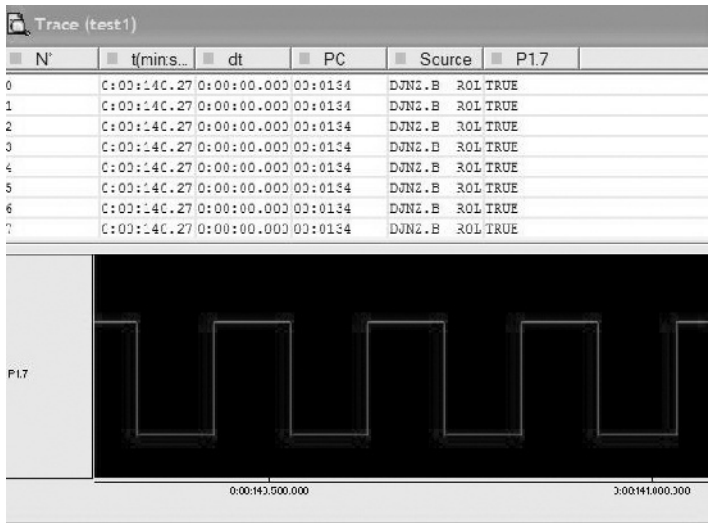


Figure 3.52 Trace window response to test1.axa program

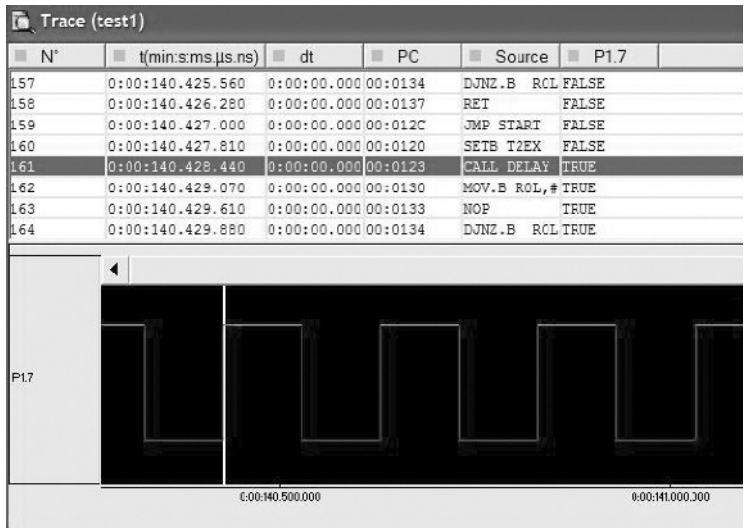


Figure 3.53 Establishing a transition time for program test1.axa

The delay time is the difference between this value and the earlier recorded value, i.e.

$$528.700\,\mu s - 428.440\,\mu s = 100.26\,\mu s$$

Multiple traces can be displayed on the Trace window. Block data such as port values can also be shown; hex values tend to be given. See Figure 3.55.

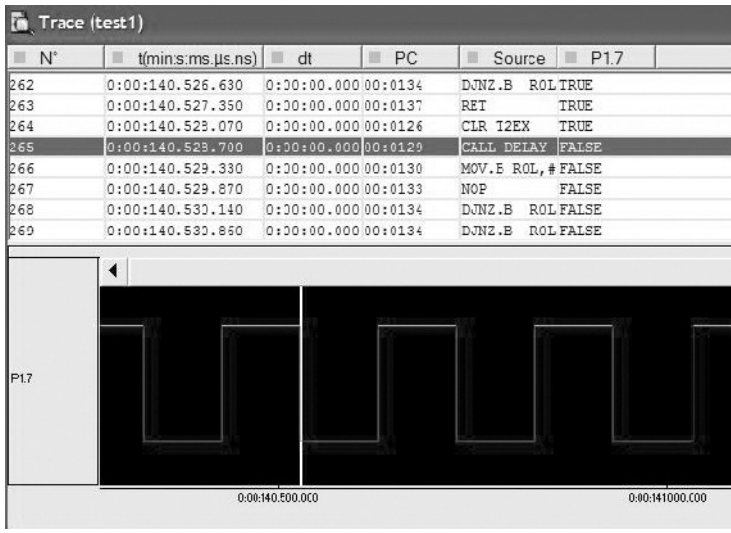


Figure 3.54 Establishing a second transition time to give time duration of a positive pulse for program test1.axa

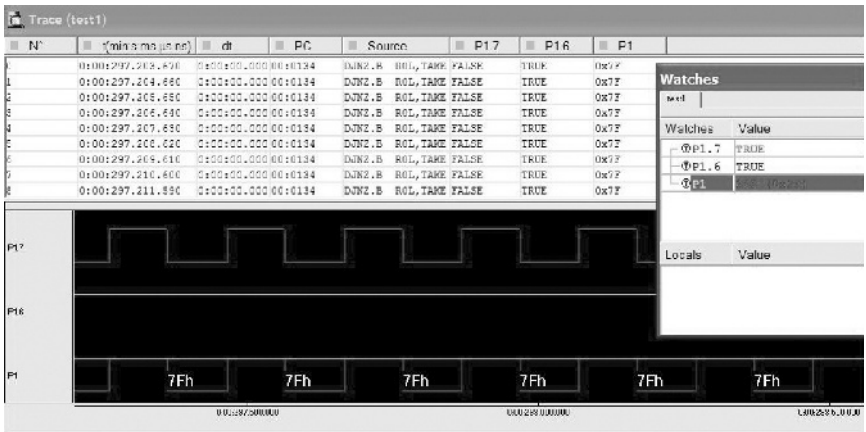


Figure 3.55 A Trace response for program test1.axa showing changes on port 1 in general and port 1, pin 7 in particular

Summary

- Software suitable for simulating and debugging programs is readily available.
- Evaluation software from Keil supports the LPC and 89C66x microcontroller families.
- Evaluation software from Raisonance supports the LPC, 89C66x and XA microcontroller families.

- Both sets of software operate within IDE.
- Both sets of software support microcontroller devices from various manufacturers.
- The software can be used for programming in both assembly language and C.
- Writing of the source program C or assembly language, syntax check, compilation to hex, program debugging/simulation all takes place within the integrated environment.
- When the debugging/simulation is satisfactorily completed, the microcontroller is programmed with the .hex version of the program.

4

P89C66x Microcontroller

4.1 Introduction

This device is a member of the 80C51 family able to execute one instruction in six clock cycles, hence providing twice the speed of a conventional 80C51. A one time programmable (OTP) configuration bit gives the user the option to select conventional 12-clock timing. This device is a single-chip 8-bit microcontroller manufactured in an advanced CMOS process. The instruction set is 100% executing and timing compatible with the 80C51 instruction set. Further information on the device, including details of the SFRs, can be found in Appendix D while details of the 80C51 Instruction set can be found in Appendix A.

Examples used in this chapter include simulation, using software that, unless otherwise stated, is available from Keil. Details regarding simulation software used in this text are covered in Chapter 3. The majority of examples in this chapter requiring programs to be written utilise assembly language. The use of high level (C) language programs is, in the main, left as an exercise for the reader although one example illustrating the use of a program written in C has been included. The solutions to all exercises in this chapter, and other chapters where relevant, can be found at the end of the book.

There are four devices in this family of microcontrollers:

1. P89C660 16 KB Flash Code Memory 512 bytes onboard RAM
2. P89C662 32 KB Flash Code Memory 1 KB onboard RAM
3. P89C664 64 KB Flash Code Memory 2 KB onboard RAM
4. P89C668 64 KB Flash Code Memory 8 KB onboard RAM

All the devices have four 8-bit ports and an onboard clock oscillator, the frequency being defined by an externally connected piezo-crystal or ceramic resonator.

The P89C66x onboard peripherals include:

- three timers – timer 0, timer 1, timer 2
- programmable counter array (PCA)
- universal asynchronous receiver transmitter (UART)
- inter integrated circuit (I^2C) interface.

4.2 Timers 0 and 1

Timers 0 and 1 are fundamentally the same and both have two 8-bit registers, timer high (TH) byte and timer low (TL) byte. Both share the timer control (TCON) register and the timer mode (TMOD) register. The arrangement is shown in Figure 4.1.

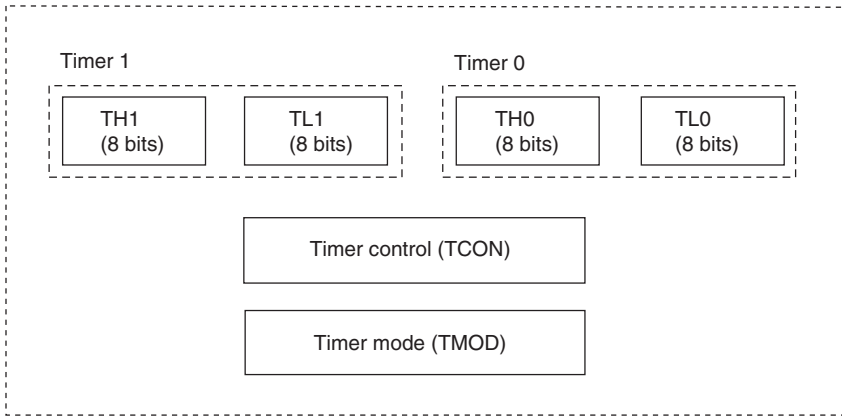


Figure 4.1 Timer 0/1 arrangement for the 89C66x family

The timers can be configured into one of the four modes:

Mode 0 TH and TL come together to form a 13-bit register where TH has 5 bits. This makes the microcontroller compatible with an earlier device.

Mode 1 TH and TL come together to form a 16-bit register.

Mode 2 TL is the working 8-bit register and TH is the automatic reload register. This mode is used to define the baud rate of the serial UART interface.

Mode 3 TH and TL registers of both timers combine to produce three 8-bit timers.

Details of the TMOD SFR are:

TMOD

GATE	C/T	M1	M0	GATE	C/T	M1	M0
Timer 1				Timer 0			

This register has two identical halves, the upper four bits for timer 1 and the lower four bits for timer 0. The bits M1 and M0 set the TMOD:

M1	M0	
0	0	mode 0
0	1	mode 1
1	0	mode 2
1	1	mode 3

MODE 1 16-BIT UP COUNTER

Figure 4.2 shows the timer in mode 1 with the TH and TL registers together forming a 16-bit register. The diagram also shows that for this microcontroller family, the timer clock is one-sixth of the microcontroller clock. The default value of TMOD is 00H, so $C/\overline{T} = 0$ and the peripheral defaults to being a timer rather than a pulse counter. The default value of GATE is 0 and this is inverted, so the OR output defaults to logic 1. The timer is turned on or off by TR1/0 (in TCON register), putting TR0 = 1 would turn timer 0 on.

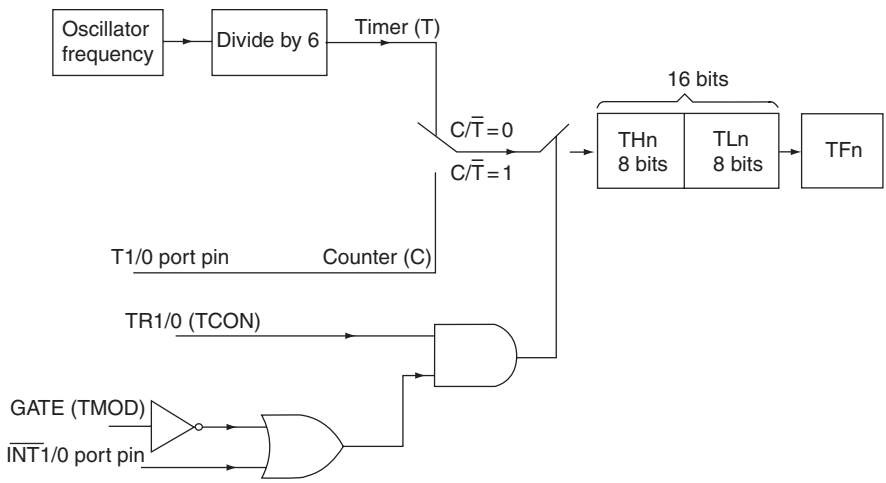


Figure 4.2 Circuit for timer 0/1 operating as a 16-bit up-counter in mode 1

In mode 1, the TH and TL registers in timer 0 or timer 1 join to form a 16-bit up counter. The counter can be loaded with a base number from which the timer can increment upwards towards the 16-bit maximum of 65535 (FFFFH). The time taken to count from the base number to the maximum count value is the required delay. Figure 4.3 shows the method of achieving the required delay.

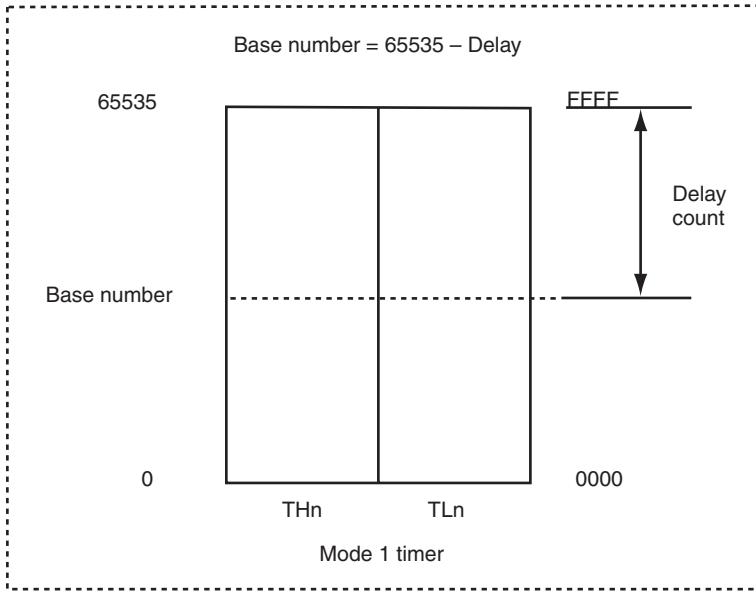


Figure 4.3 Determination of delay count for a mode 1 timer

Example 4.1
A 1 kHz square-wave signal is to be generated from pin 7 on port 1. The microcontroller clock frequency is 11.0592 MHz.

(a) Determine the required delay time.
(b) Using timer 0 determine the base numbers that must go into TH0 and TL0.

Solution
The required waveform is shown in Figure 4.4.

(a) One cycle time T of the required square-wave signal equals 1/frequency

$$T = 1/1000 = 1\,\text{ms}$$
$$\text{Delay time} = T/2 = 0.5\,\text{ms}$$

(b)
$$\text{Timer clock} = \text{microclock}/6$$
$$= 11.0592\,\text{MHz}/6 = 1.8432\,\text{MHz}$$
$$\text{Timer cycle time} = 1/1.8432\,\text{MHz} = 542.54\,\text{ns}$$
$$\text{Delay count} = (\text{delay time})/(\text{timer cycle time})$$
$$= 0.5\,\text{ms}/542.54\,\text{ns} = 922 \text{ (nearest whole number)}$$

$$\text{Base number} = 65535 - \text{delay count}$$
$$= 65535 - 922 = 64613$$

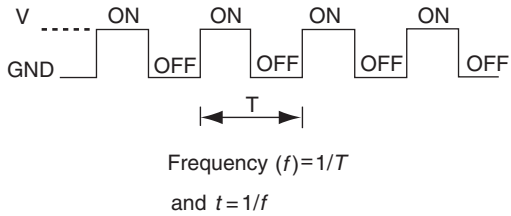


Figure 4.4 Square-wave signal to be generated at port 1, pin 7

TH0 = whole number of 64613/256
64613/256 = 252.3945313 = whole number of 252
TH0 = 252 assembly language MOV TH0, #252

TL0 = (remainder of 64613/256)252
= (0.3945313)256
TL0 = 101 assembly language MOV TL0, #101

Base number in hexadecimal

A calculator may have hex conversion; on some Casio calculators, it is accessed via the Mode button. The PC calculator in scientific view may be used; enter using decimal (Dec), then select hex (Hex). Figure 4.5 shows the value of 64613 entered with Dec selected. Clicking on the Hex button would give the hexadecimal value of FC65.

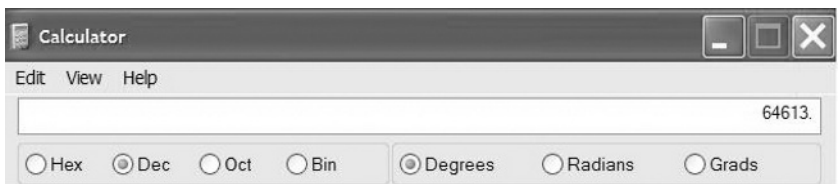


Figure 4.5 Use of the PC calculator to convert a decimal number to its hexadecimal equivalent

Alternatively the decimal numbers can be converted to hex values using the technique described in Chapter 1, e.g.:

252/16 = 15.75 = 15 and 0.75 × 16 = 12
thus 252 decimal = FC in hex

101/16 = 6.3125 = 6 and 0.3125 × 16 = 5
thus 101 decimal = 65 in hex

Whichever method used gives 64613 decimal = FC65 hex.

Loading the timer, using assembly language, can be achieved as follows:

```
MOV   TH0,#0FCH   ; 0 required before leading hex symbol
MOV   TL0,#65H
```

The software debugger/simulator will display all numbers in hex, so it is essential to be prepared.

Exercise 4.1

Repeat Example 4.1 to produce a 2 kHz square wave at port 1, pin 7. Assume the clock frequency remains at the same value.

Rollover

The timer clock increments in timer clock cycles from the base number up to the maximum value of the 16-bit register, which is FFFFH. One more increment would cause the register to rollover to 0000H and set the timer flag (TF) to 1. The TF is a bit in the TCON SFR:

TCON

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

The four least significant bits (LSB) are concerned with the trigger profiles of the interrupt signals $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$. This will be examined later in the chapter.

On power-up the default values of all the TCON bits are zero and so the timer flags TF1 and TF0 are 0. Timer 1 is turned on by making TR1=1 and it is turned off by making TR1=0; the control action of timer 0 is the same using TR0.

As soon as TR0=1, the timer 0 mode1 TH0,TL0 registers start incrementing upwards from their base number. Upon rollover the TF0 flag sets to 1 and this indicates that the delay has been completed. A possible assembly language routine for the 0.5 ms delay could be:

```
DELAY:   MOV    TH0,#0FCH   ; move hex FC into TH0
         MOV    TL0,#65H    ; move hex 65 into TL0
         SETB   TR0         ; turn timer 0 on
FLAG:    JNB    TF0,FLAG    ; jump to FLAG if TF0 is not bit (i.e.
                            ; not 1)
         CLR    TR0         ; turn timer 0 off
         CLR    TF0         ; clear TF0 to zero
         RET                ; return from subroutine
```

Example 4.2

A P89C664 microcontroller having an 11.0592 MHz clock is to be used to generate a 1 kHz square-wave signal from pin 7 of port 1. Write a suitable assembly program to achieve this.

Solution

Square-wave cycle time = 1/1 kHz = 1/1000 = 1 ms
Delay required of a square wave = half the cycle time = 0.5 ms

Timer 0 clock = (micro clock)/6 = 11.0592 MHz/6
= 1.8432 MHz
Timer 0 clock cycle time = 1/1.8432 MHz = 542.54 ns

Delay count = (delay time)/(timer clock cycle time)
= 0.5 ms/542.54 ns = 922 (to nearest whole number)

Mode 1 timer base number = 65535 − delay count
= 65535 − 922
= 64613 decimal
= FC65 hex

FC hex to go into TH0
65 hex to go into TL0

Program

```
          ORG    0           ; reset address
          SJMP   START       ; short jump over reserved area
          ORG    40H         ; program start address at 0040H
START:    MOV    TMOD,#01H   ; put Timer 0 into mode 1
AGAIN:    SETB   P1.7        ; pin 7 port 1 to logic 1 (5 volts)
          ACALL  DELAY       ; go to 0.5ms delay
          CLR    P1.7        ; pin 7 port 1 to logic 0 (0 volts)
          ACALL  DELAY       ; go to 0.5ms delay
          SJMP   AGAIN       ; repeat
DELAY:    MOV    TH0,#0FCH   ; high byte base number into TH0
          MOV    TL0,#65H    ; low byte base number into TL0
          SETB   TR0         ; turn Timer 0 on
FLAG:     JNB    TF0,FLAG    ; repeat until rollover when TF0 = 1
          CLR    TR0         ; turn Timer 0 off
          CLR    TF0         ; clear TF0 back to 0
          RET                ; return from delay subroutine
          END                ; no more assembly language after here
```

Simulation

If from Peripherals the following are chosen:

I/O Ports . . . . . Port 1

Timer . . . . . Timer 0

with breakpoints placed at the program lines:

ACALL DELAY

CLR P1.7

The simulation response will be as shown in Figure 4.6. It can be seen from Figure 4.6 that the time to the first breakpoint is shown as 0.00000271, which is 0.00271 ms. The time to the next breakpoint is 0.00050998, which is 0.50998 ms. The difference between the two values is thus:

$$0.50998\,\text{ms} - 0.00271\,\text{ms} = 0.50727\,\text{ms}$$

The contents of TL0 could be altered to bring this difference closer to 0.5 ms. To decrease the measured delay, it would be necessary to increase the base number in TL0: maybe by increasing 65H to 72H.

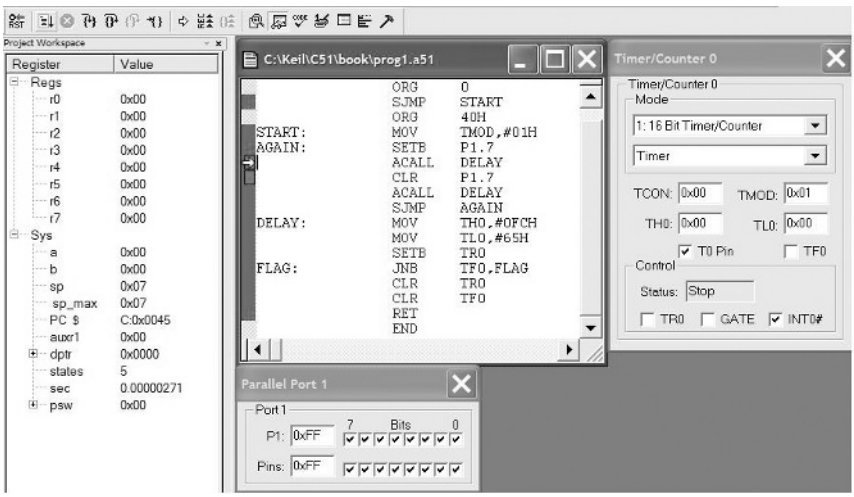


Figure 4.6 Simulation display showing the use of breakpoints

After final adjustment of TL0, it would be useful to single step through the program cycle.

If TH0 and TL0 are loaded in turn, then TCON and TR0 will change when timer 0 is turned on. TL0 increments as the program repeats at JNB. Moving the mouse cursor to the TH0 window would allow the user to change the register contents to a value of FF: similarly moving the cursor to the TL0 window would permit its value to be altered to FC. If single stepping of the program is continued, then TH0 and TL0 would be seen to roll over to 0000 and TF0 go to 1.

Alternatively, register values can be changed at the chevron sign in the Command window at the bottom of the Debug page. This is illustrated in Figure 4.7. Changes made this way have the advantage of being stored and repeated in turn by pressing the upward cursor key on the PC keyboard.

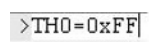


Figure 4.7 Use of Command window to change register values

Example 4.3
A P89C664 microcontroller having an 11.0592 MHz clock is to be used to generate a 1 kHz square-wave signal from pin 7 of port 1. Write a C program to achieve this.

Solution
A suitable program would be:

```
#include <reg66x.h>
#define on 1
#define off 0
sbit  SquareWavePin = P1^7; // pin 7 of port1
void delay1KHz();           // delay-on() returns nothing and
                            // takes nothing
main() {                    // start of the program
      TMOD = 0x01;          // timer1 : Gate = 0 CT = 0 M1 = 0
                            // M0=0
                            // timer0 : Gate=0 CT=0 M1=0 M0=1;
                            // mode 1
      while(1) {            // do for ever
        SquareWavePin = on; // P1.7 set to 1
        delay1KHz();        // wait for on time
        SquareWavePin = off;// P1.7 set to 0
        delay1KHz();        // wait for off time
      }                     // while()
}                           // main()
void delay1KHz() {
          TH0 = ~(922/256); // ~(3) = ~(bin:0000 0011) = bin:1111
                            // 1100 = hex : FC
          TL0 = -(922 % 256);// -(154) = -(bin:1001 1010) = bin:
                            // -0110 0110 = hex:66
          TR0 = on;         // set TR0 of TCON to run timer0
          while(!TF0);      // wait for timer0 to set the Flag TF0;
          TR0 = off;        // stop the timer0
          TF0 = off;        // clear flag TF0
}                           // delay()
```

MODE 2 EIGHT-BIT UP-COUNTER

The instruction MOV TMOD,#02H would put timer 0 into mode 2 defining an 8-bit timer using TL0 as the working register and TH0 as the automatic reload register. The circuit arrangement is shown in Figure 4.8.

In mode 2, the working register is only 8 bits wide and so the base number is 8 bits wide. When the TL register rolls over it is automatically reloaded with the contents of the TH register, whose loaded contents remain the same, so the base number goes into the TH register.

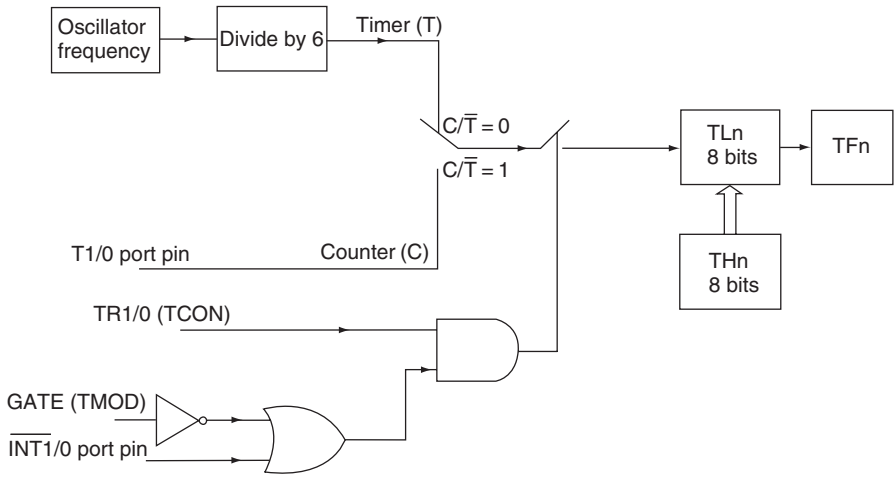


Figure 4.8 Circuit for timer 1/0 to operate as an 8-bit up-counter in mode 2

Example 4.4
A P89C664 microcontroller having an 11.0592 MHz clock is to be used to generate a 5 kHz square-wave signal from pin 7 of port 1. Write a suitable program to achieve this.

Solution

Square-wave cycle time = 1/5 kHz = 1/5000 = 0.2 ms
Delay required for a square wave = half the cycle time = 0.1ms = 100μs

Timer 0 clock = (micro clock)/6 = 11.0592 MHz/6 = 1.8432 MHz
Timer 0 clock cycle time = 1/1.8432 MHz = 542.54 ns

Delay count = (delay time)/(timer clock cycle time)
= 100 μs/542.54 ns = 184 (to nearest whole number)

Mode 2 timer base number = 225 − delay count
(225 = maximum value of 8-bit register)
= 255 − 184
= 71 decimal
= 47 hex

47 hex is to go into TH0
47 hex is to go into TL0
TL0 could start with its default value of 00H since the first half cycle would not be seen on an oscilloscope screen! The line MOV TL0,#47H can be left out of the program since after the first half cycle TL0 will automatically be reloaded with 47H from TH0.

Program

```
        ORG     0           ; reset address
        SJMP    START       ; short jump over reserved area
        ORG     40H         ; program start address at 0040H
START:  MOV     TMOD,#02H   ; put Timer 0 into mode 2
        MOV     TH0,#47H    ; auto-reload base number into TH0
AGAIN:  SETB    P1.7        ; pin 7 port1 to logic 1 (5 volts)
        ACALL   DELAY       ; go to 0.5ms delay
        CLR     P1.7        ; pin 7 port1 to logic 0 (0 volts)
        ACALL   DELAY       ; go to 0.5ms delay
        SJMP    AGAIN       ; repeat
DELAY:  SETB    TR0         ; turn Timer 0 on
FLAG:   JNB     TF0,FLAG    ; repeat until rollover when TF0=1
        CLR     TR0         ; turn Timer 0 off
        CLR     TF0         ; clear TF0 back to 0
        RET                 ; return from delay subroutine
        END                 ; no more assembly language after here
```

Simulation

Time taken to run to the first breakpoint ACALL is 0.00000380 = 3.80 μs. TH0 is loaded with 47 hex and TL0 has its default value of 00 hex. This is shown on the simulation response in Figure 4.9.

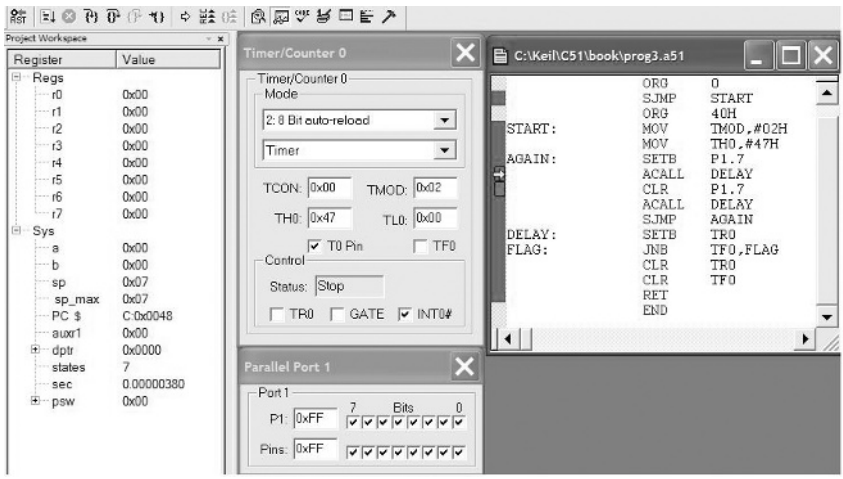


Figure 4.9 Simulation display showing the use of breakpoints

Clicking left mouse (CLM) on the simulation run button twice causes the program to come back to the first breakpoint which gives a time as shown in Figure 4.10. From Figure 4.10, the time is given as 0.00025336 s = 253.36 μs. CLM on simulation run button once more would give the time shown as:

0.00035699 s = 356.99 μs

Figure 4.10 Breakpoint timing value display

The difference is equal to 103.63 μs which is close to the 100 μs delay required for the 5 kHz square wave. Changing 47H in TH0 to 4FH would give a closer result.

Exercise 4.2
A P89C664 microcontroller having an 11.0592 MHz clock is to be used to generate a 5 kHz square-wave signal from pin 7 of port 1. Write a C program to achieve this.

TIMER INTERRUPT

When an interrupt occurs the processor pauses, saves the current program counter (PC) value into RAM designated by the stack pointer (SP) and then jumps to the interrupt vector address. The processor then carries out the instructions at the interrupt vector address and returns to the original program sequence, retrieving the previous PC data. The interrupt program must end with RETI (return from interrupt).

The P89C66x microcontroller has nine interrupts, if reset is included, as shown in Table 4.1. It can be seen from Table 4.1 that previous assembly language programs started from address 0040H in order to leave the interrupt vectors as a reserved space.

Timer interrupts can be made to occur when the TF is set at rollover. This is achieved by setting the relative bits in the interrupt enable (IE) registers.

IE0

EA	EC	ES1	ES0	ET1	EX1	ET0	EX0

EA Enable all and must always be set when interrupts are used. By putting EA = 0 any arrangement of interrupts can be disabled
EC PCA interrupt enable
ES1 I^2C interrupt enable
ES0 UART interrupt enable
ET1 Timer 1 interrupt enable
EX1 External 1 interrupt enable
ET0 Timer 0 interrupt enable
EX0 External 0 interrupt enable

IE1

–	–	–	–	–	–	–	ET2

ET2 Timer 2 interrupt enable (EA in IE0 must also be set)

Table 4.1 P89C66x interrupts

Source	Interrupt vector address	Polling priority
Reset	00H	0 (highest)
External 0	03H	1
I^2C	2BH	2
Timer 0	0BH	3
External 1	13H	4
Timer 1	1BH	5
UART	23H	6
Timer 2	3BH	7
PCA	33H	8 (lowest)

Example 4.5
Modify the program of the previous example such that a timer 0 interrupt causes the logic level on pin 7 port 1 to be toggled (switched to opposite logic level) producing a square wave of frequency 5 kHz.

Solution

Program could be

```
        ORG    0           ; reset address
        SJMP   START       ; short jump over reserved area
        ORG    0BH         ; Timer 0 interrupt vector address
        SJMP   TASK        ; go to interrupt routine
        ORG    40H         ; program start address at 0040H
START:  MOV    TMOD,#02H   ; put Timer 0 into mode 2
        MOV    TH0,#47H    ; auto-reload base number into TH0
        SETB   EA          ; enable all
        SETB   ET0         ; enable Timer 0 interrupt
        SETB   TR0         ; turn Timer 0 on
AGAIN:  SJMP   AGAIN       ; stay here till interrupt occurs
TASK:   CPL    P1.7        ; complement (i.e. toggle) pin 7 port 1
        RETI               ; return from interrupt routine
        END                ; end of assembly language
```

Simulation
The response is shown in Figure 4.11. The interrupt window is shown in Figure 4.11. When in debug the Interrupt window is obtained from Peripherals on the top menu bar. Putting a breakpoint at the TASK label and running the program to this point give a timing of 0.00014594 s = 145.94 μs. Running the simulation once more the timing increases to 0.00024631 s = 246.31 μs. The difference between this and the previous value is 100.37 μs, which is very close to the required 100 μs.

Exercise 4.3
Modify the C program of Exercise 4.2 such that a timer 0 interrupt causes the logic level on pin 7 port 1 to be toggled (switched to opposite logic level) producing a square wave of frequency 5 kHz.

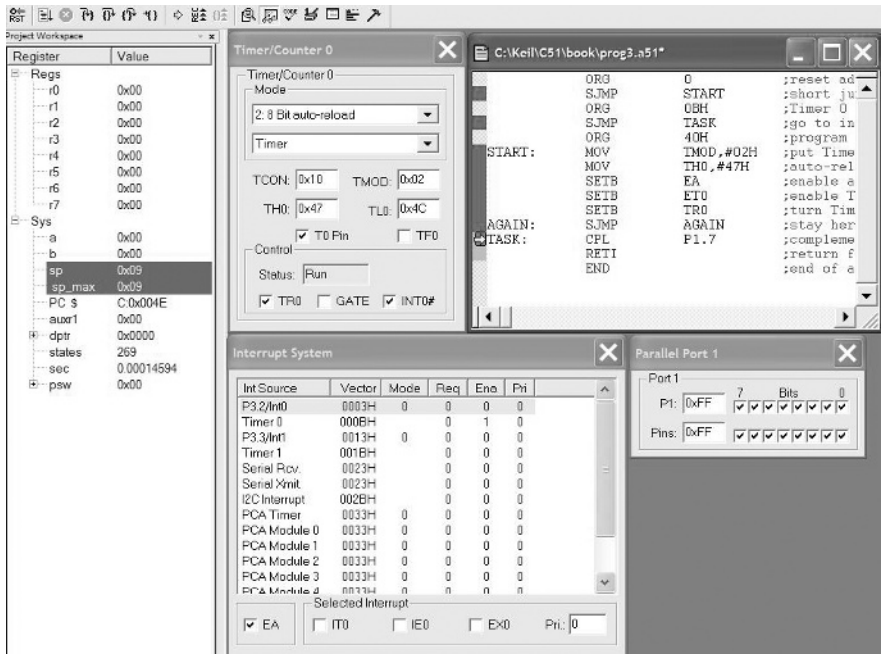


Figure 4.11 Simulation display with breakpoint used to determine timing

4.3 Timer 2

The previous program could be viewed as the basis of a simple multitasking system where the microcontroller performed a task of complementing pin 7 on port 1 every 100 µs. The auto-reload and automatic clearing of the Timer Flag meant that once the timer reload register had been set up and the timer turned on it could be left to continually interrupt every 100 µs. Because the working register TL0 is only 8 bits wide the time duration of the interrupt signal is small.

The P89C664 has Timer 2, which has 16-bit auto-reload giving a maximum count of 65536 (2^{16}). Timer 2 has three operating modes:

1. capture mode
2. 16-bit auto-reload mode
3. baud rate generator mode.

Auto-reload is the default mode. Capture mode causes data in TL2 and TH2 to be transferred to the capture registers RCAP2L and RCAP2H when there is a 1-to-0 transition on T2EX (port 1.1).

Timer 1 can be used as the serial port baud rate generator but has limitations on the minimum baud rate. For example, with an oscillator frequency of

11.0592 MHz the baud rate generation can only go down to the standard rate of 4800, for lower values the oscillator frequency must be lowered. Timer 2 having 16-bit auto-reload gets over this problem. (See Section 4.9.)

The control register associated with timer 2 is T2CON.

T2CON

TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2	CP/RL2

TF2	rollover or overflow flag, it must be cleared by software
TR2	turns timer 2 on (1) or off (0)
C/T2	increments TL2, TH2 by onboard timer (0) or external negative edge on port1.0 (T2)
CP/RL2	when 1 (capture mode), when 0 (auto-reload on rollover)
EXF2	external flag set to 1 when there is a negative transition on port 1.1 (T2EX)
EXEN2	external enable flag, when 1 allows capture or reload following a negative transition on T2EX
RCLK, TCLK	when 1, baud rate generator mode

Example 4.6

A P89C664 microcontroller has an oscillator frequency of 11.0592 MHz. Write a program that causes timer 2 to generate an interrupt every 10 ms toggling pin 7 on port 1.

Solution

Oscillator frequency = 11.0592 MHz
Therefore timer clock = 1.8432 MHz
Timer clock cycle = 542.54 ns
Delay time count = 10 ms/542.54 ns = 18432 (nearest whole number)
Timer 2 base number = 65535 − 18432 = 47103 decimal = B7FFH

In timer 2 the reload register for TH2 is RCAP2H (capture register 2) and the reload register for TL2 is RCAP2L. ET2 in IEN1 enables timer 2 interrupt.

IEN1

–	–	–	–	–	–	–	ET2

The evaluation version of the assembly language software does not have timer 2 SFRs; therefore the program starts by equating (EQU) the SFR labels to their hex addresses. This information is obtained from the microcontroller data sheet, see Appendix D.

The program uses OR logic (ORL) to force logic 1 in the SFRs without affecting other bits, and it also uses AND logic (ANL) to force logic 0.

Program

```
RCAP2H   EQU   0CBH          ; sfr address = CBH
RCAP2L   EQU   0CAH          ; sfr address = CAH
IEN1     EQU   0E8H          ; ser address = E8H
T2CON    EQU   0C8H          ; sfr address = C8H
         ORG   0             ; reset address
         SJMP  START         ; jump over reserved area
         ORG   3BH           ; Timer2 interrupt address
         SJMP  TASK          ; jump to interrupt task
         ORG   40H           ; program start address
START:   MOV   RCAP2H,#0B7H  ; B7H into RCAP2H
         MOV   RCAP2L,#0FFH  ; FFH into RCAP2L
         SETB  EA            ; enable all interrupts
         ORL   IEN1,#01H     ; enable Timer2(ET2) interrupt
         ORL   T2CON,#04H    ; turn Timer2 on
AGAIN:   SJMP  AGAIN         ; stay here till interrupt
TASK:    CPL   P1.7          ; toggle P1.7
         ANL   T2CON,#7FH    ; clear Timer2 flag(TF2)
         RETI                ; return from interrupt
         END                 ; end of assembly language
```

Simulation

The simulation response is shown in Figure 4.12. Setting the breakpoint at the TASK label and running the simulation would initially give a large time count in the sec register because TH2 and TL2 (T2 in the timer simulation window) start with their default values of zero. In Figure 4.12 the sec count is seen to be 0.03556261 which is 35.56261 ms. Running the simulation once more would see the sec register change to 0.04556315 which is 45.56315 ms, a difference of approximately 10 ms.

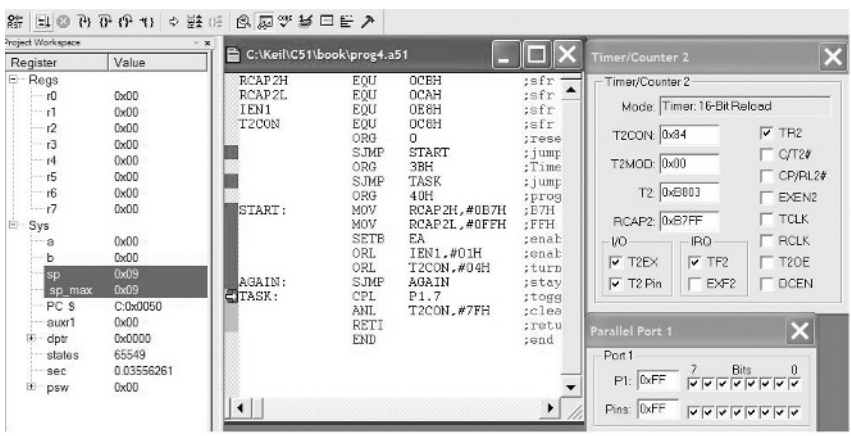


Figure 4.12 Simulation display with breakpoint used to determine timing

Exercise 4.4
A P89C664 microcontroller has an oscillator frequency of 11.0592 MHz. Write a C program that causes timer 2 to generate an interrupt every 10 ms toggling pin 7 on port 1.

4.4 External interrupt

Negative edge transitions on PORT 3 pins 2 ($\overline{\text{INT0}}$) and 3 ($\overline{\text{INT1}}$) can cause interrupts; their interrupt vector addresses are 03H and 13H respectively. A possible circuit arrangement is shown in Figure 4.13. Figure 4.13 shows a switch circuit where the voltage on P3.3 (port 3 pin 3) is normally 5 V. Pressing the switch causes a negative edge transition as the voltage switches down from 5 V to 0 V. If the switch is pressed and held then a logic 0 level is held on P3.3. These are the two external interrupt conditions; it can be either edge triggered (transition logic 1 to 0) or level triggered (logic 0).

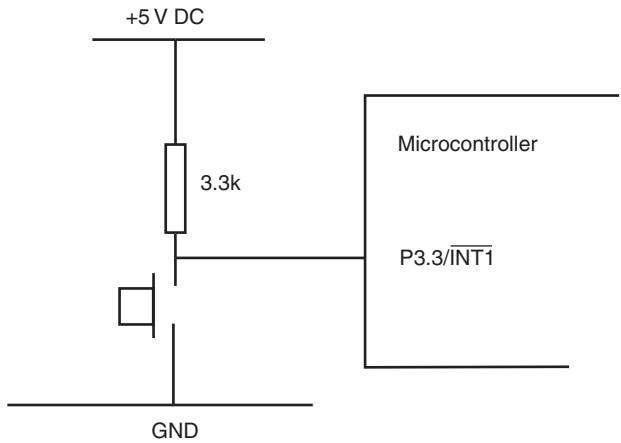


Figure 4.13 Circuit arrangement to produce an interrupt on port 3, pin 3. A similar arrangement can be used for port 3, pin 2

Example 4.7
A P89C664 microcontroller has an oscillator frequency of 11.0592 MHz. Write an assembly language program that complements the logic level on port 1 pin 7 when an edge triggered interrupt occurs on port 3 pin 3 ($\overline{\text{INT1}}$).

Solution
The four least significant bits of the TCON register are used to set the external interrupt parameters.

TCON

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

IT1=1 $\overline{\text{INT1}}$ (P3.3) interrupt activated on a negative edge transition. IE1 flag set to 1 when there is a negative edge transition on $\overline{\text{INT1}}$ (P3.3), cleared automatically when servicing the interrupt. IT1=0 $\overline{\text{INT1}}$ (P3.3) interrupt activated on logic 0 level. IE1 flag set to 1 when there is a logic 0 level on $\overline{\text{INT1}}$ (P3.3), cleared when the logic level on P3.3 goes back to logic 1.

The process for IT0 and IE0 is similar to the above.

Program

```
        ORG    0          ; reset address
        SJMP   START      ; jump over reserved area
        ORG    13H        ; INT1 address vector
        SJMP   TASK       ; jump to interrupt routine
        ORG    40H        ; program start address
START:  SETB   IT1        ; interrupt edge triggered
        SETB   EA         ; enable all set interrupts
        SETB   EX1        ; enable INT1 interrupt
AGAIN:  SJMP   AGAIN      ; stay here till interrupt
TASK:   CPL    P1.7       ; interrupt task
        RETI              ; return from interrupt
        END               ; no more assembly language
```

Simulation

The simulation response is shown in Figure 4.14. The activity of IT1 and IE1 can be seen on the Watch Window or TCON in the timer 1 window. In Figure 4.14 the simulation uses the timer 1 window. When single stepping through the program, the position reached in Figure 4.14 is when P3.3 has just gone to logic 0. At this point Reg in the Interrupt System window goes to 1. TCON changes from 0x04 (SETB IT1) to 0x0C (IT1 and IE1 both set).

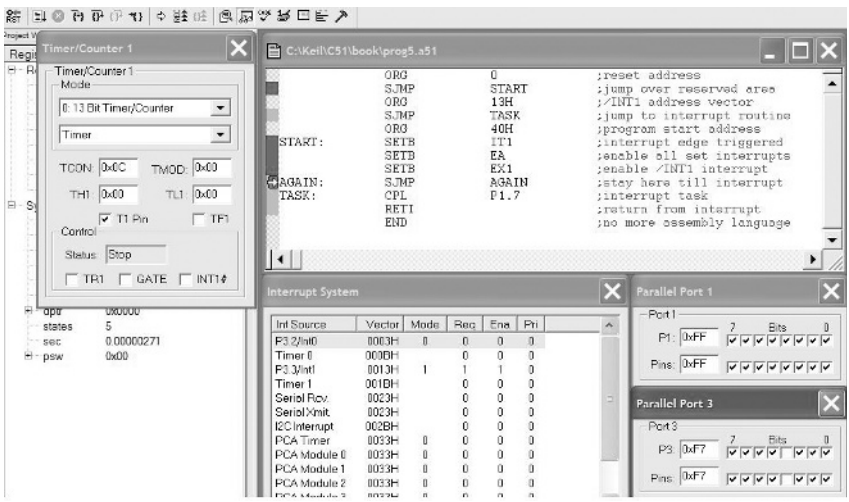


Figure 4.14 Simulation display showing the effect of a program on timer 1

Single stepping once more will cause the program cursor to go to SJMP TASK; TCON to go back to 0x04 and Reg go back to 0. Single stepping twice more would cause P1.7 to go to logic 0.

Exercise 4.5
A P89C664 microcontroller has an oscillator frequency of 11.0592 MHz. Write a C program that complements the logic level on port 1 pin 7 when an edge-triggered interrupt occurs on port 3 pin 3 ($\overline{\text{INT1}}$).

4.5 Interrupt priority

Table 4.1 shows the order in which the interrupts are polled; for example it is seen that timer 0 interrupt is polled or checked before timer 1. The order in which the interrupts are serviced may be set by using two interrupt priority tables, together they give four levels of interrupt as shown in Table 4.2.

Interrupt priority (IP)

PT2	PPC	PS1	PS0	PT1	PX1	PT0	PX0

IPH (high byte)

PT2H	PPCH	PS1H	PS0H	PT1H	PX1H	PT0H	PX0H

PT2 Timer 2
PPC PCA
PS1 I^2C
PS0 UART
PT1 Timer 1
PX1 External 1 ($\overline{\text{INT1}}$)
PT0 Timer 0
PX0 External 0 ($\overline{\text{INT1}}$)

Table 4.2 Priority levels

IPH.x	IP.x	Levels order
0	0	0 (lowest)
0	1	1
1	0	2
1	1	3 (highest)

Example 4.8
Assuming a P89C664 microcontroller is to be used, write an assembly language program that causes timer 1 to have a higher priority than timer 0.

Solution

```
MOV  IP,#0AH     ; A = 1010  PT1(IP.3) = 1 PT0  (IP.1) = 1
MOV  IPH,#08H    ; 8 = 1000  PT1(IPH.3) = 1 PT0 (IPH1) = 0
```

Timer 0 has a priority level of 1 and timer 1 has a priority level of 3, so even if timer 0 interrupt has not finished servicing its task, timer 1 will interrupt the task.

Program

```
IEN0     EQU   0A8H       ; sfr address = A8H
IPH      EQU   0B7H       ; sfr address = B7H

         ORG   0          ; reset address
         SJMP  START      ; jump over reserved area
         ORG   0BH        ; Timer 0 interrupt vector address
         SJMP  TASK0      ; jump to Timer 0 int task
         ORG   1BH        ; Timer 1 interrupt vector address
         SJMP  TASK1      ; jump to Timer 1 int task
START:   MOV   TMOD,#22H  ; Timer 0 & Timer 1 in mode 2
         MOV   IEN0,#8AH  ; EA and Timer 1&2 interrupts
         MOV   TH0,#0F8H  ; hex F8 into TH0
         MOV   TH1,#0EEH  ; hex EE into TH1
         MOV   TL0,#0F8H  ; hex F8 into TL0
         MOV   TL1,#0EEH  ; hex EE into TL1
         MOV   IP,#0AH    ; Timer 0 priority 1 and
         MOV   IPH,#08H   ; Timer 1 priority 3
         MOV   TCON,#50H  ; turn Timers 0 and 1 on
AGAIN:   SJMP  AGAIN      ; stay here till interrupt
TASK0:   CPL   P1.0       ; Timer 0 task, cpl P1.0
         MOV   R0,#55H    ; trivial tasks, 55H to R0
         MOV   A,R0       ; register R0 to Accumulator
         MOV   P2,A       ; Accumulator to port 2
         MOV   R2,#88H    ; 88H to register R2
         MOV   A,R2       ; R2 to Accumulator
         MOV   P2,A       ; Accumulator to port 2
         CPL   P1.0       ; complement P1.0
         RETI             ; return from Task0 interrupt
TASK1:   CPL   P1.1       ; complement port 1 pin 1
         RETI             ; return from Task1 interrupt
         END              ; end of assembly language
```

Simulation

The simulation response is shown in Figure 4.15. It may be seen from the interrupt system window in Figure 4.15 that timer 0 has a priority (Pri) of 1 and timer 1 has the higher priority of 3. Both timers are in 8-bit auto-reload (mode 2). It may also be seen that pins 0 and 1 of port 1 are both at logic 0.

The simulation has been single stepped and the simulation cursor is at RETI in the timer 1 interrupt TASK1. The timer 0 interrupt TASK0 starts off with CPL P1.0 and finishes with the same instruction, so when TASK0 is complete port 1 pin 0 should be showing logic 1. The simulation as shown in Figure 4.15 is at the point where the higher priority timer 1 interrupt has interrupted timer 0

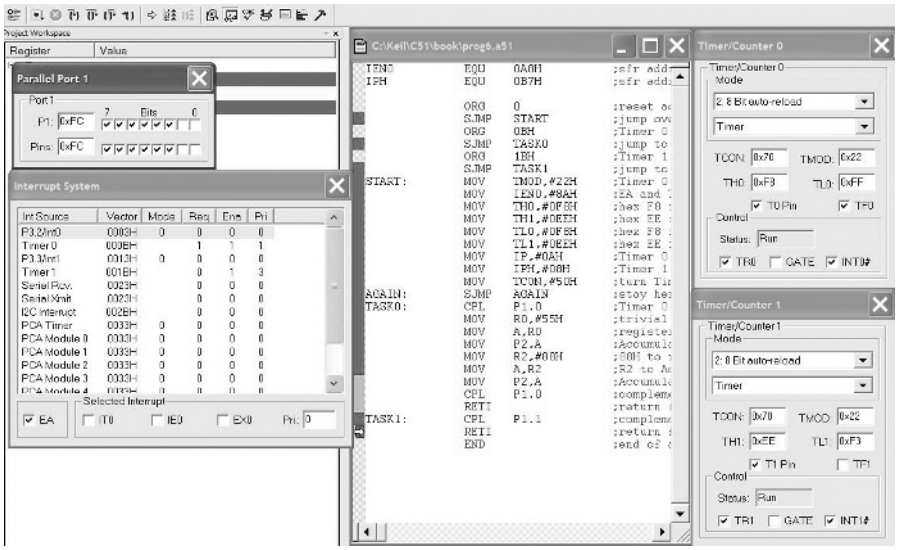


Figure 4.15 Simulation display showing the priority levels of timers 0 and 1

TASK0 before completion, preventing it from reaching the final CPL instruction. When timer 1 interrupt has completed TASK1 the microcontroller returns to the point at which it left timer 0 TASK0 and completes the task.

Exercise 4.6
Assuming a P89C664 microcontroller is used, write a C program that causes timer 1 to have a higher priority than timer 0.

4.6 Programmable counter array (PCA)

The PCA has a 16-bit timer and five 16-bit capture/compare modules each of which can be put into one of seven different configurations. Each of the five modules has a port pin associated with it that may be an input (e.g. interrupt) or an output (e.g. signal out). The 16-bit Timer/Counter is the time base for each of the five modules as can be seen in Figure 4.16. Each of the five modules has a Compare/Capture Mode register (CCAPMn) for selecting one of the seven configurations.

CCAPMn ($n = 0,1,2,3,4$)

–	ECOMn	CAPPn	CAPNn	MATn	TOGn	PWMn	ECCFn

ECOMn when = 1, enable comparator
CAPPn when = 1, capture on positive edge
CAPNn when = 1, capture on negative edge

MATn when = 1, match between counter and capture registers flags an interrupt
TOGn when = 1, port pin toggles when MATn condition occurs
PWMn when = 1, pulse width modulation (PWM) mode
ECCFn when = 1, enables flags (CCFn) in CCON SFR to generate interrupts

Details of the module modes for the CCAPMn register are shown in Table 4.3 for each bit of the register.

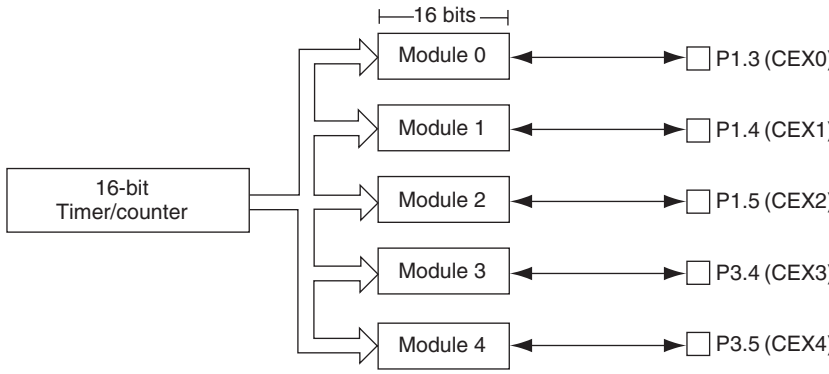


Figure 4.16 Programmable counter array (PCA)

Table 4.3 CCAPMn module modes

X	0	0	0	0	0	0	0	No operation
X	X	1	0	0	0	0	X	16-bit capture by positive edge on CEXn
X	X	0	1	0	0	0	X	16-bit capture by negative edge on CEXn
X	X	1	1	0	0	0	X	16-bit capture by transition on CEXn
X	1	0	0	1	0	0	X	16-bit software timer
X	1	0	0	1	1	0	X	16-bit high speed output
X	1	0	0	0	0	1	0	8-bit PWM
X	1	0	0	1	X	0	X	Watchdog timer

The two other SFRs associated with the PCA are the counter mode register (CMOD) and the counter control register (CCON).

CMOD

CIDL	WDTE	–	–	–	CPS1	CPS0	ECF

CIDL when = 0 (PCA continues during idle mode)
when = 1 (PCA gated off during idle mode)

WDTE when = 0 (disables watchdog timer)
when = 1 (enables watchdog timer)

CPS1	CPS0	
0	0	PCA time base runs at (micro oscillator frequency)/6
0	1	PCA time base runs at (micro oscillator frequency)/2
1	0	PCA time base runs at Timer 0 overflow
1	1	PCA time base runs at external clock on port 1 pin 2 (ECI) (maximum=micro oscillator frequency/4)

ECF = 1 (enables counter overflow interrupt, enables CF bit in CCON SFR)
= 0 (disables counter overflow interrupt)

CCON

CF	CR	–	CCF4	CCF3	CCF2	CCF1	CCF0

CF counter overflow flag
CR when = 1, PCA time base runs
when = 0, PCA time base stops
CCFn interrupt flag, set by hardware when a match or capture occurs. Cleared by software

4.7 Pulse width modulation (PWM)

The use of PWM allows a variable DC average voltage to drive small inductive loads such as a small DC motor. The PWM frequency has to be much faster than the movement of the application. An example of waveforms produced using PWM is shown in Figure 4.17.

The DC average is achieved by variation of the on/off ratio in a cycle. In Figure 4.17 the top signal has an average of $(5\,V \times 9)/10 = 4.5\,V$, while the lower signal has an average of $(5\,V \times 6)/10 = 3\,V$.

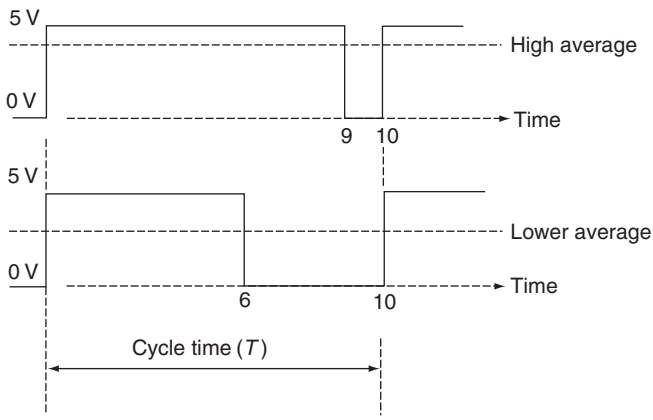


Figure 4.17 Waveforms showing the effect of pulse width modulation (PWM) in varying the average DC voltage over a cycle period (T)

PULSE WIDTH MODULATION (PWM) USING THE PCA

A possible arrangement is shown in Figure 4.18. The example is taking n=1, so the PWM output is from port 1 pin 4 (P1.4). To configure the PCA into PWM mode set bits ECOM1 and PWM1 in the CCAPM1 SFR to 1. The PWM is 8 bits and Figure 4.18 shows the comparison is between the low byte CL of the PCA Timer/Counter and the low byte CCAP1L. The high byte CCAP1H is used to automatically reload CCAP1L when it goes to zero. CCAP1L goes to zero when CL has incremented up to its value. CCAP1H is effectively a marker fixing the on (e.g. 5 V) off (0 V) ratio of the PWM signal. Because the PWM is 8 bits, the cycle time of the PWM is 256 PCA timer clock cycles.

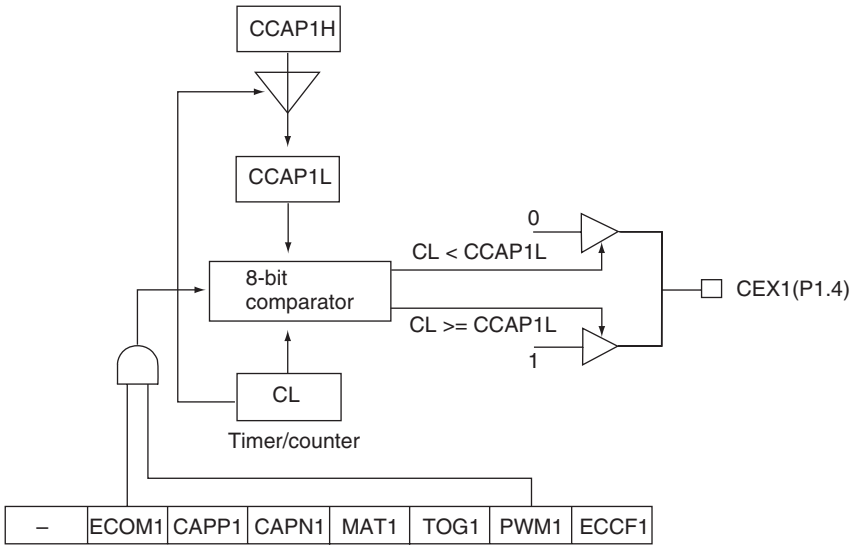


Figure 4.18 Use of the PCA to produce a PWM output from port 1, pin 4

In one PWM cycle CL increases from zero up towards the CCAP1L value (automatically loaded from CCAP1H). During this period, when CL < CCAP1L, the PWM output is logic 0 (0 V). When CL = CCAP1L the latter momentarily goes to zero but is immediately reloaded from CCAP1H. CL continues to increase and for the period CL >= CCAP1L the PWM output is logic 1 (e.g. 5 V). The effect is illustrated in Figure 4.19.

Example 4.9

A P89C664 microcontroller has an oscillator frequency of 11.0592 MHz. Write an assembly language program that will cause the PCA to generate a PWM signal from pin 4 of port 1 with a Mark (logic 1) Space (logic 0) ratio of 6 to 4. The PCA timer clock frequency should be one-sixth of the microcontroller oscillator frequency.

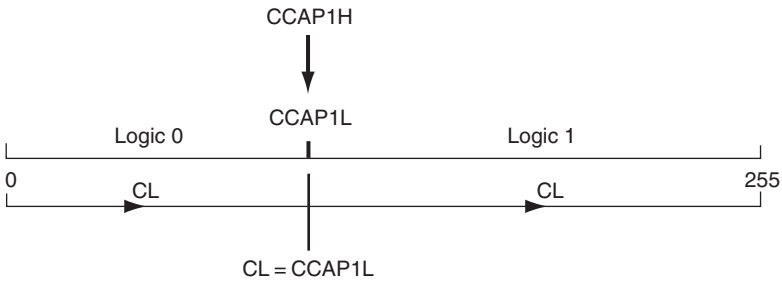


Figure 4.19 Effect on output logic level as CL increases from 0 to 255. Transition occurs as CL increases above the value in CCAP1L

Solution

Ratio 6:4 = 6 + 4 periods = 10 periods
8 bits = 0 to 255 = 256 increments
Therefore one period = 256/10 = 25.6 increments per period
Mark (logic 1) = 6 periods
Hence Mark = 6 × 25.6 = 154 increments (nearest whole number)
CCAP1L (and CCAP1H) = 256 − 154 = 102 = decimal = 66 hex

Since the PCA timer clock frequency = (micro oscillator frequency)/6, the CMOD SFR can assume its default value of 00H. The CR bit in the CCON SFR will have to be set to 1 to turn the PCA time base on.

$$\text{PCA timer clock frequency} = 11.0592\,\text{MHz}/6 = 1.8432\,\text{MHz}$$

$$\text{PCA timer cycle time} = \frac{1}{1.8432}\,\text{MHz} = 542.54\,\text{ns}$$

$$\text{Logic 0 is held for } 102 \times 542.54\,\text{ns} = 55.3\,\mu\text{s}$$

$$\text{Logic 1 is held for } 154 \times 542.54\,\text{ns} = 83.6\,\mu\text{s}$$

Program

```
CCAP1H   EQU    0FBH          ; sfr address
CCAP1L   EQU    0EBH          ; sfr address
CCAPM1   EQU    0C3H          ; sfr address
CCON     EQU    0C0H          ; sfr address

         ORG    0             ; microcontroller reset address
         SJMP   START         ; jump over reserved area
         ORG    40H           ; program start address
START:   ORL    CCAPM1,#42H   ; set ECOM1 and PWM1
         MOV    CCAP1L,#102   ; load 6:4 count
         MOV    CCAP1H,#102   ; 6:4 count reload
         ORL    CCON,#40H     ; set CR to turn PCA timer on
STAY:    SJMP   STAY          ; stay here whilst generating PWM
         END                  ; no more assembly language
```

Simulation

The simulation response is shown in Figure 4.20. The program generates the PWM whilst it remains at the SJMP STAY line, so there is nowhere to measure time using a breakpoint. The Raisonance software has the Trace feature where the signals on the port pins can be displayed. Signal times can be measured from the table above the traces.

From Figure 4.20 the trace cursor is on a leading edge, selected from the table above by clicking the mouse on the TRUE condition at 199 µs. Scrolling to the first FALSE after this at 282 µs gives that logic 1 is held for 83 µs, which is quite accurate. The trace was run under animation; the chosen options were: continual mode, rolling trace, maximum number of records = 1000.

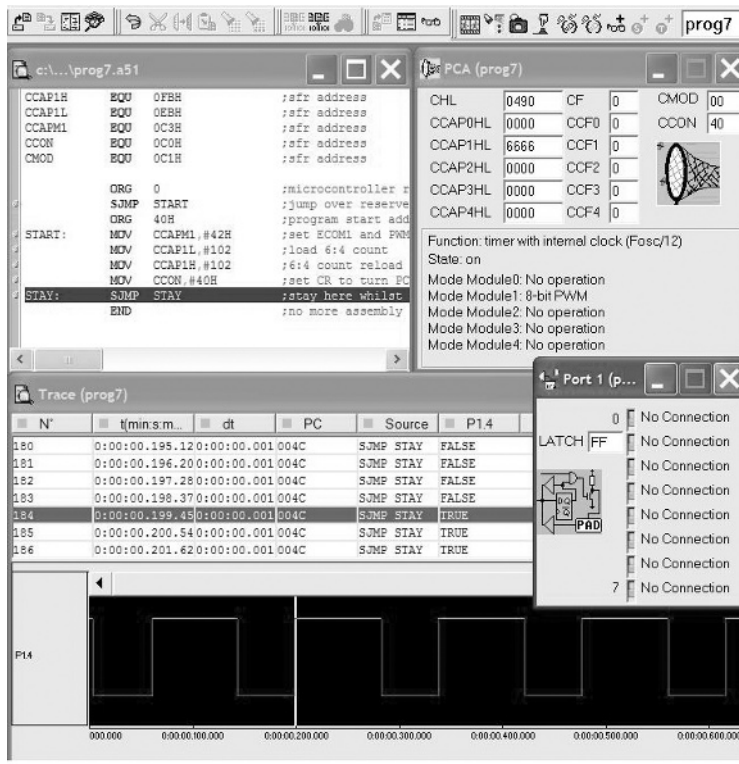


Figure 4.20 Simulation display showing the use of the Trace window

Exercise 4.7

A P89C664 micrcontroller has an oscillator frequency of 11.0592 MHz. Write a C program that will cause the PCA to generate a PWM signal from pin 4 of port 1 with a Mark (logic 1) Space (logic 0) ratio of 2 to 8. The PCA timer clock frequency should be one-sixth of the microcontroller oscillator frequency.

4.8 Watchdog timer

If it is allowed to run unchecked the watchdog timer automatically causes a main system reset. This is particularly useful if the system is to operate in a noisy environment where interference may cause the microcontroller-based system to malfunction.

In the P89C66x microcontroller family the watchdog timer is available by using module 4 of the PCA. Figure 4.21 outlines its configuration. The Watchdog is enabled with WDTE = 1 in the CMOD register. Once turned on the PCA Timer/Counter increments up from zero. If the 16-bit CH, CL register ever matches the 16-bit CCAP4H CCAP4L register setting then a main system reset occurs and the operating program runs from the beginning.

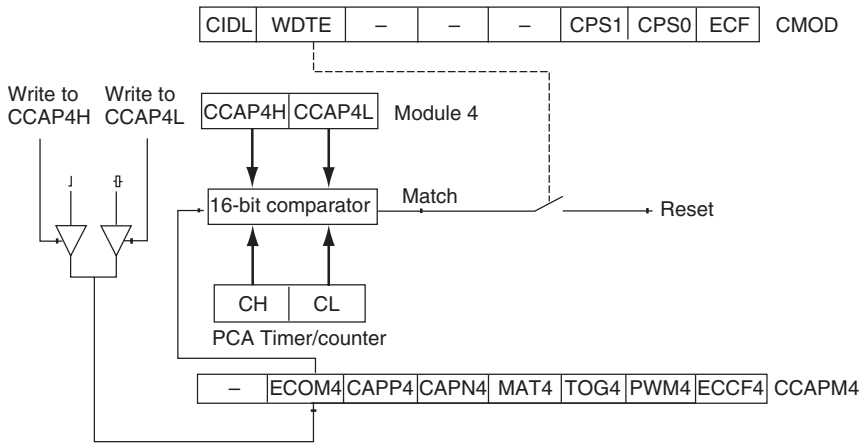


Figure 4.21 Use of module 4 of the PCA to facilitate the watchdog timer

To prevent the match CCAP4H, CCAP4L are periodically changed preventing CH, CL from reaching a matching value. This is done by periodically changing the value of the high byte CCAP4H to the current value of the PCA timer (CH) + (FFH).

Example 4.10
Write a simple assembly language program that toggles pin 7 of port 1 and also incorporates the use of the watchdog timer.

Solution

```
CCAPM4   EQU   0C6H          ; sfr address
CCAP4L   EQU   0EEH          ; sfr address
CCAP4H   EQU   0FEH          ; sfr address
CCON     EQU   0C0H          ; sfr address
CMOD     EQU   0C1H          ; sfr address
CL       EQU   0E9H          ; sfr address
CH       EQU   0F9H          ; sfr address
```

```
        ORG     0               ; reset address
        SJMP    START           ; jump over reserved area
        ORG     40H             ; program start address
START:
        ORL     CCON,#40H       ; turn PCA timer on
        ORL     CMOD,#40H       ; enable watchdog WDTE = 1
        MOV     CCAPM4,#48H     ; CCAPM4 to watchdog
        MOV     CCAP4L,#0FFH    ; maximum initially into
        MOV     CCAP4H,#0FFH    ; compare
STAY:
        CPL     P1.7            ; complement port 1 pin 7
        MOV     R0,#99H         ; register decrement
LOOP:   DJNZ    R0,LOOP         ; delay
        ACALL   WATCHDOG        ; call watchdog refresh
        SJMP    STAY            ; repeat complement
WATCHDOG:
        MOV     CCAP4L,#0       ; make CCAP4L zero
        MOV     A,CH            ; get current CH value
        ADD     A,#0FFH         ; add FF to CH value and
        MOV     CCAP4H,A        ; put answer in CCAP4H
        RET                     ; return from refresh
        END                     ; no more assembly language
```

Simulation

Figure 4.22 shows the simulation response. Single stepping down to the line after MOV CCAPM4,#48H which selects the watchdog timer gives the response shown in the compare capture register window. Moving the mouse cursor over the line 4 0000H watchdog timer and clicking the left mouse button will cause all the module bits to change to mode 4.

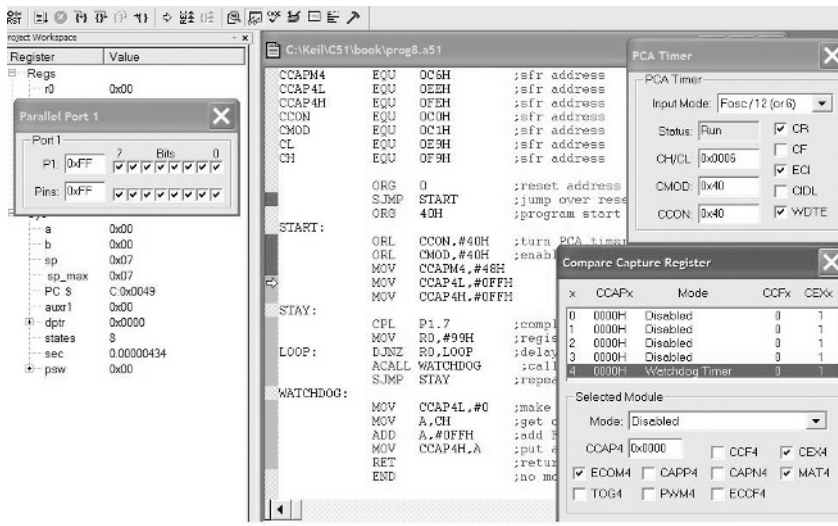


Figure 4.22 Simulation display showing details of the compare capture register

Single stepping further will cause the program to become stuck in the delay loop. Going to the Command window at the bottom left of the PC screen and, at the chevron (>), typing R0 = 1 and then pressing the enter key, would allow a check on the register window.

If single stepping is continued the watchdog subroutine can be gone through and the contents of the CH/CL and CCAP4 windows checked. Clicking the left mouse button on simulation run should cause pin 7 in the port 1 window to flicker on and off.

Coming out of the simulation and commenting out the line that calls up the watchdog routine (by putting a semicolon at the beginning of it) produces the result as shown in Figure 4.23. Recompiling the program and running the simulation should cause the Command window to report that the system is being continually reset. See Figure 4.24.

```
;               ACALL  WATCHDOG        ;call watchdog refresh
```

Figure 4.23 Use of a semicolon to 'comment out' a program line

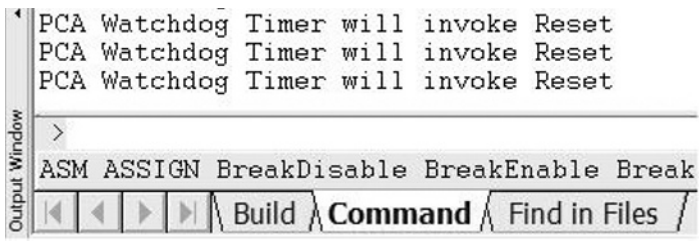


Figure 4.24 Command window indication that the system is being continually reset

Exercise 4.8
Write a simple C program that toggles pin 7 of port 1 and also incorporates the use of the watchdog timer.

4.9 Universal asynchronous receive transmit (UART)

UARTs are used for serial communication between systems; they can be either half duplex (send or receive) or full duplex (send and receive at the same time). Also known as an RS232 connection the microcontroller UART can provide the connection with a PC or another microcontroller-based system. Figure 4.25 illustrates possible connection arrangements. In a minimum connection there could be only two transmission lines, transmit (Tx) and receive (Rx) as shown in Figure 4.26. The data is conveyed as a bit stream, either transmit or receive, and the speed is defined by the baud rate i.e. the bits per second.

The UART has four modes of operation, 0 to 3. Modes 0 and 2 have fixed baud rates, mode 0 is one-sixth of the oscillator frequency, and mode 2 is 1/16

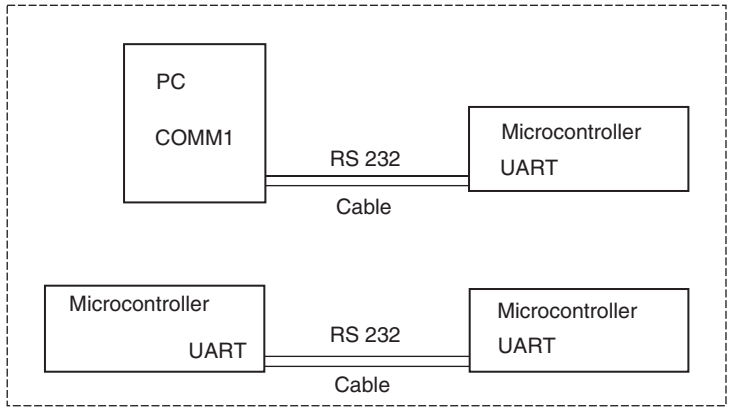


Figure 4.25 Use of RS232 interface between PC and microcontroller or between two microcontrollers

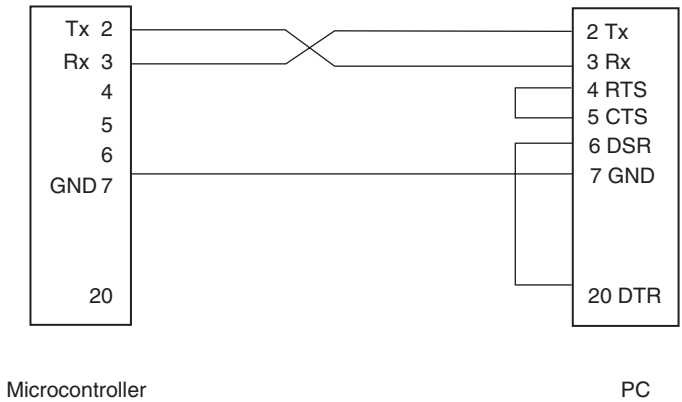


Figure 4.26 RS 232 transmit (Tx) and receive (Rx) connections between a PC and microcontroller

or 1/32 of the oscillator frequency. For modes 1 and 3 the baud rate can be selected, a typical range is:

75, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400

Modes 0 and 1 are used for connection between two devices. Modes 2 and 3 are used for master slave multiprocessor systems, in principle there could be one master microcontroller and up to 255 slave microcontrollers.

In mode 1 ten bits are used to specify an RS232 frame consisting of 1 start bit (logic 0), 8 data bits and 1 stop bit (logic 1). For example the ASCII bit pattern 0100 0001 (hex 41) represents the character A and is transmitted as shown in Figure 4.27; least significant bit (LSB) first.

The baud rate is defined by using one of the onboard timers usually timer 1 in mode 2 and for the P89C66x microcontroller, timer 2.

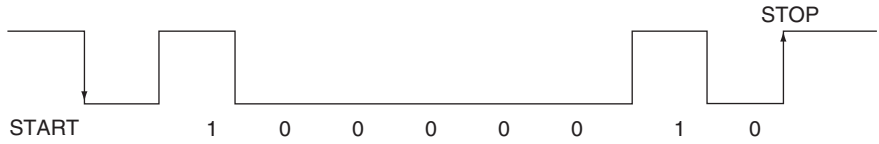


Figure 4.27 ASCII bit pattern 01000001 (41H) for character 'A'

BAUD RATE USING TIMER 1

In timer mode 2, timer high byte (TH1) is used as the automatic reload register. The equation is:

$$\text{TH1} = 256 - \frac{2^{\text{SMOD}} \times \text{Oscillator frequency}}{192 \times \text{Baud rate}}$$

SMOD = 0 for UART modes 0, 1, 3
SMOD = 0 for UART mode 2 bit speed (oscillator frequency)/32
SMOD = 1 for UART mode 2 bit speed (oscillator frequency)/16

As an indication of possible values to be loaded into TH1, using the above equation, and assuming an oscillator frequency of 11.0592 MHz and UART mode 1, the TH1 values for baud rates of 38400, 19200, 9600, 300, 150 would be:

38400	TH1 = 254.5	
19200	TH1 = 253	FDH
9600	TH1 = 250	FAH
300	TH1 = 64	40H
150	TH1 = −128	

The content of TH1 should be a positive, whole number. The baud rate of 38 400 gave 254.5 and the baud rate of 150 gave −128, so there are limits using timer 1. Of course one solution would be to change the oscillator frequency, another solution is to use timer 2. If you are using timer 2 for something else and insist on very low baud rates it is possible to use timer 1 in mode 1 (16 bit) with the interrupt enabled and the interrupt routine doing a software reload.

SERIAL PORT CONTROL (SCON) REGISTER

SCON

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0	SM1	
0	0	mode 0
0	1	mode 1
1	0	mode 2
1	1	mode 3

SM2 when £ = 1, enables multiprocessor operation in modes 2 and 3
REN when £ = 1, enables serial reception
TB8 Used in multiprocessor operation in modes 2 and 3
RB8 Used in multiprocessor operation in modes 2 and 3
TI Transmit interrupt flag, set when byte transmission is completed. Must be cleared by software
RI Receive interrupt flag, set when a byte in the serial buffer (SBUF) is ready for retrieval. Must be cleared by software

Practical tip. Although not apparent in the device data sheet and not required in the simulation, transmission start-up problems may occur in the hardware if TI is not initially set to 1 by the software.

Example 4.11
A P89C664 microcontroller has an oscillator frequency of 11.0592 MHz. Using timer 1 and configuring the UART in mode 1 write an assembly language program that transmits the ASCII character A at a baud rate of 9600.

Solution

Program

```
S0CON    EQU    98H          ; SCON sfr address
S0BUF    EQU    99H          ; SBUF sfr address

         ORG    0            ; reset address
         SJMP   START        ; jump over reserved area
         ORG    40H          ; program start address
START:   MOV    S0CON,#42H   ; serial mode 1, TI set
         MOV    TMOD,#20H    ; timer 1 mode 2
         MOV    TH1,#0FAH    ; baudrate 9600
         MOV    TL1,#0FAH    ; TL1 also initially set
         SETB   TR1          ; turn timer 1 on

AGAIN:   MOV    S0BUF,#'A'   ; ASCII of A into S0BUF
HERE:    JNB    TI,HERE      ; stay here till TI set
         CLR    TI           ; clear TI
         SJMP   AGAIN        ; repeat
         END                 ; end of assembly language
```

Simulation
The simulation response is shown in Figure 4.28. If a breakpoint is inserted at CLR TI and the program kept running to this point, a string of the character A

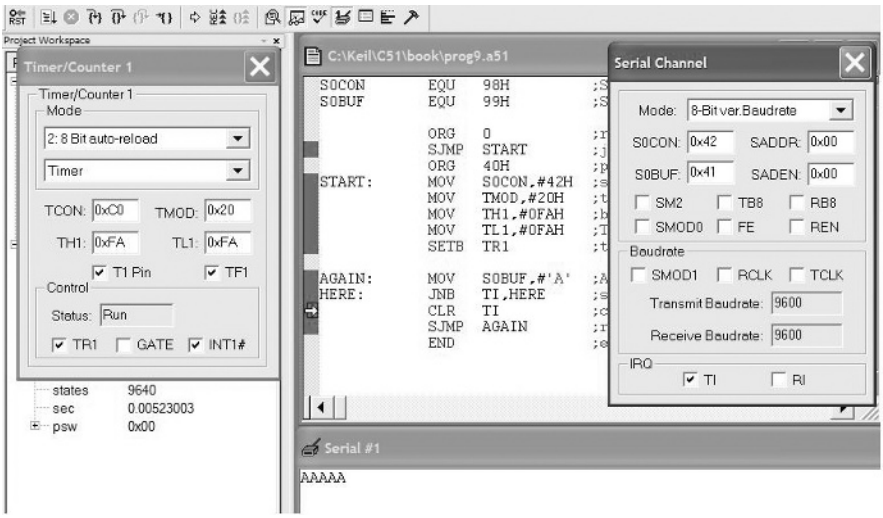


Figure 4.28 Simulation display showing a string of character 'A' in the serial window, using timer 1

will be generated in the serial window. Serial Window # 1 is obtained from View on the top menu bar. Text is cleared from this window by moving the cursor over the text and then right clicking the mouse.

Exercise 4.9
A P89C664 microcontroller has an oscillator frequency of 11.0592 MHz. Using timer 1 and configuring the UART in mode 1 write a C program that transmits the ASCII character A at a baud rate of 9600.

BAUD RATE USING TIMER 2

The equation is:

$$RCAP2 = 65536 - \frac{\text{Oscillator frequency}}{16 \times \text{Baud rate}}$$

Example 4.12
Assuming an oscillator frequency of 11.0592 MHz and UART mode 1 determine the timer 2. Capture values for baud rates of 38400, 19200, 9600, 300, 150.

Solution

38400	RCAP2 = 65518 = FFEEH	RCAP2H = 0FFH	RCAP2L = 0EEH
19200	RCAP2 = 65500 = FFDEH	RCAP2H = 0FFH	RCAP2L = 0DEH
9600	RCAP2 = 65464 = FFB8H	RCAP2H = 0FFH	RCAP2L = 0B8H
300	RCAP2 = 63232 = F700H	RCAP2H = 0F7H	RCAP2L = 00H
150	RCAP2 = 60928 = EE00H	RCAP2H = 0EEH	RCAP2L = 00H

Example 4.13

A P89C664 microcontroller has an oscillator frequency of 11.0592 MHz. Using timer 2 and configuring the UART in mode 1 write an assembly language program that transmits the ASCII character A at a baud rate of 9600.

Solution

```
S0CON    EQU    98H          ; SCON sfr address
S0BUF    EQU    99H          ; SBUF sfr address
RCAP2L   EQU    0CAH         ; sfr address
RCAP2H   EQU    0CBH         ; sfr address
T2CON    EQU    0C8H         ; Timer 2 control sfr address

         ORG    0            ; reset address
         SJMP   START        ; jump over reserved area
         ORG    40H          ; program start address
START:   MOV    S0CON,#42H   ; serial mode 1, TI set
         MOV    RCAP2H,#0FFH ; baudrate 9600
         MOV    RCAP2L,#0B8H ;
         ORL    T2CON,#34H   ; turn Timer 2 on

AGAIN:   MOV    S0BUF,#'A'   ; ASCII of A into S0BUF
HERE:    JNB    TI,HERE      ; stay here till TI set
         CLR    TI           ; clear TI
         SJMP   AGAIN        ; repeat
         END                 ; end of assembly language
```

Simulation

The simulation response is shown in Figure 4.29. Again the simulation is run to a breakpoint at the instruction CLR TI.

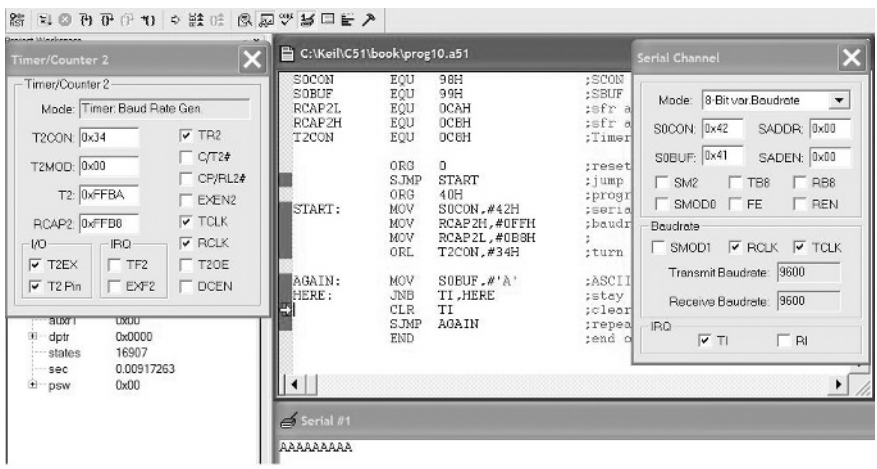


Figure 4.29 Simulation display showing a string of character 'A' in the serial window, using timer 2

Exercise 4.10
A P89C664 microcontroller has an oscillator frequency of 11.0592 MHz. Using timer 2 and configuring the UART in mode 1 write a C program that transmits the ASCII character A at a baud rate of 9600.

SENDING A LINE OF TEXT

Example 4.14
Write an assembly language program that repeatedly sends the line of text 'Roses are red'.

Solution

```
S0BUF    EQU    99H               ; SBUF sfr address
S0CON    EQU    98H               ; SCON sfr address

         ORG    0                 ; reset address
         SJMP   START             ; jump over reserved area
         ORG    40H               ; program start address
START:   MOV    S0CON,#42H        ; serial mode 1, TI set
         MOV    TMOD,#20H         ; timer 1 mode 2
         MOV    TH1,#0FAH         ; baudrate 9600
         MOV    TL1,#0FAH         ; TL1 also initially set
         SETB   TR1               ; turn timer 1 on

TEXT:    MOV    DPTR,#MSG1        ; Data Pointer to message address
NEXTCH:  MOV    A,#0              ; zero the previous character
         MOVC   A,@A + DPTR       ; character into A
         CJNE   A,#7EH,TRXCH      ; checking end of message, ~= 7EH
         MOV    A,#0DH            ; carriage return = 0DH
         ACALL  SEND              ; call up send routine
         MOV    A,#0AH            ; line feed = 0AH
         ACALL  SEND              ; call up send routine
         SJMP   TEXT              ; repeat line of text
TRXCH:   ACALL  SEND              ; send text character
         INC    DPTR              ; increment data pointer
         SJMP   NEXTCH            ; prepare to send next character
SEND:    JNB    TI,SEND           ; check SBUF clear to send
         CLR    TI                ; clear TI
         MOV    S0BUF,A           ; send contents of A
         RET                      ; return from subroutine
MSG1:    DB     'Roses are red ~' ; text message
         END                      ; no more assembly language
```

Simulation
With a breakpoint at SJMP TEXT, the simulation response is as shown in Figure 4.30.

Exercise 4.11
Write a C program that repeatedly sends the line of text 'Ashes to ashes, dust to dust'.

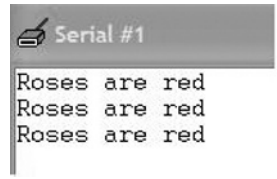


Figure 4.30 Simulation display showing a text message displayed in the serial window

RECEIVING A CHARACTER

Example 4.15
Write an assembly language program that receives a character into the serial buffer (S0BUF) of the UART and writes the hex value of the character onto port 1. The character capture process is to start as the result of the UART being interrupted. The UART should be configured as mode 1.

Solution
The TI bit in the serial control (S0CON) can be left at its default value of zero but the receive bit (REN) must be set.

S0CON

SM0	SM1	SM2	REN	TB8	RB8	TI	RI
0	1	0	1	0	0	0	0

Thus S0CON = 50H. When the character byte is received RI will set but this must be cleared by the program to enable other receive interrupts to occur. In the simulation we should expect RB8 to set when the character byte has been received. TB8 and RB8 are mainly used in modes 2 and 3 but in mode 1 RB8 is set by the stop bit, which is the last bit of the 10-bit mode 1 data frame.

The interrupt enable (IEN0) register bits must be set.

IEN0

EA	EC	ES1	ES0	ET1	EX1	ET0	EX0
1	0	0	1	0	0	0	0

Thus IEN0 = 90H. The UART interrupt vector address is at 0023H.

Program

```
S0BUF    EQU    99H          ; SBUF sfr address
S0CON    EQU    98H          ; SCON sfr address
IEN0     EQU    0A8H         ; interrupt sfr address
         ORG    0            ; reset address
         SJMP   START        ; jump over reserved area
         ORG    23H          ; UART interrupt address
```

```
        SJMP    RXBUF       ; jump to interrupt routine
        ORG     40H         ; program start address

START:  MOV     SOCON,#50H  ; mode 1, REN enabled
        MOV     TH1,#0FAH   ; 9600 baud
        MOV     TMOD,#20H   ; timer 1 mode 2
        MOV     IEN0,#90H   ; UART interrupt enabled
        SETB    TR1         ; turn timer 1 on
STAY:   SJMP    STAY        ; stay here, wait for interrupt

RXBUF:  JNB     RI,RXBUF    ; check for received byte
        CLR     RI          ; clear RI
        MOV     A,S0BUF     ; move character from buffer to A
        MOV     P1,A        ; hex value onto port 1
        RETI                ; return from interrupt
        END                 ; no more assembly language
```

Simulation

The simulation response is shown in Figure 4.31. Running the simulation would cause little activity until the character byte is entered. The character can be entered at the command prompt at the bottom of the screen.

> sin = 'A', see Figure 4.32.

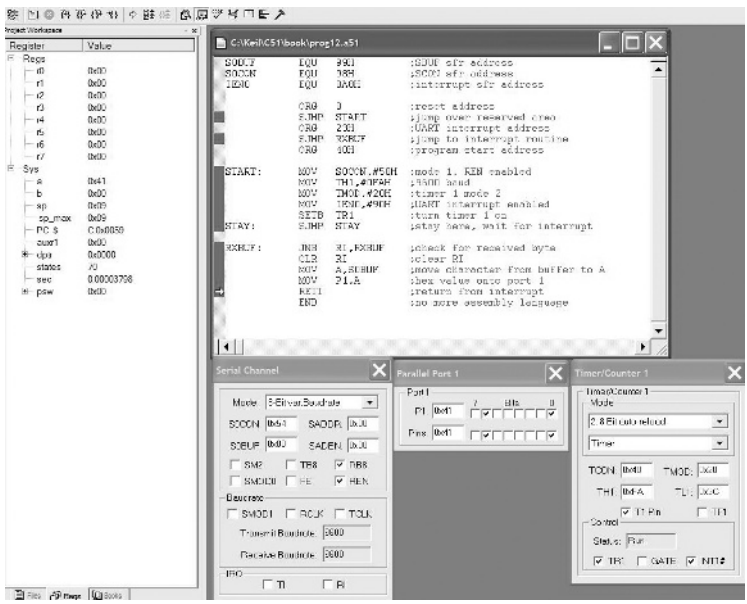


Figure 4.31 Simulation display for a program to write a hex character on to port 1 as a result of the UART being interrupted

Figure 4.32 Entering the character byte in the Command window

Exercise 4.12
Write a C program that receives a character into the serial buffer (S0BUF) of the UART and writes the hex value of the character onto port 1. The character capture process is to start as the result of the UART being interrupted. The UART should be configured as mode 1.

4.10 Inter integrated circuit (IIC or I^2C)

Commonly referred to as I squared C, the I^2C bus or IIC bus was originally developed as a control bus for linking microcontroller and peripheral ICs for Philips consumer products on a PCB. The simplicity of a 2-wire bus that combined both address and data bus functions was quickly adopted in such diverse applications as:

- telecommunications
- automotive dashboards
- energy management systems
- test and measurement products
- medical equipment
- point of sales terminals
- security systems.

This patented Philips method of serial data transmission uses two lines, one for a serial clock (SCL) and the other for serial data (SDA). The SDA line is bi-directional, i.e. data can go up it or down it. There are various microcontrollers in the Philips Semiconductors family having I^2C capability; the programs in this text are based on the P80C554. The P89C66x pin 7 on port 1 is the SDA line and that pin 6 is the SCL line. When used for I^2C these two pins configure as open drain and it is necessary to have a pull-up resistor from each pin to 5 V, the P89C664 board used 3.3 k resistors.

There are other microcontrollers belonging to the 80C51 family that have I^2C SFRs, for example the P87LPC764. The P87LPC764 microcontroller is designed to send and receive I^2C data as bits and requires extra programming to group them into bytes. The class of microcontrollers to which the P89C664 belongs sends and receives the data as bytes.

All I^2C slave ICs have the ability to return Acknowledge (A) signals (active low) back to the Master IC sending to them.

I^2C has three modes; all 8-bit bidirectional, dependent on devices and clock speeds:

- standard mode (up to 100 kbps (k bits per second))
- fast mode (up to 400 kbps)
- high-speed mode (up to 3.4 Mbps).

There are four modes of operation:

1. master transmitter
2. master receiver
3. slave receiver
4. slave transmitter.

The master is the microcontroller while the slave is the device addressed by the microcontroller. In a system having two microcontrollers the master at a particular time is the one issuing the commands.

Philips manufacture a whole range of I^2C slave devices, a small range is:

- memories, EEPROM and static RAM
- data converters
- LCD drivers
- I/O ports
- clock/calendars
- DTMF/tone generators
- TV decoders
- teletext decoders
- video processors
- audio processors.

The arrangement for connecting a microcontroller to I^2C devices is shown in Figure 4.33. For the purposes of explaining the I^2C bus the devices shown in Figure 4.33 will be used, i.e. the P89C664 microcontroller and the slave devices, PCF 8582, a 256-byte CMOS EEPROM and PCF 8591, an 8-bit A/D and D/A converter, which has 4 ADC channels.

The P89C664 has four I^2C SFRs; they are:

S1CON Serial 1 Control
S1STA Serial 1 Status
S1DAT Serial 1 Data
S1ADR Serial 1 Address

The S1ADR is used only in the slave transmitter mode, which will not be covered in this chapter. The S1DAT register is used to transmit data in much the same way as the SBUF register was used for the RS232 UART operation. You may recall that when data was transmitted in the UART SBUF the program waited, via a continuous loop, until the transmit interrupt (TI) bit was set. A similar method is used for the I^2C bus, i.e. data is

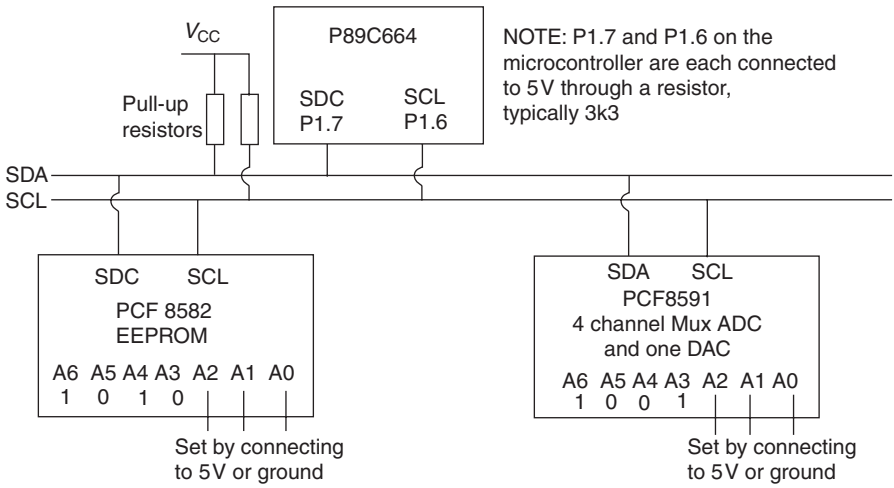


Figure 4.33 P89C664 microcontroller connected to slave devices via the I²C bus

put into S1DAT and then the program continually loops until the serial interrupt (SI) bit is set.

Only the five most significant bits of the S1STA register are used, and these are used to give information on the success or failure of each part of the I²C serial transmission. Data transfer on the I²C bus takes the form as shown in Figure 4.34.

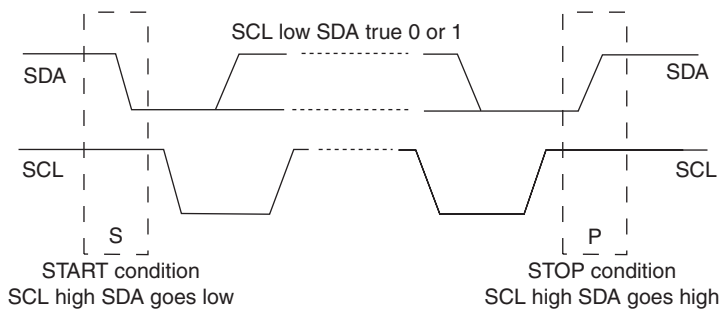


Figure 4.34 Data transfer on the I²C bus

The S1CON register is very important; it controls each part of the serial transmission and it is worth looking at this register in some detail.

S1CON

CR2	ENS1	STA	STO	SI	AA	CR1	CR0

CR2, 1, 0 are used to define the serial clock speed as shown in Table 4.4.

The P89C664 experiment board had a value of $f_{OSC} = 11.0592\,\text{MHz}$.

ENS1	enable serial 1 when = 1 enables the I^2C
STA	start, used to generate starts, refer to I^2C protocol later
STO	stop, used to generate stops, refer to I^2C protocol later
SI	serial interrupt
AA	assert acknowledge

These last four control bits are very important in the use of the I^2C bus.

Table 4.4 Serial clock rates

CR2	CR1	CR0	f_{osc} divided by
0	0	0	128
0	0	1	112
0	1	0	96
0	1	1	80
1	0	0	480
1	0	1	60
1	1	0	30
1	1	1	48 × (256 − reload value Timer 1)

USE OF THE SI BIT

SI is usually cleared by software; SI is set when a function completes.

Example 4.16
To illustrate the effect of the SI bit consider the following assembly language programs:

```
; program to send a Start (STA = 1)
         SETB   STA          ; set STA = 1
         CLR    SI           ; clear SI
LOOP:    JNB    SI,LOOP      ; continually loop until SI = 1, then
                             ; STA will = 1
; program to send a Stop (STO = 1)
         SETB   STO          ; set STO = 1
         CLR    SI           ; clear SI
LOOP:    JNB    SI,LOOP      ; continually loop until SI = 1, then
                             ; STO will = 1
; program to send data e.g.#04h
         CLR    STA          ; clear start (STA = 0)
         MOV    S1DAT,#04H   ; put hex 4 into s1dat
         CLR    SI           ; clear SI
LOOP:    JNB    SI,LOOP      ; continually loop until SI = 1, then
                             ; s1dat will contain #04H
; program to set up transmission speed, send a stop, send a start,
; clear SI
```

```
; this is an example of the start of a typical IIC program
; the 89C664 fOSC is 11.0592 MHz, divide by 112 for clock cycle time
          MOV     S1CON,#45H  ; set ENS1, set AA, clear SI
          SETB    STA         ; set STArt
          CLR     SI          ; clear SI
LOOP:     JNB     SI,LOOP     ; continually loop till SI is set
; program to send Assert Acknowledge (AA)
; Note that AA is active low; a clear AA is sent to assert the
; acknowledge
          CLR     AA          ; clear AA to assert acknowledge
          CLR     SI          ; clear SI
LOOP:     JNB     SI,LOOP     ; continually loop till SI is set
```

USING THE PCF8582 EEPROM

This 256-byte memory device can be written to by the microcontroller and retain the information even though the power is turned off. It is specified to have data retention for at least 10 years and would be very useful for battery-powered remote sensing devices and many more applications. It is in an 8-pin package as shown by Figure 4.35.

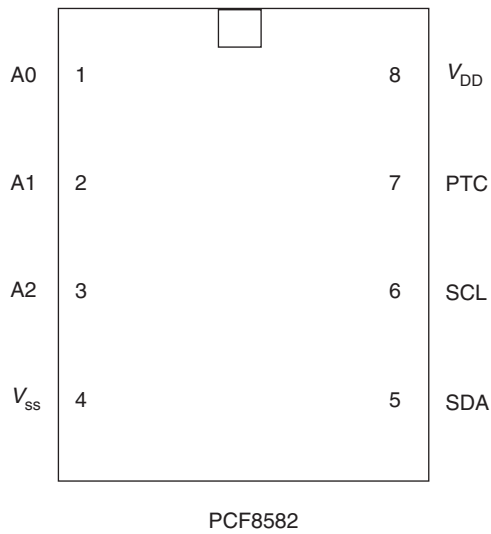


Figure 4.35 Pin-out details of the PCF 8582 EEPROM

Pins 1, 2 and 3 are hardwired by the engineer to define the slave address of the device. With three definable address pins it means that up to 2^3 or eight 8582 EEPROMs can be addressed on the I^2C bus. The first four address bits are internally configured, for the PCF8582 the address table is:

1	0	1	0	A2	A1	A0

In the following examples A2, A1 and A0 are all connected to ground or 0 V. The full slave address is:

1010000X

where the least significant bit X is 0 for write data and 1 for read data. The hex address for writing a byte is A0 and for reading a byte the address is A1.

pin 4 is ground (0 V)
pin 5 is serial data SDA
pin 6 is serial clock SCL
pin 7 is programming timing control (an output and may be left unconnected)
pin 8 is the 5 V DC power supply

The I^2C bus has only two lines and therefore there is a certain protocol to go through in order to store or retrieve data. Each device data sheet has a block diagram to explain the necessary protocol and for the PCF8582 this is shown by:

S	SLAVE ADDRESS	0	A	WORD ADDRESS	A	DATA BYTE	A	P

Send a start
Send the slave address + 0 for write
Send word address
Send data byte
Send a stop

Example 4.17
Write an assembly language program to write a byte to the PCF8582 EEPROM chip.

Solution

```
S1CON    EQU    0D8H           ; sfr address
S1DAT    EQU    0DAH           ; sfr address
         ORG    0              ; reset address
         SJMP   START          ; jump over reserved area
         ORG    40H            ; program start address
START:   MOV    S1CON,#45H     ; set clock speed, set ENS1,set AA,
                               ; clear SI
         SETB   S1CON.5        ; set STA
LOOP1:   JNB    S1CON.3,LOOP1  ; continually loop till SI = 1
         CLR    S1CON.5        ; clear start, donot want repeated
                               ; start
         MOV    S1DAT,#0A0H    ; send eeprom address + write to
                S1DAT          ;
         CLR    S1CON.3        ; clear SI
```

```
LOOP2:   JNB    S1CON.3,LOOP2   ; continually loop till SI = 1
         MOV    S1DAT,#04H      ; send eeprom internal address to
                S1DAT           ;
         CLR    S1CON.3         ; clear SI
LOOP3:   JNB    S1CON.3,LOOP3   ; continually loop till SI = 1
         MOV    S1DAT,#66H      ; send data byte to s1dat
         CLR    S1CON.3         ; clear SI
LOOP4:   JNB    S1CON.3,LOOP4   ; continually loop till SI = 1
         SETB   S1CON.4         ; set STO = 1
         CLR    S1CON.3         ; clear SI
LOOP5:   JNB    S1CON.3,LOOP5 ; loop till SI = 1 and stop is set
AGAIN:   SJMP   AGAIN           ; forever loop, a way of stopping
         END                    ; end of assembly language
```

Simulation

The simulation response is shown in Figure 4.36. If the breakpoints are set as shown and the program run to each one, the response may be checked on the I^2C Interface window. Figure 4.36 shows that the Master has just transmitted the slave address + Write information. The Status shows 0x20 which agrees with the data sheet information. The communication information is available by clicking the left mouse button on the I^2C Communication button.

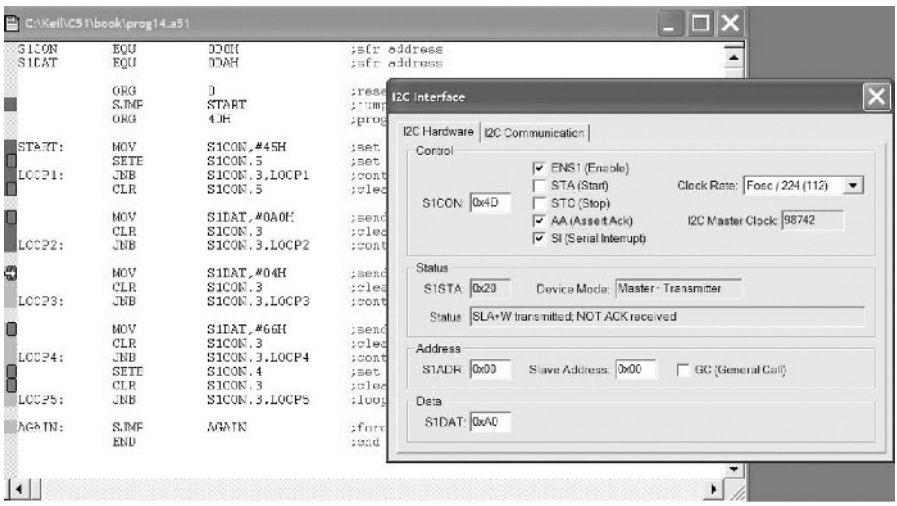


Figure 4.36 Simulation display showing the I^2C interface hardware window

The result is shown in Figure 4.37. Figure 4.37 shows that the microcontroller is in Master Transmitter mode. It shows the slave address as 50! (means 50 hex). This is A0 with the least significant bit removed and the remaining bits

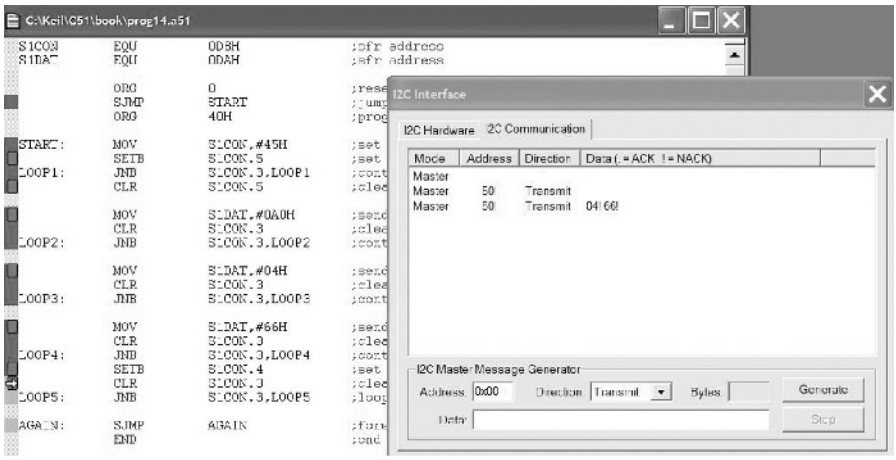


Figure 4.37 Simulation display showing I^2C interface communications window

A6, A5, A4, A3, A2, A1, A0 = 101 0000 = 50 hex. The address within the EEPROM is 04hex and 66hex is the byte written into this address.

Exercise 4.13
Write a C program to write a byte to a PCF8582 EEPROM chip.

TO READ A BYTE OF DATA

The protocol block diagram is given below.
The last acknowledge after Data Byte and before P is sent by the microcontroller master.

S	SLAVE ADDRESS	0	A	WORD ADDRESS	A	S	SLAVE ADDRESS	1	A	DATA BYTE	A	P

The first slave address has 0 at the end for Write Word Address. The second slave address has 1 at the end for Read Data Byte.

Send a start
Send the slave address +0 for Write
Send word address
Send a repeated start
Send the slave address +1 for Read
byte transfers to S1DAT
Master generates acknowledge
Send a Stop

Example 4.18
Write an assembly language program to read a byte from PCF8582 EEPROM chip.

Solution

```
; program to read a byte
S1CON    EQU    0D8H
S1DAT    EQU    0DAH
         ORG    0              ; reset start address
         SJMP   START          ; jump over reserved address space
         ORG    40H            ; program start address
START:   MOV    S1CON,#45H     ; set speed, ENS1, set AA, clr SI
         SETB   S1CON.5        ; set STArt
LOOP1:   JNB    S1CON.3,LOOP1  ; wait till complete
         CLR    S1CON.5        ; ensure no repeated start
         MOV    S1DAT,#0A0H    ; write to slave address
         CLR    S1CON.3        ; clear SI
LOOP2:   JNB    S1CON.3,LOOP2  ; wait till complete
         MOV    S1DAT,#04H     ; data byte stored at address 04h
         CLR    S1CON.3
LOOP3:   JNB    S1CON.3,LOOP3  ; wait till complete
         SETB   S1CON.5        ; generate a STArt
         CLR    S1CON.3
LOOP4:   JNB    S1CON,LOOP4    ; wait till start is complete
         CLR    S1CON.5        ; ensure no repeated start
         MOV    S1DAT,#0A1H    ; send slave address to bus + Read
         CLR    S1CON.3
LOOP5:   JNB    S1CON.3,LOOP5  ; wait till complete
         CLR    S1CON.2        ; master sends acknowledge, recall
                               ; acknowledge
         CLR    S1CON.3        ; is active low, clr sends
                               ; acknowledge
LOOP6:   JNB    S1CON.3,LOOP6  ; wait till sent
         SETB   S1CON.4        ; microcontroller master generates
                               ; a stop
         CLR    S1CON.3
LOOP7:   JNB    S1CON.3,LOOP7  ; wait till stop is sent
AGAIN:   SJMP   AGAIN          ; forever loop, a way of stopping
         END                   ; end of assembly language
```

Exercise 4.14
Write a C program to read a byte from a PCF8582 EEPROM chip.

Summary

The P89C66x microcontroller:

- is a member of the 80C51 family with enhanced speed compared to the conventional 80C51 device;
- uses Flash Code memory with four 8-bit ports and an onboard clock oscillator;

- has three 16-bit timers, timer 0, 1 and 2. Timers 0 and 1 are virtually the same and can be configured into one of four possible modes, mode 0, 1, 2 and 3. Timer 2 has three operating modes i.e. Capture, 16-bit auto-reload and baud rate generator mode;
- has nine interrupts;
- allows negative-edge transitions or levels to generate external interrupts with specific interrupt vector address locations;
- can operate with specified interrupt priority levels;
- has a PCA consisting of a 16-bit timer and five 16-bit Capture/Compare modules. Each of the latter can be utilised in one of seven different configurations. The PCA can be used to provide a PWM signal;
- has a watchdog timer, which is available using module 4 of the PCA. The watchdog timer if allowed to run unchecked will cause a system reset;
- has a UART facility used for serial communication. The UART has four modes of operation, modes 0,1, 2 and 3;
- has an I^2C interface for linking the device with I^2C compatible peripherals using a 2-wire bus for address and data communication.

5

Low Pin Count (LPC) Devices

5.1 Introduction

The P8xLPCxxx family of microcontrollers are designed for low pin count applications with high functionality and a wide range of performance. There are two types of LPC devices, namely P87LPC76x with EPROM/OTP code memory ranging from 1 KB to 4 KB and P89LPC9xx with 4 KB to 8 KB of flash memory. At the moment there are several different LPC microcontrollers, namely P87LPC760, P87LPC761, P87LPC762, P87LPC764, P87LPC767, P87LPC768 and P87LPC769, all with EPROM/OTP code memory and P89LPC921, P89LPC922, P89LPC930, P89LPC931 and P89LPC932 with flash code memory. Table 5.1 shows some of the characteristics of eight of these devices.

Table 5.1 shows that the code memory for the P87LPC76x devices ranges from 1 KB to 4 KB, while the RAM memory is 128 bytes for all devices in the range. The I/O pin count for this group varies from 12 pins for the P87LPC760 device, 14 pins for the P87LPC761 and 18 pins for the rest of P87LPC76x series.

All the LPC range incorporates I^2C and UART serial interfaces and a Watchdog (Wd) timer. The P87LPC767, P87LPC768 and P87LPC769 devices all contain analog to digital converter (ADC) circuitry onboard. The P87LPC768 has pulse width modulation (PWM) while the P87LPC769 has digital to analog converter (DAC) circuitry onboard.

The P89LPC932 has 8 KB of Flash code memory, 128 bytes of RAM data memory and special features such as the capture/compare unit (CCU) and serial peripheral interface (SPI).

Information on details such as PWM, CCU and SPI can be found in Appendix F, which is for the 89LPC932 device but contains data relevant to the features of the other devices mentioned above.

In this chapter we shall concentrate on two of the above devices, namely P87LPC769 and P89LPC932, and use some application examples to show the working of some of the features in each device. The examples and exercises use C language programming. It is left as an exercise for the reader if assembly

Table 5.1 Characteristics of 87LPC76x and 89LPC9xx devices

Part number	Memory		Timer/counters				I/O pins	Serial interfaces	Special features	A/D bits/ch
	Flash	EPROM	RAM	PWM	CCU	Wd				
87LPC760		1 k	128	N	N	Y	12	I^2C, UART		
87LPC761		2 k	128	N	N	Y	14	I^2C, UART		
87LPC762		2 k	128	N	N	Y	18	I^2C, UART		
87LPC764		4 k	128	N	N	Y	18	I^2C, UART		
87LPC767		4 k	128	N	N	Y	18	I^2C, UART	ADC	8/4
87LPC768		4 k	128	Y	N	Y	18	I^2C, UART	ADC PWM	8/4
87LPC769		4 k	128	N	N	Y	18	I^2C, UART	ADC DAC	8/4
89LPC932	8 k	512 (EEPROM)	128	Y	Y	Y	26	I^2C, UART SPI	Analog Com.	

language programming is required. Simulation has been used for the first example in this chapter only. Details of the simulation software, suitable for the LPC devices and available for downloading to the user's PC, can be found in Chapter 3. Again, it is left as an exercise for the reader if simulation is required in order to follow the remaining examples and exercises in this chapter.

5.2 P87LPC769

The 87LPC769 is a 20-pin single-chip microcontroller designed for LPC applications. A member of the Philips LPC family, the 87LPC769 offers programmable oscillator configurations for high and low speed crystals or RC operation, wide operating voltage range, programmable port output configurations, selectable Schmitt trigger inputs, LED drive outputs and a built-in watchdog timer. The 87LPC769 is based on an accelerated 80C51 processor architecture that executes instructions at twice the rate of standard 80C51 devices.

Features of the LPC769 include:

- four-channel multiplexed 8-bit ADC;
- two DAC outputs;
- 4 KB EPROM code memory;
- 128 byte RAM data memory;
- 32-byte customer code EPROM;
- two 16-bit counter/timers;
- two analog comparators;
- full duplex UART;
- I^2C communication port;
- eight keypad interrupt inputs, plus two additional external interrupt inputs;
- four interrupt priority levels;

- watchdog timer with separate on-chip oscillator, requiring no external components. The watchdog timeout time is selectable from 8 values;
- oscillator fail detect. The watchdog timer has a separate fully on-chip oscillator, allowing it to perform an oscillator fail detect function;
- configurable on-chip oscillator with frequency range and RC oscillator options (selected by user programmed EPROM bits). The RC oscillator option allows operation with no external oscillator components;
- programmable port output configuration options: quasi-bidirectional, open drain, push-pull, input only;
- selectable schmitt trigger port inputs;
- LED drive capability (20 mA) on all port pins;
- 15 I/O pins minimum. Up to 18 I/O pins using on-chip oscillator and reset options;
- only power and ground connections are required to operate the 87LPC769 when fully on-chip oscillator and reset options are selected.

Figure 5.1 shows the block diagram of the device while Figure 5.2 shows the pin configuration diagram. In the section that follows we shall concentrate on analog functions and use ADC, DAC and comparator examples to show some of the applications of the LPC769 device.

5.3 Analog functions

The pins that are used for analog functions must have their digital outputs (except for DAC output pins) and their digital inputs disabled. Digital outputs are disabled by putting the port output into the input only (high impedance) mode. This is done by configuring the appropriate bits of the port output mode registers PxM1 and PxM2 as shown in Table 5.2.

Table 5.2 87LPC769 port output mode configurations

PxM1y	PxM2y	Port output mode
0	0	Quasi-bidirectional
0	1	Push-pull
1	0	Input only (high impedance)
1	1	Open drain

Digital inputs of port 0 are disabled through use of port 0 digital input disable (PT0AD) register, by setting the corresponding bit in the PT0AD.

ANALOG TO DIGITAL CONVERTER

The 87LPC769 incorporates a four channel, 8-bit ADC. The A/D inputs are alternate functions on four port 0 pins. These are P0.3 as AD0, P0.4 as AD1,

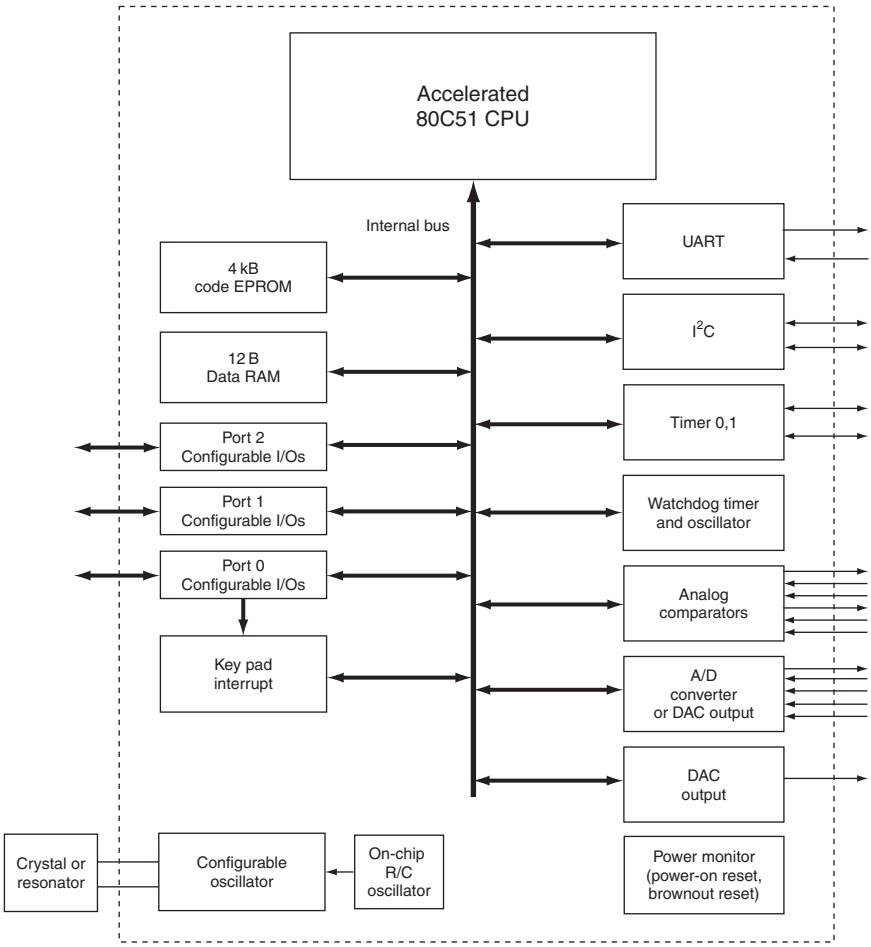


Figure 5.1 Block diagram of the 87LPC769 microcontroller

P0.5 as AD2 and P0.6 as AD3. The A/D power supply and references are shared with the processor power pins, V_{DD} and V_{SS}. The ADC circuitry consists of a 4-input analog multiplexer and an 8-bit successive approximation ADC.

The SFR ADCON controls the ADC. Details of ADCON are:

7	6	5	4	3	2	1	0
ENADC	ENDAC1	ENDAC0	ADCI	ADCS	RCCLK	AADR1	AADR0

where the bit functions are:

ENADC When ENADC = 1, the A/D is enabled and conversions may take place. Must be set 10 µs before a conversion is started. ENADC cannot be cleared while ADCS or ADCI is 1.

ENDAC1 When ENDAC1 = 1, DAC1 is enabled to provide an analog output voltage.

ENDAC0 When ENDAC0 = 1, DAC0 is enabled to provide an analog output voltage.

ADCI A/D conversion complete/interrupt flag. This flag is set when an A/D conversion is completed.

ADCS A/D start. Setting this bit by software starts the conversion of the selected A/D input. ADCS remains set while the A/D conversion is in progress and is cleared automatically upon completion.

RCCLK When RCCLK = 0, the CPU clock is used as the A/D clock. When RCCLK = 1, the internal RC oscillator is used as the A/D clock.

AADR1, 0 These bits select the A/D channel to be converted:

AADR1	AADR0	A/D input selected
0	0	AD0 (P0.3)
0	1	AD1 (P0.4)
1	0	AD2 (P0.5)
1	1	AD3 (P0.6)

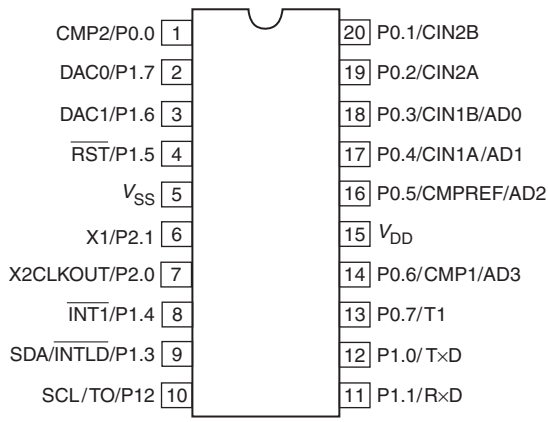


Figure 5.2 Pin configuration for the 87LPC769 microcontroller

A/D conversion

The A/D must be enabled by setting the ENADC bit at least 10 µs before a conversion is started to allow time for the A/D to stabilise. Prior to the beginning of an A/D conversion, one analog input pin must be selected for conversion via the AADR1 and AADR0 bits. These bits cannot be changed while the A/D is performing a conversion. Setting the ADCS bit, which remains set while the conversion is in progress starts an A/D conversion.

When the conversion is complete, the ADCS bit is cleared and the ADCI bit is set. When ADCI is set, it will generate an interrupt if the interrupt system is enabled; the A/D interrupt is enabled (via the EAD bit in the IE1 register), and the A/D interrupt is the highest priority pending interrupt. When a conversion is complete, the result is contained in the register DAC0. This value will not change until another conversion is started. Before another A/D conversion may be started, the ADCI bit must be cleared by software. The A/D channel selection may be changed by the same instruction that sets ADCS to start a new conversion, but not by the same instruction that clears ADCI.

A/D timing

The A/D may be clocked in one of two ways. The default is to use the CPU clock as the A/D clock source. When used in this manner the A/D completes a conversion in 31 machine cycles. The A/D may also be clocked by the on-chip RC oscillator, even if the RC oscillator is not used as the CPU clock. This is accomplished by setting the RCCLK bit in ADCON.

Example 5.1

Use of ADC

```
/*************************************************************
* Chapter 5                                                  *
* ADC application of 87LPC769                                *
* April 2003                                                 *
*                                                            *
* This program reads AD0, AD1, AD2 and AD3 one at            *
* a time and stores it in the ACC for other uses             *
************************************************************* /
#include <REG769.H>
/*************************************************************
* START of the PROGRAM                                       *
************************************************************* /
void main (void) {
  unsigned char channel;
/*************************************************************
* Disable P0, ADC pins digital Outputs and Inputs            *
* AD3 = P0.6, AD2 = P0.5, AD1 = P0.4, AD0 = P0.3             *
************************************************************* /
  P0M2& = ~ 0x78;                    /*Set Pins for Input Only*/
  P0M1| = 0x78;                           /*P0M2 = 0& P0M1 = 1 */
  PT0AD = 0x78;                     /*Disable Digital Inputs */
/*************************************************************
* Enable the A/D Converter and use the CPU clock             *
* as the A/D clock.                                          *
************************************************************* /
```

```
  ENADC = 1;                          /*enable ADC, 10 µs before conv.*/
  RCCLK = 0;                                        /*use CPU clock*/
  channel = 0;                              /*set to the first channel*/
/************************************************************
* Perform conversions forever.                                *
************************************************************ /
  while (1) {
/************************************************************
* Update the channel number and store it in ADCON.            *
************************************************************ /
    channel = (channel + 1)%4;                    /*update channel*/
    ADCON& = ~ 0x03;                             /*clear channel no*/
    ADCON | = channel;                          /*set the channel no*/
/************************************************************
* Start a conversion and wait for it to complete.             *
************************************************************ /
    ADCI = 0;                                  /*Clear conversion flag*/
    ADCS = 1;                                      /*Start conversion */
    while (ADCI == 0);                        /* Wait for conversion end */
    ACC = DAC0;                               /* send the results to Acc*/
    ADCI = 0;                                  /*Clear conversion flag*/
  }
}
```

Simulation

The Keil simulation package can be used to demonstrate the simulation of the 87LPC769 microcontroller. The Keil µ Vision2 package is well suited for the analogue functions of the LPC since there is a special ADC window available, which shows all the internal registers related to ADC and DAC conversions such as ADCON, DAC0 and channel select. The window also shows the bits related to the ADC/DAC functions such as ADCS and ENDADC. The values of the channels could also be set using analogue input channels windows. The ADC window is shown in Figure 5.3.

As described in Chapter 3, the simulation begins by going to Project on the top menu bar and selecting New Project. For the chip vendor and particular device, the P87LPC769 should be chosen from the Philips directory. Details are shown in Figure 5.4.

Figure 5.4 illustrates that information about the device, such as memory capability (128 bytes of RAM, 4 KB of on-chip programmable EPROM), etc. is available in the window. The C file should now be added to the project and the simulation started by clicking on the button with the red letter d as shown in Figure 5.5.

The window for the program should appear as shown in Figure 5.6.

Next Peripherals is chosen from the top menu bar and then A/D Converter as shown in Figure 5.7.

The ADC window, shown in Figure 5.3, should appear, allowing different values to be set for each channel prior to simulation. See Figure 5.8.

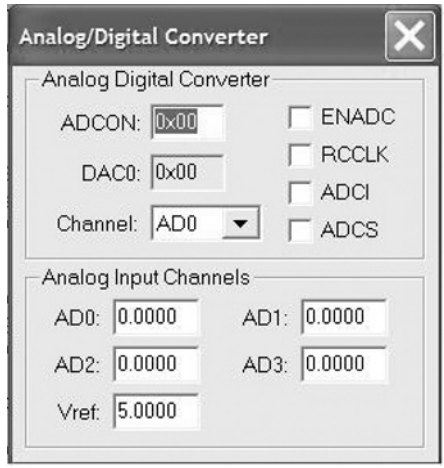


Figure 5.3 Keil μ Vision2 analog/digital converter window

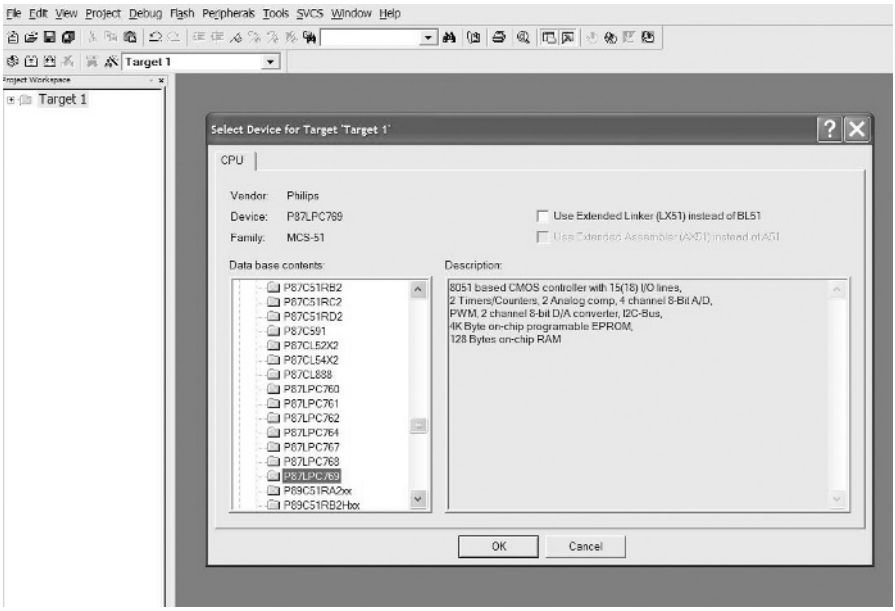


Figure 5.4 Selection of chip vendor and device type

Figure 5.5 Icon to start/stop debug session

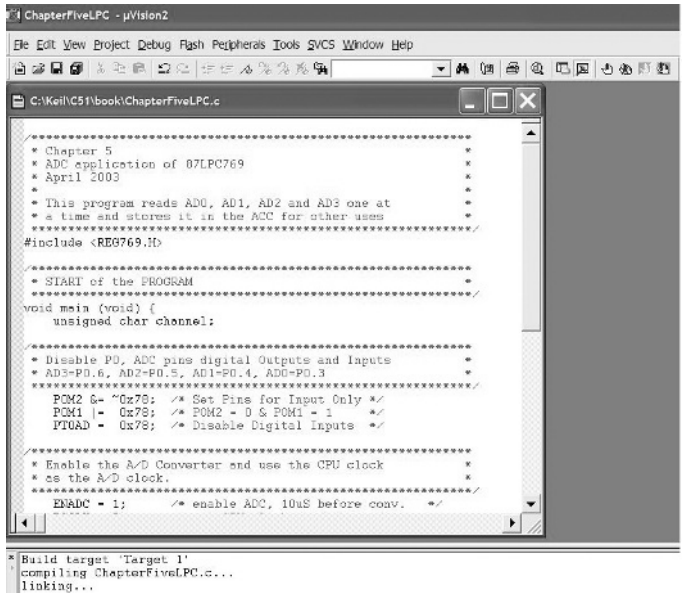


Figure 5.6 Simulation window for analog/digital converter program

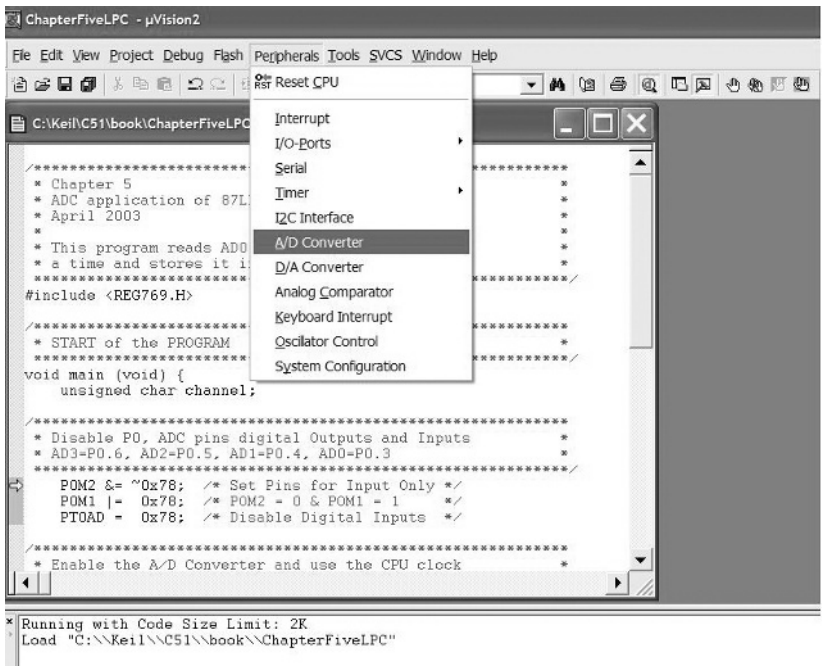


Figure 5.7 Selection of analog/digital converter window

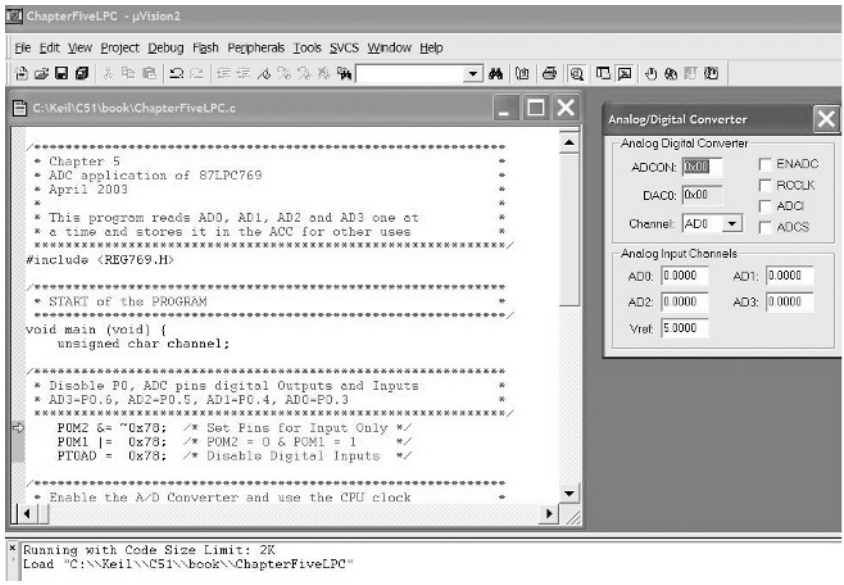


Figure 5.8 Simulation window with the analog/digital converter window added

Placing the cursor in the AD1 window allows the value to be set to 2.5 (this would be volts). Similarly the value for AD2 may be set to, say, 4.0 and AD3 to 5.0. The effect of these changes is shown in Figure 5.9.

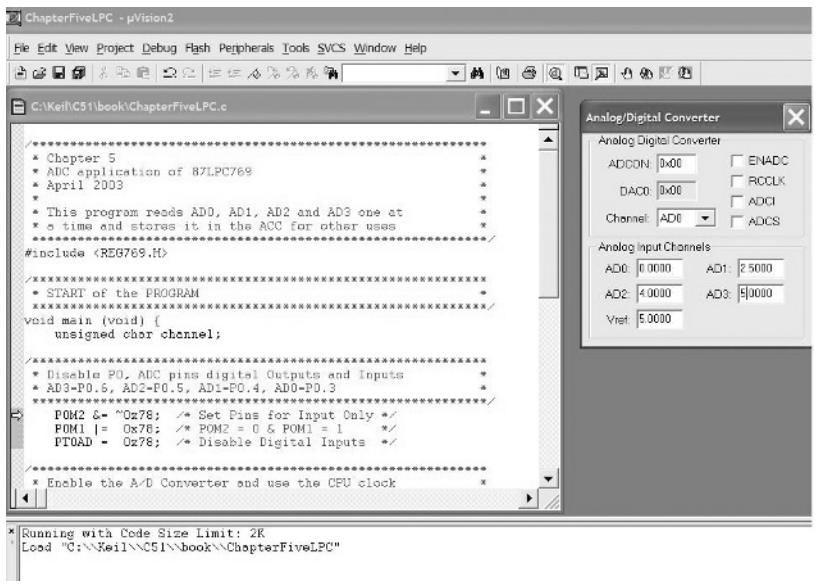


Figure 5.9 Changing the values of the analog input channels

Single stepping through the program may be achieved by pressing the appropriate icon from the debug menu (see Figure 5.10), or pressing key F11. Details regarding the debug menu and other icons used for debug operations are described in Chapter 3.

Figure 5.10 Icon for single stepping the program

When single stepping through the program it can be seen that values in the ADCON (A/D control register), channel (selected channel), ADCI (A/D conversion complete/interrupt flag bit), ADCS (A/D start bit) and DAC0 (A/D 0 value) windows change as the steps progress. It should be noted that ENADC (enable A/D) would be set only once, at the beginning of the program. The effect is shown in Figure 5.11.

Figure 5.11 shows the values for each channel; for example channel AD0 set originally to '0.0' V, is indicated by DAC0 of 0x00 while that of AD3 set to '5.0' indicates a value of 0xFF. Note that AD1 was set to '2.5' V, which is indicated by a DAC0 value of 0x80 (128).

Exercise 5.1
An 87LPC769 microcontroller is to be used to read an analog input ranging from 0 V to 5 V on the ADC0, and display the results on two seven-segment displays (which are driven by display drivers), with the following interface:

P0.3 – P0.0 connected to A, B, C and D of Driver 1 (least significant digit 0.0 – 0.9)
P1.3 – P1.0 connected to A, B, C and D of Driver 2 (most significant digit 0 – 5)

Write a 'C' program to do this.

DAC OUTPUTS

The 87LPC769 provides a two channel, 8-bit DAC function. DAC0 is also a part of the ADC and it should not be enabled while the A/D is active. Digital outputs must be disabled on the DAC output pins while the corresponding DAC is enabled, as described under Analog Functions. The DACs use the power supply as the references, V_{DD} as the upper reference and V_{SS} as the lower reference. The DAC output is generated by a tap from a resistor ladder and is not buffered. The maximum resistance to V_{DD} or V_{SS} from a DAC output is 10 kΩ. Care must be taken with the loading of the DAC outputs in order to avoid distortion of the output voltage. DAC accuracy is affected by noise, generated on-chip and elsewhere in the application. Since the 87LPC769 power pins are used for the DAC references, the power supply also affects the accuracy of the DAC outputs.

Figure 5.11 Analog/digital converter window showing values for channels AD0, AD1, AD2 and AD3

Example 5.2

Use of DAC

```
/*************************************************************
* Chapter 5                                                  *
* DAC application of 87LPC769                                *
* April 2003                                                 *
*                                                            *
* This program generates a sawtooth waveform on              *
* DAC1 and DAC0 of the P87LPC769 microcontroller             *
************************************************************* /
#include <REG769.H>
/*************************************************************
```

```
* START of the PROGRAM                                                *
************************************************************** */
void main (void) {
  unsigned int i;
/*************************************************************
* Disable P1, DAC pins digital Outputs and set the              *
* DAC1 = P1.6, DAC0 = P1.7 to Input Only (Hi z)                 *
************************************************************** */
  P1M2& = ~0xC0;                          /*Set Pins for Input Only*/
  P1M1| = 0xC0;                               /*P1M2 = 0 & P1M1 = 1*/
/*************************************************************
* Disable the A/D Converter because of DAC0                     *
* AND Enable the D/A Converters                                 *
************************************************************** */
  ADCI = 0;                       /*Clear A/D conversion complete flag*/
  ADCS = 0;                          /*Clear A/D conversion start flag*/
  ENADC = 0;                               /*Disable the A/D Converter*/
  ENDAC0 = 1;                                          /*Enable DAC0*/
  ENDAC1 = 1;                                          /*Enable DAC1*/
/*************************************************************
* Create a sawtooth waveform on DAC0 and the opposite           *
* sawtooth waveform on DAC1.                                    *
************************************************************** */
  while (1) {
    for (i = 0; i < 255; i++){
      DAC0 = i;
      DAC1 = 0xFF - i;
    }
  }
}
```

Exercise 5.2
An 87LPC769 microcontroller is to be used to generate a triangular waveform on DAC0. Write a 'C' program to do this.

5.4 Analog comparators

The P87LPC769 provides two analog comparators. Because of the input and output options the comparators can be used in a number of different configurations. Comparator operation is such that the output is a logical one (which may be read in a register and/or routed to a pin) when the positive input (one of two selectable pins) is greater than the negative input (selectable from a pin or an internal reference voltage). Otherwise the output is a zero. Each comparator may be configured to cause an interrupt when the output value changes.

COMPARATOR CONFIGURATION

There are two comparator control registers, CMP1 for comparator 1 and CMP2 for comparator 2. These control registers are identical and their bits are as shown:

7	6	5	4	3	2	1	0
–	–	CEn	CPn	CNn	OEn	COn	CMFn

where the bit functions are:

7, 6	Reserved for future use. Should not be set to 1 by user programs.
CEn	Comparator enable. When set by software, the corresponding comparator function is enabled. Comparator output is stable 10 μs after CEn is first set.
CPn	Comparator positive input select. When 0, CINnA is selected as the positive comparator input. When 1, CINnB is selected as the positive comparator input.
CNn	Comparator negative input select. When 0, the comparator reference pin CMPREF is selected as the negative comparator input. When 1, the internal comparator reference Vref is selected as the negative comparator input.
OEn	Output enable. When 1, the comparator output is connected to the CMPn pin if the comparator is enabled (CEn = 1). This output is asynchronous to the CPU clock.
COn	Comparator output, synchronised to the CPU clock to allow reading by software. Cleared when the comparator is disabled (CEn = 0).
CMFn	Comparator interrupt flag. This bit is set by hardware whenever the comparator output COn changes state. This bit will cause a hardware interrupt if enabled and of sufficient priority. Cleared by software and when the comparator is disabled (CEn = 0).

The overall connections to both comparators are shown in Figure 5.12. There are eight possible configurations for each comparator, as determined by the control bits in the corresponding CMPn register.

Example 5.3

Comparator configuration

```
/*************************************************************
* Chapter 5                                                  *
* Comparator application of 87LPC769                         *
* April 2003                                                 *
*                                                            *
* This program configures CMP1 with CIN1B (P0.3)             *
* as positive input and CMPREF(P0.5) as the                  *
```

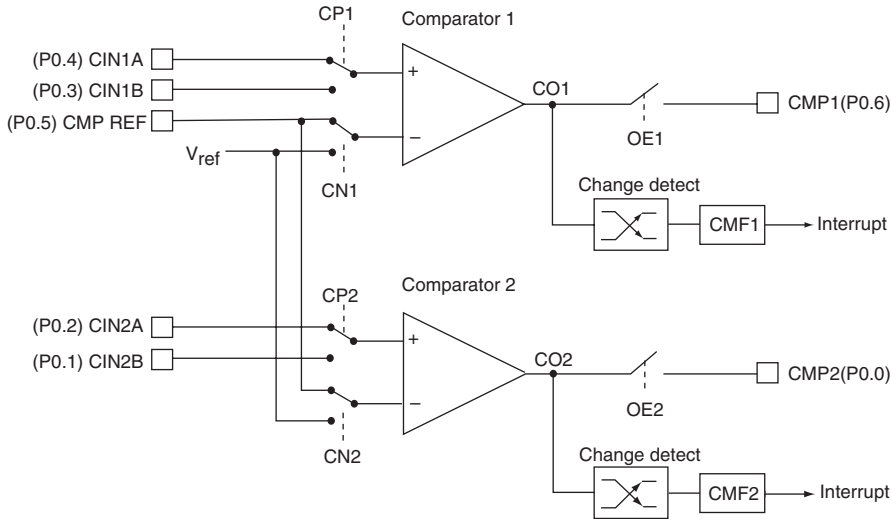


Figure 5.12 Comparator configurations of the 87LPC769 microcontroller

```
* negative input.                                                    *
* CMP2 is configured with internal V_ref (1.28 V) as                 *
* negative input and CIN2A (P0.2) as positive input.                 *
* Both comparator outputs CMP1 (P0.6) and CMP2                       *
*(P0.0) are gated to output pins.                                    *
********************************************************************/
#include <REG769.H>
/*******************************************************************
* START of the PROGRAM                                              *
********************************************************************/
void main (void) {
  unsigned char i;
/*******************************************************************
* Disable P0, digital Outputs and Inputs                            *
* CMPREF = P0.5                                                     *
* CIN1A = P0.4, CIN1B = P0.3, CMP1 = P0.6                           *
* CIN2A = P0.2, CIN2B = P0.1, CMP2 = P0.0                           *
********************************************************************/
  P0M2& =~0x0C;                          /*Set Pins for Input Only*/
  P0M1| = 0x0C;                                /*POM2 = 0&POM1 = 1*/
  PT0AD = 0x0C;                          /*Disable Digital Inputs*/
/*******************************************************************
* Set CIN1B(P0.3) as +ve input, CMPREF as -ve                       *
* input and CMP1 Out(P0.0)                                          *
* -   -  CEn CPn CNn OEn COn CMFn                                   *
* 0   0   1   1   0   1   0   0                                     *
********************************************************************/
CMP1 = 0x34;
/*******************************************************************
```

```
* Set CIN2A(P0.2) as +ve input, V_ref as -ve                    *
* input and CMP2 Out(P0.6)                                      *
* -   -  CEn CPn CNn OEn COn CMFn                               *
* 0   0   1   0   1   1   0   0                                 *
*****************************************************************/
  CMP2 = 0x2C;
/*****************************************************************
* Do nothing delay 10 µs.                                       *
*****************************************************************/
  for (i = 0; i<=10; i++)
          ;
  while (1)                                    /*Loop Forever */
          ;
}
```

Exercise 5.3
Configure CMP1 with CIN1A (P0.4) as a positive input and Vref (1.28 V) as the negative input and CMP2 with internal CMPREF (P0.5) as a negative input and CIN2B (P0.1) as the positive input. Both comparator outputs CMP1 (P0.6) and CMP2 (P0.0) are to be gated to output pins.

5.5 P89LPC932

The P89LPC932 device is based on a high performance processor architecture that executes instructions in two to four clocks, six times the rate of standard 80C51 devices. Many system level functions have been incorporated into the P89LPC932 in order to reduce component count, board space and system cost.

The P89C932 contains many features, some of which are summarised as follows:

- 8 KB Flash code memory with 1 KB erasable sectors and 64-byte erasable page size;
- 256-byte RAM data memory; 512-byte auxiliary on-chip RAM;
- 2-byte customer Data EEPROM on-chip allows serialisation of devices, storage of set-up parameters, etc;
- two 16-bit counter/timers. Each timer may be configured to toggle a port output upon timer overflow or to become a pulse width modulation (PWM) output;
- real-time clock that can also be used as a system timer;
- capture/compare unit (CCU) provides PWM, input capture and output compare functions;
- two analog comparators with selectable inputs and reference source;
- enhanced UART with fractional baud rate generator, break detect, framing error detection, automatic address detection and versatile interrupt capabilities;
- 400 kHz byte-wide I^2C communication port;

- SPI communication port;
- eight keypad interrupt inputs, plus two additional external interrupt inputs;
- four interrupt priority levels;
- watchdog timer with separate on-chip oscillator, requiring no external components. The watchdog time-out time is selectable from 8 values;
- LED drive capability (20 mA) on all port pins. A maximum limit is specified for the entire chip;
- 23 I/O pins minimum (28-pin package). Up to 26 I/O pins while using on-chip oscillator and reset options.

Figure 5.13 shows the block diagram while Figure 5.14 shows the pin configuration diagram for the 89LPC932 device. In the following sections we will concentrate on the serial peripheral interface (SPI), I^2C serial communication and EEPROM functions and use examples to show some of the applications of the device.

5.6 Serial peripheral interface (SPI)

Together with the usual serial communication interfaces, the LPC932 device provides another high-speed serial communication interface, called the SPI interface. SPI is a full-duplex, synchronous communication bus with Master and Slave operation modes. Communication of up to 3 Mbit/s can be supported in either Master or Slave mode.

The SPI interface has four pins: SPICLK, MOSI, MISO and $\overline{SS}$. SPICLK, MOSI and MISO are typically tied together between two or more SPI devices. Data flows from master to slave on the MOSI (master out slave in) pin and flows from slave to master on the MISO (master in slave out) pin. The SPICLK signal is output in the master mode and is input in the slave mode. If the SPI system is disabled, i.e. SPEN (SPCTL.6) = 0 (reset value), these pins are configured for port functions.

$\overline{SS}$ is the optional slave select pin. In a typical configuration, an SPI master asserts one of its port pins to select one SPI device as the current slave. An SPI slave device uses its $\overline{SS}$ pin to determine whether it is selected. The $\overline{SS}$ is ignored if any of the following conditions are true:

- If the SPI system is disabled, i.e. SPEN (SPCTL.6) = 0 (reset value).
- If the SPI is configured as a master, i.e. MSTR (SPCTL.4) = 1, and P2.4 is configured as an output (via the P2M1.4 and P2M2.4 SFR bits).
- If the $\overline{SS}$ pin is ignored, i.e. SSIG (SPCTL.7) bit = 1, this pin is configured for port functions.

Note that even if the SPI is configured as a master (MSTR = 1), it can still be converted to a slave by driving the $\overline{SS}$ pin low (if P2.4 is configured as input and SSIG = 0). Should this happen, the SPIF bit (SPSTAT.7) will be set. Typical connection of a simple Master Slave is shown in Figure 5.15.

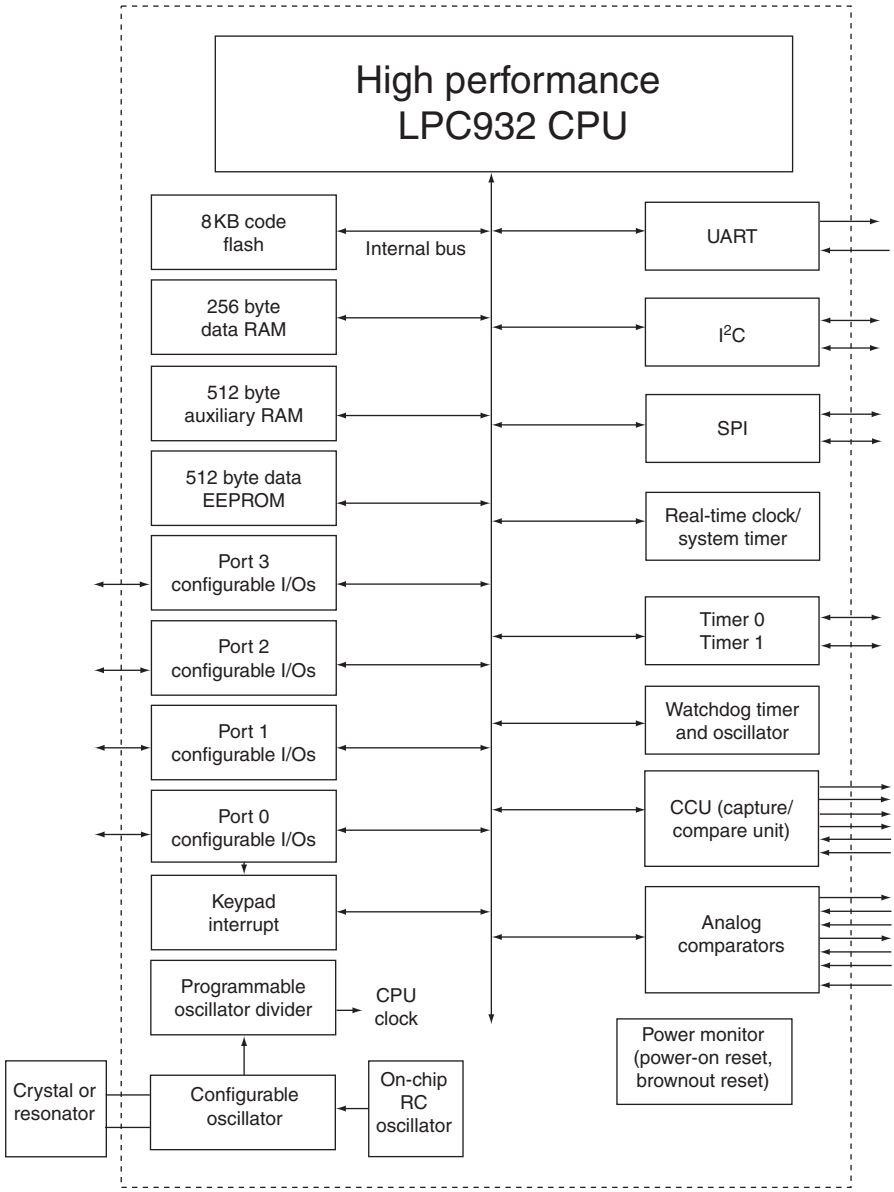


Figure 5.13 Block diagram of the 89LPC932 microcontroller

SPI CONFIGURATION

The LPC932 provides three registers for SPI programming. These are SPiCon-TroL register SPCTL, SPiSTATus register, SPSTAT and SPiDATa register SPDAT. The details of these registers are discussed below.

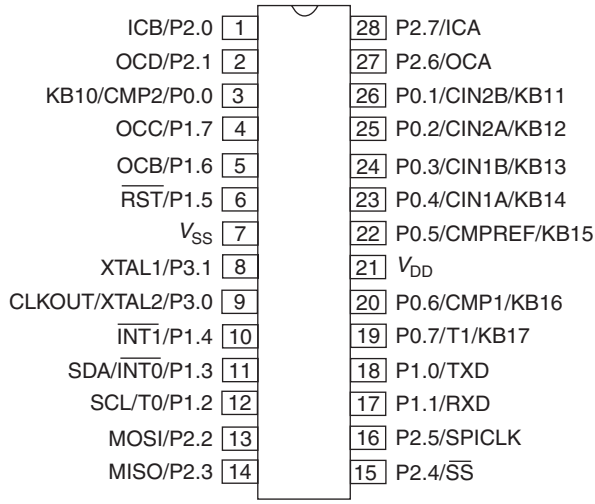


Figure 5.14 Pin configuration for the 89LPC932 microcontroller

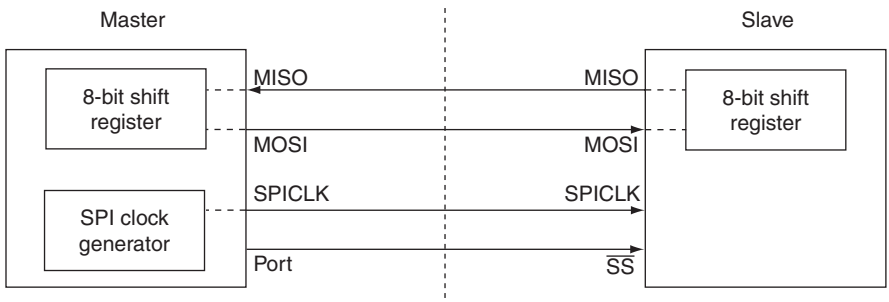


Figure 5.15 Simple master and slave connection

SPCTL register

7	6	5	4	3	2	1	0
SSIG	SPEN	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

where the bit functions are:

SSIG SS IGnore. If set = 1, MSTR (bit 4) decides whether the device is a master or slave. If cleared = 0, the $\overline{SS}$ pin decides whether the device is master or slave. The $\overline{SS}$ pin can be used as a port pin.

SPEN — SPI Enable. If set = 1, the SPI is enabled. If cleared = 0, the SPI is disabled and all SPI pins will be port pins.

DORD — SPI Data ORDer.1: The LSB of the data word is transmitted first.
0: The MSB of the data word is transmitted first.

MSTR — Master/Slave mode Select (see Table 5.3).

CPOL — SPI Clock Polarity. 1: SPICLK is high when idle. The leading edge of SPICLK is the falling edge and the trailing edge is the rising edge.
0: SPICLK is low when idle. The leading edge of SPICLK is the rising edge and the trailing edge is the falling edge.

CPHA — SPI CLock Phase select. 1: Data is driven on the leading edge of SPICLK and is sampled on the trailing edge.
0: Data is driven when $\overline{SS}$ is low (SSIG = 0) and changes on the trailing edge of SPICLK, and is sampled on the leading edge. (*Note:* If SSIG = 1, the operation is not defined.)

SPR1, SPR0 — SPI Clock Rate Select

SPR1	SPR0	SPI clock rate
0	0	CCLK/4
0	1	CCLK/16
1	0	CCLK/64
1	1	CCLK/128

SPSTAT register

7	6	5	4	3	2	1	0
SPIF	WCOL	–	–	–	–	–	–

where the bit functions are:

SPIF — SPI transfer completion Flag. When a serial transfer finishes, the SPIF bit is set and an interrupt is generated if both ESPI (IEN1.3) bit and the EA bit are set. If $\overline{SS}$ is an input and is driven low when SPI is in master mode, and SSIG = 0, this bit will also be set. The SPIF flag is cleared in software by writing '1' to this bit.

WCOL — SPI Write COLlision flag. The WCOL bit is set if the SPI data register is written during a data transfer. The WCOL flag is cleared in software by writing a '1' to this bit. Bits 5–0 reserved for future use. Should not be set to 1 by user program.

Table 5.3 Master and slave selection

SPEN (SPCTL.6)	SSIG (SPCTL.7)	P2M2.4	$\overline{SS}$ Pin	MSTR (SPCTL.4)	Master or Slave Mode	MISO	MOSI	SPICLK	Remarks
0	X	X	P2.4[1]	X	SPI Disabled	P2.3[1]	P2.2[1]	P2.5[1]	SPI disabled. P2.2, P2.3, P2.4, P2.5 are used as port pins.
1	0	X	0	0	Slave	Output	Input	Input	Selected as slave.
1	0	X	1	0	Slave	Hi-Z	Input	Input	Not selected. MISO is high impedance to avoid bus contention.
1	0	0	0	1(−> 0)[2]	Slave	Output	Input	Input	P2.4/$\overline{SS}$ is configured as an input or quasi-bidirectional pin. SSIG is 0. Selected externally as slave if $\overline{SS}$ is selected and is driven low. The MSTR bit will be cleared to '0' when $\overline{SS}$ becomes low.
1	0	0	1	1	Master	Input	Hi-Z	Hi-Z	MOSI and SPICLK are at high impedance to avoid bus contention. Note that the user must pull-up or pull-down SPICLK (depending on CPOL – SPCTL.3) to avoid a floating SPICLK.
1	0	1	X	1	Master	Input	Output	Output	MOSI and SPICLK are push-pull.
1	1	X	P2.4[1]	0	Slave	Output	Input	Input	
1	1	X	P2.4[1]	1	Master	Input	Output	Output	

1. Selected as a port function.
2. The MSTR bit changes to '0' automatically when $\overline{SS}$ becomes low in input mode and SSIG is 0.

SPDAT register

7	6	5	4	3	2	1	0
MSB							LSB

EXAMPLES OF SPI ON 89LPC932

Example 5.4

SPI master

```
/*************************************************************
* Chapter 5                                                  *
* SPI Master application of 89LPC932                         *
* April 2003                                                 *
*                                                            *
* This program writes some data to some slave Devices.       *
* Assumes, P0.0 = Device0.ss pin                             *
* Assumes, P0.1 = Device1.ss pin                             *
* Assumes, P0.2 = Device2.ss pin                             *
* Assumes, P0.3 = Device3.ss pin                             *
************************************************************* /
#include <Reg932.h>
sbit Device0 = P0^0;
sbit Device1 = P0^1;
sbit Device2 = P0^2;
sbit Device3 = P0^3;
/*************************************************************
* Write one byte to the SPI                                  *
************************************************************* /
void SPI_Write(unsigned char dat) {
  SPDAT = dat;                          /*write Data to SPI bus*/
  while ((SPSTAT & 0x80) == 0);              /*wait completion*/
  SPSTAT| = 0x80;                 /*clear SPIF by writing 1 to it*/
}
/*************************************************************
* START of the PROGRAM                                       *
************************************************************* /
void main (void) {
/*************************************************************
* Port 2 to quasi-bidirectional                              *
* MOSI = P2.2, MISO = P2.3, SPICLK = P2.4, SS = P2.5         *
************************************************************* /
    P2M1 = 0xC3;
    P2M2 = 0xC3;
/*************************************************************
* configure SPI                                              *
* SS = 1   MSTR determines device is master/slave            *
```

```
* SPEN = 1  Enable SPI                                                *
* DORD = 1  LSB of the data is transmitted first                       *
* MSTR = 1  device is master                                           *
* CPOL = 1  SPICLK is high when idle. The leading edge of SPICLK is    *
*   falling edge.                                                      *
* CPHA = 1  data is driven on the leading edge of SPICLK and sampled   *
*   on the trailing edge                                               *
* SPR1 = 0  SPI clock rate = CCLK/4                                    *
* SPR0 = 0                                                            *
********************************************************************* /
    SPCTL = 0xFC;
/*************************************************************
* send A, B, C and D to devices continuously                   *
************************************************************* /
while (1) {
  Device0 = 0;                                  /*select Device 0*/
  SPI_Write(0 × 41);                          /*write A to Device 0*/
  Device0 = 1;                                 /*Deselect Device 0*/
  Device1 = 0;                                  /*select Device 1*/
  SPI_Write(0 × 42);                          /*write B to Device 1*/
  Device1 = 1;                                 /*Deselect Device 1*/
  Device2 = 0;                                  /*select Device 2*/
  SPI_Write(0 × 43);                          /*write C to Device 2*/
  Device2 = 1;                                 /*Deselect Device 2*/
  Device3 = 0;                                  /*select Device 3*/
  SPI_Write(0 × 44);                          /*write D to Device 3*/
  Device3 = 1;                                 /*Deselect Device 3*/
  }                                                      /* while( ) */
}                                                         /* main( ) */
```

Exercise 5.4

Write a C program to write text 'Hassan' to a slave Device. MSB is to be transmitted first, and clock rate to be CCLK/128. Assume P0.0 = Device0.ss pin.

Example 5.5

SPI Slave

```
/*************************************************************
* Chapter 5                                                    *
* SPI Slave application of 89LPC932                            *
* April 2003                                                   *
*                                                              *
* This program writes some data to Master Devices.             *
* Note: SS pin (P2^4) must be set to 0 for slave               *
* to be active.                                                *
************************************************************* /
```

```
#include <Reg932.h>
/*************************************************************
* Write one byte to the SPI                                  *
*************************************************************/
void SPI_Write(unsigned char dat)
{
    SPDAT = dat;                          /*write Data to SPI bus*/
    while ((SPSTAT & 0 × 80) == 0);           /* wait completion */
    SPSTAT| = 0x80;               /*clear SPIF by writing 1 to it */
}
/*************************************************************
* START of the PROGRAM                                       *
************************************************************* /
void main (void) {
/*************************************************************
* Port 2 to quasi-bidirectional                              *
* MOSI = P2.2, MISO = P2.3, SPICLK = P2.4, SS = P2.5         *
************************************************************* /
  P2M1 = 0xE3;
  P2M2 = 0xE3;
/*************************************************************
* configure SPI                                              *
* SS    = 0  MSTR determines device is master/slave          *
* SPEN = 1  Enable SPI                                       *
* DORD = 1  LSB of the data is transmitted first             *
* MSTR = 0  device is slave                                  *
* CPOL = 1  SPICLK is high when idle. The leading edge of    *
*             SPICLK is falling edge.                        *
* CPHA = 1  data is driven on the leading edge of SPICLK and *
*           sampled on the trailing edge.                    *
* SPR1 = 0  SPI clock rate = CCLK/4                          *
* SPR0 = 0                                                   *
************************************************************* /
  SPCTL = 0x6C;
  while (1) {
            SPI_Write('V');                 /* write A to Master */
            SPI_Write('W');                 /* write B to Master */
            SPI_Write('X');                 /* write C to Master */
            SPI_Write('Y');                 /* write D to Master */
  }                                                 /* while( ) */
}                                                    /* main( ) */
```

5.7 EEPROM memory

The LPC932 has a 512 byte electrically erasable program read only memory (EEPROM) that can be used to store configuration parameters. The Data EEPROM is SFR based, byte readable, byte writeable and

erasable. The user can read, write and fill the memory via three SFRs and one interrupt:

- Address Register (DEEADR) is used for address bits 7–0 (bit 8 is in the DEECON register).
- Control Register (DEECON) is used for address bit 8, set-up operation mode and status flag bit.
- Data Register (DEEDAT) is used for writing data to, or reading data from, the Data EEPROM.

DEECON

7	6	5	4	3	2	1	0
EEIF	HVERR	ECTL1	ECTL0	–	–	–	EADR8

where the bit functions are:

EEIF Data EEPROM interrupt flag. Set when a read or write finishes. Reset by software.

HVERR Reserved for future use. Should not be set to 1 by user program.

ECLT1, ECTL0 Operation mode selection:

ECLT1	ECLT0	Selection
0	0	Byte read/write mode
1	0	Row (64 bytes) fill
1	1	Block fill (512 bytes)

bit 3 Reserved for future use. Should not be set to 1 by user program

bit 2 Reserved for future use. Should not be set to 1 by user program

bit 1 Reserved for future use. Should not be set to 1 by user program

EADR8 Most significant address (bit 8) of the Data EEPROM.

OPERATION MODES

Byte Mode In this mode data can be read and written to one byte at a time. Data is in the DEEDAT register and the address is in the DEEADR register.

Row Fill In this mode the addressed row (64 bytes, with address DEEADR.5 – 0 ignored) is filled with the DEEDAT pattern. To erase the entire row to 00H or program the entire row to FFH, write 00H or FFH to DEEDAT prior to row fill.

Block Fill In this mode all 512 bytes are filled with the DEEDAT pattern. To erase the block to 00H or program the block to FFH, write 00H or FFH to DEEDAT prior to the block fill. Prior to using this command EADR8 must be set = 1.

In any mode, after the operation finishes, the hardware will set EEIF bit. An interrupt can be enabled via the IEN1.7 bit. If IEN1.7 and the EA bits are set, it will generate an interrupt request. The EEIF bit is cleared by software.

DATA EEPROM READ

To read a byte from EEPROM the steps shown below should be followed:

- Write to DEECON with ECTL1, ECL0 = '00' and correct bit 8 address to EADR8.
- Without writing to the DEEDAT register, write address bits 7–0 to DEEADR.
- If both the EIEE (IEN1.7) bit and the EA (IEN0.7) bit are '1's, wait for the Data EEPROM interrupt then read or poll the EEIF bit until it is set to '1'. If EIEE or EA is '0', the interrupt is disabled, only polling is enabled.
- Read the Data EEPROM data from the DEEDAT SFR.

DATA EEPROM WRITE

To write a byte to EEPROM the steps shown below should be followed:

- Write to DEECON with ECTL1, ECL0 = '00' and correct bit 8 address to EADR8.
- Write the data to the DEEDAT register.
- Write address bits 7–0 to DEEADR.
- If both the EIEE (IEN1.7) bit and the EA (IEN0.7) bit are '1's, wait for the Data EEPROM interrupt then read/poll the EEIF bit until it is set to '1'. If EIEE or EA is '0', the interrupt is disabled and only polling is enabled. When EEIF is '1', the operation is complete and data is written.

DATA EEPROM ROW FILL

To write a row of 64 bytes to the EEPROM the following steps should be taken.

- Write to DEECON with ECTL1, ECTL0 = '10' and correct bit 8 address to EADR8.
- Write the fill pattern to the DEEDAT register.
- Write address bits 7–0 to DEEADR. Note that address bits 5–0 are ignored.
- If both the EIEE (IEN1.7) bit and the EA (IEN0.7) bit are '1's, wait for the Data EEPROM interrupt then read/poll the EEIF (DEECON.7) bit until it is set to '1'. If EIEE or EA is '0', the interrupt is disabled and only polling is enabled. When EEIF is '1', the operation is complete and row is filled with the DEEDAT pattern.

DATA EEPROM BLOCK FILL

To write array of 512 bytes to the EEPROM the following steps should be taken.

- Write to DEECON with ECTL1, ECTL0 = '11'. Set bit EADR8 = 1.
- Write the fill pattern to the DEEDAT register.
- Write any address to DEEADR. Note that the entire address is ignored in a block fill operation.
- If both the EIEE (IEN1.7) bit and the EA (IEN0.7) bit are '1's, wait for the Data EEPROM interrupt then read/poll the EEIF (DEECON.7) bit until it is set to '1'. If EIEE or EA is '0', the interrupt is disabled and only polling is enabled. When EEIF is '1', the operation is complete.

EXAMPLES OF EEPROM USING THE 89LPC932

Example 5.6
EEPROM Write

```
/*************************************************************
* Chapter 5                                                  *
* LPC932 EEPROM byte write applications                      *
* April 2003                                                 *
*                                                            *
* This program writes some data to EEPROM memory             *
************************************************************* */
#include <Reg932.h>
#define dataAddress 4
/*************************************************************
* Write one byte to the EEPROM                               *
************************************************************* */
void writeByte(unsigned int adr, unsigned char dat)
{
  DEECON = 0b00000000;                    /*write byte operation*/
  DEEDAT = dat;                                 /*set write data*/
  DEEADR = (unsigned char) adr;                    /*start write*/
  while((DEECON & 0x80) == 0);            /*wait until completes*/
}
/*************************************************************
* START of the PROGRAM                                       *
************************************************************* */
void main (void) {
  unsigned char myData = 'H';
  writeByte(dataAddress,myData);          /* write H to address 4 */
  while(1);
  }
```

Exercise 5.5
Write a C program to fill a row of 64 bytes with text 'X'on the EEPROM on address 0 onwards.

Exercise 5.6
Write a C program to fill a block of 512 bytes with text 'Y'.

Example 5.7
EEPROM Read

```
/*************************************************************
* Chapter 5                                                  *
* LPC932 EEPROM byte Read applications                       *
* April 2003                                                 *
*                                                            *
* This program writes some data to EEPROM memory             *
* and then reads the same data back                          *
************************************************************* /
#include <Reg932.h>
#define dataAddress 4
/*************************************************************
* Read one byte from the EEPROM                              *
*************************************************************/
unsigned char readByte(unsigned int adr)
{
  DEECON = 0x00;                                  /*address*/
  DEEADR = (unsigned char) adr;                  /*start read*/
  while((DEECON&0x80) == 0);            /*wait until completes */
  return DEEDAT;                           /* return data read */
}
/*************************************************************
* START of the PROGRAM                                       *
*************************************************************/
void main (void) {
  P1 = readByte(dataAddress);         /*read address 4 of EEPROM*/
  while(1);
}
```

Exercise 5.7
Write a C program to write 'Hassan' 'Fred' 'David' to EEPROM in location 10 hex onwards and then read this data back and send then one character at a time to port P0.

Summary

- Low pin count (LPC) devices are available:
  1. in the 87LPC76x range with up to 4 KB EPROM/OTP code memory
  2. in the 89LPC9xx range with up to 8 KB of flash code memory.
- LPC devices incorporate serial interfaces including I^2C, UART and SPI, according to type.
- LPC devices contain special features such as ADC, DAC and PWM, according to type.
- LPC devices can use some pins for analog functions. This permits the use of analog comparators as well as onboard ADC and DAC functions.
- Analog comparators on the 87LPC76x range have up to eight possible configurations.
- The 89LPC932 incorporates a high-speed serial peripheral interface (SPI) and contains three registers for SPI programming.
- The 89LPC932 has 512 bytes of EEPROM that is SFR based, byte readable, byte writeable and erasable. The memory can be read, written to and filled using three SFRs and one interrupt.

6

The XA 16-bit Microcontroller

6.1 Introduction

The eXtended Architecture (XA) microcontroller was introduced by Philips Semiconductors as the 16-bit version of their 8051 microcontroller. This chapter describes the XAG49, which is a Flash version, having 64 KB of program memory. See Figure 6.1. As shown in Figure 6.1 the peripherals include two UARTs, three timers 0, 1 and 2 and a Watchdog timer. More information about the peripherals, SFRs etc., is contained in Appendix F.

It is most important to note that in the XA microcontroller, the Watchdog timer is on by default and if its action is not required then one of the first things to do is to program it off. Having stated this, it should be said that the current version of the Flash programming software WinISP (windows in system programming) disables the Watchdog during programming of the device. Nevertheless the authors think it is better to include the three programming lines that turn off the Watchdog in case later versions or other ISP software do not disable it.

The Watchdog is on by default for other XA family members that use external PROMs for program memory. The three assembly language lines that disable the Watchdog are,

```
MOV.B WDCON, #0
MOV.B WFEED1, #A5H
MOV.B WFEED2, #5AH
```

This will be covered in more detail later.

Notice the .B extension to the MOV instruction; the XA microcontroller is a 16-bit device and this describes the width of some internal registers. Data is moved in bytes (MOV.B) or words (MOV.W); the default is word (MOV).

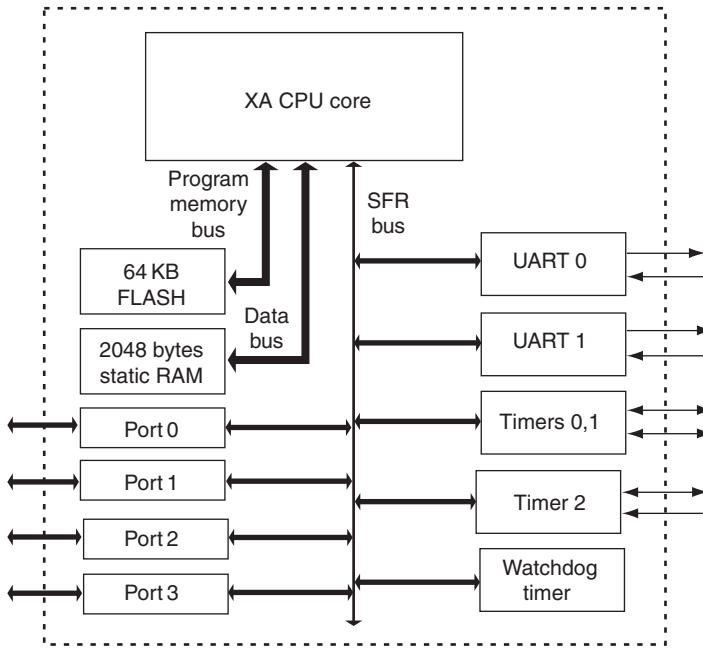


Figure 6.1 XA block diagram

The timers 0, 1 and 2 are better organised than in the 8-bit devices. In the P89C66x and standard 8051 microcontrollers, mode 0 of the timers 0 and 1 is configured as a 13-bit timer so that the device could be compatible with the earlier 8048 microcontroller. In the XA, mode 0 for timers 0 and 1 is a 16-bit auto-reload timer/counter.

The P89C66x family has a timer 2 but it only has three modes of operation i.e. 16-bit auto reload and counting up or down, 16-bit capture and baud rate generator. Timer 2 in the XA has four modes of operation, the first being 16-bit auto-reload counting up while the other three modes are the same as the P89C66x.

Additionally, on the P89C66x the timer clock is fixed at one-sixth of the oscillator frequency. The XA has pre-scalar bits in the system configuration register (SCR) that give three options for timer clock of one quarter, 1/16 or 1/64 of the oscillator frequency.

The beginning of the XA assembly language program is different from that of the 8051. The assembly language programs for the P89C664 commence with:

```
        ORG    0          ; reset address
        SJMP   START      ; jump over area reserved for interrupts
        ORG    40H        ; program start address
START:
```

while assembly language programs for the XA commence with:

```
        ORG     0               ; reset start address
        DW      8F00H,START     ; Define Word hex 8F00
        ORG     120H            ; program start address
START:
```

The XA has many more interrupts than the 8-bit 8051 microcontrollers and this accounts for the higher program start address.

The action of the XA to the line DW 8F00H,START at the reset address is to put the word, 8F00H in the above example, into the 16-bit PSW register, then run the program from the label address (START in this example).

The program is written for compilation and simulation using the evaluation software from www.raisonance.com. This excellent package can be downloaded as an 8051 and XA combination.

Example 6.1

This first program is aimed at toggling pin 7 on port 1 at a frequency of 1 kHz, copying as far as possible the 1 kHz square-wave program written for the 8-bit P89C664 microcontroller in Chapter 4. The XAG49 flash microcontroller in this example has an oscillator frequency of 11.0592 MHz. The timer clock pre-scalar is at its default value of (oscillator frequency)/4.

Solution

Square-wave cycle time = 1/1 kHz = 1/1000 = 1 ms
Time delay required of a square wave = half the cycle time = 0.5 ms

Timer 0 clock = (micro clock)/4 = 11.0592 MHz/4 = 2.7648 MHz
Timer 0 clock cycle time = 1/2.7648 MHz = 361.69 ns

Delay count = (delay time)/(timer clock cycle time)
= 0.5 ms/361.69 ns = 1382 (to nearest whole number)

Mode 1 timer base number = 65535 − delay count
= 65535 − 1382
= 64153 decimal
= FA99 hex

Program

```
$INCLUDE(REGXAG49.INC)          ; list of sfr addresses
        ORG     0               ; reset address
        DW      8F00H,START     ; define word hex8F00
        ORG     120H            ; program start address
START:  MOV.B   WDCON,#0        ; watchdog control off
        MOV.B   WFEED1,#0A5H    ; watchdog feed1
```

```
        MOV.B   WFEED2,#5AH     ; watchdog feed2
        MOV.B   TMOD,#01H       ; Timer 0 into mode 1
AGAIN:  SETB    P1.7            ; pin 7 port 1 to logic 1
        CALL    DELAY           ; call delay routine
        CLR     P1.7            ; pin 7 port 1 to logic 0
        CALL    DELAY           ; call delay routine
        JMP     AGAIN           ; repeat pin 7 toggling
DELAY:  MOV.B   TH0,#0FAH       ; FAH into Timer 0 high byte
        MOV.B   TL0,#99H        ; 99H into Timer 0 low byte
        SETB    TR0             ; turn Timer 0 on
FLAG:   JNB     TF0,FLAG        ; wait till Flag sets at rollover
        CLR     TR0             ; turn Timer 0 off
        CLR     TF0             ; clear flag TF0 to zero
        RET                     ; return from sub-routine
        END                     ; end of assembly language
```

Note that, apart from MOV.B and DW 8F00H, there are a couple of instruction changes i.e. SJMP becomes JMP and ACALL becomes CALL.

Simulation

The simulation window is shown in Figure 6.2.

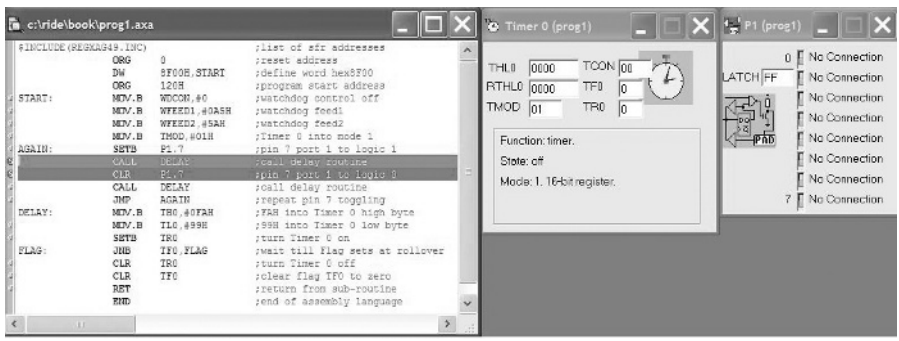


Figure 6.2 Simulation response showing the use of breakpoints

Placing a breakpoint on the first CALL and another at CLR P1.7 and then running the program to the first breakpoint will give a time interval which can be observed at the bottom of the PC screen as shown in Figure 6.3(a).

This example shows 0.003 ms, which is 3 µs. This number can be reset to zero by pressing Ctrl T and the program run to the next breakpoint, which gives the time shown in Figure 6.3(b). From Figure 6.3(b) the time interval is 0.501 ms, which is close to the required value of 0.5 ms.

Figure 6.3(a) Simulation response showing a breakpoint time interval

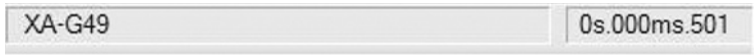


Figure 6.3(b) Simulation response showing a time interval between breakpoints

This Raisonance software is able to show digital traces in the simulation. Using the trace window and scrolling to the first transition and then clicking the left mouse button on the table line will produce a vertical cursor on the trace. The effect is shown in Figure 6.4. From this figure the table shows 82.489410 ms. Scrolling to the next transition, which is the first False in this example, gives a time of 82.991970 ms. The difference is 0.50256 ms. Trace options used were continual mode, maximum records = 2000.

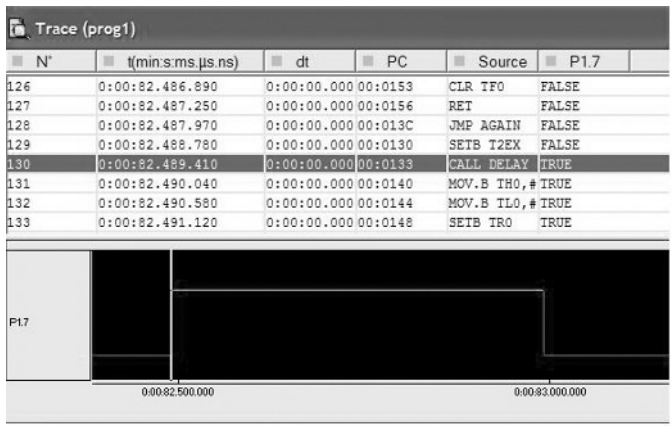


Figure 6.4 Simulation response showing trace window and transition times

Exercise 6.1
An XAG49 microcontroller having an 11.0592 MHz clock is to be used to generate a 1 kHz square-wave signal from pin 7 of port 1. Write a C program to achieve this.

6.2 XA registers

The XA general registers are shown in Figure 6.5. The registers are 16-bit wide although registers R0 to R7 may be accessed as high byte or low byte. As with the 8051 family, registers R0 to R3 are available in four banks and these banks

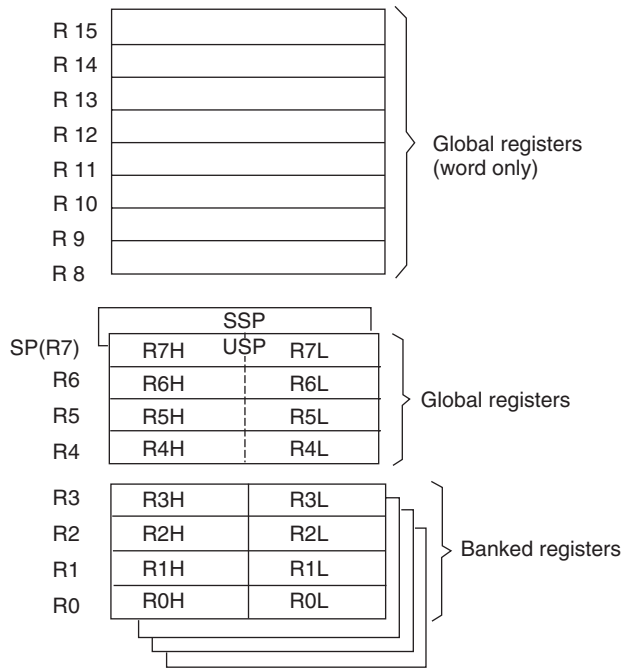


Figure 6.5 XA general registers

are selected by two bits RS0 and RS1 in the program status word high (PSWH) byte register:

PSWH

SM	TM	RS1	RS0	IM3	IM2	IM1	IM0

SM = 1 System mode
SM = 0 User mode

The XA can be used in a multitasking system, operating in system mode (system designer) or user mode (product user). In Figure 6.5, Register 7 (R7) is either the system stack pointer SSP (SM = 1) or the user stack pointer USP (SM = 0).

TM = 1 Trace mode on, SM must = 1
TM = 0 Trace mode off

Putting TM = 1 causes the XA to pause after each instruction and a monitor program could display any of the XA registers. This could be used for system debug.

RS1, RS0 Register bank selection, RS1, RS0 = 0, 0 is default bank 0
IM3, IM2, IM1, IM0 Interrupt mask levels. 1111 = F is the highest level

The PSW low (PSWL) byte register contains arithmetic and logic unit (ALU) information.

PSWL

C	AC	–	–	–	V	N	Z

C carry flag
AC auxiliary Carry flag
V overflow flag
N negative result flag
Z zero result flag

The program line in the program of Example 6.1 occurring at reset (ORG 0)

DW 8F00H, START

puts the XA in system mode (SM) and set the highest interrupt level, i.e.

8F00H = 1000 1111 0000 0000; 1000 1111 in PSWH 0000 0000 in PSWL

PSWH

SM	TM	RS1	RS0	IM3	IM2	IM1	IM0
1	0	0	0	1	1	1	1

6.3 Watchdog timer

The watchdog timer protects the system by causing a main reset when the watchdog underflows as a result of a failure of software to feed the timer prior to it reaching its overflow count. Unusually the XA watchdog timer is on by default and if not required it must be programmed off.

The watchdog timer has four SFRs:

WFEED1 feed part 1
WFEED2 feed part 2
WDL WatchDog auto-reLoad
WDCON WatchDog CONtrol

The watchdog timer arrangement is shown in Figure 6.6. Bit functions of the WDCON SFR are:

WDCON

PRE2	PRE1	PRE0	–	–	WDRUN	WDTOF	–

PREn Watchdog pre-scale
WDRUN = 1 (watchdog on)
= 0 (watchdog off)
WDTOF WatchDog TimeOut Flag

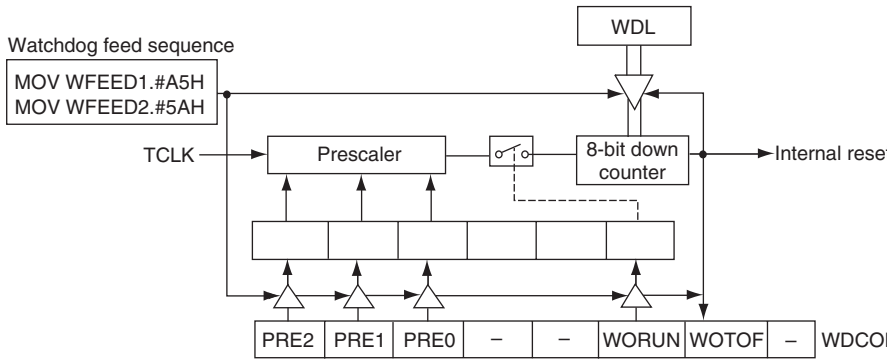


Figure 6.6 XAG49 watchdog timer

WDCON default (reset) value = E4H = 1110 0100. Hence watchdog on by default. The required routine to turn it off is:

```
MOV.B WDCON,#0
MOV.B WFEED1,#0A5H
MOV.B WFEED2,#5AH
```

and this must be the very first program task.

Using the watchdog would be good industrial practice. Local electro-magnetic interference (EMI) could freeze the circuit operation and a watchdog induced reset would automatically restart the circuit. The watchdog time delay, t_D, is given by:

$$t_D = t_{OSC} \times N \times P(W + 1)$$

where:

t_{OSC} = crystal oscillator cycle time
N = main timer prescale value
P = watchdog prescale value
W = watchdog 8-bit WDL value

N may be set using the system control register (SCR) SFR:

SCR

–	–	–	–	PT1	PT0	CM	PZ

PT1	PT0	N
0	0	4
0	1	16
1	0	64
1	1	–

$N = 4$ is the default. P may be set using WDCON:

PRE2	PRE1	PRE0	*P*
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

$P = 4096$ is the default value.

Example 6.2
An XAG49 microcontroller system having an 11.0592 MHz crystal oscillator uses the watchdog. WDL value is 8AH, SCR is 04H and WDCON is 64H. Determine the maximum time in which the watchdog refresh routine must be applied.

Solution

$$f_{OSC} = 11.0592 \times 10^6 \text{ Hz}$$
$$t_{OSC} = 1/f_{OSC} = 90.422 \text{ ns}$$

SCR = 04H = 0000 0100 binary, hence $N = 16$.
WDCON = 64H = 0110 0100 binary, hence $P = 256$.
WDL = 8AH = $(8 \times 16 + 10)$ decimal = 138.
The watchdog time delay, $t_D = 90.422 \times 10^{-9} \times 16 \times 256(138 + 1) = 51.48$ ms.

WATCHDOG REFRESH PROGRAM ROUTINE

Using values from Example 6.2, the instructions to produce the required values of *N*, *P* and *W* are:

```
MOV.B SCR,#04H
MOV.B WDCON,#64H
MOV.B WDL,#8AH
```

The following subroutine must be continually applied within 51.48 ms time periods to prevent the watchdog from resetting.

```
CLR    EA                ; disable all interrupts
MOV.B  WFEED1,#0A5H
MOV.B  WFEED2,#5AH
SETB   EA                ; enable all interrupts
RET
```

Example 6.3
The program of Example 6.1 is to be modified so that the watchdog is enabled with a time-out delay of approximately 0.6 ms.

Solution

$$t_D = t_{OSC} \times N \times P(W + 1) = 90.42\,ns \times 4 \times 32(51 + 1) = 0.602\,ms$$

The two watchdog feed lines are written into the 0.5 ms delay, so the watchdog should never time-out.

Program

```
$INCLUDE(REGXAG49.INC)          ; list of sfr addresses
         ORG    0               ; reset address
         DW     8F00H,START     ; define word hex8F00
         ORG    120H            ; program start address
START:   MOV.B  WDCON,#04H      ; watchdog pre-scale = 32
         MOV.B  WDL,#51          ; watchdog auto-reload = 51
         MOV.B  SCR,#0           ; timer clock pre-scale = 4
         MOV.B  TMOD,#01H        ; Timer0 into mode 1
AGAIN:   SETB   P1.7             ; pin7 port 1 to logic 1
         CALL   DELAY            ; call delay routine
         CLR    P1.7             ; pin7 port 1 to logic 0
         CALL   DELAY            ; call delay routine
         JMP    AGAIN            ; repeat pin7 toggling
DELAY:   MOV.B  TH0,#0FAH        ; FAH into Timer0 high byte
         MOV.B  TL0,#99H         ; 99H into Timer0 low byte
         SETB   TR0              ; turn Timer0 on
FLAG:    JNB    TF0,FLAG         ; wait till flag sets at rollover
         CLR    TR0              ; turn Timer0 off
         CLR    TF0              ; clear flag TF0 to zero
         MOV.B  WFEED1,#0A5H     ; feed the watchdog
         MOV.B  WFEED2,#5AH      ; feed the watchdog
         RET                     ; return from sub-routine
         END                     ; end of assembly language
```

Simulation
The simulation response is shown in Figure 6.7. In the simulation, putting a breakpoint at MOV.B TMOD,#01H and running the program from reset, the simulation will stop at this breakpoint. Pressing GO again, the program repeats at the AGAIN label so that, provided the watchdog is fed in time, the simulation should not stop at the breakpoint again.

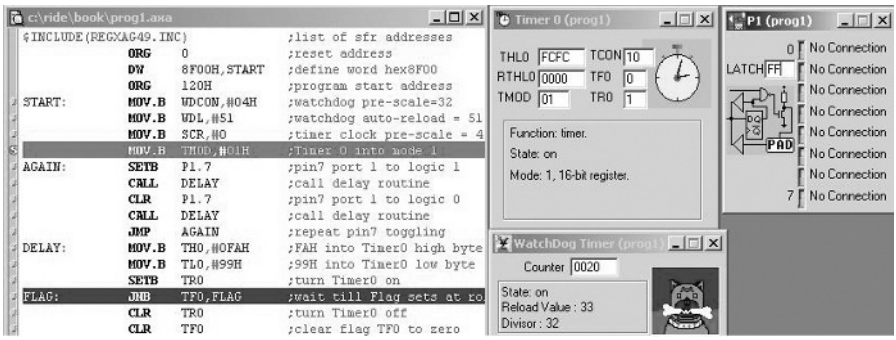


Figure 6.7 Simulation response illustrating the effect of the watchdog timer

Coming out of the simulation and changing WDL to #34 (0.4 ms), recompiling and returning to the simulation shows that repeated pressing of the GO button results in the program continually sticking at the breakpoint having arrived there from a watchdog reset.

The counter in the watchdog simulation window is that of the WDL register and shows it decrementing. For the watchdog delay of 0.6 ms WDL starts from 33H (51 decimal) and before WDL reaches zero the 0.5 ms delay ends and the watchdog is refreshed. For the watchdog delay of 0.4 ms (WDL = #34 or #22H) the 0.5 ms has not completed when WDL decrements to zero and the watchdog causes a main reset.

Exercise 6.2
Write the program for Example 6.3 using C language.

6.4 UART

The equation for the timer range value (TRV) when using the XA depends on the timer and mode used since the timer range (TR) can be either 65536 (16-bit timer) or 256 (8-bit timer). Timers 2 and 1 in mode 0 are 16-bit timers while timer 1 in mode 2 is an 8-bit timer.

$$TRV = TR - \frac{\text{Oscillator frequency}}{N \times 16 \times \text{Baud rate}}$$

N is the timer clock pre-scaler set by bits PT1 and PT0 in the system configuration register (SCR); this register is shown in earlier text on the watchdog timer. If $TR = 65536$ then TRV is converted into hex and put into reload registers RTL1 and RTH1 for timer 1 mode 0 and registers T2CAPL and T2CAPH when Timer 2 is used. If $TR = 256$ then TRV is 8 bits and is loaded into RTL1.

Example 6.4
(a) Timer 1 mode 2 is to be used with oscillator frequency = 11.0592 MHz, $N = 4$, baud rate = 9600. Determine the value for TRV and derive the program lines that would place the correct values in the timer registers.
(b) What would be the changes necessary to part (a) if timer 2 is used?

Solution
(a) $TRV = 238$ and the program lines could be:

```
MOV.B  TMOD,#20H    ; timer 1 mode 2
MOV.B  RTL1,#238    ; timer 1 reload TL1
MOV.B  TL1,#238     ; TL1 also initially set
SETB   TR1          ; turn timer 1 on
```

(b) If timer 2 had been used then TRV = 65518 decimal = FFEE hex and the program lines could be:

```
MOV.B  TH2,#0FFH    ; FFhex into timer 2 high byte
MOV.B  TL2,#0EEH    ; EEhex into timer 2 low byte
MOV.B  T2CAPH,#0FFH ; FFhex into timer 2 high byte capture
MOV.B  T2CAPL,#0EEH ; EEhex into timer 2 low byte capture
OR.B   T2CON,#34H   ; enable Rx and Tx clocks and timer 2 on
```

T2MOD not used since timer 2 defaults to 16-bit baud rate generator.

Example 6.5
Write an assembly language program that repeatedly sends two lines of text:
'Roses are red'
'Violets are blue'.

Solution
A comparison can be made between this UART program and that for the 8-bit P89C664 in Chapter 4 (Example 4.14) where the requirement was to send a single line of text. The basic program structure is modified to send two rows of text. Of course the XA has a different start and the watchdog is turned off and also the XA uses bytes and words. The registers used are either word length e.g. R6 or byte length e.g. R3L (L for low byte).

Look at the alternative to using the DPTR (data pointer):

MOVC R3L, [R6 +]

R6 contains the message address; the above instruction moves the contents of the address (message characters) into the low byte of R3 and increments R6 (points to the next character).

Program

```
$INCLUDE(REGXAG49.INC)              ; list of sfr addresses
         ORG    0                   ; reset address
         DW     8F00H,START         ; define PSW
         ORG    120H                ; program start address
START:   MOV.B  WDCON,#0            ; watchdog control off
         MOV.B  WFEED1,#0A5H        ; watchdog feed1
         MOV.B  WFEED2,#5AH         ; watchdog feed2
         MOV.B  S0CON,#42H          ; serial mode 1, TI set
         MOV.B  TMOD,#20H           ; timer 1 mode 2
         MOV.B  RTL1,#238           ; timer 1 reload TL1
         MOV.B  TL1,#238            ; TL1 also initially set
         SETB   TR1                 ; turn timer 1 on
FIRST:   MOV.B  R4L,#1              ; message marker
TEXT1:   MOV.W  R6,#MSG1            ; message1 address into R6
         JMP    NEXTCH              ; jump over message2
TEXT2:   MOV.W  R6,#MSG2            ; message2 address into R6
NEXTCH:  MOVC.B R3L,[R6+]           ; contents of R6 into R3L,
                                    ; increment R6
         CJNE   R3L,#7EH,NEXT       ; check for end of message,~=7EH
         MOV.B  R3L,#0DH            ; carriage return into R3L
         CALL   SEND                ; send carriage return
         MOV.B  R3L,#0AH            ; line feed into R3L
         CALL   SEND                ; send line feed
         DJNZ   R4L,FIRST           ; decrement R4L, if not zero
                                    ; message1
         JMP    TEXT2               ; if R4L zero message2
NEXT:    CALL   SEND                ; send current character
         JMP    NEXTCH              ; get next character
SEND:    JNB    TI,SEND             ; check SBUF clear to send
         CLR    TI                  ; clear TI
         MOV.B  S0BUF,R3L           ; send current character
         RET                        ; return from subroutine
MSG1:    DB     'Roses are red~'    ; message1
MSG2:    DB     'Violets are blue~' ; message2
         END                        ; end of assembly language
```

Simulation

The simulation can be run with the animation button set. The animation button is shown in Figure 6.8.

Figure 6.8 Animation button

In the simulation the blue horizontal cursor moves through the program and the message text prints out on the UART Buffer window as shown in Figure 6.9.

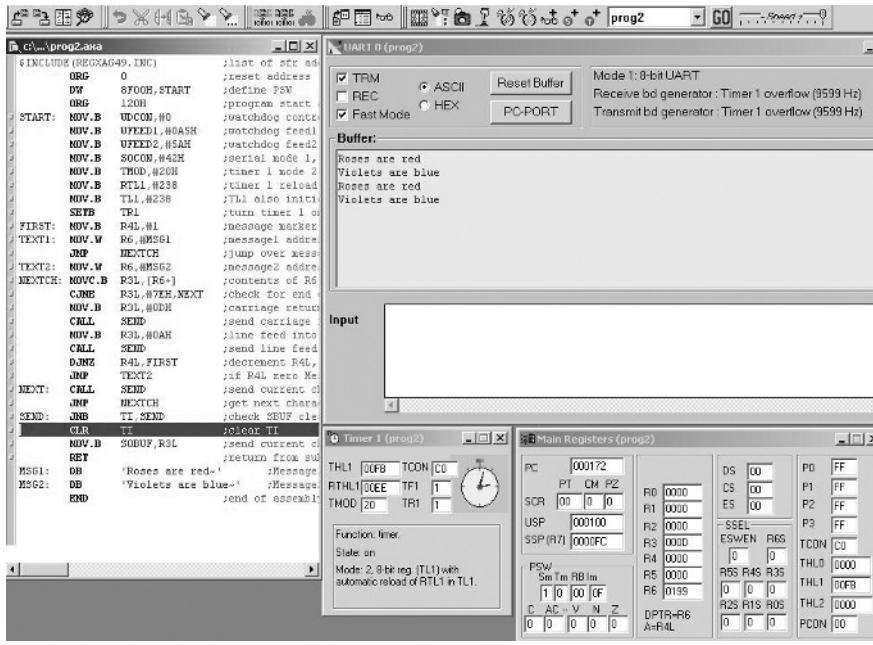


Figure 6.9 Simulation response showing the message printed in the UART buffer window

Exercise 6.3
Implement Example 6.5 in C.

6.5 8051 compatibility

The XA-G3 was the first XA on the market and was introduced as the 16-bit version of the 8051 microcontroller and there are bits in the SCR (system configuration register) to help the 8051 designer. The XAG49 is basically the flash version of the XA-G3.

SCR

–	–	–	–	PT1	PT0	CM	PZ

CM = 1 8051 compatibility mode

PZ = 1 Page Zero, forces the code and data memory to be limited to 16 address lines.

When CM = 1 the general-purpose registers are re-allocated. The registers are shown in Figure 6.10. Also there is a special 8-bit PSW register that looks like the PSW register in the P89C664:

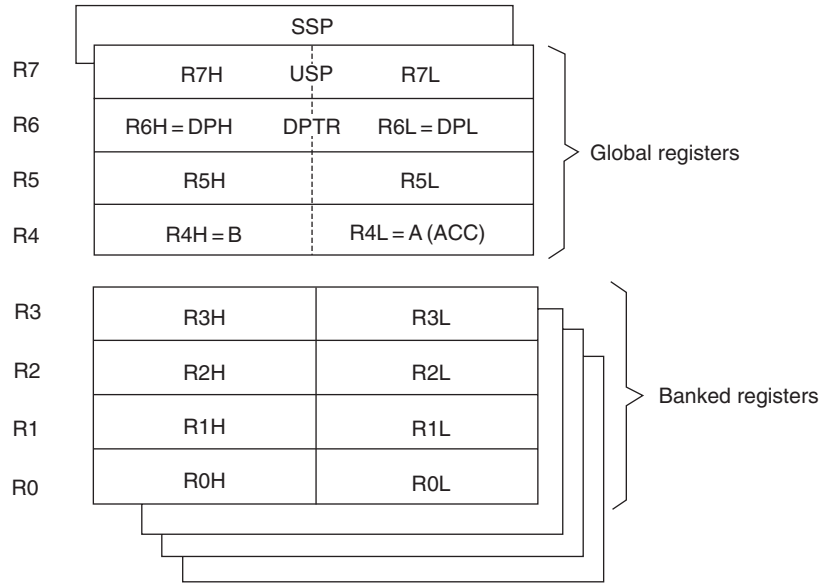


Figure 6.10 XAG49 general-purpose registers

PSW51

C	AC	F0	RS1	RS0	OV	F1	P

Translation software used to be available to convert 8051 programs to XA but it was not satisfactory. The XA programs start up differently and there is the watchdog, on by default. There is the need to use bytes (B) and words (W) and there is double-word (D), a 32-bit instruction, used by the division instruction.

It is the authors' opinion that the XA is best treated as a new microcontroller in its own right and not one that 8051 programs should be forced into. The Flash XAG49 is a particularly useful 16-bit microcontroller having two UARTs and a good set of timers. The default watchdog is an integral part of the device and far more robust than that of the P89C66x family; use of a watchdog is good practice for reliable systems.

6.6 Interrupts

The interrupt table for the 8-bit P89C66x microcontroller family showed eight interrupts which, apart from the two external interrupts $\overline{\text{INT0}}$, $\overline{\text{INT1}}$, occurred as a result of actions by the onboard peripherals. Collectively these are described as event interrupts. There was another type of interrupt, although it was not emphasised as such, and that is the reset. The action of operating the reset always starts the system running again from the reset address; reset is the

highest overriding interrupt. In XA terms, the reset is described as an exception interrupt that is serviced immediately.

The XA has four types of interrupts:

1. exception interrupts
2. trap interrupts
3. event interrupts
4. software interrupts.

Details of exception interrupts are provided in Table 6.1.

Table 6.1 XA exception interrupts
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

Exception interrupts are serviced as soon as they occur since each represents some important event or problem that must be dealt with before normal operation can resume. Reset has a higher priority than the other exceptions and is always serviced immediately, aborting other exceptions.

Example 6.6
The following program actually has a line:

DIVU.B R4L,#0 (register 4 low byte is divided by zero)

It should be appreciated that the purpose of the program is to demonstrate an exception handling routine. The divide-by-zero interrupt vector address is 0010H. When the divide-by-zero happens the program goes to a UART routine

that could send a message to the host PC. In a small system, the message might be sent to an alphanumeric LCD.

Solution

Program

```
$INCLUDE(REGXAG49.INC)              ; list of sfr addresses
          ORG       0               ; reset address
          DW        8F00H,START     ; SM = 1, IM = F
          ORG       0010H           ; divide-by-zero interrupt
                                    ; vector
          DW        8A00H,TEXT      ; go to message
          ORG       120H            ; program start address
START:    MOV.B     WDCON,#0        ; watchdog control off
          MOV.B     WFEED1,#0A5H    ; watchdog feed1
          MOV.B     WFEED2,#5AH     ; watchdog feed2
          MOV.B     S0CON,#42H      ; serial mode 1, TI set
          MOV.B     TMOD,#20H       ; timer 1 mode 2
          MOV.B     TH1,#0FAH       ; baudrate 9600
          MOV.B     TL1,#0FAH       ; TL1 also initially set
          SETB      TR1             ; turn timer 1 on
;Divide by zero
          MOV.B     R4L,#44         ; load R4L with 44
          DIVU.B    R4L,#0          ; divide R4L by zero
STAY:     JMP       STAY            ; stay here after message
TEXT:     MOV.W     R6,#MSG1        ; message1 address into R6
NEXTCH:   MOVC.B    R3L,[R6+]       ; contents of R6 into R3L,
                                    ; increment R6
          CJNE      R3L,#7EH,NEXT   ; check for end of message,
                                    ; ~ = 7EH
          MOV.B     R3L,#0DH        ; carriage return into R3L
          CALL      SEND            ; send carriage return
          MOV.B     R3L,#0AH        ; line feed into R3L
          CALL      SEND            ; send line
          RETI                      ; return from interrupt
NEXT:     CALL      SEND            ; send current character
          JMP       NEXTCH          ; get next character
SEND:     JNB       TI,SEND         ; check SBUF clear to send
          CLR       TI              ; clear TI
          MOV.B     S0BUF,R3L       ; send current character
          RET                       ; return from subroutine
MSG1:     DB        'Divide by zero~' ; Message1
          END                       ; end of assembly language
```

Simulation

The simulation response is shown in Figure 6.11. It is useful to single step until the exception interrupt occurs; at this rate it would be possible to see that the SSP stacks down to 00FA. The SPs in the 8051 microcontrollers stack up.

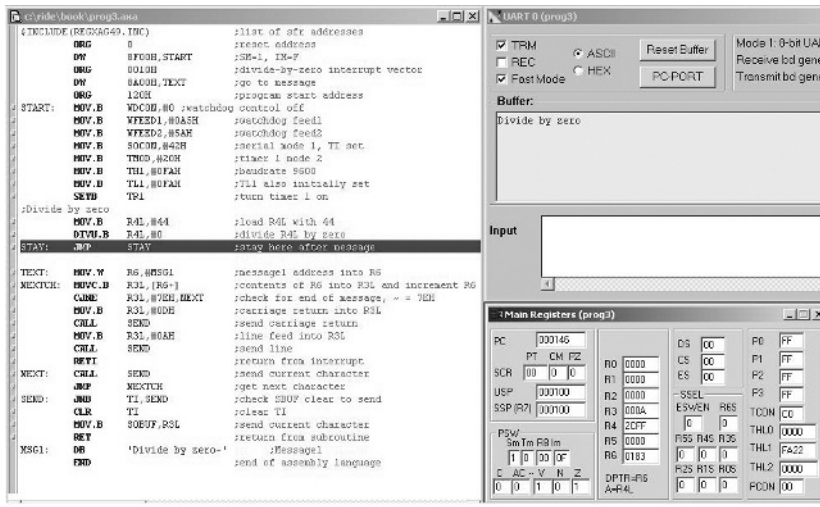


Figure 6.11 Simulation response showing the effect on system registers of an exception interrupt

If the simulation is then run by pressing Go (the program is also stopped with this button), it should be possible to see in the Main Registers window that Z (divide-by-zero) and V (overflow) in the PSW are set to 1.

Also R3L contains 0A (line feed was the last action of the message routine). R4H contains hex 2C (the equivalent decimal is 44). R4L contains FF; this is as big as 8 bits gets! In mathematical terms 44/0 might tend to infinity. The message appears in the UART buffer.

Exercise 6.4

Using C language, write a program that demonstrates division-by-zero exception and, in your exception function, provide a method that sends a message to the UART.

Recall that when event interrupts were used for the 8-bit P89C664 microcontroller the IE register had to be configured and priorities assigned. This will also be done later for the XA event interrupts. This type of preparation is not necessary for exception and trap interrupts since they are activated as soon as they happen. However, exception interrupts have a higher priority than traps and reset has the highest priority of all. Details of trap interrupts are shown in Table 6.2.

It should be noted that a trap interrupt is a type of exception interrupt. It occurs immediately and there is no prioritising. The use of trap interrupts is illustrated in Figure 6.12.

The XA has two operating modes, system and user (or application). This is in common with other microcontrollers capable of sustaining and supporting real-time multitasking systems. To manage tasks between these two modes the XA

Table 6.2 XA 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

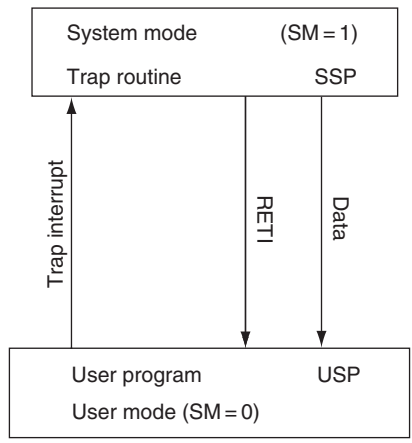


Figure 6.12 Use of trap interrupts

has two SPs, SSP and USP. The control between the two is done using the SM bit in the PSWH.

Trap interrupts occur in user mode programs but are serviced in the controlling SM program.

Example 6.7
The following program is designed to illustrate the use of a trap interrupt that causes pin 7 of port 1 to toggle.

Solution

Program

```
$INCLUDE (REGXAG49.INC)              ; sfr addresses
          ORG       0                ; reset address
          DW        8F00H,START      ; SM = 1, IM = F
          ORG       54H              ; Trap 5 vector address
          DW        8F00H,SYSTEM     ; go to System routine
          ORG       120H             ; program start address
START:
;watchdog off
          MOV.B     WDCON,#0         ; watchdog control off
          MOV.B     WFEED1,#0A5H     ; watchdog feed1
          MOV.B     WFEED2,#5AH      ; watchdog feed2
;assign SSP(System Stack Pointer)and USP values
          MOV.W     R7,#0800H        ; SSP = 0800H
          CLR       SM               ; SM = 0 User mode
          MOV.W     R7,#0800H        ; USP = 0800H
; User mode routine runs whilst P1.0 is high
USER:
          SETB      P1.4             ; port 1 pin 4 to logic1
          CLR       P1.4             ; port 1 pin 4 to logic0
          JB        P1.0,USER        ; test pin 0 for logic1
          TRAP      #05              ; activate Trap if pin 0 = 0
          JMP       USER             ; repeat User routine
;System mode routine runs when Trap 5 is executed
SYSTEM:
          SETB      P1.7             ; port 1 pin 7 to logic1
          CLR       P1.7             ; port 1 pin 7 to logic0
          RETI                       ; return from interrupt
          END                        ; end of assembly language
```

Simulation

The simulation response is shown in Figure 6.13. Selecting the Animation button (shown in Figure 6.8) and then pressing GO causes the simulation to continually repeat the first three lines of the user routine. Because the port pins default to logic 1, the switch test on pin 0 is high.

Stopping the simulation and looking at the main registers would show that register 7 (R7) is assigned to the USP; it was left there when the SP values were assigned. Also looking in the PSW window would show that SM = 0, confirming that the system is in user mode.

Moving the cursor to the LATCH space on the port 1 window and clicking the left hand mouse button to the right of the F, the least significant byte, and changing it to an E will cause pin 0 to go to logic 0. This would cause a colour change from green to red.

Now continuing the simulation by single stepping until the trap instruction executes when the program jumps to the system routine. If the main registers are checked it will show that SM has now gone to 1 and R7 is pointing to the

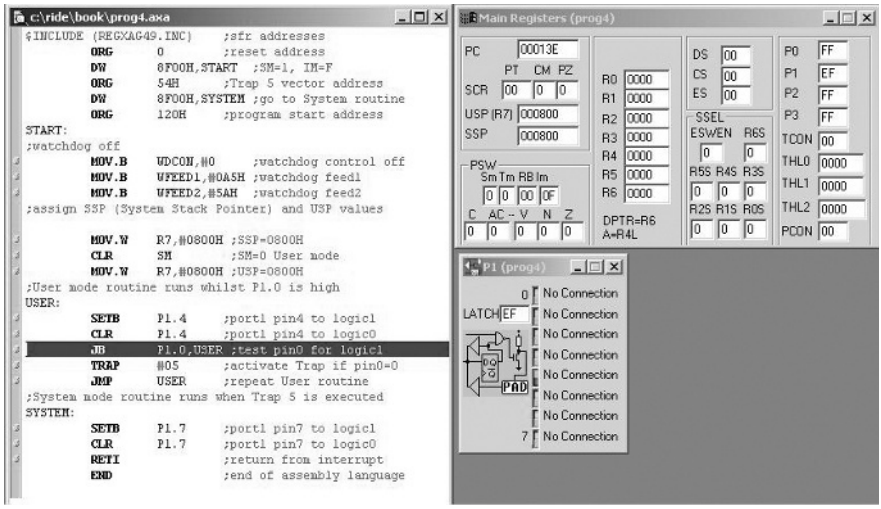


Figure 6.13 Simulation response showing the effect on system registers of a trap interrupt

SSP. Execution of the trap instruction has taken the microcontroller from user mode to SM.

If single stepping is continued, pin 7 will be toggled and when RETI is executed the microcontroller would return to user mode.

Exercise 6.5

Use a C program to demonstrate the trap events, and prove the C program works by using simulation similar to that used in Example 6.7.

Event interrupts are illustrated in Table 6.3. Each event interrupt has seven priorities associated with it, 9 (lowest) to 15 (highest).

Table 6.3 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 (PRI1)	9
Serial port 1 Tx	TI.1	00AC–00AF	ETI1	IPA5.6 – 4 (PTI1)	10

The interrupt priority column shows three bits associated with each event interrupt and this could imply a range 0–7 but event interrupts must be read as having a range (8 + 1) to (8 + 7) i.e. 9–15. This is to avoid confusion with software interrupts, explained later, which have a lower priority range of 1(lowest) to 7(highest). A value of 0 in the event interrupt priority field will disable the interrupt.

Example 6.8
A program to illustrate the use of event interrupts will use the following program lines:

```
MOV.B IPA0,# 20H   ; Timer 0 priority = 10(i.e. 8 + 2)
MOV.B IPA1,# 03H   ; External interrupt 1 priority = 11(i.e. 8 + 3)
```

So external 1 interrupt would have a greater priority than timer interrupt 0 even though the latter has a higher arbitration ranking.

Solution

Program

```
$INCLUDE(REGXAG49.INC)       ; sfr addresses
         ORG    0            ; reset address
         DW     8200H,START  ; SM=1,IM=2
         ORG    84H          ; Timer 0 interrupt vector
         DW     8A00H,TIMER  ; IM=10, go to Timer 0 int.
         ORG    88H          ; External1 interrupt vector
         DW     8B00H,EXTNL  ; IM=11, go to Extnl1 int.
         ORG    120H         ; program start address
START:   MOV.B  WDCON,#0     ; Watchdog off
         MOV.B  WFEED1,#0A5H
         MOV.B  WFEED2,#5AH
         MOV.B  TMOD,#02H    ; Timer 0 in mode2
         MOV.B  TL0,#0DDH    ; hexDD into Timer 0 low byte
         MOV.B  RTL0,#0DDH   ; hexDD into Timer 0 Reload
         MOV.B  IPA1,#03H    ; Extnl int. priority=11
         MOV.B  IPA0,#20H    ; Timer 0 int. priority=10
         MOV.B  IEL,#86H     ; EA and Ex1, ET0 enables
         SETB   TR0          ; turn Timer 0 on
STAY:    JMP    STAY         ; stay here wait for int.
TIMER:   SETB   P1.7         ; Timer 0 interrupt routine
         CLR    P1.7         ; toggling pin 7 on port1
         JMP    TIMER        ; repeat
EXTNL:   SETB   P1.4         ; External1 interrupt
         CLR    P1.4         ; toggling pin 4 on port1
         JMP    EXTNL        ; repeat
         END                 ; end of assembly language
```

Simulation
The simulation response is shown in Figure 6.14. Note that the first interrupt mask (IM), third line of the program, is less than those of the event interrupts.

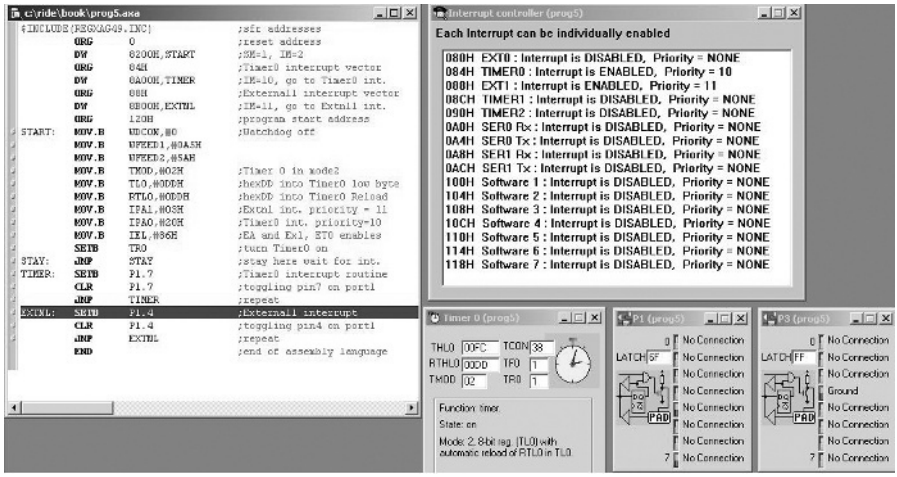


Figure 6.14 Simulation response showing the effect of an event interrupt

This has no effect on the reset because reset is the highest priority exception vector.

In addition to the interrupt window, timer 0 and port 1 windows, the port 3 window is also accessed. External 1 interrupt is activated by making pin 3 of port 3 go to logic 0. Pressing the animation button (shown in Figure 6.8) and pressing GO will run the simulation. Quite soon the timer 0 register will overflow and the interrupt sequence will run, toggling pin 7 on port 1.

With the simulation it is possible to position the cursor at strategic spots and by adjusting the cursor position the arrowhead cursor changes to a pointing finger; clicking the left mouse button would allow the values pointed at to be changed. While the timer 0 interrupt program is running if the mouse cursor is moved over pin 3 of port 3 then the pin 3 voltage level can be changed to ground and immediately the external 1 interrupt would break into the timer 0 interrupt routine and pin 4 would be toggled. If reset is pressed the program re-runs straight to the external 1 interrupt because pin 3 on port 3 would stay low and the interrupt is level activated.

Exercise 6.6

Write a C program to configure the timer 0 in mode 2 with interrupt priority of 10 and enable external interrupt 1 with priority of 11. Then show the effects of these in simulation as was illustrated in Example 6.8.

Software interrupts are shown in Table 6.4. Software interrupts are similar to event interrupts except they are activated by software writing to the appropriate interrupt request bit in the relevant SFR. There are two SFRs:

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

The primary purpose of the software interrupt is to provide an organised way in which portions of the event interrupt routine may be executed at a lower priority level than the one at which the service routine began.

Table 6.4 XA 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)

Example 6.9

In the next program a timer 0 event interrupt is set up with a priority of 9 and a software interrupt is set up with a priority of 1. Half way through the event interrupt, its priority is lowered to zero allowing the software interrupt to be activated.

The idea is that whilst the low priority software interrupt is active it could be interrupted by other important event interrupts.

Solution

Program

```
$INCLUDE(REGXAG49.INC)        ; sfr addresses
         ORG    0             ; reset address
         DW     8000H,START   ; SM=1, IM=0
         ORG    84H           ; Timer 0 interrupt vector
         DW     8900H,TIMER0  ; goto Timer 0 interrupt
         ORG    100H          ; SWR1 interrupt vector
         DW     8000H,SWINT1  ; goto SWR1 interrupt
         ORG    120H          ; program start address
;Watchdog off
START:   MOV.B  WDCON,#0      ; Watchdog control off
         MOV.B  WFEED1,#0A5H  ; Watchdog feed1
         MOV.B  WFEED2,#5AH   ; Watchdog feed2
;intialise Timer 0
         MOV.B  TMOD,#02H     ; Timer 0 in mode 2
         MOV.B  TL0,#0EEH     ; TL0 loaded with hexEE
         MOV.B  RTL0,#0EEH    ; hexEE into Reload
;initialise interrupts
         MOV.B  IEL,#82H      ; Enable all and Timer 0
```

```
          MOV.B   IPA0,#10H      ; Timer 0 priority = 9
          OR.B    SWE,#01H       ; SWI priority = 1
; Timer 0 on
          SETB    TR0            ; turn Timer 0 on
STAY:     JMP     STAY           ; stay wait for interrupts
; Timer 0 Event interrupt
TIMER 0:  SETB    P1.0           ; set pin 0 to 1
          CLR     P1.0           ; clr pin 0 to 0
          MOV.B   IPA0,#0        ; lower Timer 0 priority = 0
          SETB    SWR1           ; activate SWI priority = 1
; Software interrupt
SWINT1:   CLR     SWR1           ; to allow repeat SWI
          SETB    P1.1           ; set pin 1 to 1
          CLR     P1.1           ; clear pin 1 to 0
          MOV.B   IPA0,#10H      ; up Timer 0 priority = 9
          RETI                   ; return from interrupt
          END                    ; end of assembly language
```

Simulation

The simulation response is shown in Figure 6.15. If the interrupt, port 1 and timer 0 windows are accessed, the animation button pressed, and the program single stepped through, it should be possible to see timer 0 event interrupt enabled and priority set to 9. Then the software interrupt would be enabled with a priority of 1. The TL0 and RTL0 registers have a large number, so it

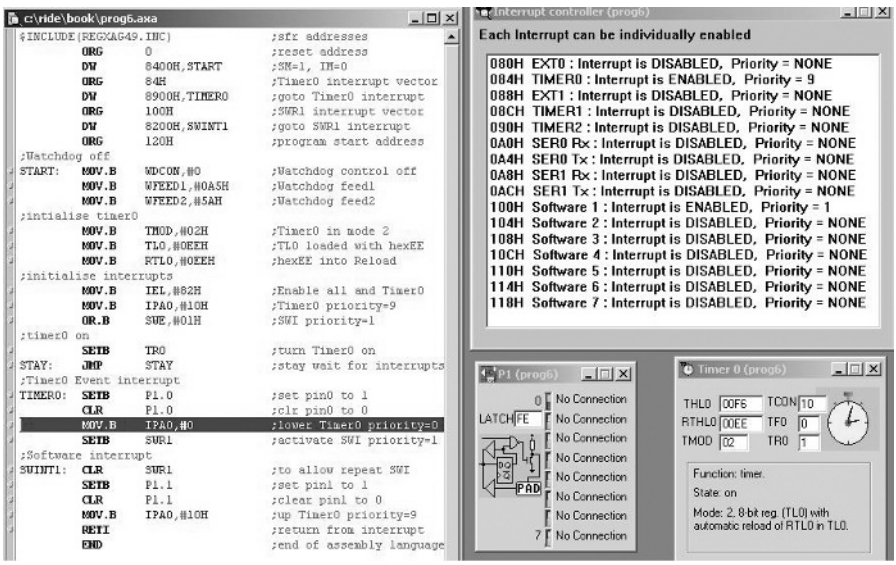


Figure 6.15 Simulation response showing the effect of a software interrupt

would not take long in single step mode to roll over and cause a timer 0 event interrupt.

The event interrupt task is simple, pin 0 on and off. Next the event priority is reduced to zero and this would allow the priority 1 software interrupt to become active. Again the task is simple, pin 1 on and off, before the event interrupt is restored to its priority level of 9.

EVENT INTERRUPT AND SOFTWARE INTERRUPT PRIORITISING

Figure 6.16 gives an example. Event priority level 10 starts off, and then a level 12 interrupts it. When the level 12 finishes, the level 10 resumes; a level 5 software interrupt comes in but this must wait for all event interrupts to finish. Whilst level 10 is happening, a level 8 is activated but has to wait until level 10 finishes. Level 10 finishes and then level 8 is serviced. Only after all the event interrupts are over can the level 5 software interrupt be serviced.

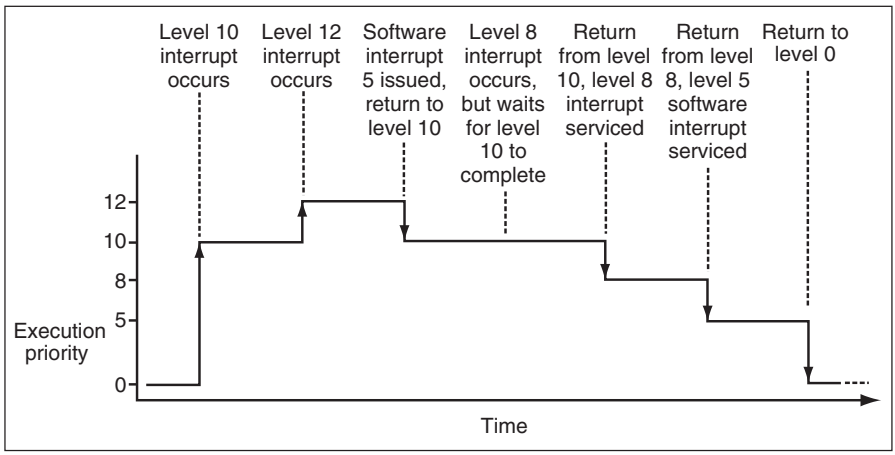
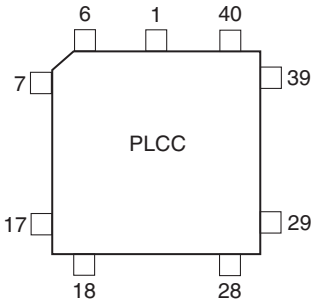


Figure 6.16 Example illustrating event and software interrupt prioritising

Remember software interrupts have priority levels 1 (low) to 7 (high) whilst event interrupts have priority levels 9 (low) to 15 (high). Level 8 disables the event interrupt and lets the software interrupt in. Typical XA hardware is shown in Figure 6.17.



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/R×D1	28	P2.4/A16D12
7	P1.5/T×D1	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/R×D0	33	ALE
12	NC	34	$\overline{NC}$
13	P3.1/T×D0	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	XTAL 2	42	P0.1/A5D1
21	XTAL 1	43	P0.0/A4D0
22	V_{SS}	44	V_{DD}

Figure 6.17 Pin functions for the XAG49 PLCC package

Summary

- The XA microcontroller is the 16-bit version of the 8051 device.
- The device has a watchdog timer which is on by default.
- The watchdog timer has a time delay that can be set, within a maximum value, by the user.
- The XA has 16-bit registers.
- The XA has two UARTs.
- The device has four types of interrupts.
- The device has two operating modes – system and user.

7

Project Applications

7.1 Introduction

The text for earlier chapters has concentrated on particular devices with explanations, and relevant programs, on the use of the onboard peripherals of the specified device. The examples presented in the earlier chapters are of a relatively trivial nature in order to illustrate the use of timers, interrupts, etc. This chapter will present examples of a more complex nature designed to achieve a specific objective but using the principles outlined in the preceding chapters. The reader is invited to develop the projects further by adding to, or modifying, the original project. This is not offered as an exercise and no solutions are provided since the alterations to the original project could take different forms. The use of simulation should enable the reader to establish whether any alteration carried out on the original project results in the required outcome.

7.2 Project 1: speed control of a small DC motor

The requirement is to use a microcontroller to drive a DC motor in both forward and reverse directions of shaft rotation and to implement a two-speed (fast and slow) arrangement. Switches are to be used to produce the two speeds and effect a reversal of shaft rotation. A possible arrangement is shown in the block diagram of Figure 7.1, which uses a P89C664 microcontroller.

The method of speed control is by the pulse width modulation (PWM) technique, described in Chapter 4, using the P89C664 device. Putting pin 5 on port 1 (P1.5) to logic 0 and applying the PWM to pin 4 (P1.4) causes the motor shaft to rotate. Holding pin 4 at logic 0 and applying the PWM to pin 5 causes the shaft rotation to reverse. This control of forward or reverse rotation is achieved by the bridge design of the motor drive circuit.

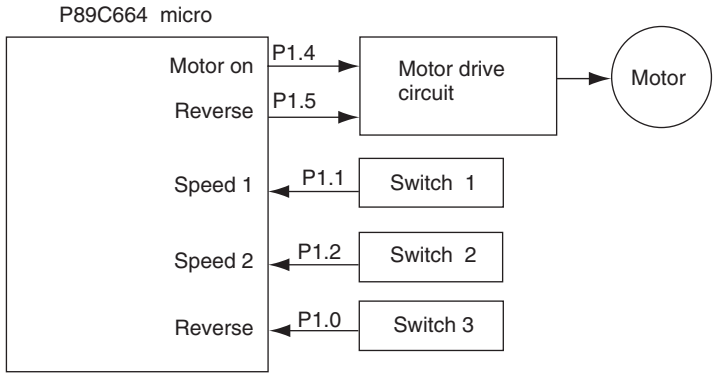


Figure 7.1 Block diagram for speed control of a small DC motor using a P89C664 microcontroller

If a switch is not pressed then the motor shaft remains stationary. For the purposes of this example it is assumed that switch 1 sets a PWM ratio of 6:4 and switch 2 sets a PWM ratio of 9:1.

From Chapter 4, the description of the PWM technique shows that a 6:4 ratio has a total of $6 + 4$ periods (i.e. 10 periods in total), so a 9:1 ratio will have the same total of 10 periods. A 6:4 ratio means 6 periods at logic 1 and 4 periods at logic 0 whereas a 9:1 ratio has 9 periods at logic 1 and 1 period at logic 0 giving a higher average value over 10 periods. The latter arrangement will give a higher DC value over the 10 periods and hence produce a higher speed of shaft rotation than that produced by the former arrangement.

The capture registers CCAP1L (low) and CCAP1H (high) are both 8 bits and therefore have 2^8 or 256 increments.

$$\text{Increments per period} = \frac{256}{10} = 25.6$$

Therefore $6{:}4 = 154$ increments at logic 1, 102 increments at logic 0. Ratio $9{:}1 =$ 230 increments at logic 1, 26 increments at logic 0. It is assumed that the switches are normally at logic 1 and switch to logic 0 when pressed. A possible arrangement is shown in Figure 7.2.

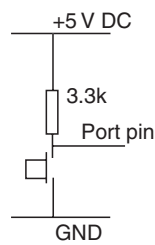


Figure 7.2 Circuit for achieving logic 1/0 levels

MOTOR DRIVE

A bridge arrangement is shown in Figure 7.3. The circuit utilises complementary pair NPN/PNP transistors T2/T3 and T4/T5. The motor takes no more than half an ampere while the diodes greatly reduce the induced voltages caused by quickly switching currents.

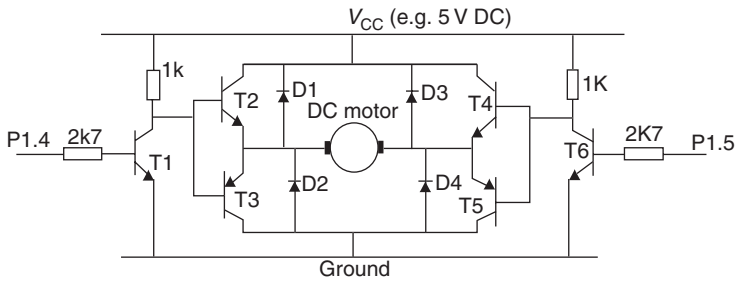


Figure 7.3 A bridge drive circuit using the inputs from port pins P1.4 and P1.5 to control the DC motor speed and direction of rotation

Motor off

If P1.4 and P1.5 are both held at logic 0, the collectors of T1 and T6 will both be high. Thus T2 and T4 will be ON (conducting) while T3 and T5 will be OFF (non-conducting) and there will be no conduction path through the motor between the 5 V supply rail and ground.

Motor on

If P1.5 is held at logic 0, the collector of T6 will be high, T4 will be ON and T5 will be OFF. If a PWM signal is applied to P1.4 then when the PWM is high at logic 1, T1 collector will be low; so T2 will be OFF and T3 will be ON. Hence there is a conduction path from ground through transistor T3, the motor and through transistor T4 up to the 5 V supply rail.

Motor reverse

If P1.4 is held at logic 0, then transistor T2 will be ON and T3 will be OFF. If a PWM signal is applied to P1.5 then when PWM is high, transistor T4 will be OFF, T5 will be ON giving a reverse conduction path through transistor T5, the motor and transistor T2.

PROGRAM PLAN

The program is to drive the motor in one direction from pin 4 of port 1 using the PWM method constructed from the programmable counter array (PCA). Reverse is achieved by applying the PWM through pin 5 of port 1.

Two speeds are possible: fast (9:1) and slow (6:4). Reverse and the speeds are chosen from active-low on/off switches. See Figure 7.1.

The program is forced to check the switches by the action of the active low interrupt $\overline{\text{INT0}}$ on pin 2 of port 3. The following program lines declare the interrupt vector address at 0003H and point to the interrupt chosen sequence, CHECK.

```
ORG 03H        ; external interrupt 0 address
SJMP CHECK     ; jump to interrupt routine
```

START

CFH = 1100 1111 binary and this forces pins 4 and 5 of port 1 to be zero and so the first action of the program is to turn off the motor.

SETB IT0 Sets the interrupt to occur on a high-to-low transition (i.e. negative edge) of a switch action on pin 2 of port 3 ($\overline{\text{INT0}}$).

MOV IEN0,#81H Puts binary 1000 0001 into IE register IEN0 to enable the action of $\overline{\text{INT0}}$.

CHECK

The switches on port 1 pins 0 (REVERSE), 1 (6:4 SPEED1), 2 (9:1 SPEED2) are normally at logic 1. When they are pressed they go to logic 0 and JNB (jump if not bit) becomes active and sends the program to the corresponding routine.

SPEED

For SPEED1 and SPEED2, the PWM action is through pin 4 on port 1 whereas for the reverse rotations, SPEED1R and SPEED2R, the PWM action is through pin 5 on port 1. So the action of the first two program lines of these four subroutines is to disable the PWM action on the opposite pins.

Program

```
$INCLUDE (REG66x.INC)           ; sfr addresses
          ORG   0               ; reset address
          SJMP  START           ; jump to start
          ORG   03H             ; external interrupt 0 address
          SJMP  CHECK           ; jump to interrupt routine
          ORG   40H             ; program start address
START: MOV P1,#0CFH             ; motor drives to zero
          SETB  IT0             ; interrupts on negative edge
          MOV   IEN0,#81        ; external int INT0 enabled
STAY:     SJMP  STAY            ; stay here till int occurs
CHECK:    JNB   P1.0,REVERSE    ; if selected goto reverse
          JNB   P1.1,SPEED1     ; goto speed1 6:4
          JNB   P1.2,SPEED2     ; goto speed2 9:1
          SJMP  CHECK           ; check switches again
SPEED1:   ANL   CCAPM2,#0FDH    ; disable PWM drive on P1.5
          CLR   P1.5            ; put P1.5 to logic 0
```

```
            ORL   CCAPM1,#42H    ; set ECOM1 and PWM1 (P1.4)
            MOV   CCAP1L,#102    ; load 6:4 count
            MOV   CCAP1H,#102    ; 6:4 count reload
            ORL   CCON,#40H      ; set CR to turn PCA timer on
            RETI                 ; return from interrupt
SPEED2:     ANL   CCAPM2,#0FDH   ; disable PWM drive on P1.5
            CLR   P1.5           ; put P1.5 to logic 0
            ORL   CCAPM1,#42H    ; set ECOM1 and PWM1 (P1.4)
            MOV   CCAP1L,#26     ; load 9:1 count
            MOV   CCAP1H,#26     ; 9:1 count reload
            ORL   CCON,#40H      ; set CR to turn PCA timer on
            RETI                 ; return from interrupt
REVERSE:    JNB   P1.1,SPEED1R   ; goto speed1 reverse
            JNB   P1.2,SPEED2R   ; goto speed2 reverse
            SJMP  CHECK          ; check input switches
SPEED1R:    ANL   CCAPM1,#0FDH   ; disable PWM drive on P1.4
            CLR   P1.4           ; put P1.4 to logic 0
            ORL   CCAPM2,#42H    ; set ECOM2 and PWM2 (P1.5)
            MOV   CCAP2L,#102    ; load 6:4 count
            MOV   CCAP2H,#102    ; 6:4 count reload
            ORL   CCON,#40H      ; set CR to turn PCA timer on
            RETI                 ; return from interrupt
SPEED2R:    ANL   CCAPM1,#0FDH   ; disable PWM drive on P1.4
            CLR   P1.4           ; put P1.4 to logic 0
            ORL   CCAPM2,#42H    ; set ECOM2 and PWM2
            MOV   CCAP2L,#26     ; load 9:1 count
            MOV   CCAP2H,#26     ; 9:1 count reload
            ORL   CCON,#40H      ; set CR to turn PCA timer on
            RETI                 ; return from interrupt
            END                  ; end of assembly language
```

Simulation

This uses the Raisonance software (see Chapter 3 for details). With the Trace window activated, settings used are Mode = Continual, Maximum number of records = 500. P1.4 and P1.5 are set in the Watches window and have Trace added to them. Port 1 (motor drive and switches) and port 3 (active low interrupt on pin 2) are also displayed, as are the PCA and main registers windows. See Figure 7.4.

CLM on the Animation icon (two red characters in film). Pin 1 on port 1 may be changed to ground by moving the mouse cursor over the pin till the arrow changes to a pointing finger, then CRM and selecting ground. CLM on GO will run the simulation. Pins 4 and 5 should go to ground in the port and Trace windows. Whilst the simulation is running, moving the mouse cursor over pin 2 (3rd pin down) of port 3 and changing it to ground will cause an active low interrupt. Eventually the Trace window should show the 6:4 PWM signal. If the trace labels do not show then CLM on P1.4 and P1.5 buttons at the top of the trace window (see Figure 7.5).

As an exercise the reader is invited to experiment with different combinations of the control switches. However, remember that the changed response will only happen when pin 2 on port 3 has a high (V_{cc}) to low (ground) transition.

Figure 7.6 shows the simulation response at the time when the 6:4 speed on port 1 pin 4 has changed to a reverse rotation 9:1 speed driven from port 1 pin 5.

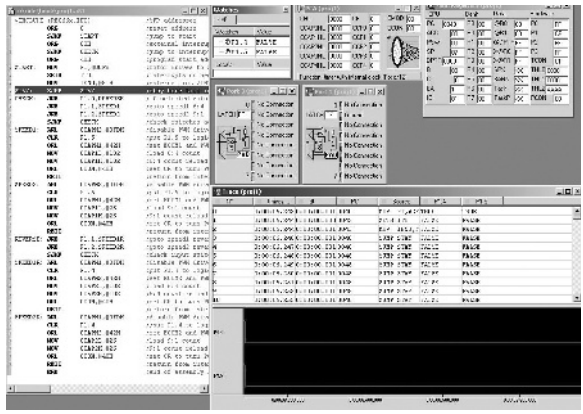


Figure 7.4 Simulation response showing project 1 program window and other relevant windows

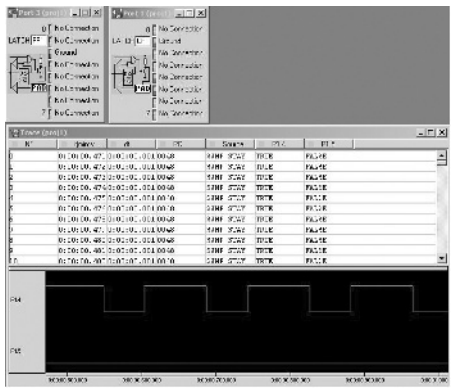


Figure 7.5 Simulation response showing the trace waveforms for a 6:4 PWM signal on P1.4

Other registers to observe during simulation include the PC and SP both in the main registers window. The PC contains the program address and the SP is increased when an interrupt occurs and is restored to its default value of 07H when the program returns from interrupt.

PROGRAM DEVELOPMENTS

1. It is possible to progressively increment the PWM ratio from a low speed value (e.g. 10:90) up towards a higher speed value (e.g. 90:10) and to stay at a set speed when a switch is released.
2. Dynamic breaking is sudden and can be caused by putting the motor into reverse for a very short time (e.g. 100 ms) and then turning off the PWM.
3. Where would you put a register decrement delay routine to cause a time difference (software de-bounce) between the occurrence of the interrupt and the testing of the control switches?

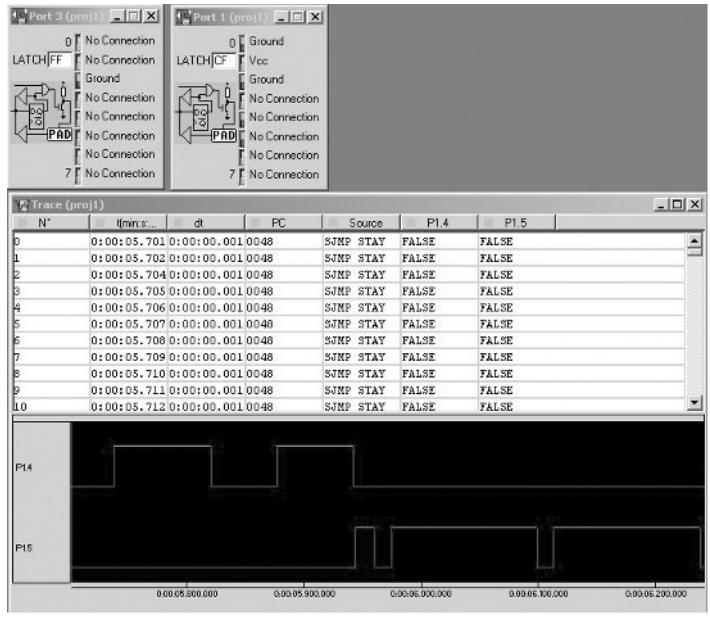


Figure 7.6 Simulation response showing the trace waveforms for a 9:1 PWM signal on P1.5

7.3 Project 2: speed control of a stepper motor

The requirement is to use a microcontroller to drive a stepper motor in both forward and reverse directions of shaft rotation and to implement a two-speed (fast and slow) arrangement. Switches are to be used to produce the two speeds and effect a reversal of shaft rotation. A possible arrangement is shown in the block diagram of Figure 7.7, which uses a P89C664 microcontroller.

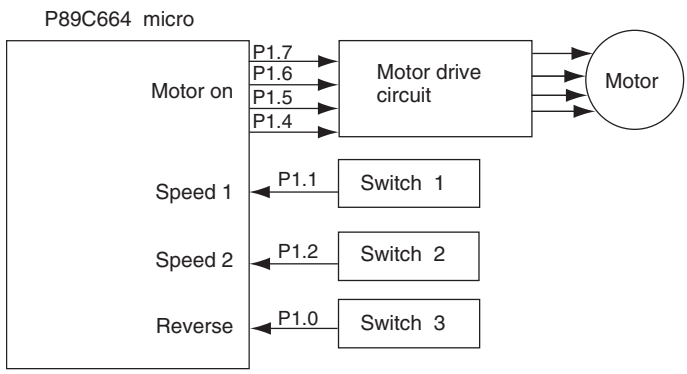


Figure 7.7 Block diagram for speed control of a stepper motor using a P89C664 microcontroller

Generally a stepper motor has four sets of coils. One end of each coil may be connected together and then connected to DC supply. The remaining four ends may be driven through transistors either separately or in integrated circuit form.

A four-bit code sequence continuously applied to the drive circuit from the microcontroller port causes the motor shaft to rotate in angular steps. Cheap (e.g. £12) stepper motors have step angles of 7.5 degrees whereas more expensive (e.g. £45) motors have step angles of 1.8 degrees. Step resolution and turning force (i.e. torque) may be improved by using a step-down gearbox.

The stepping code sequence may be obtained from the motor manufacturer or distributor. The program in this example uses a common four-step sequence of A 9 5 6 that, if sent continuously, would cause the motor shaft to rotate. Figure 7.8 shows the driving signals from the port pins.

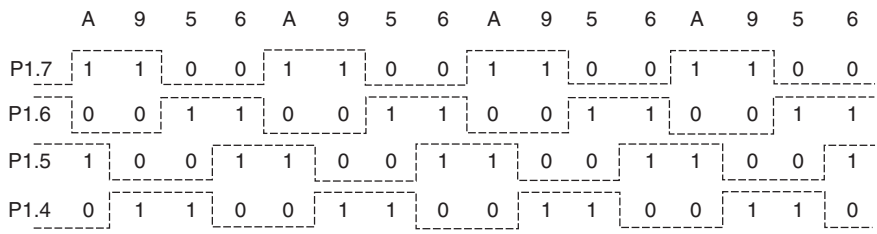


Figure 7.8 Signal sequence from the port pins to cause the stepper motor to rotate

Sending the code in reverse 6 5 9 A causes the motor shaft rotation to reverse. The rotation speed depends on the delay each step is held for. Details of a suitable drive circuit are shown in Figure 7.9.

The transistors (TR) must be chosen to easily handle the coil current. If the value of coil current is not given by the motor supplier then it is possible to measure the coil resistance with a multimeter (a typical value would be 15 ohms). Dividing V_{cc} by the coil resistance gives a good estimate of the coil current; double this value and select a transistor that has this current as its maximum-rated value. In this way the transistors will not run hot. The value of resistor R is chosen to control the input current of the transistor. The transistor current ratio is given in component catalogues as h_{FE}, which is device forward current gain in common-emitter mode. In this circuit h_{FE} is basically the coil current divided by the input current to the transistor. Thus the transistor input current is:

$$\text{input current} = (\text{coil current})/h_{FE}$$

The 74LS04 logic gate comprises eight inverter buffer circuits. Using two in series will restore the voltage level at the input to resistor R to the same value as the output from the relevant port pin. The voltage from the 74LS04 logic gate to turn on the transistor is 5 V. The voltage input to the transistor on the other side of the resistor is approximately 0.7 V; so the voltage difference across the resistor R is $(5 - 0.7)\,\text{V} = 4.3\,\text{V}$.

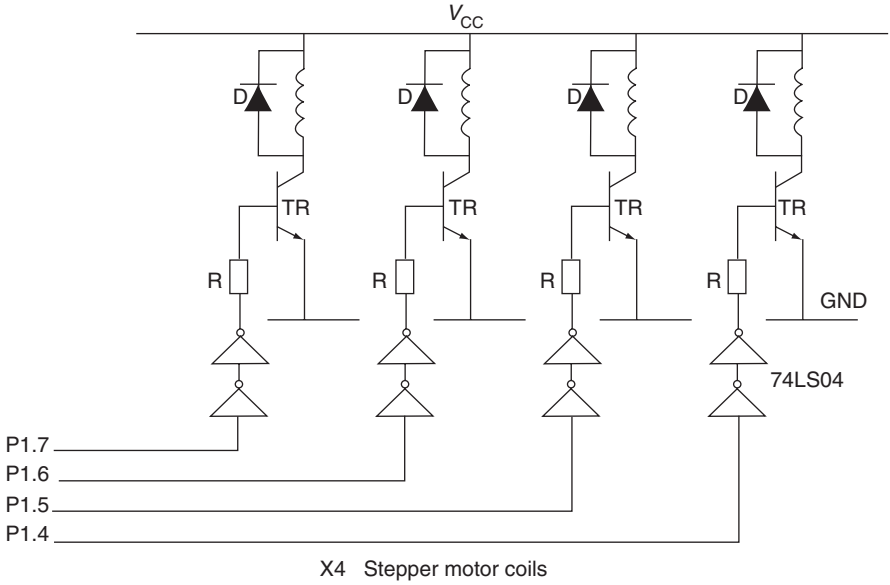


Figure 7.9 Suitable circuit arrangement to provide a drive for a stepper motor

The resistor value is given by:

$$R = (\text{voltage across the resistor})/\text{input current}$$
$$R = (4.3 \times h_{FE})/(\text{coil current})$$

A logic 1 (5 V) from the microcontroller port pin is applied through the two inverter gates of the 74LS04 to the resistor R. This sets up 0.7 V to the transistor base that causes the transistor to behave as an electronic switch, turning the device on and allowing current to flow through the coil. The logic gates act as a buffer ensuring that the microcontroller port pin is not current loaded. The diodes D reduce the large induced voltages caused when the current is suddenly switched on or off.

TIMER VALUES FOR ROTATION SPEED

Consider a 7.5 degree stepper motor having a step sequence of A 9 5 6. Assume it is desired to make the shaft rotate at 60 revolutions per minute or one revolution every second.

360/7.5 = 48 steps in a revolution and the program action will basically be step-delay; so this means 48 delays = one revolution.

$$48 \text{ delays} = 1 \text{ s}$$
$$1 \text{ delay} = (1/48) \text{ s} = 20.833 \text{ ms}$$

Suppose the microcontroller crystal frequency is 11.0592 MHz. If a P89C664 microcontroller is used the timer clock frequency is (11.0592/6) MHz.

Timer clock cycle time = 6/11.0592 MHz = 542.54 ns
Timer increments to roll-over = 20.833 ms/542.54 ns = 38400
Timer mode 1 base number = 65535 − 38400 = 27135 = 69FF hex

Similarly for a shaft speed of 40 revolutions per minute:

1 delay = 1.5/48 = 31.25 ms
Timer increments to roll-over = 57599
Timer mode 1 base number = 65535 − 57599 = 7936 = 1F00 hex
Let us assume: speed1 = 40 revs per minute, timer mode 1 base = 1F00H
TH0 = 1FH
TL0 = 00H
speed2 = 60 revs per minute; timer mode 1base = 69FFH
TH0 = 69H
TL0 = 0FFH

PROGRAM PLAN

When an interrupt occurs the program jumps to the interrupt routine and the SP is incremented. On completion of the interrupt the main program resumes its action and the SP assumes its prior value. It is possible for interrupts to occur within interrupt routines, these are called nested interrupts but even nested interrupts must be orderly and the SP must assume its prior value.

The SP (default value 07H) points to an address in RAM. If it were allowed to increase indefinitely without returning (RETI) to its prior value it would increment into higher RAM space possibly corrupting register values including the SFRs.

In project 1 the PWM program controlling the DC motor used a negative edge transition interrupt completing very quickly after the PWM rate was configured. Once set up the PWM signal was continually transmitted from the port 1 pin.

The stepper motor program is different in that the program continually applies the stepping sequence code. This time the interrupt is level sensitive and RETI is applied upon completion of the four-step sequence.

AT THE START

Pins 7, 6, 5, 4 of port 1 are turned off by MOV P1,#0FH so that current is not flowing through the coils. IT0 is not SETB and assumes its default value of zero making the external switched interrupt $\overline{\text{INT0}}$ level sensitive. Timer 0 is set to mode 1 by MOV TMOD,#01H making it a 16-bit timer. $\overline{\text{INT0}}$ interrupt is enabled with MOV IEN0,#81H.

CHECK

The three active low switches, which are also connected to pin 3.2 ($\overline{\text{INT0}}$), are checked.

The forward and reverse data sequences are given at the bottom of the program opposite labels FORWARD and REVERSE. The sequences are defined using DB (define byte). The last number in each sequence 0F0H is used to mark the end of the stepping sequence.

This program uses the data pointer (DPTR), which is a 16-bit register. It points to the first byte of the stepping data sequence using MOV DPTR, #REVERSE and MOV DPTR,#FORWARD. It may be incremented to the next byte with INC DPTR.

Program

```
$INCLUDE(REG66x.INC)              ; sfr addresses
         ORG   0                  ; reset address at 0000H
         SJMP  START              ; jump over reserved area
         ORG   03H                ; INT0 interrupt address
         SJMP  CHECK              ; jump to interrupt routine
         ORG   40H                ; program start address
START:   MOV   P1,#0FH            ; motor drives off
         MOV   TMOD,#01H          ; timer 0 in mode 1
         MOV   IEN0,#81H          ; INT0 interrupt enabled
STAY:    SJMP  STAY               ; stay till int. level changes
CHECK:   JNB   P1.0,REVERS        ; check for reverse
         JNB   P1.1,SPEED         ; check for speed1
         JNB   P1.2,SPEED         ; check for speed2
         SJMP  CHECK              ; keep checking switches
REVERS:  JB    P1.0,SPEED         ; jump next if rev not chosen
         MOV   DPTR,#REVERSE      ; dptr = reverse data address
         SJMP  NEXT               ; jump next if rev chosen
SPEED:   MOV   DPTR,#FORWARD      ; dptr = forward data address
NEXT:    MOV   A,#0               ; accumulator A = 0
         MOVC  A,@A + DPTR        ; data at dptr address into A
         CJNE  A,#0F0H,NEXONE     ; next data if not sequence end
         SJMP  LOOP               ; sequence end so goto RETI
NEXONE:  MOV   P1,A               ; data to Port 1
         JNB   P1.2,SPEED2        ; check if speed2 chosen
         ACALL DELAY1             ; if not then call DELAY1
         SJMP  OVER               ; and jump over DELAY2
SPEED2:  ACALL DELAY2             ; call DELAY2 for speed2
OVER:    INC   DPTR               ; increment data pointer
         SJMP  NEXT               ; repeat the data port loop
LOOP:    RETI                     ; check level active interrupt
DELAY1:  MOV   TH0,#1FH           ; DELAY1 high byte
         MOV   TL0,#00H           ; DELAY1 low byte
         SJMP  MISS               ; jump over DELAY2
DELAY2:  MOV   TH0,#69H           ; DELAY2 high byte
```

```
         MOV   TL0,#0FFH              ; DELAY2 low byte
MISS:    SETB  TR0                    ; turn timer 0 on
FLAG:    JNB   TF0,FLAG               ; stay till timer rolls over
         CLR   TR0                    ; turn timer 0 off
         CLR   TF0                    ; clear timer 0 flag
         RET                          ; return from subroutine
FORWARD: DB    0A7H,97H,57H,67H,0F0H ; forward data + stop
REVERSE: DB    67H,57H,97H,0A7H,0F0H ; reverse data + stop
         END                          ; end of assembly language
```

Simulation

Check interrupt

The port 1 and port 3 windows should be accessed; also the interrupt controller window can be viewed by selecting View on the main menu bar and then Hardware Peripherals. The result is shown in Figure 7.10.

As Figure 7.10 shows, a breakpoint is required on the line.

```
STAY:  SJMP  STAY
```

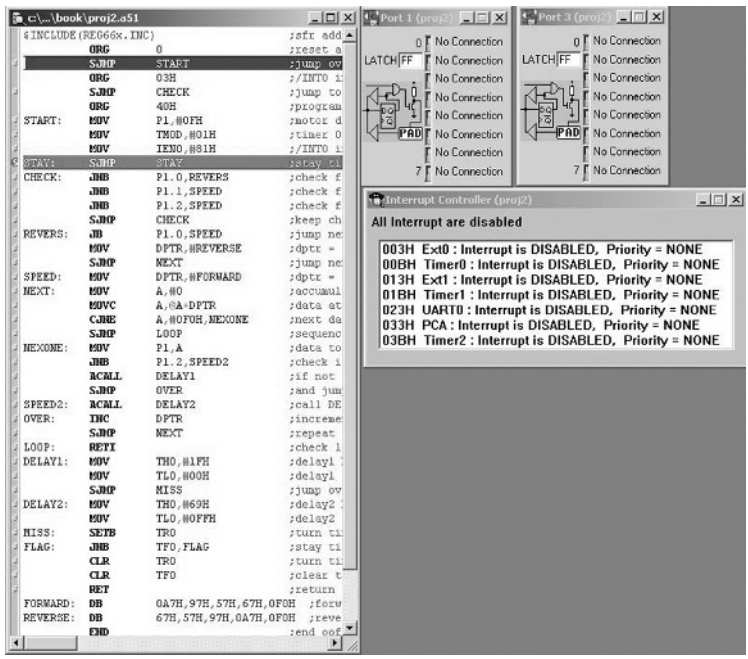


Figure 7.10 Simulation response showing project 2 program window and other relevant windows

Then selecting Animation (icon, two red characters in film) and then GO should indicate that the interrupt is enabled. Also the pins 7,6,5,4 on port 1 should go to logic 0 (see Figure 7.11).

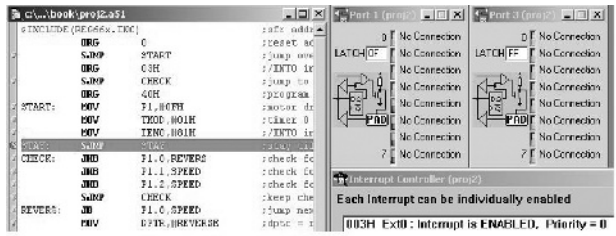


Figure 7.11 Simulation response indicating the port 1 and port 3 changes

Changing pin 2 on port 3 to ground activates the interrupt. Selecting GO should then see the program continually checking the level of the active low switches.

Checking the data pointer (DPTR) contents

The interrupt window should now be cancelled and the Main Registers window accessed by choosing View on the top menu bar and then Main Registers. The result is shown in Figure 7.12. Pin 1 (second one down) on port 1 should be set to ground to simulate the choice of Speed1 while pin 2 (third one down) on port 3 should be set to ground to cause a level zero interrupt.

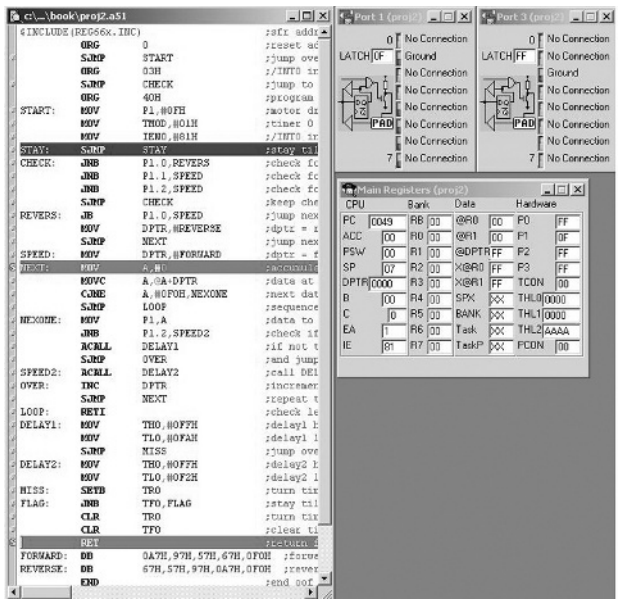


Figure 7.12 Simulation response with port 1, port 3 and main register windows and breakpoints set

A breakpoint is required on the line NEXT: MOV A,#0 (this is after the line that loads the DPTR with the FORWARD address). Another breakpoint is

required on the line RET. This is one byte before FORWARD. It is not possible to put a breakpoint on the FORWARD line because it does not have an instruction; DB (define byte) is a directive.

These changes are illustrated in Figure 7.12. Also Figure 7.12 gives plenty of information in the main registers window. As indicated:

Stack pointer (SP) default value of 07H
Accumulator A value
Values on the Ports
TH and TL (THL) values in Timers 0, 1 and 2
Program counter (PC) with current program address
Data pointer (DPTR) contents, default address is 0000H

Resetting the simulation (CLM on icon with finger pointing to red button) and, without using Animation, CLM on GO causes the program to run to the first breakpoint. At this point DPTR shows 0090 which is the hex address of the first byte 0A7H on the FORWARD: line.

Pressing GO again results, after a short interval, in the program reaching the second breakpoint. The PC should show 008F, one byte before 0090 the address in the DPTR. At this juncture look again at the information contained in the main registers window, which will show information set by the program. It will not show port 1, pin 1 and port 3, pin 2 at ground because these were set (simulated) by external hardware. It should show port 1 going to 0F because this was the action of the first program line MOV P1,#0FH.

At reset TCON SFR assumed its default value of 00.

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

At the first breakpoint TCON is 02 showing that IE0 had set to 1, indicating that an interrupt had occurred.

At reset the SP = 07 pointing to address 0007H in onboard RAM. At the first breakpoint SP had changed to 09, pointing to address 0009H in onboard RAM indicating that an address had been stored at locations 0007H, 0008H.

The contents of the onboard RAM may be checked by selecting from the top menu bar:

View− > Data dump . . . − > Data View

The result is shown in Figure 7.13. From Figure 7.13 it can be seen that the RAM contents at specific locations are:

Location 0007 = 00, location 0008 = 49

0049H is the address of the program line STAY: SJMP STAY, this is the address the program returns to when the interrupt has completed.

Pressing GO again, so that the program runs to the next breakpoint, and checking the SP, shows that it has increased to 0B. Checking the RAM again

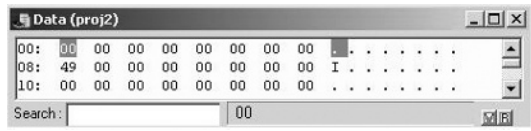


Figure 7.13 Window showing data stored in RAM for project 2 at specified breakpoint

gives the result indicated in Figure 7.14. From Figure 7.14 it can be seen that the RAM contents at specific locations are:

$$\text{Location } 0009 = 00, \text{ location } 000\text{A} = 70$$

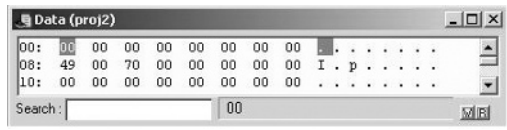


Figure 7.14 Window showing data stored in RAM for project 2 at a subsequent specified breakpoint

0070H is the address of SJMP OVER, the next line after the first call. Placing a breakpoint there can check this.

Measuring the step time delay

The Animation should be off and the existing breakpoints removed. P1.0 and P1.1 should be kept at ground. A breakpoint should be placed on either side of the program line FLAG: JNB TF0,FLAG in the time delay routine. This is indicated in Figure 7.15.

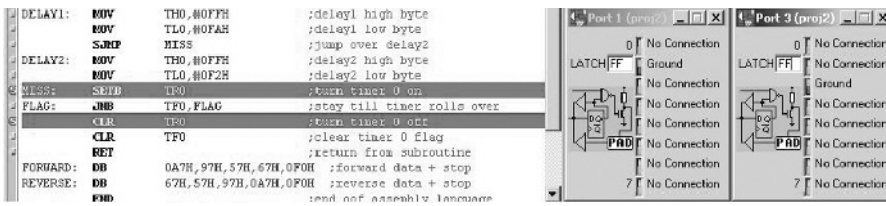


Figure 7.15 Simulation window showing the breakpoints to be used for measuring the step time delay

Pressing reset and then GO will cause the program to run to the first breakpoint. The timing panel at the bottom right of the screen will have some value as shown in Figure 7.16(a).

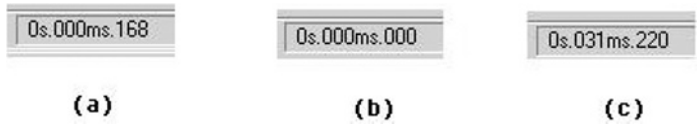


Figure 7.16 (a) Possible initial timing value at the first breakpoint. (b) Putting zero in the timing panel. (c) Time to the next breakpoint giving the step time response for a step motor speed of 40 rpm

Pressing Ctrl T will zero the value in the timing panel as shown in Figure 7.16(b), while pressing GO will cause the program to run to the next breakpoint when the panel should show the duration of the step time delay. See Figure 7.16(c). By calculation the time delay was 31.250 ms; as shown in Figure 7.16(c) the simulation value is very close to the calculated value.

If now P1.1 is changed to V_{cc} and P1.2 to ground and the above procedure repeated, a step time value as shown in Figure 7.17 results. From Figure 7.17 the step time delay is 20.814 ms, which is close enough to the calculated value to be acceptable.

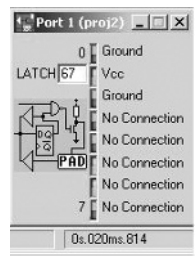


Figure 7.17 Step time response for a step motor speed of 60 rpm

Stepping signal patterns

First of all the existing breakpoints should be removed. The actual time delays are too long to quickly generate signal traces, so it is suggested that TH0 is changed to #0FFH for Delay1 and Delay2 and that TL0 is changed to #0FAH for Delay1 and #0F2H for Delay2; the signal patterns will be the same.

The Watches window should then be set up using View → Watch and the signals P1.4, P1.5, P1.6, P1.7 added; the signals should also be added to the Trace. The Trace window should then be accessed using View → Trace → View and the Trace options set using View → Trace → Options. The Mode should be set to Continual while the Maximum number of records should be set to 500 and rolling trace should be ticked.

Running the Trace with Animation on should show the signals setting up. Running without Animation and stopping after several seconds shows the complete pattern. This is illustrated in Figure 7.18.

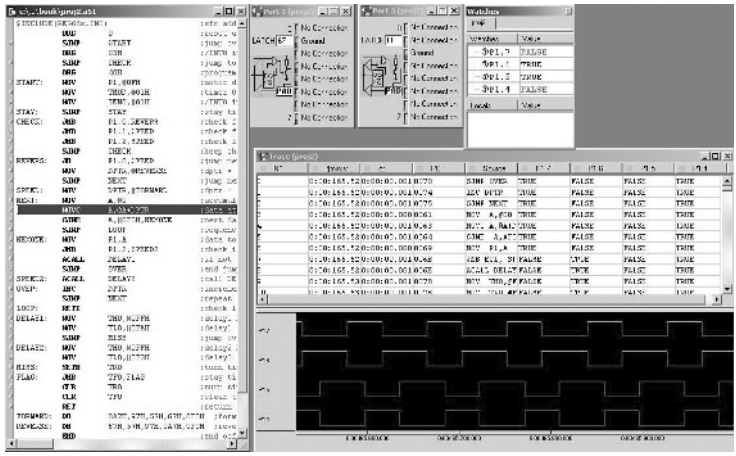


Figure 7.18 Simulation response showing trace patterns on specified port pins for project 2

PROGRAM DEVELOPMENTS

1. Check that the program drives the stepping motor when Reverse is chosen but Speed1 and Speed2 are not.
2. Modify the program so that reverse only occurs when one of the speeds are chosen.
3. Assuming four switches are connected to port 2 modify the program so that the motor shaft stops after completion of 1 to 10 complete revolutions.

7.4 Project 3: single wire multiprocessor system

Serial master/slave buses exist either as onboard peripherals or as a separate chip set. The P89C66x microcontroller family has the I^2C bus as an onboard peripheral, the principles of which have been explained in Chapter 4 (see also Appendix D) and an example given. The Philips LPC932 microcontroller has the SPI bus as well as the I^2C.

For I^2C systems the slave device is a special chip that is able to return an acknowledge signal. The SPI slaves can be special devices, but non-special chips employing serial shift registers to move data in and out can also be used.

The I^2C bus is a two-wire system whereas in its simple mode the number of wires required for an SPI bus depends on the number of slave devices. This project presents a single wire bus system where the Master and Slaves are all microcontrollers using their UARTs in mode 3, which is multiprocessor mode. An example having two slaves is represented by Figure 7.19. The receive (Rx) and transmit (Tx) pins of all the microcontrollers are connected together and data can be shifted out of the master or into it but not at the same time.

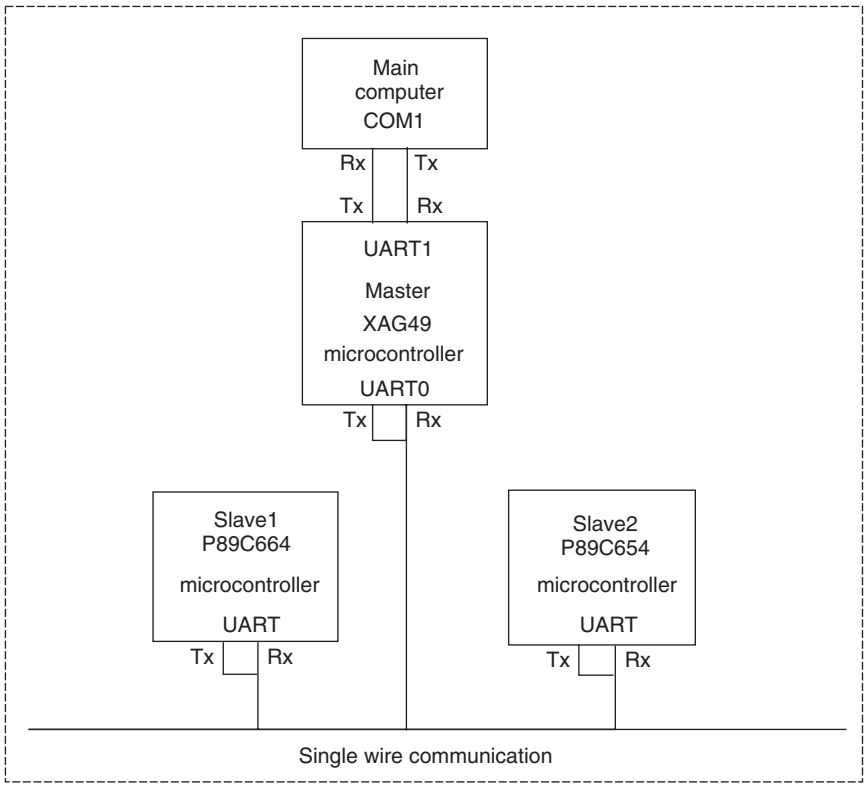


Figure 7.19 Block diagram of a single wire communication system

The UART has four operation modes i.e. 0, 1, 2 and 3. Modes 0 and 1 are used between two devices. Mode 0 is a fixed transfer rate dependent on the microcontroller crystal frequency. Mode 1 is a variable transfer rate and can be set at various baud rates; this example uses a baud rate of 9600 bits per second. Modes 2 and 3 are used between a master device and multiple slave devices. Mode 2 is the fixed transfer rate and mode 3 is the variable baud rate. In theory there could be 256 slaves (i.e. 8-bit address) but in practice too many slave devices would cause loading effects.

If this system uses the same microcontroller type for master and slaves then mode 2 would be preferable to mode 3 since it would not be necessary to program timers for baud rate generators. This example has an XA as a master and two slave devices using the P89C664; hence mode 3 is to be used. A communication protocol exists for data transfer between master and slave devices as shown in Figure 7.20.

Details of the Serial Control (SCON) register are:

Serial control (SCON) register

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Mode 3 SM0 = 1	SM1 = 1
REN	Enables serial reception
TI	Transmit interrupt flag
RI	Receive interrupt flag

SM2 = 1 Multiprocessor (master with all slaves) communication
SM2 = 0 Dual (master with addressed slave) communication

TB8 = 1 Transmitted bit 8 (all slaves can be interrupted, provided SM2 = 1)
TB8 = 0 Only addressed slave is interrupted because its SM2 = 0

RB8 = 1 Transmitted bit (TB8 = 1) is received into slave

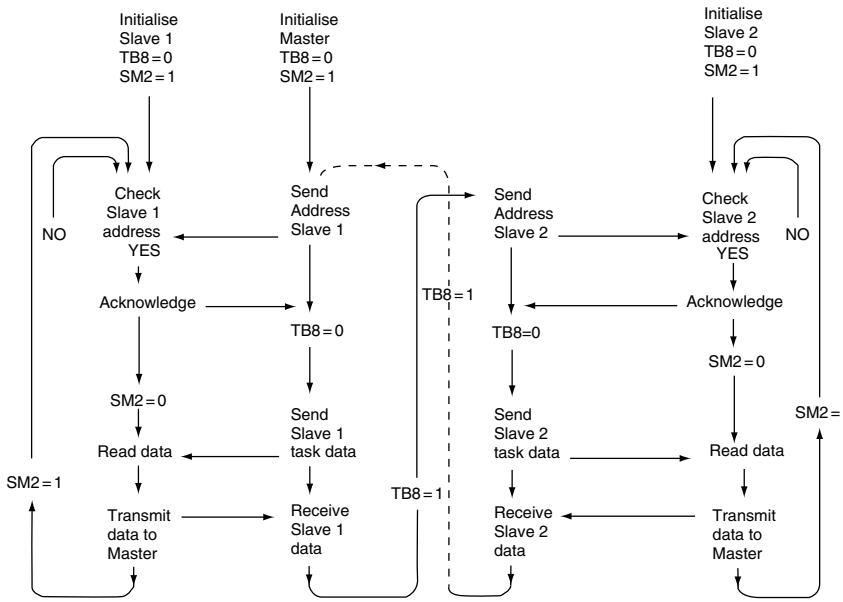


Figure 7.20 Communication protocol for master/slave communication

PROGRAM PLAN

The main controlling bits are SM2 and TB8. SM2 = 1 enables multiple processor communication. Initially the master device and all the slave devices are set up for multiple communication. When the master sets TB8 = 1 it interrupts all the slave devices.

In this first interrupt the master sends through its UART the address of the slave it wishes to communicate with. The chosen slave acknowledges and clears its SM2 = 0. The master clears its TB8 = 0. The remaining slaves continue with

SM2 = 1. In this condition any further UART interrupts from the master are only received by the chosen slave whose SM2 = 0.

A slave device cannot be serially interrupted if its SM2 = 1 and the Master TB8 = 0. The master now interrupts the chosen slave and sends data to it. Figure 7.20 shows that the chosen slave may return data to the master, although in this example it is not the case.

When communication between the master and slave is complete the slave sets SM2 = 1 and the master sets TB8 = 1, resuming the condition for multiple communication with all the slave devices. The master then seeks to address the next slave device.

In this example both slave processors are each driving a stepper motor, the master sends data that alters the step hold delay and in this way varies their rotation speed.

As suggested by Figure 7.19 the system information to the master could come from a host PC. Since the XA processor has two UARTs, one of them could be used to communicate with the host PC whilst the second could be used to interface with the slave processors.

It is possible in a complex system for the P89C664 slaves to communicate with each other via their I^2C connections since the I^2C system is capable of multimaster communications. In this example the slave data has been arbitrarily chosen and originates from the master device.

Master program

```
$INCLUDE(REGXAG49.INC)          ; sfr addresses
          ORG     0             ; reset address
          DW      8F00H,START   ; define word hex8F00
          ORG     120H          ; program start address
START:    MOV.B   WDCON,#0      ; watchdog control off
          MOV.B   WFEED1,#0A5H  ; watchdog feed1
          MOV.B   WFEED2,#5AH   ; watchdog feed2
          MOV.B   S0CON,#0F8H   ; mode 3 multi-processor
          MOV.B   TMOD,#20H     ; timer 1 into mode 2
          MOV.B   RTL1,#238     ; set for 9600 baud
          MOV.B   TL1,#238      ; set for 9600 baud
          SETB    TR1           ; turn timer 1 on
LOOP:     MOV.B   R4L,#01H      ; slave 1 address into R4L
          CALL    ADDRESS       ; send slave 1 address
          MOV.B   R4L,#33H      ; slave 1 data into R4L
          CALL    SLVDATA       ; send slave 1 data
          MOV.B   R4L,#02H      ; slave 2 address into R4L
          CALL    ADDRESS       ; send slave 2 address
          MOV.B   R4L,#66H      ; slave 2 data into R4L
          CALL    SLVDATA       ; send slave 2 data
          JMP     LOOP          ; repeat
ADDRESS:  SETB    TB8           ; set to interrupt all slaves
          JMP     SEND          ; jump to SBUF
SLVDATA:  CLR     TB8           ; communicate with chosen slave
SEND:     MOV.B   S0BUF,R4L     ; address or data into UART SBUF
```

```
WAITTI:   JNB     TI,WAITTI     ; transmit contents of SBUF
          CLR     TI            ; clear TI to enable repeat
WAITRI:   JNB     RI,WAITRI     ; await acknowledge from slave
          CLR     RI            ; clear RI to enable repeat
          RET                   ; return from transmit routine
          END                   ; end of assembly language
```

Simulation

Having entered and compiled the program the simulation response is shown in Figure 7.21. With the Animation on and CLM on GO, the simulation will run until it tests the Receive Interrupt (RI), which is the acknowledgement back from the slave. CLM on Stop (same icon as for GO).

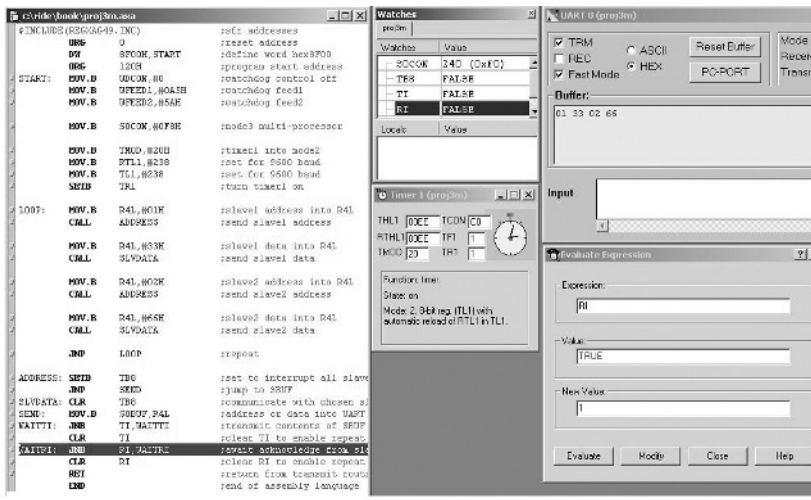


Figure 7.21 Master simulation for project 3

Positioning the cursor on RI in the watches window and CRM will cause a small window to appear. Choosing Evaluate will produce an Evaluate Expression window for RI. If 1 is typed into the New Value space and CLM on Modify then RI in the watches window will change to True.

The Evaluate Expression window should be left in place to enable the simulation to get past the RI test point. CLM on GO will continue the simulation. If the S0CON register is checked when the TB8 value changes, it should equal 1 for slave addresses and be equal to 0 for slave data.

Slave program

```
$INCLUDE(REG66x.INC)          ; sfr addresses
          ORG   0             ; reset address
          SJMP  START         ; jump to start
          ORG   23H           ; UART interrupt address
```

```
            SJMP    SERIAL          ; jump to interrupt
            ORG     40H             ; program start
START:      MOV     SCON,#0F0H      ; mode3, SM = 1
            MOV     IEN0,#90H       ; enable UART interrupt
            MOV     TMOD,#21H       ; timer1 mode2, timer0 mode1
            MOV     TH1,#0FAH       ; timer1 baudrate 9600
            MOV     TL1,#0FAH       ; TL1 also initially set
            SETB    TR1             ; turn timer1 on
            MOV     A,#0FAH         ; initial accumulator value
;stepper motor
MOTOR:      MOV     P1,#0A7H        ; step1 = A
            ACALL   DELAY           ; step hold delay
            MOV     P1,#97H         ; step2 = 9
            ACALL   DELAY           ; step hold delay
            MOV     P1,#57H         ; step3 = 5
            ACALL   DELAY           ; step hold delay
            MOV     P1,#67H         ; step4 = 6
            ACALL   DELAY           ; step hold delay
            SJMP    MOTOR           ; back to step1
;step hold delay
DELAY:      MOV     TH0,A           ; main delay value from Master
            MOV     TL0,#0FFH       ; LSByte delay value
            SETB    TR0             ; turn timer0 on
FLAG:       JNB     TF0,FLAG        ; stay till timer0 roll-over
            CLR     TR0             ; turn timer0 off
            CLR     TF0             ; clear timer0 flag
            RET                     ; return from delay
; UART interrupt from Master
SERIAL:     CLR     RI              ; clear RI, set by interrupt
            MOV     A,SBUF          ; SBUF contents into A
            CJNE    A,#01H,OTHER    ; RETI if Master not selects #01H
            CLR     SM2             ; SM2 = 0, leave multiproc comm
WAITRI:     JNB     RI,WAITRI       ; next interrupt will be data
            CLR     RI              ; clear RI, set by interrupt
            MOV     A,SBUF          ; SBUF data into A
            SETB    SM2             ; SM2 = 1, back to multiproc comm
OTHER:      RETI                    ; return from interrupt
            END                     ; no more assembly language
```

Simulation

The simulation response for this program is shown in Figure 7.22. The timer 0 and timer 1 windows are accessed; timer 0 for the step hold delay and timer 1 for the serial baud rate generator. In the watches window SCON, SM2, RI, TI, A, SBUF have been inserted. The evaluate expression window has been set up for SBUF and RI, from the Watches window. The port 1 window has been accessed, as has the interrupt controller window, which has been resized.

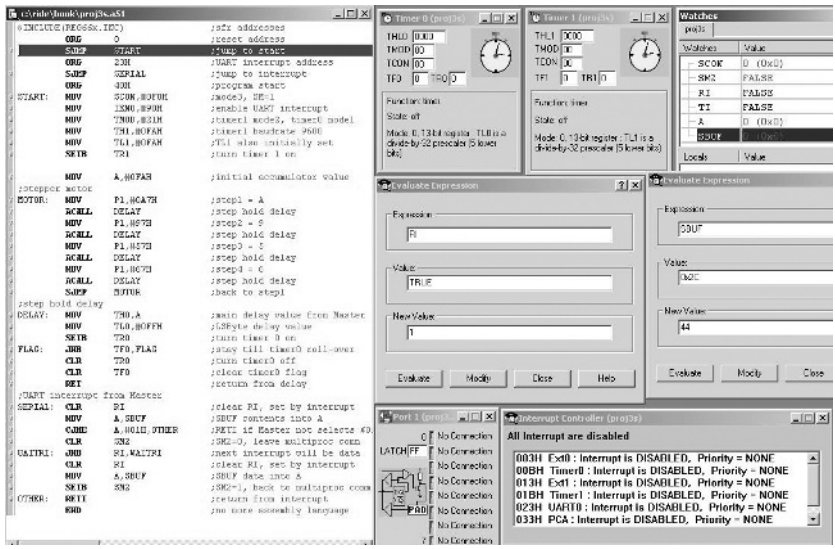


Figure 7.22 Slave simulation for project 3

Putting the Animation on and CLM on GO will cause the simulation to put A7H (1010 0111) onto port 1. While viewing the simulation on the PC screen remember that green represents logic 1. The simulation will stick in the step hold delay whilst Timer 0 increments up from FAFFH towards FFFFH and roll-over.

Halting the animation by CLM on Stop and entering 0x01 as a new value in the SBUF evaluate expression window and then CLM on Modify will alter the value in SBUF. This could be confirmed by checking in the watches window that SBUF is 0x01, which is the address of the first slave. Putting 1 as a New Value in the RI evaluate expression window and CLM on Modify will cause the Slave to interrupt when the simulation continues. In the THL0 window of Timer 0 the number may be edited, using the PC cursor and keyboard, to FFFA so that it is close to roll-over. See Figure 7.23.

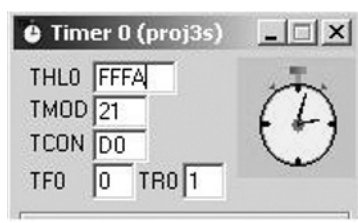


Figure 7.23 Timer 0 window, used to alter the value in THL0

The simulation should be continued by single stepping. Because RI is set the slave is interrupted and the simulation cursor will jump to the interrupt vector address at 0023H. The simulation will then jump to the UART interrupt. RI will

be cleared and the contents of the SBUF (the slave address) transferred to accumulator A. The slave address is checked at the third interrupt program line. If it is not the address sent by the master to the SBUF then the slave will jump to RETI and leave the communication with the master. In this example the slave program is that for address #01H.

If single stepping is continued the program should clear SM2 to 0 and the slave should come out of multiprocessing mode and have single communication with the master. The simulation should stick at WAITRI: awaiting the next interrupt from the master, which will convey the step, hold delay data.

Modifying the new value in the SBUF evaluate expression window to 0x33 and then modifying the new value in the RI evaluate expression window to 1 and continuing single stepping will cause 0x33 to be transferred from the SBUF into accumulator A and the simulation will leave the interrupt.

The stepper motor hold delay should now have changed; the timer 0 base number has changed to THL0 = 33FF, TH0 = 33H and TL0 = FFH. The value in TH0 can be changed by the master.

PROGRAM DEVELOPMENTS

1. What difference is required to write the program for slave 2?
2. How would the simulation be different?
3. Modify slave 1 program such that the slave returns data to the master. The data could be number of revolutions completed since last communication.
4. What limitations would there be on this return data?
5. Modify the master program such that the slave data comes from a host PC via the other XA UART.

7.5 Project 4: function generator

The requirement is to design a function generator, using the P87LPC769 microcontroller, with the minimal amount of external components, to generate sine, square and sawtooth waveforms. The output of the circuit is not designed to source an output current to the circuits under test and a buffer circuit is required to enhance the current sourcing capability and also provide a low output resistance for the function generator. A block diagram of a possible circuit arrangement is shown in Figure 7.24.

PROGRAM PLAN

To generate the required waveforms, the P87LPC769's DAC0 is configured as a digital to analogue converter by disabling the ADC and enabling DAC0. This could be achieved by clearing ADCI (A/D conversion complete flag), ADCS (A/D conversion start flag) and ENADC (enable ADC).

The DAC0 is enabled by disabling P1 (DAC1 = P1.6 and DAC0 = P1.7) pins digital output and setting them to input only (Hi Z). This achieved by

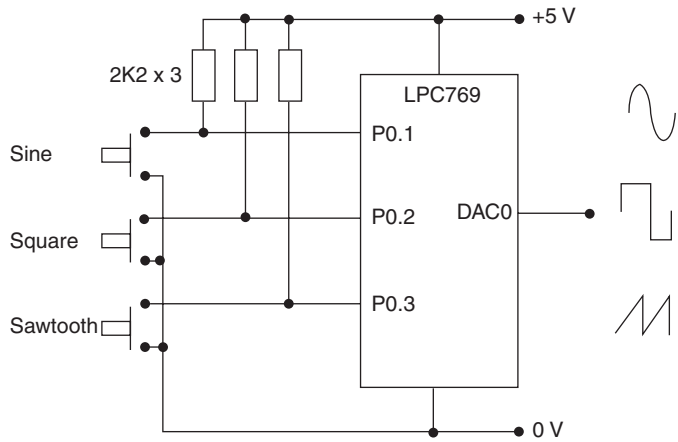


Figure 7.24 Block diagram of function generator using a P87LPC769 microcontroller

using P1M2 and P1M1 registers and setting their appropriate pin to 0 and 1 respectively. Finally the DAC0 should be enabled by setting the ENDAC0 to 1.

For the generation of the waveforms, the sine wave requires most of the work. For this a set of 180 sine wave values, scaled to 255 (8-bit register), is generated and is put into the DAC0 one at a time. The square wave only uses two values of 255 (5 V) and 0 (0 V). In this program to generate equal time of logic 1 and logic 0, the value of 255 is written 3 times and 0 twice. Running a simple 'for' loop and inputting the values of the loop into the DAC0 generates the sawtooth waveform.

Program

```
/*************************************************************
* Chapter seven :PROJECT s                                   *
* 87LPC769 Function generator                                *
* June 2003                                                  *
*                                                            *
* This program generates Sine, Square and Sawtooth waves on  *
* the DAC0 of the P87LPC769 microcontroller                  *
*************************************************************/
#include <REG769.H>
sbit SineKey     = P0^1;                  /* press this key to generate Sine */
sbit SquareKey   = P0^2;                /* press this key to generate Square */
sbit SawtoothKey = P0^3;             /* press this key to generate Sawtooth */
code unsigned char Sine[] = {                          /* Sine values */
  127,131,136,140,145,149,153,158,162,166,170,175,
  179,183,187,191,194,198,202,205,209,212,215,218,
  221,224,227,230,232,235,237,239,241,243,245,246,
  248,249,250,251,252,253,253,254,254,254,254,254,
```

```
  253,253,252,251,250,249,248,246,245,243,241,239,
  237,235,232,230,227,224,221,218,215,212,209,205,
  202,198,194,191,187,183,179,175,170,166,162,158,
  153,149,145,140,136,131,127,123,118,114,109,105,
  101, 96, 92, 88, 84, 79, 75, 71, 67, 64, 60, 56,
   52, 49, 45, 42, 39, 36, 33, 30, 27, 24, 22, 19,
   17, 15, 13, 11, 9,  8,  6,  5,  4,  3,  2,  1,
    1,  0,  0,  0,  0,  0,  1,  1,  2,  3,  4,  5,
    6,  8,  9, 11, 13, 15, 17, 19, 22, 24, 27, 30,
   33, 36, 39, 42, 45, 49, 52, 56, 60, 63, 67, 71,
   75, 79, 84, 88, 92, 96,101,105,109,114,118
};
/*************************************************************
* START of the PROGRAM                                       *
************************************************************* /
void main (void) {
 unsigned char i;
/*************************************************************
* Disable the A/D Converter because of DAC0 AND              *
* Enable the D/A Converter.                                  *
************************************************************* /
   ADCI = 0;              /* Clear A/D conversion complete flag */
   ADCS = 0;                 /* Clear A/D conversion start flag */
   ENADC = 0;                      /* Disable the A/D Converter */
/*************************************************************
* Disable P1, DAC pins digital Outputs and set the           *
* DAC1 = P1.6, DAC0 = P1.7 to Input Only (Hi z)              *
************************************************************* /
   P1M2 & =~ 0xC0;                    /* Set Pins for Input Only */
   P1M1 |= 0xC0;                                /*P1M2 = 0 & P1M1 = 1*/
/*************************************************************
* Enable the D/A Converter                                   *
************************************************************* /
   ENDAC0 = 1;                                    /* Enable DAC0 */
/*************************************************************
* Create the waveforms on DAC0                               *
************************************************************* /
   while(1){                                     /* Run for ever */
            while ( !SquareKey ) {     /* if key pressed Run this */
                    DAC0 = 255;    /*-------------------------*/
                    DAC0 = 255;    /* generate 5 volts, 3 times and
                                      0 volts */
                    DAC0 = 255;   /*once, since while() statement
                                    adds */
                    DAC0 = 0;                  /* to the 0 volts. */
                    DAC0 = 0;       /*-------------------------*/
            }
            while ( !sawtoothKey )   { /* if key pressed Run this */
              for(i = 0; i < 255; i++)
              DAC0 = i;
```

```
        }
        while ( !SineKey )  {              /* if key pressed Run this */
           for(i = 0; i < 179; i++)
           DAC0 = Sine[i];
    }                                                    * while(1) */
}                                                         /* main() */
}
```

Simulation

The best way to simulate this program is to use Keil software since it contains a register for DAC0. For the simulation the values of P0.1, P0.2 and P0.3 should be set to zero for sine, square and sawtooth generation. If more than one is set to zero, the last waveform continues to be generated. During debugging the DAC0 window should be displayed using peripherals and then D/A from the debug menu. Also port 0 should be displayed, so that the port pins P0.1, P0.2 and P0.3 can be set and reset.

After opening the Keil software a project should be created. A C file should be added to the project and the project compiled. Details are shown in Figure 7.25.

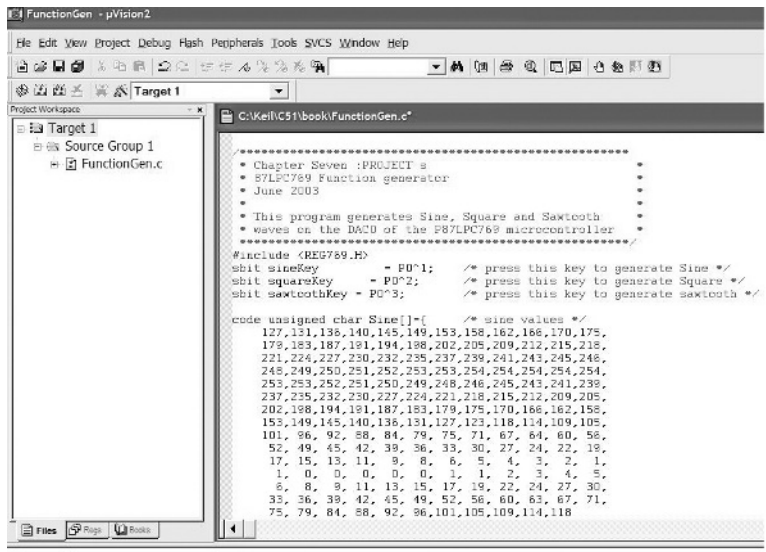


Figure 7.25 Keil µ Vision2 Simulation window

Using the debugger, the DAC0 and port 0 registers should be displayed. Initially P0.1 should be set to zero and the program stepped through. The value of the DAC0 will change as the sine wave is generated. See Figure 7.26.

Resetting the microcontroller and setting pin P0.2 to logic zero should run the square-wave section of the program. A value of 4.9902 V can be observed in the capture, shown in Figure 7.27, which represents logic 1.

Finally resetting the controller and set pin P0.3 to logic zero should generate a sawtooth waveform on the DAC0 output. See Figure 7.28.

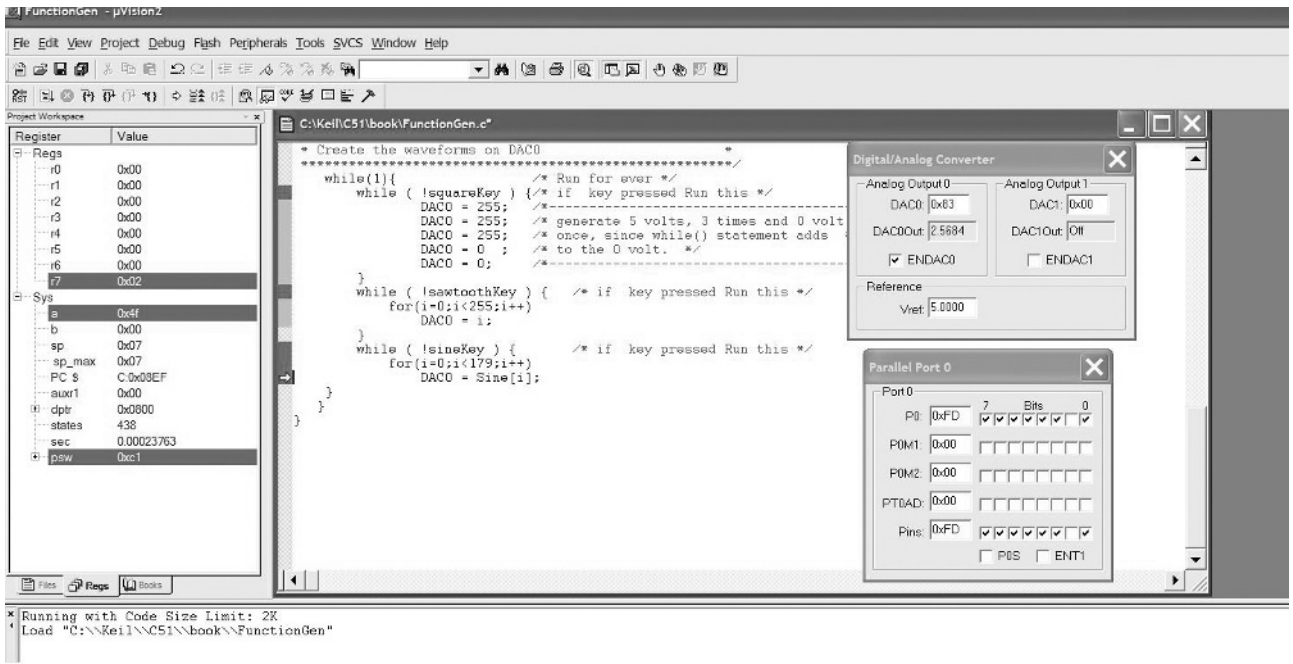


Figure 7.26 Simulation window, with the DAC0 and port 0 windows added, for sine waveform generation

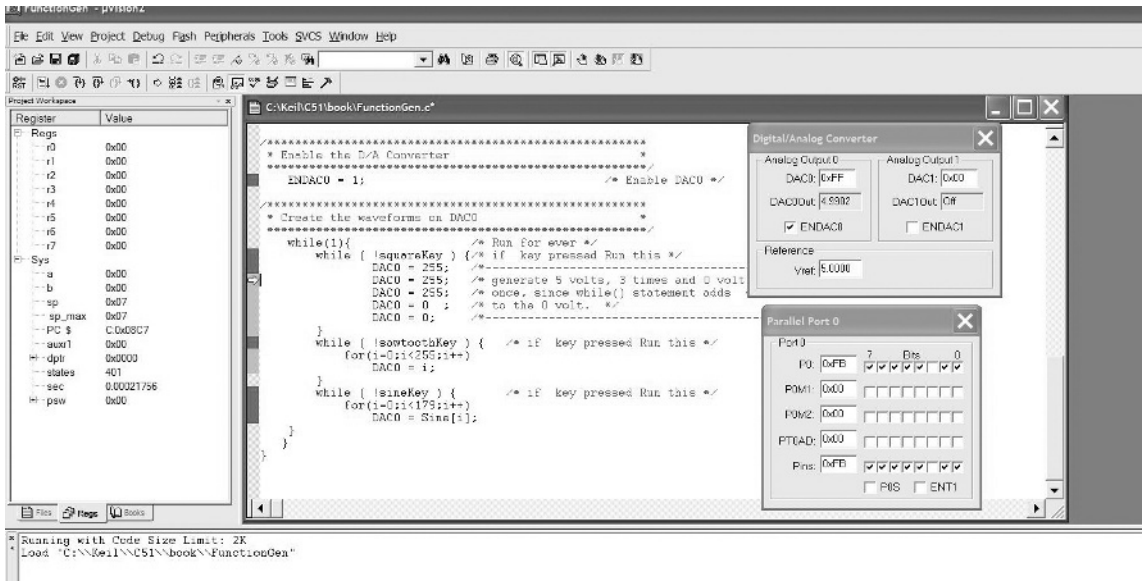


Figure 7.27 Simulation window, with the DAC0 and port 0 windows added, for square-waveform generation

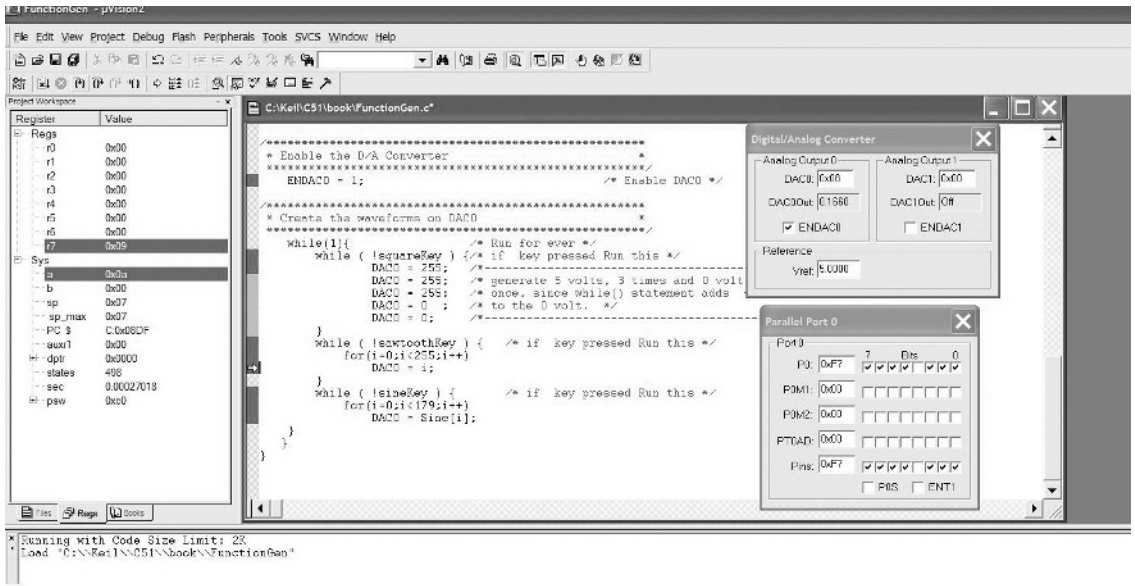


Figure 7.28 Simulation window, with the DAC0 and port 0 windows added, for sawtooth waveform generation

Waveforms generated by the circuit and shown on an oscilloscope are shown in Figure 7.29. The oscilloscope settings for the amplitude and timebase for each waveform were as follows:

Sine wave: 1V/cm and 1 ms/cm
Square wave: 1 V/cm and 2 μs/cm
Sine wave: 1 V/cm and 1ms/cm.

PROGRAM DEVELOPMENTS

1. Modify the software and hardware so that the frequency could be increased or decreased. Use two port pins, one for increasing and one for decreasing the frequency.
2. Use two more port pins to adjust the amplitude of the waveforms.
3. Add more waveforms of your choice to the project e.g. a triangular waveform.

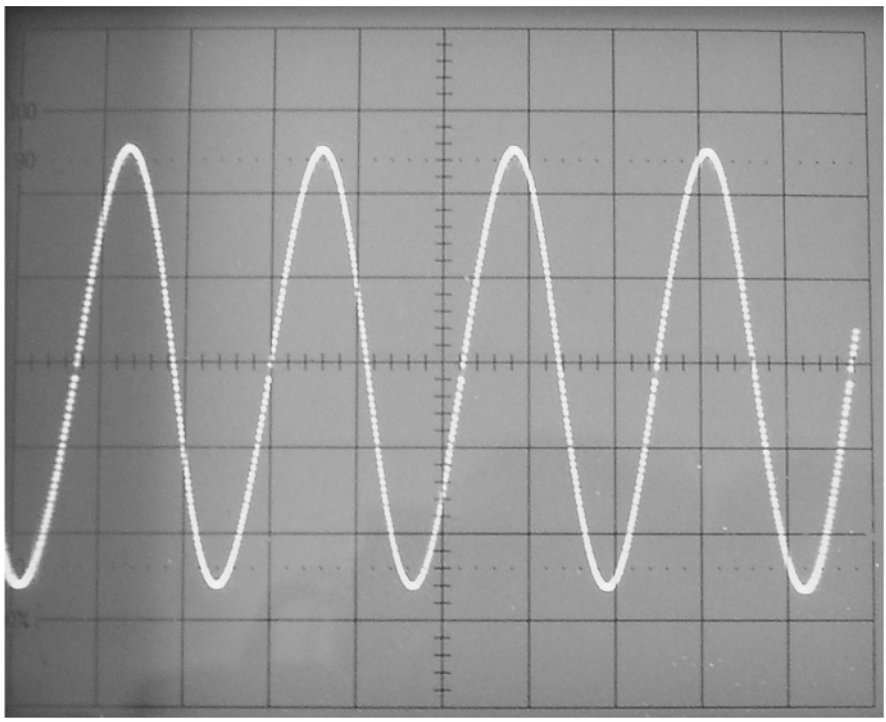


Figure 7.29(a) Function generator output response for sine waveform generation

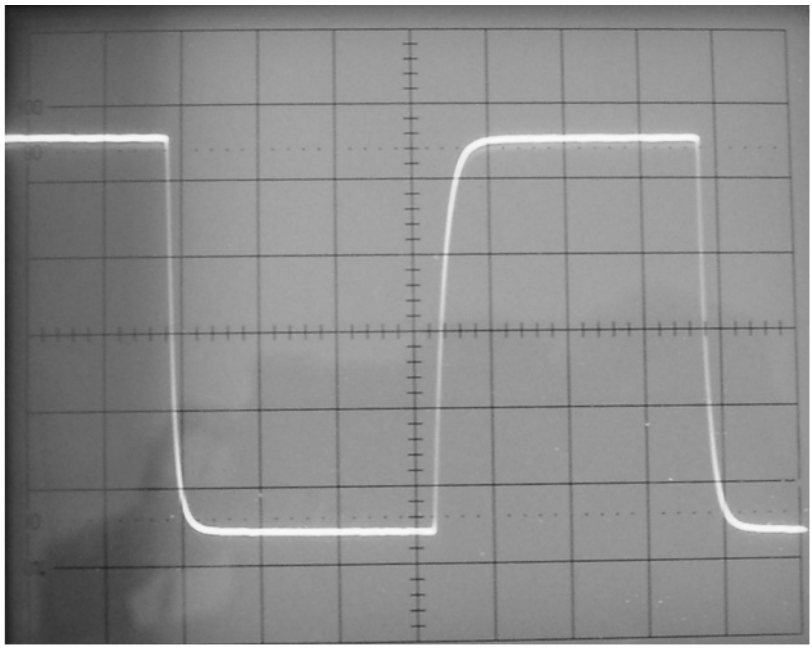


Figure 7.29(b) Function generator output response for square-waveform generation

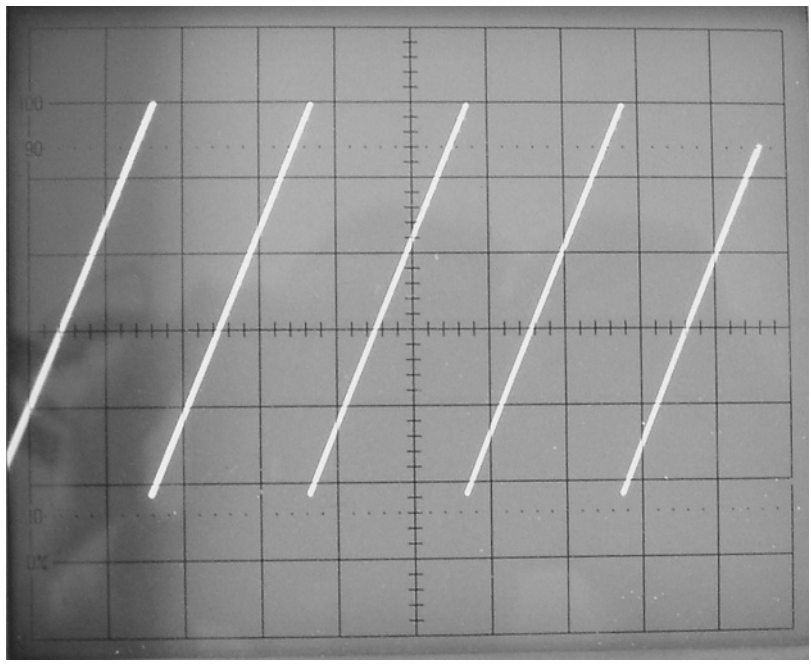


Figure 7.29(c) Function generator output response for sawtooth waveform generation

Solutions to Exercises

Chapter 1

EXERCISE 1.1

1023

EXERCISE 1.2

15
$0 \times 128 + 0 \times 64 + 0 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1$
$15 = 00001111$

250
$1 \times 128 + 1 \times 64 + 1 \times 32 + 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1$
$250 = 11111010$

EXERCISE 1.3

200
Answer $200/16 = 12$ remainder 8
$200 = $ C8 Hex

EXERCISE 1.4

(1) $10000110 = 1000\ 0110$
86 Hex
$8 \times 16 + 6 \times 1 = 134$ Decimal

(2) $10011000011 = 0100\ 1100\ 0011$
4 C3 Hex
$4 \times 256 + 12 \times 16 + 3 \times 1 = 1219$ Decimal

EXERCISE 1.5

No, there is no difference between the two instructions in terms of results. They both add '1' to the register A.

EXERCISE 1.6

No, there is no difference between the two instructions in terms of results. They both take '1' from the register A.

EXERCISE 1.7

A = 00 Hex
B = 5E Hex

EXERCISE 1.8

A = 17 Hex
B = 01 Hex

EXERCISE 1.9

A = 2D H = 00101101 B & 3B H = 00111011

0010 1101 And Logical
0011 1011

0010 1001
A = 0010 1001 B = 29 H = 41 Dec

EXERCISE 1.10

R0 = 38 H = 00111000 B & 9A H = 10011010

0011 1000 OR Logical
1001 1010

1011 1010
R0 = 1011 1010 B = BA H = 186 Dec

EXERCISE 1.11

P0 = 125 Dec = 01111101 B

10000010 Complement

P0 = 1000 0010 B = 82 H = 130 Dec

EXERCISE 1.12

A = 128 Dec = 10000000 B & B = 2 Dec = 00000010 B

RR A A = 0100 0000 & B = 0000 0010
RL B A = 0100 0000 & B = 0000 0100

RR A A = 0010 0000 & B = 0000 0100
RR A A = 0001 0000 & B = 0000 0100
RL B A = 0001 0000 & B = 0000 1000

Therefore A = 16 Dec & B = 8 Dec

EXERCISE 1.13

With reference to Figure exercise 1.13

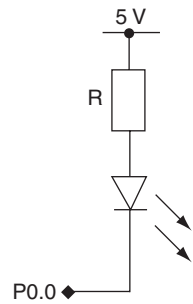


Figure exercise 1.13

$$R = (5\,V - V_f)/I_f = (5\,V - 0.7\,V)/10\,mA = 4.3\,V/10\,mA = 430\,\Omega$$

EXERCISE 1.14

MOV PSW, #18H

EXERCISE 1.15

ACALL calls up a subroutine; the subroutine must always have RET as its last operation. ACALL range is limited to +127 places forward or −128 places backward.

AJMP, similar to ACALL, jumps to addresses, which has a similar range but no return from it.

EXERCISE 1.16

JNC The program jumps to a relative position in the program if carry is 0.

JNB The program jumps to a relative position in the program if specified bit = 0.

EXERCISE 1.17

Clock frequency = 11.0592 MHz
Therefore clock cycle = 1/11.0592 MHz = 90.423 ns

Signal frequency = 20 kHz
Therefore signal cycle = 1/20 kHz = 50 μs

Delay = 25 μs = (54 + 12 × number) 90.423 ns

Therefore number = ((25 μs/90.423 ns) − 54)/12 = 18 decimal (to the nearest whole number)

```
$INCLUDE (REG66X.INC)        ; lists all sfr addresses
          ORG    0           ; sets start address to 0
          SJMP   START       ; short jump to START label
          ORG    0040H       ; puts next program line at address
                               0040H
START:JB        P1.0,PULSE   ; jump to PULSE if pin0 port 1 is logic 1
                CLR P1.7     ; otherwise clear pin7 port 1 to zero
                SJMP START   ; go to START check switch
PULSE:SETB      P1.7         ; set pin7 on port 1 to logic 1
          ACALL DELAY
          NOP                ; hold logic 1 on pin7 port 1
          NOP
          NOP
          NOP
          CLR    P1.7        ; clear pin7 on port 1 to logic 0
          ACALL DELAY
          AJMP   START       ; go to START check switch
DELAY:    MOV    R0,#18
TAKE:     DJNZ   R0,TAKE
          RET
          END                ; no more assembly language
```

Chapter 4

EXERCISE 4.1

(a) One cycle time T of the required square-wave signal equals 1/frequency.
$T = 1/2000 = 0.5$ ms
Delay time $= T/2 = 0.25$ ms

(b) Timer clock = micro clock/6
= 11.0592 MHz/6 = 1.8432 MHz
Timer cycle time = 1/1.8432 MHz = 542.54 ns

Delay count = (Delay time)/(Timer cycle time)
= 0.25 ms/542.54 ns = 461 (nearest whole number)
Base number = 65535 − Delay count
= 65535 − 461 = 65074
TH0 = whole number of 65074/256
65074/256 = 254.1953125 = whole number of 254
TH0 = 254
TL0 = (remainder of 65074/256) × 256
= (0.1953125) × 256 = 50
i.e. base number = 65074 decimal = FE32 Hex

EXERCISE 4.2

```
#include <reg66x.h>
#define on   1
#define off  0
sbit  SqaureWavePin = P1^7;    // pwm = pin 7 of the PORT1
void delay5KHz();              // delay-on() returns nothing and
                               // takes nothing
main() {                       // start of the program
       TMOD = 0x02;            // Timer1 : Gate = 0 CT = 0 M1 = 0 M0 = 0
                               // Timer0 : Gate = 0 CT = 0 M1 = 1
                               // M0 = 0 ; mode 2
       TH0 = -184;             // -(184) = -(bin:1011 1000) =
                               // bin:0100 1000 = hex:48
       TL0 = -184;             // load TL0 for the first cycle
       while(1) {              // do for ever
       SquareWavePin = on;     // P1.7 set to 1
       delay5KHz();            // wait for on time
       SquareWavePin = off;    // P1.7 set to 0
       delay5KHz();            // wait for off time
  }                            // while()
}                              // main()
void delay5KHz() {
  TR0 = on;                    // set TR0 of TCON to run Timer0
  while(!TF0);                 // wait for Timer0 to set the
                               // Flag TF0
  TR0 = off;                   // stop the Timer0
  TF0 = off;                   // clear flag TF0
}                              // delay()
```

EXERCISE 4.3

```
#include <reg66x.h>
sbit  SquareWavePin = P1^7;
void T0intET0() interrupt 1 using 1 {  // Timer0 Interrupt Service
                                       // Routine
    SquareWavePin = ~ SquareWavePin;   // toggle Output
```

```
}
main() {                                  // start of the program
         TMOD = 0x02;                     // Timer1 : Gate = 0 CT = 0
                                          // M1 = 0 M0 = 0
                                          // Timer0: Gate = 0 CT = 0
                                          // M1 = 1 M0 = 0 ; mode2
         TH0 = -184;                      // -(184) = -(bin:1011 1000)
                                          // = bin:0100 1000 = hex : 48
         ET0 = 1;                         // Enable Timer0 interrupt
         EA = 1;                          // Enable All interrupt
         TR0 = 1;                         // start Timer0
         while(1);                        // end of the program
}                                         // main()
```

EXERCISE 4.4

```
#include <reg66x.h>
sbit  SquareWavePin = P1^7;
void T2intET2() interrupt 7 using 1 {  // Timer 2 Interrupt Service
                                       // Routine
    SquareWavePin = ~SquareWavePin;    // toggle Output
}
main() {                               // start of the program
         TH2 = 0xB7;                   // The Timer starts with
         TL2 = 0xFF;                   // correct values, first time
         RCAP2H = 0xB7;                // B7H into RCAP2H
         RCAP2L = 0xFF;                // FFH into RCAP2L
         T2CON = 0x00;                 // Timer 2 : TF2 = 0 EXF2= 0
                                       // RCLK = 0
                                       // EXEN2 = 0 TR2 = 0 CT2= 0
                                       // CP/RL2 = 0;AutoReload
         ET2 = 1;                      // Enable Timer 0 interrupt
         EA = 1;                       // Enable All interrupt
         TR2 = 1;                      // start Timer 0
         while(1);
}                                      // end of the program
```

EXERCISE 4.5

```
#include <reg66x.h>
sbit  Pin1 = P1^7;
void EX1intEX1() interrupt 2 using 1 {   // External 1 Interrupt
                                         // Service Routine
        Pin1 = ~Pin1;                    // toggle Output
}
main() {                                 // start of the program
        IT1 = 1;                         // set -ve edge triggered
        EX1 = 1;                         // Enab1PPPXTERNAL 1
                                         // interrupt
```

```
        EA = 1;                                  // Enable All interrupt
        while(1);
}                                                // end of the program
```

EXERCISE 4.6

```
#include <reg66x.h>
sbit outT0 = P1^0;
sbit outT1 = P1^1;
void T0intET0() interrupt 1 using 1 {    // Timer0 Interrupt
                                         // Service Routine
       outT0 = ~outT0;                   // toggle Output
       ACC = 0x55;
       P2 = ACC;
       ACC = 0x88;
       P2 = ACC;
       outT0 = ~outT0;                   // toggle Output
}
void T1intET1() interrupt 3 using 3 {    // Timer1 Interrupt
                                         // Service Routine
       outT1 = ~outT1;                   // toggle Output
}
main() {                                 // start of the program
       TH1 = 0xEE;                       //
       TH0 = 0xF8;                       //
       TL1 = 0xEE;                       //
       TL0 = 0xF8;                       //
       TMOD = 0x22;                      // Gate = 0 C/T = 0 M1 = 1
                                         //  M0 = 0 both timers:
                                         //  autoreload
                                         // Interrupt Priority (IP)
       IP = 0x0A;                        // PT2 = 0 PPC = 0 PS1 = 0
                                         // PS0 = 0 PT1 = 1 PX1 = 0
                                         // PT0 = 1 PX0 = 0
                                         // IPH(High byte)
       IPH = 0x08;                       // PT2H = 0 PPCH = 0 PS1H = 0
                                         // PS0H = 0 PT1H = 1 PX1H = 0
                                         // PT0H = 0 PX0H = 0
       ET0 = 1;                          // Enable Timer0 interrupt
       ET1 = 1;                          // Enable Timer1 interrupt
       EA = 1;                           // Enable All interrupt
       TR0 = 1;                          // start Timer0
       TR1 = 1;                          // start Timer1
       while(1);
}                                        // end of the program
```

EXERCISE 4.7

Ratio2: 8 = 2 + 8 periods = 10 periods
8 bits = 0 to 255 = 256 increments

Therefore one period = 256/10 = 25.6 increments per period.
Mark (logic 1) = 2 periods
Therefore = 2 × 25.6 = 51 increments (nearest whole number)
CCAP1L (CCAP1H) = 256 − 51 = 205 decimal = CD, and when
CL < CCAP1L, means CEX1 = 0.

Since the PCA timer clock frequency = (micro oscillator frequency)/6. The CMOD SFR can assume its default value of 00H. The CR bit in the CCON SFR will have to be set to 1 to turn the PCA time base on.

PCA timer clock frequency = 11.0592 MHz/6 = 1.8432 MHz
PCA timer cycle time = 1/1.8432 MHz = 542.54 ns
Logic 0 is held for 205 × 542.54 ns = 111.22 μs
Logic 1 is held for 51 × 542.54 ns = 22.69 μs

```
include <reg66x.h>
main() {                              // start of the program
       CCAPM1 |= 0x42;                // set ECOM1 and PWM1
       CCAP1L = 205;                  // load 2 : 8 count
       CCAP1H = 205;                  // count reload
       CR = 1;                        // turn on PCA timer in CCON
       while(1);
}                                     // end of the program
```

EXERCISE 4.8

```
#include <reg66x.h>
sbit output = P1^7;                   // label the output pin
int i;

void updateWDT();                     // prototype
main() {                              // start of the program
       CR = 1;                        // turn on PCA timer in CCON
       CMOD = 0x40;                   // WDT Enable
       CCAPM4 = 0x48;                 // set ECOM4 and MAT4, ie to WDT
       CCAP4L = 0xFF;                 // maximum into compare
       CCAP4H = 0xFF;                 // FFFF

  while(1){
          output = ~output;
          //for(i = 0;i < 99;i++)
          //;                         // some delay or other tasks
          updateWDT();
          }
}                                     // end of the program
void updateWDT(){
  CCAP4L = 0;
  CCAP4H = CH + 0xFF;
}
```

EXERCISE 4.9

```
#include <reg66x.h>
main() {                                  // start of the program
                                          // Serial mode 1
       SCON = 0x42;                       // SM0 = 0, SM1 = 1, SM2 = 0,
                                          // REN = 0, TB8 = 0, RB8 = 0,
                                          // TI = 0, RI = 0
                                          // Timer 1 in mode 2
       TMOD = 0x20;                       // T1:Gat = 0, C/T = 0, M1 = 1,
                                          // M0 = 0, T0:G = 0, C/T = 0,
                                          // M1 = 0, M0 = 0
       TH1 = 0xFA;                        // Baudrate = 9600
       TL1 = 0xFA;
       TR1 = 1;                           // start Timer 1
       while(1){
                SBUF = 'A';               // load 'A' into Serial
                                          // BUFfer
                while(!TI);               // wait for completion of
                                          // transmission
                TI = 0;                   // clear transmission flag
       }                                  // while(1)
}                                         // end of the program
```

EXERCISE 4.10

```
#include <reg66x.h>
main() {                        // start of the program
                                // Serial mode 1
       SCON = 0x42;             // SM0 = 0, SM1 = 1, SM2 = 0,
                                // REN = 0, TB8 = 0, RB8 = 0, TI = 0,
                                   RI = 0
                                // Timer 1 in mode 2
       RCAP2H = 0xFF;           // Baudrate = 9600
       RCAP2L = 0xB8;
       TCLK = 1;                // Tx CLocK flag,forces use of T2
                                // in mode 1
                                // 3
       TR2 = 1;                 // start Timer 1
       while(1){
                SBUF = 'A';  // load 'A' into Serial BUFfer
                while(!TI);// wait for completion of transmission
                TI = 0;     // clear transmission flag
       }                        // while(1)
}                               // end of the program
```

EXERCISE 4.11

```
#include <reg66x.h>
char message[] = ''Ashes to Ashes, dust to dust~'';
```

```
int charPos;
void sendChar(char ch){                  // send one character
  SBUF = ch;                             // load character into Serial
                                         // BUFfer
  while(!TI);                            // wait for completion of
                                         // transmission
  TI = 0;                                // clear transmission flag
}                                        // sendChar()
void sendMessage(){                      // send a string
    charPos = 0;                         // reset to first character
    while(message[charPos] != '~'){
    sendChar(message[charPos]);
    charPos++;                           // point to the next
                                         // character
    }                                    // while(message[charPos]
                                         // != '~')
    sendChar(0x0D);                      // send carriage return
    sendChar(0x0A);                      // send line feed
}                                        // sendMessage()
main() {                                 // start of the program
                                         // Serial mode 1
       SCON = 0x42;                      // SM0 = 0, SM1 = 1, SM2 = 0,
                                         // REN = 0, TB8 = 0, RB8 = 0,
                                         // TI = 0, RI = 0
                                         // Timer 1 in mode 2
       TMOD = 0x20;                      // T1:Gat = 0, C/T = 0, M1 = 1,
                                         // M0 = 0 T0:G = 0, CT = 0,
                                         // M1 = 0, M0 = 0
       TH1 = 0xFA;                       // Baudrate = 9600
       TL1 = 0xFA;
       TR1 = 1;                          // start Timer 1
       while(1)
    sendMessage();
}                                        // end of the program
```

EXERCISE 4.12

```
#include <reg66x.h>
void SerialPort() interrupt 4 using 3 {  // Serial port Interrupt
                                         // Service Routine
  while(!RI);                            // wait for interrupt flag
  RI = 0;                                // clear flag
  P1 = SBUF;                             // send data to port 1
}
main() {                                 // start of the program
                                         // Serial mode 1
       SCON = 0x50;                      // SM0 = 0, SM1 = 1, SM2 = 0,
                                         // REN = 1, TB8 = 0, RB8 = 0,
                                         // TI = 0, RI = 0
```

```
                                        // Timer 1 in mode 2
        TMOD = 0x20;                    // T1:Gat = 0, C/T = 0,
                                        // M1 = 1, M0 = 0 T0:G = 0,
                                        // C/T = 0, M1 = 0, M0 = 0
        TH1 = 0xFA;                     // Baudrate = 9600
        TL1 = 0xFA;
        ES0 = 1;                        // Enable serial port
                                        // interrupt
        EA = 1;                         // Enable All interrupt
        TR1 = 1;                        // start Timer 1
        while(1);
                                        // wait for serial interrupt
}                                       // end of the program
```

EXERCISE 4.13

```
#include <reg66x.h>
main() {                // start of the program
                        // IIC Serial clock speed 1/112 CR2 = 0,
                        // CR1 = 0,CR0 = 1
        S1CON = 0x45;   // CR2,ENS1 = 1,STA = 0, STO = 0,SIO = 0,
                        // AA = 1, CR1 = 0, CR0 = 0
        STA = 1;        // start IIC
        while(!SI);     // wait for serial interrupt
        STA = 0;        // clear start bit, donot want repeated
                        // start
        S1DAT = 0xA0;   // send EEPROM address + write to S1DAT
        SI = 0;         // clear SI bit
        while(!SI);     // wait for serial interrupt
        S1DAT = 0x04;   // send EEPROM internal address
        SI = 0;         // clear SI bit
        while(!SI);     // wait for serial interrupt
        S1DAT = 0x66;   // send byte to S1DAT
        SI = 0;         // clear SI bit
        while(!SI);     // wait for serial interrupt
        STO = 1;        // stop IIC
        SI = 0;         // clear SI bit
        while(!SI);     // wait for serial interrupt
        while(1);       // do nothing for ever
}                       // end of the program
```

EXERCISE 4.14

```
#include <reg66x.h>
main() {                // start of the program
                        // IIC Serial clock speed 1/112 CR2 = 0,
                        // CR1 = 0, CR0 = 1
        S1CON = 0x45;   // CR2,ENS1 = 1,STA = 0,STO = 0,SIO = 0,
                        // AA = 1,CR1 = 0,CR0 = 1
```

```
      STA = 1;            // start IIC
      while(!SI);         // wait for serial interrupt
      STA = 0;            // clear start bit, do not want repeated
                          // start
      S1DAT = 0xA0;       // write to slave address
      SI = 0;             // clear SI bit
      while(!SI);         // wait till complete
      S1DAT = 0x04;       // data byte stored at address 04 Hex
      SI = 0;             // clear SI bit
      while(!SI);         // wait till complete
      STA = 1;            // generate a STArt
      SI = 0;             // clear SI bit
      while(!SI);         // wait till start complete
      STA = 0;            // ensure no repeated start
      S1DAT = 0xA1;       // send slave address to bus + read
      SI = 0;             // clear SI bit
      while(!SI);         // wait for serial interrupt
      AA = 0;             // master sends acknowledge
      SI = 0;             // clear SI bit
      while(!SI);         // wait till sent
      STO = 0;            // master microcontroller sends a stop
      SI = 0;             // clear SI bit
      while(!SI);         // wait till sent
      while(1);           // do nothing for ever
}                         // end of the program
```

Chapter 5

EXERCISE 5.1

```
/************************************************************
* Chapter 5 Exercise5.1                                     *
* ADC application of 87LPC769                               *
* April 2003                                                *
*                                                           *
* This program reads AD0 and displays the results           *
* on two seven segments as for example 3.7                  *
************************************************************ /
#include <REG769.H>
unsigned char Volts, Volts_tenth;
/************************************************************
* START of the PROGRAM                                      *
************************************************************ /
void main (void) {
  unsigned char channel;
/************************************************************
* Disable P0, ADC pins digital Outputs and Inputs           *
* AD3 = P0.6, AD2 = P0.5, AD1 = P0.4, AD0 = P0.3            *
************************************************************ /
```

```
  P0M2 &= ~0x40;                          /* Set Pin for Input Only */
  P0M1|= 0x40;                                /* P0M2 = 0 & P0M1 = 1 */
  PT0AD = 0x40;                          /* Disable Digital Inputs */
/*************************************************************
* Enable the A/D Converter and use the CPU clock              *
* as the A/D clock.                                           *
************************************************************* */
  ENADC = 1;                      /* enable ADC, 10µs before conv.*/
  RCCLK = 0;                                    /* use CPU clock */
  channel = 3;                        /* set to the first channel */
/*************************************************************
* Update the channel number and store it in ADCON.            *
************************************************************* */
  ADCON &= ~0x03;                        /* clear channel number */
  ADCON|= channel;                      /* set the channel number */
/*************************************************************
* Perform conversions forever.                                *
************************************************************* */
  while (1) {
/*************************************************************
* Start a conversion and wait for it to complete.             *
************************************************************* */
  ADCI = 0;                             /* Clear conversion flag */
  ADCS = 1;                                  /* Start conversion */
  while (ADCI == 0);                      /* Wait conversion end */
  Volts = (unsigned char) DAC0/51;
  P1 = Volts;
  Volts_tenth = (unsigned char) DAC0%51;
  P0 = Volts_tenth/5;
  ADCI = 0;                             /* Clear conversion flag */
  }
}                                          /* end of the program */
```

EXERCISE 5.2

```
/*************************************************************
* Chapter 5 Exercise 5.2                                      *
* DAC application of 87LPC769                                 *
* April 2003                                                  *
*                                                             *
* This program generates triangular wave on the               *
* DAC0 of the P87LPC769 microcontroller                       *
************************************************************* */
#include <REG769.H>
/*************************************************************
* START of the PROGRAM                                        *
************************************************************* */
```

```
void main (void) {
  unsigned int i;
/************************************************************
* Disable P1, DAC pins digital Outputs and set the          *
* DAC1 = P1.6, DAC0 = P1.7 to Input Only ( Hi z )           *
************************************************************ /
  P1M2 & = ~0xC0;                    /* Set Pins for Input Only */
  P1M1| = 0xC0;                            /* P1M2=0 & P1M1=1 */
/************************************************************
* Disable the A/D Converter because of DAC0                 *
* AND Enable the D/A ConverterS                             *
************************************************************ /
  ADCI = 0;               /*Clear A/D conversion complete flag */
  ADCS = 0;                  /* Clear A/D conversion start flag */
  ENADC = 0;                       /* Disable the A/D Converter */
  ENDAC0 = 1;                                 /* Enable DAC0 */
/************************************************************
* Create a sawtooth wave on DAC0 and the opposite           *
* sawtooth wave on DAC1.                                    *
************************************************************ /
  while (1) {
    for (i = 0; i < 255; i++)
    DAC0 = i;
  for (i = 255; i > 0; i--)
    DAC0 = i;
  }
}
```

EXERCISE 5.3

```
/************************************************************
* Chapter 5 Exercise5.3                                     *
* Comparator application of 87LPC769                        *
* April 2003                                                *
*                                                           *
* Configures CMP1 with CIN1A (P0.4) as positive             *
* input and Vref(1.28V) as the negative input and           *
* CMP2 is configured with internal CMPREF(P0.5) as          *
* negative input and CIN2B(P0.1) as positive input.         *
* Both comparator outputs CMP1(P0.6) and CMP2(P0.0)         *
* are gated to output pins.                                 *
************************************************************ /
#include <REG769.H>
/************************************************************
* START of the PROGRAM                                      *
************************************************************ /
void main (void) {
  unsigned char i;
/************************************************************
```

```
* Disable P0, digital Outputs and Inputs                              *
* CMPREF = P0.5                                                       *
* CIN1A = P0.4, CIN1B = P0.3, CMP1 = P0.6                             *
* CIN2A = P0.2, CIN2B = P0.1, CMP2 = P0.0                             *
********************************************************************/
  P0M2 &= ~0x0C;                          /* Set Pins for Input Only */
  P0M1|= 0x0C;                                 /* P0M2 = 0 &P0M1 = 1 */
  PT0AD = 0x0C;                           /* Disable Digital Inputs */

/*******************************************************************
* Set CIN1A( P0.3 ) as +ve input, Vref as -ve                         *
* input and CMP1 Out( P0.0 )                                          *
* - - CEn CPn CNn OEn COn CMFn                                        *
* 0 0 1 0 1 1 0 0                                                     *
********************************************************************/
 CMP1 = 0x2C;

/*******************************************************************
* Set CIN2B( P0.2) as +ve input, CMPREF as -ve                        *
* input and CMP2 Out( P0.6 )                                          *
* - - CEn CPn CNn OEn COn CMFn                                        *
* 0 0 1 1 0 1 0 0                                                     *
********************************************************************/
  CMP2 = 0x34;
/*******************************************************************
* Do nothing delay 10 µs                                              *
********************************************************************/
for (i = 0; i <= 10; i++);
while (1){};                                          // Loop Forever
}
```

EXERCISE 5.4

```
/*******************************************************************
* Chapter 5 Exercise5.4                                               *
* SPI Master application of 89LPC932                                  *
* April 2003                                                          *
*                                                                     *
* This program writes some data to some slave                         *
* Devices.                                                            *
* Assumes, P0.0 = Device0.ss pin                                      *
********************************************************************/
#include <Reg932.h>
sbit Device0 = P0^0;
/*******************************************************************
* Write one byte to the SPI                                           *
********************************************************************/
void SPI_Write(unsigned char dat)
{
  SPDAT = dat;                              /* write Data to SPI bus */
```

```
  while ((SPSTAT & 0x80) == 0);                /* wait completion */
  SPSTAT| = 0x80;                  /* clear SPIF by writing 1 to it */
}
/*************************************************************
* START of the PROGRAM                                       *
************************************************************* /
void main (void) {
/*************************************************************
* Port 2 to quasi-bidirectional                              *
* MOSI = P2.2, MISO = P2.3, SPICLK = P2.4, SS = P2.5         *
************************************************************* /
  P2M1 = 0xC3;
  P2M2 = 0xC3;
/*************************************************************
* configure SPI                                              *
* SS = 1 MSTR determines device is master/slave              *
* SPEN = 1 Enable SPI                                        *
* DORD = 0 MSB of the data is transmitted first              *
* MSTR = 1 device is master                                  *
* CPOL = 1 SPICLK is high when idle. The leading             *
*          edge of SPICLK is falling edge.                   *
* CPHA = 1 data is driven on the leading edge of             *
*          SPICLK and sampled on the trailing edge           *
* SPR1 = 1 SPI clock rate = CCLK/128                         *
* SPR0 = 1                                                   *
************************************************************* /
SPCTL = 0xDF;
Device0 = 0;                                  /* select Device 0 */
while (1) {
SPI_Write('H');                            /* write H to Device 0 */
SPI_Write('A');                            /* write A to Device 0 */
SPI_Write('S');                            /* write S to Device 0 */
SPI_Write('S');                            /* write S to Device 0 */
SPI_Write('A');                            /* write A to Device 0 */
SPI_Write('N');                            /* write N to Device 0 */
  }
}
```

EXERCISE 5.5

```
/*************************************************************
* Chapter 5 Exercise5.5                                      *
* LPC932 EEPROM byte read and write applications             *
* April 2003                                                 *
*                                                            *
* This program writes to a row of 64 bytes to                *
* EEPROM memory                                              *
************************************************************* /
```

```
#include <Reg932.h>
#define dataAddress 0
/*************************************************************
* Write a row of 64 bytes to the EEPROM                      *
************************************************************* /
void writeByteRow(unsigned int adr, unsigned char dat)
{
  DEECON = 0x20 ;                          /* row of 64 bytes write */
  DEEDAT = dat;                                 /* set write data */
  DEEADR = (unsigned char) adr;                   /* start write */
  while((DEECON&0x80) == 0);                /* wait until complete */
}

/*************************************************************
* START of the PROGRAM                                       *
************************************************************* /
void main (void) {
  writeByteRow(dataAddress,'X');                /* write to EEPROM */
}
```

EXERCISE 5.6

```
/*************************************************************
* Chapter 5 Exercise5.6                                      *
* LPC932 EEPROM byte read and write applications             *
* April 2003                                                 *
*                                                            *
* This program writes 512 bytes to EEPROM memory             *
************************************************************* /
#include <Reg932.h>
#define dataAddress 0

/*************************************************************
* Write one byte to the EEPROM                               *
************************************************************* /
void writeByte(unsigned int adr, unsigned char dat)
{
  DEECON = 0x30;                                  /* block write */
  DEEDAT = dat;                                 /* set write data */
  DEEADR = (unsigned char)adr;                    /* Any address */
  while((DEECON&0x80) == 0);                /* wait until complete */
}
/*************************************************************
* START of the PROGRAM                                       *
************************************************************* /
void main (void) {
  writeByte(dataAddress,'Y');                   /* write to EEPROM */
}
```

EXERCISE 5.7

```
/*************************************************************
* Chapter 5 Exercise5.7                                      *
* LPC932 EEPROM byte read and write applications             *
* April 2003                                                 *
*                                                            *
* This program writes some data to EEPROM memory             *
* and then reads the same data back                          *
************************************************************* /
#include <Reg932.h>
/*************************************************************
* Write one byte to the EEPROM                               *
************************************************************* /
void writeByte(unsigned int adr, unsigned char dat)
{
  DEECON = 0x00;                          /* byte read/write */
  DEEDAT = dat;                            /* set write data */
  DEEADR = (unsigned char) adr;               /* start write */
  while((DEECON&0x80) == 0);             /*wait until complete */
}
/*************************************************************
* read one byte from the EEPROM                              *
************************************************************* /
unsigned char readByte(unsigned int adr)
{
  DEECON = 0x00;                          /* byte read/write */
  DEEADR = (unsigned char) adr;                /* start read */
  while((DEECON&0x80) == 0);             /*wait until complete */
  return DEEDAT;                               /* return data */
}
/*************************************************************
* START of the PROGRAM                                       *
************************************************************* /
void main (void) {
  writeByte(0x10,'H');                        /* write to EEPROM */
  writeByte(0x11,'A');                        /* write to EEPROM */
  writeByte(0x12,'S');                        /* write to EEPROM */
  writeByte(0x13,'S');                        /* write to EEPROM */
  writeByte(0x14,'A');                        /* write to EEPROM */
  writeByte(0x15,'N');                        /* write to EEPROM */
  writeByte(0x16,' ') ;                       /* write to EEPROM */
  writeByte(0x17,'F');                        /* write to EEPROM */
  writeByte(0x18,'R');                        /* write to EEPROM */
  writeByte(0x19,'E');                        /* write to EEPROM */
  writeByte(0x1A,'D');                        /* write to EEPROM */
  writeByte(0x1B,' ') ;                       /* write to EEPROM */
  writeByte(0x1C,'D');                        /* write to EEPROM */
  writeByte(0x1D,'A');                        /* write to EEPROM */
  writeByte(0x1E,'V');                        /* write to EEPROM */
```

```
  writeByte(0x1F,'I');                          /* write to EEPROM */
  writeByte(0x20,'D');                          /* write to EEPROM */

  P0 = readByte(0x10);                         /* read from EEPROM */
  P0 = readByte(0x11);                         /* read from EEPROM */
  P0 = readByte(0x12);                         /* read from EEPROM */
  P0 = readByte(0x13);                         /* read from EEPROM */
  P0 = readByte(0x14);                         /* read from EEPROM */
  P0 = readByte(0x15);                         /* read from EEPROM */
  P0 = readByte(0x16);                         /* read from EEPROM */
  P0 = readByte(0x17);                         /* read from EEPROM */
  P0 = readByte(0x18);                         /* read from EEPROM */
  P0 = readByte(0x19);                         /* read from EEPROM */
  P0 = readByte(0x1A);                         /* read from EEPROM */
  P0 = readByte(0x1B);                         /* read from EEPROM */
  P0 = readByte(0x1C);                         /* read from EEPROM */
  P0 = readByte(0x1D);                         /* read from EEPROM */
  P0 = readByte(0x1E);                         /* read from EEPROM */
  P0 = readByte(0x1F);                         /* read from EEPROM */
  P0 = readByte(0x20);                         /* read from EEPROM */
}
```

Chapter 6

EXERCISE 6.1

```
/*************************************************************
* Chapter XA Exercise6.1                                     *
* Timer 0 programming Application                            *
* April 2003                                                 *
*                                                            *
* This toggles P1.7 port pin at 1KHz                         *
************************************************************* /
#include <REGXAG49.H>
sbit  SquareWavePin = P1^7;                /* pin 7 of the port 1 */
void delay();                        /* declare the delay function */

/*************************************************************
* START of the PROGRAM                                       *
************************************************************* /
void main (void) {
  WDCON = 0;                                    /* watchdog off */
  WFEED1 = 0xA5;
  WFEED2 = 0x5A;
  TMOD = 0x01;                                 /* Timer 0 in mode 1 */
  while(1) {                                     /* do for ever */
           SquareWavePin = 1;              /* pin 7 of port 1 set to 1 */
           delay();                        /* produce delay of 0.5ms */
           SquareWavePin = 0;              /* pin 7 of port 0 set to 1 */
```

```
          delay();                    /* produce delay of 0.5ms */
  }                                                    /* while() */
}                                                      /* main() */

/*************************************************************
* produce a delay of about 0.5ms                              *
************************************************************* /
void delay(){
  TH0 = 0xFA;                              /* set the high byte */
  TL0 = 0x99;                               /* set the low byte */
  TR0 = 1;                                     /* start timer 0 */
  while(!TF0);                            /* wait for roll-over */
  TR0 = 0;                                     /* stop timer 0 */
  TF0 = 0;                                     /* clear flag 0 */
}                                                    /* delay() */
```

EXERCISE 6.2

```
/*************************************************************
* Chapter XA Exercise6.2                                      *
* WatchDog programming Application                            *
* April 2003                                                  *
*                                                             *
* This toggles P1.7 port pin at 1KHz                          *
************************************************************* /
#include <REGXAG49.H>
sbit  SquareWavePin = P1^7;               /* pin 7 of the port 1 */
void delay();                      /* declare the delay function */
/*************************************************************
* START of the PROGRAM                                        *
************************************************************* /
void main (void) {
  WDCON = 4;                          /* watchdog pre-scale = 32 */
  WDL = 51;                          /* watchdog auto-reload = 51 */
  SCR = 0;                           /*timer clock pre-scale = 4 */
  TMOD = 0x01;                              /* timer 0 in mode 1 */
  while(1) {                                     /* do for ever */
          SquareWavePin = 1;          /* pin 7 of port 1 set to 1 */
          delay();                    /* produce delay of 0.5ms */
          SquareWavePin = 0;          /* pin 7 of port 0 set to 1 */
          delay();                    /* produce delay of 0.5ms */
 }                                                   /* while() */
}                                                     /* main() */

/*************************************************************
* produces a delay of about 0.5ms                             *
************************************************************* /
void delay(){
  TH0 = 0xFA;                              /* set the high byte */
  TL0 = 0x99;                               /* set the low byte */
```

```
  TR0 = 1;                                        /* start timer 0 */
  while(!TF0);                                  /* wait for roll-over */
  TR0 = 0;                                         /* stop timer 0 */
  TF0 = 0;                                         /* clear flag 0 */
  WFEED1 = 0xA5;                                /* feed the watchdog */
  WFEED2 = 0x5A;                                /* feed the watchdog */
}                                                          /* delay() */
```

EXERCISE 6.3

```
/************************************************************
* Chapter XA Exercise6.3                                    *
* UART programming application                              *
* April 2003                                                *
*                                                           *
* This sends two messages to UART continuously              *
************************************************************ /
#include <REGXAG49.H>
code char MessageOne[] = ''Roses are red ~ '' ;
code char MessageTwo[] = ''Violets are blue ~ '' ;
const char CR = 0x0D;                          /* Carriage Return */
const char LF = 0x0A;                                /* Line Feed */
void send( unsigned char ch );

/************************************************************
* START of the PROGRAM                                      *
************************************************************ /
void main (void) {
  unsigned int i;
    WDCON = 0;                                /* watchdog control off */
  WFEED1 = 0xA5;                                 /* feed the watchdog */
  WFEED2 = 0x5A;                                 /* feed the watchdog */
  SCON = 0x42;                                /* Serial mode 1, TI set */
  TMOD = 0x20;                                     /* Timer1 in mode 2 */
  RTL1 = 238;                                    /* Timer1 reload set */
  TL1 = 238;                                  /* TL1 also set initially */
TR1 = 1;                                              /* start Timer1 */
while(1) {                                              /* do for ever */
    i = 0;
    while(MessageOne[i] != '~ ') {
      send(MessageOne[i]);
      i++;
    }
    send(CR);                                  /* send Carriage Return */
    send(LF);                                        /* send Line Feed */
    i = 0;
    while(MessageTwo[i] != '~ ') {
      send(MessageTwo[i]);
      i++;
    }
```

```
      send(CR);                              /* send Carriage Return */
      send(LF);                                    /* send Line Feed */
}                                                        /* while(1)*/
}                                                          /* main()*/
/*************************************************************
* this function sends one character                                 *
************************************************************* /
void send(unsigned char ch){
  while(!TI);                        /* wait for SBUF clear to send */
    TI = 0;                                              /* clear TI */
    S0BUF = ch;
}                                                           /* send() */
```

EXERCISE 6.4

```
/*************************************************************
* Chapter XA Exercise6.4                                            *
* Exception programming Application                                 *
* April 2003                                                        *
*                                                                   *
* This sends a messages to UART when an exception                   *
* occurs.                                                           *
************************************************************* /
#include <REGXAG49.H>
code char Message[] = ''Divide By Zero Exception ~ '' ;
const char CR = 0x0D;                              /* Carriage Return */
const char LF = 0x0A;                                    /* Line Feed */
/*************************************************************
* START of the PROGRAM                                              *
************************************************************* /
void main (void) {
  WDCON = 0;                                  /* watchdog control off */
  WFEED1 = 0xA5;                                 /* feed the watchdog */
  WFEED2 = 0x5A;                                 /* feed the watchdog */
  SCON = 0x42;                                /* Serial mode 1, TI set */
  TMOD = 0x20;                                    /* timer 1 in mode 2 */
  TH1 = 0xFA;                                              /* TH1 set */
  TL1 = 0xFA;                               /* TL1 also set initially */
  TR1 = 1;                                          /* start timer 1 */
  while(1) {                                     /* stay here for ever */
           TL0 = 44;                                 /* divide by zero */
           TL0 = TL0/0;
  }
}                                                           /* main() */
/*************************************************************
* sends one character to the UART                                   *
************************************************************* /
void send(unsigned char ch){
  while(!TI);                        /* wait for SBUF clear to send */
```

```
  TI = 0;                                                /* clear TI */
  S0BUF = ch;
}                                                          /* send() */
/*************************************************************
* handles divide by zero exceptions                          *
************************************************************* /
void DivideByZero() exception 4 using 1 {
  int i = 0;
  while(Message[i] != '~ ') {
    send(Message[i]);
    i++;
  }
  send(CR);                                  /* send carriage return */
  send(LF);                                        /* send line feed */
}                                       /* DivideByZero() exception */
```

EXERCISE 6.5

```
/*************************************************************
* Chapter XA Exercise6.5                                     *
* Trap programming application                               *
* April 2003                                                 *
*                                                            *
* This program continuously toggles P1.4, while P1.0 is      *
* at logic 1, and calls the trap 5 to toggle P1.7 when       *
* P1.0 is logic 0.                                           *
************************************************************* /
#include <REGXAG49.H>
sbit port1Pin4 = P1^4;
sbit port1Pin0 = P1^0;
sbit port1Pin7 = P1^7;
/*************************************************************
* handles trap                                               *
************************************************************* /
int myTrap5() trap 5 {
                    port1Pin7 = 1;
                    port1Pin7 = 0;
                    return 0;
}
/*************************************************************
* START of the PROGRAM                                       *
************************************************************* /
void main (void) {
  WDCON = 0;                                 /* watchdog control off */
  WFEED1 = 0xA5;                                /* feed the watchdog */
  WFEED2 = 0x5A;                                /* feed the watchdog */
  SM = 0;                                /* SM = 0, therefore user mode */
    while(1) {
```

```
    while( port1Pin0 ){                          /* while port1 pin0 = 1 */
      port1Pin4 = 1;                                  /* port1, pin4 = 1 */
      port1Pin4 = 0;                                  /* port1, pin4 = 0 */
    }
  myTrap5();
  }
}                                                                /* main() */
```

EXERCISE 6.6

```
/*************************************************************
* Chapter XA Exercise6.6                                     *
* Interrupt Priority programming Application                 *
* April 2003                                                 *
*                                                            *
* This program changes the priority of the external 1 to one *
* higher than that of Timer0.                                *
************************************************************* /
#include <REGXAG49.H>
sbit port1Pin4 = P1^4;
sbit port1Pin7 = P1^7;
/*************************************************************
* Timer0 interrupt                                           *
************************************************************* /
void Timer0(void) interrupt 1 {
  while(1){
           port1Pin7 = 1;
           port1Pin7 = 0;
  }
}
/*************************************************************
* External 1 interrupt                                       *
************************************************************* /
void EX1ternal() interrupt 2 priority 11 {
  port1Pin4 = 1;
  port1Pin4 = 0;
}
/*************************************************************
* START of the PROGRAM                                       *
************************************************************* /
void main (void) {
  WDCON = 0;                                  /* watchdog control off */
  WFEED1 = 0xA5;                                 /* feed the watchdog */
  WFEED2 = 0x5A;                                 /* feed the watchdog */
  TMOD = 0x02;                                    /* Timer0 in mode 2 */
  TL0 = 0xDD;                              /* Timer0 low byte set to DD */
  RTL0 = 0xDD;                              /* Timer0 reload set to DD */
  IPA1 = 0x03;                  /* External Int. Priority = 11(8 + 3)*/
```

```
  IPA0 = 0x20;                    /* Timer0 Int. Priority = 10(8 + 2)*/
  IEL = 0x86;                             /* Enable EA, Ex1 and ET0 */
  TR0 = 1;                                        /* start Timer0 */
  while(1);                          /* wait here for interrupts */
}                                                       /* main() */
```

Appendix A

8051 Instruction Set

A.1 Introduction

The instructions for the 8051 device are dependent on the clock frequency and are completed in a number of clock cycles. The basic 8051 device operates on a minimum 12 clock cycles per instruction basis and this is reflected in the notes that follow each type of instruction described below. However, some members of the 8051 family operate on a minimum of 6 clock cycles per instruction, hence performance is twice as fast as the basic 8051 for a specified clock frequency. Other members of the 8051 family operate on a minimum of 2 clock cycles with a consequent increase in operating speed for a given clock frequency. Details of any variation in instruction timing for a particular 8051 family member referred to in this text are given in the relevant appendix.

A.2 Notes on instruction set

Rn	Registers R7–R0 of the currently selected register bank.
direct	8-bit internal data location address. This could be internal data RAM or an SFR.
@Ri	8-bit internal data RAM location addressed indirectly through Register Ri (R0 or Rl).
addr 16	16-bit destination address. Used by LCALL and LJMP. A branch can be anywhere within the 64 KB program memory space.
addr 11	11-bit destination address. Used by ACALL and AJMP. The branch would be within the same 2 KB of program memory.
bit	Direct addressed bit in internal data RAM or SFR.
#data	8-bit constant included in the instruction.
#data 16	16-bit constant included in the instruction.
C*	Carry bit in the PSW register.
/	Complement byte or bit.
rel	Signed (two's complement) 8-bit offset byte.

A.3 Data transfer instructions

MOV A, Rn	[A]	$<---$	[Rn]
MOV A, direct	[A]	$<---$	[direct]
MOV A, @Ri	[A]	$<---$	[Ri{M}]
MOV A, #data	[A]	$<---$	data
MOV Rn, A	[Rn]	$<---$	[A]
MOV Rn, direct	[Rn]	$<---$	[direct]
MOV Rn, #data	[Rn]	$<---$	data
MOV direct, A	[direct]	$<---$	[A]
MOV direct, Rn	[direct]	$<---$	[Rn]
MOV direct, direct	[direct]	$<---$	[direct]
MOV direct, @Ri	[direct]	$<---$	[Ri{M}]
MOV direct, #data	[direct]	$<---$	data
MOV @Ri, A	[Ri{M}]	$<---$	[A]
MOV @Ri, direct	[Ri{M}]	$<---$	[direct]
MOV @Ri, #data	[Ri{M}]	$<---$	data
MOV DPTR, #data 16	[DPTR]	$<---$	data (16-bit)
MOVC A, @A + PC	[A]	$<---$	[[[A] + [PC]]{M}]
MOVC A,@A + DPTR	[A]	$<---$	[[[A] + [DPTR]]{M}]
MOVX A,@Ri	[A]	$<---$	[Ri{M}]
MOVX A, @ DPTR	[A]	$<---$	[DPTR {M}]
MOVX @DPTR,A	[DPTR{M}]	$<---$	[A]
PUSH direct	[SP]	$<---$	[SP] + 1
	[SP{M}]	$<---$	[direct]
POP direct	[direct]	$<---$	[SP{M}]
	[SP]	$<---$	[SP] − 1
XCH A,Rn	[A]	$<-->$	[Rn]
XCH A,direct	[A]	$<-->$	[direct]
XCH A,@Ri	[A]	$<-->$	[Ri{M}]
XCHD A,@Ri	$[A_{3-0}]$	$<-->$	$[Ri\{M_{3-0}\}]$

The majority of data transfer instructions consist of 24-clock cycles. The exceptions are:

MOV A,#data
MOV Rn,A
MOV Rn,#data
MOV direct,A
MOV @Ri,A
MOV @Ri,#data
XCH A,Rn
XCH A,direct
XCH A,@Ri
XCHD A,@Ri

which consist of 12-clock cycles.

A.4 Arithmetic instructions

ADD A,Rn	[A]	$<---$	[A] + [Ri]
ADD A,direct	[A]	$<---$	[A] + [direct]
ADD A,@Ri	[A]	$<---$	[A] + [Ri{M}]
ADD A,#data	[A]	$<---$	[A] + data
ADDC A,Rn	[A]	$<---$	[A] + [Ri] + C^*
ADDC A,direct	[A]	$<---$	[A] + [direct] + C^*
ADDC A,@Ri	[A]	$<---$	[A] + [Ri{M}] + C^*
ADDC A,#data	[A]	$<---$	[A] + data + C^*
SUBB A, Rn	[A]	$<---$	[A] − [Rn] − C^*
SUBB A, direct	[A]	$<---$	[A] − [direct] − C^*
SUBB A,@Ri	[A]	$<---$	[A] − [Ri{M}] − C^*
SUBB A,#data	[A]	$<---$	[A] − data − C^*
INC A	[A]	$<---$	[A] + 1
INC Rn	[Rn]	$<---$	[Rn] + 1
INC direct	[direct]	$<---$	[direct] + 1
INC @Ri	[Ri{M}]	$<---$	[Ri{M}] + 1
DEC A	[A]	$<---$	[A] − 1
DEC Rn	[Rn]	$<---$	[Rn] − 1
DEC direct	[direct]	$<---$	[direct] − 1
DEC @Ri	[Ri{M}]	$<---$	[Ri{M}] − 1
INC DPTR	[DPTR]	$<---$	[DPTR] + 1
MUL AB	$[A_{7-0}]$	$<---$	[A] × [B]
	$[B_{15-8}]$	$<---$	[A] × [B]
DIV AB	$[A_{15-8}]$	$<---$	[A]/[B]
	$\{B_{7-0}]$	$<---$	remainder
DA A	if $[A_{3-0}] > 9$,		OR Aux $C^* = 1$
	then $[A_{3-0}]$	$<---$	$[A_{3-0}] + 6$
	if $[A_{7-4}] > 9$,		OR Aux $C^* = 1$
	then $[A_{7-4}]$	$<---$	$[A_{7-4}] + 6$

The majority of arithmetic instructions consist of 12-clock cycles. The exceptions are:

INC DPTR

which takes 24-clock cycles and:

MUL AB
DIV AB

both of which take 48-clock cycles.

A.5 Logical instructions

ANL A, Rn	[A]	$<---$	[A] AND [Rn]
ANL A, direct	[A]	$<---$	[A] AND [direct]

ANL A, @Ri	[A]	$<---$	[A] AND [Ri{M}]
ANL A, #data	[A]	$<---$	[A] AND data
ANL direct, A	[direct]	$<---$	[direct] AND [A]
ANL direct, #data	[direct]	$<---$	[direct] AND data
ORL A, Rn	[A]	$<---$	[A] OR [Rn]
ORL A, direct	[A]	$<---$	[A] OR [direct]
ORL A, @Ri	[A]	$<---$	[A] OR [Ri{M}]
ORL A, #data	[A]	$<---$	[A] OR data
ORL direct, A	[direct]	$<---$	[direct] OR [A]
ORL direct, #data	[direct]	$<---$	[direct] OR data
XRL A, Rn	[A]	$<---$	[A] XOR [Rn]
XRL A, direct	[A]	$<---$	[A] XOR [direct]
XRL A, @Ri	[A]	$<---$	[A] XOR [Ri{M}]
XRL A, #data	[A]	$<---$	[A] XOR data
XRL direct, A	[direct]	$<---$	[direct] XOR [A]
XRL direct, #data	[direct]	$<---$	[direct] XOR data
CLR A	[A]	$<---$	0
CPL A	[A]	$<---$	[/A]
RL A	$[A_{n+1}]$	$<---$	$[A_n]$, n = 0–6
	$[A_0]$	$<---$	$[A_7]$
RLC A	$[A_{n+1}]$	$<---$	$[A_n]$, n = 0–6
	$[A_0]$	$<---$	C*
	C*	$<---$	$[A_7]$
RRA	$[A_n]$	$<---$	$[A_{n+1}]$, n = 0–6
	$[A_7]$	$<---$	$[A_0]$
RRC A	$[A_n]$	$<---$	$[A_{n+1}]$, n = 0–6
	$[A_7]$	$<---$	C*
	C*	$<---$	$[A_0]$
SWAP A	$[A_{3-0}]$	$<---->$	$[A_{7-4}]$

The majority of logical instructions consist of 12-clock cycles. The exceptions are:

ANL direct,#data
ORL direct,#data
XRL direct,#data

which take 24-clock cycles.

A.6 Boolean variable manipulation instructions

CLR C	C*	$<---$	0
CLR bit	bit	$<---$	0
SETB C	C*	$<---$	1
SETB bit	bit	$<---$	1
CPL C	C*	$<---$	/C*
CPL bit	bit	$<---$	/bit

ANL C, bit	C*		$<---$	C* AND bit
ANL C,/bit	C*		$<---$	C* AND /bit
ORL C, bit	C*		$<---$	C* OR bit
ORL C,/bit	C*		$<---$	C* OR /bit
MOV C, bit	C*		$<---$	bit
MOV C,/bit	C*		$<---$	/bit
JC rel		[PC]	$<---$	[PC] + 2
	if $C^* = 1$	[PC]	$<---$	[PC] + rel
JNC rel		[PC]	$<---$	[PC] + 2
	if $C^* = 0$	[PC]	$<---$	[PC] + rel
JB bit, rel		[PC]	$<---$	[PC] + 3
	if bit = 1	[PC]	$<---$	[PC] + rel
JNB bit, rel		[PC]	$<---$	[PC] + 3
	if bit = 0	[PC]	$<---$	[PC] + rel
JBC bit, rel		[PC]	$<---$	[PC] + 3
	if bit = 1	bit	$<---$	0
		[PC]	$<---$	[PC] + rel

The majority of Boolean variable manipulation instructions take 24-clock cycles. The exceptions are:

CLR C
CLR bit
SETB C
SETB bit
CPL C
CPL bit
MOV C,bit

all of which take 12-clock cycles.

A.7 Program branching instructions

ACALL addr 11	[PC]	$<---$	[PC] + 2
	[SP]	$<---$	[SP] + 1
	[SP{M}]	$<---$	$[PC_{7-0}]$
	[SP]	$<---$	[SP] + 1
	[SP{M}]	$<---$	$[PC_{15-8}]$
	$[PC_{10-0}]$	$<---$	addr 11
LCALL addr 16	[PC]	$<---$	[PC] + 3
	[SP]	$<---$	[SP] + 1
	[SP]{M}]	$<---$	$[PC_{7-0}]$
	[SP]	$<---$	[SP] + 1
	[SP{M}]	$<---$	$[PC_{15-8}]$
	$[PC_{15-0}]$	$<---$	addr 16
RET	$[PC_{15-8}]$	$<---$	[SP{M}]
	[SP]	$<---$	[SP] − 1

		$[PC_{7-0}] <---$	[SP{M}]
		[SP] <---	[SP] − 1
RETI		$[PC_{15-8}] <---$	[SP{M}]
		[SP] <---	[SP] − 1
		$[PC_{7-0}] <---$	[SP{M}]
		[SP] <---	[SP] − 1
AJMP addr 11		[PC] <---	[PC] + 2
		[PC] <---	addr 11
LJMP addr 16		[PC] <---	[PC] + 3
		[PC] <---	addr 16
SJMP rel		[PC] <---	[PC] + 2
		[PC] <---	[PC] + rel
JMP @ A + DPTR		[PC] <---	[A] + [DPTR]
JZ rel		[PC] <---	[PC] + 2
	if [A] = 0	[PC] <---	[PC] + rel
JNZ rel		[PC] <---	[PC] + 2
	if [A] − 0	[PC] <---	[PC] + rel
CJNE A, direct, rel		[PC] <---	[PC] + 3
	if A < > [direct]	[PC] <---	[PC] + rel
	if A < [direct]	C* <---	1
	else	C* <---	0
CJNE A, #data,rel		[PC] <---	[PC] + 3
	if A < > data	[PC] <---	[PC] + rel
	if A < data	C* <---	1
	else	C* <---	0
CJNE Rn, #data, rel		[PC] <---	[PC] + 3
	if [Rn] < > data	[PC] <---	[PC] + rel
	if [Rn] < data	C* <---	1
	else	C* <---	0
CJNE @Ri, #data, rel		[PC] <---	[PC] + 3
	if [Ri{M}] < > data		
		[PC] <---	[PC] + rel
	if [Ri{M}] < > data		
		C* <---	1
	else	C* <---	0
DJNZ Rn, rel		[PC] <---	[PC] + 2
		[Rn] <---	[Rn] − 1
	if [Rn] = 0	[PC] <---	[PC] + rel
DJNZ direct, rel		[PC] <---	[PC] + 2
	[direct]	<---	[direct] − 1
	if [Rn] = 0	[PC] <---	[PC] + rel
NOP		No operation	

All program branching instructions take 24-clock cycles except for:

NOP

which takes 12-clock cycles.

Appendix B

Philips XA Microcontroller – XA and 8051 Instruction Set Differences

B.1 Arithmetic

8051 INSTRUCTIONS

INC ADD ADDC DA DEC SUBB MUL DIV

ADDITIONAL INSTRUCTIONS FOR THE XA

ADDS
ADD Short signed value (4-bit: +7 to −8) to destination.

Example:
ADDS Rd, #data4

[Rd] $<---$ [Rd] + data4

SUB (.b, .w)
SUBtract without borrow.

Example:
SUB Rd, Rs

[Rd] $<---$ [Rd] – [Rs]

MULU (.b, .w)
MULtiply Unsigned (8 × 8, 16 × 16).

Example:
MULU.b Rd, Rs

[RdH] $<---$ most significant byte of [Rd] × [Rs]
[RdL] $<---$ least significant byte of [Rd] × [Rs]

Example:
MULU.w Rd, Rs

[Rd + 1] <– – – most significant byte of [Rd] × [Rs]
[Rd] <– – – least significant byte of [Rd] × [Rs]

DIVU (.b, .w, .d)
Divide Unsigned (8/8, 16/8, 32/16).

Example:
DIVU.b Rd, Rs

[RdL] <– – – 8-bit integer portion of [Rd]/[Rs]
[RdH] <– – – 8-bit remainder of [Rd]/[Rs]

Example:
DIVU.w Rd, Rs

[RdL] <– – – 8-bit integer portion of [Rd]/[Rs]
[RdH] <– – – 8-bit remainder of [Rd]/[Rs]

Example:
DIVU.d Rd, Rs

[Rd] <– – – 16-bit integer portion of [Rd]/[Rs]
[Rd + 1] <– – – 16-bit remainder of [Rd]/[Rs]

SEXT (.b, .w)
Sign EXTend N flag (sign bit) into destination register.

Example:
SEXT.b Rd

[Rd] <– – – FF if N = 1
[Rd] <– – – 00 if N = 0

Example:
SEXT.w Rd

[Rd] <– – – FFFF if N = 1
[Rd] <– – – 0000 if N = 0

B.2 Logical

8051 INSTRUCTIONS

CLR bit SETB bit CPL CLR RL RR RLC RRC ANL OR XOR

ADDITIONAL INSTRUCTIONS FOR THE XA

NEG (.b, .w)
NEGate destination register (two's complement).

Example:
NEG Rd

$$[Rd] \quad <---- \quad [\overline{Rd}] + 1$$

LSR (.b, .w, .d)
Logical Shift Right destination register, 1–31 number of bits.

Example:
LSR Rd, Rs (see Figure B.1)

Figure B.1

ASR (.b, .w, .d)
Arithmetic Shift Right destination register, 1–31 number of bits.

Example:
ASR Rd, Rs (see Figure B.2)

Figure B.2

ASL (.b, .w, .d)
Arithmetic Shift Left destination register, 1–31 number of bits.

Example:
ASL Rd, Rs (see Figure B.3)

Figure B.3

NORM (.b, .w, .d)
Logically shifts left the contents of the destination register until MSB is set, storing the number of shifts performed in the source register.

Example:
NORM Rd, Rs (see Figure B.4)

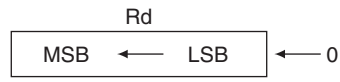


Figure B.4

CMP (.b, .w)
CoMPares the source with the destination by performing a two's complement binary subtraction of source from destination. Some flags are affected.

Example:
CMP Rd, Rs

[Rd] – [Rs]

AND (.b, .w)
ANDs bitwisely the contents of the source to the destination.

Example:
AND Rd.Rs

[Rd] $<---$ [Rd] · [Rs]

B.3 Control transfer

8051 INSTRUCTIONS (UNCONDITIONAL)

SJMP rel LJMP addrl6 AJMP addrll LCALL addrl6 ACALL addrll

ADDITIONAL INSTRUCTIONS FOR THE XA (UNCONDITIONAL)

FJMP
Far JUMP absolute causes an unconditional branch to the absolute memory location. The target address is always forced to be EVEN.

Example:
FJMP addr24

[PC(23 – 0)] $<---$ addr24
[PC(0)] $<---$ 0

CALL
CALL subroutine relative branches unconditionally in the range of +65 534 bytes to −65 536 bytes.

Example:
CALL rel26

[PC]	$<---$	[PC] + 3
[SP]	$<---$	[SP] − 4
[SP(mem)]	$<---$	[PC(23 − 0)]
[PC]	$<---$	[PC] + rell6
[PC(0)]	$<---$	0

FCALL
Far CALL subroutine absolute causes an unconditional branch to an absolute location in the 16 MB of XA address.

Example:
FCALL addr24

[PC]	$<---$	[PC] + 4
[SP]	$<---$	[SP] − 4
[SP(mem)]	$<---$	[PC(23 − 0)]
[PC]	$<---$	addr24
[PC(0)]	$<---$	0

BR
Unconditional BRanch subroutine causes an unconditional branch to a location in the range of +254 bytes to −256 bytes.

Example:
BR rel8

[PC]	$<---$	[PC] + 2
[PC]	$<---$	[PC] + rel8
[PC(0)]	$<---$	0

8051 INSTRUCTIONS (CONDITIONAL)

JB JC JNC JNZ JZ JBC CJNE DJNZ

ADDITIONAL INSTRUCTIONS FOR THE XA (CONDITIONAL)

BOV
Branch if OVerflow flag is set to a location in the range of +254 bytes to −256 bytes.

Example:
BOV rel8

```
[PC]                  <---  [PC] + 2
If [V-flag] = 1 then
[PC]                  <---  [PC] + rel8
[PC(0)]               <---  0
```

BNV
Branch No oVerflow, branches if overflow flag is not set, to a location in the range of +254 bytes to −256 bytes.

Example:
BNV rel8

```
[PC]                  <---  [PC] + 2
If [V-flag] = 0 then
[PC]                  <---  [PC] + rel8
[PC(0)]               <---  0
```

BPL
Branch PLus, branch to a location in the range of +254 bytes to −256 bytes if N flag is not set.

Example:
BPL rel8

```
[PC]                  <---  [PC] + 2
If [N-flag] = 0 then
[PC]                  <---  [PC] + rel8
[PC(0)]               <---  0
```

BCC
Branch if Carry Clear, branches to a location in the range of +254 bytes to −256 bytes if carry flag is not set.

Example:
BCC rel8

```
[PC]                  <---  [PC] + 2
If [C-flag] = 0 then
[PC]                  <---  [PC] + rel8
[PC(0)]               <---  0
```

BCS
Branch if Carry Set, branches to a location in the range of +254 bytes to −256 bytes if carry flag is set.

Example:
BCS rel8

```
[PC]                    <----   [PC] + 2
If [C-flag] = 1 then
[PC]                    <----   [PC] + rel8
[PC(0)]                 <----   0
```

BEQ
Branch if EQual, branches to a location in the range of +254 bytes to −256 bytes if zero flag is set.

Example:
BEQ rel8

```
[PC]                    <----   [PC] + 2
If [Z-flag] = 1 then
[PC]                    <----   [PC] + rel8
[PC(0)]                 <----   0
```

BNE
Branch if Not Equal, branches to a location in the range of +254 bytes to −256 bytes if zero flag is reset.

Example:
BNE rel8

```
[PC]                    <----   [PC] + 2
If [Z-flag] = 0 then
[PC]                    <----   [PC] + rel8
[PC(0)]                 <----   0
```

BG
Branch Greater, branches to a location in the range of +254 bytes to −256 bytes if the last 'compare' instruction had a destination value that was greater than the source value in an 'unsigned operation'.

Example:
BG rel8

```
[PC]                              <----   [PC] + 2
If [Z-flag] OR [C-flag] = 0 then
[PC]                              <----   [PC] + rel8
[PC(0)]                           <----   0
```

BGE
Branch Greater than or Equal to, branches to a location in the range of +254 bytes to −256 bytes if the last 'compare' instruction had a destination value that was greater than or equal to the source value in a 'signed operation'.

Example:
BGE rel8

```
[PC]                                  <----  [PC] + 2
If [N-flag] XOR [V-flag] = 0 then
[PC]                                  <----  [PC] + rel8
[PC(0)]                               <----  0
```

BGT
Branch Greater Than, branches to a location in the range of +254 bytes to −256 bytes if the last 'compare' instruction had a destination value that was greater than the source value in a 'signed operation'.

Example:
BGT rel8

```
[PC]                                  <----  [PC] + 2
If ([Z-flag] OR [N-flag]) XOR [V-flag] = 0 then
[PC]                                  <----  [PC] + rel8
[PC(0)]                               <----  0
```

BLE
Branch Less than or Equal to, branches to a location in the range of +254 bytes to −256 bytes if the last 'compare' instruction had a destination value that was less than or equal to the source value in a 'signed operation'.

Example:
BLE rel8

```
[PC]                                  <----  [PC] + 2
If ([Z-flag] OR [N-flag]) XOR [V-flag] = 1 then
[PC]                                  <----  [PC] + rel8
[PC(0)]                               <----  0
```

BLT
Branch Less Than, branches to a location in the range of +254 bytes to −256 bytes if the last 'compare' instruction had a destination value that was less than the source value in a 'signed operation'.

Example:
BLT rel8

```
[PC]                                  <----  [PC] + 2
If [N-flag] OR [V-flag] = 1 then
[PC]                                  <----  [PC] + rel8
[PC(0)]                               <----  0
```

BMI
Branch MInus, branches to a location in the range of +254 bytes to −256 bytes if N-flag is set.

Example:
BMI rel8

[PC]	< − − −	[PC] + 2
If [N-flag] = 1 then		
[PC]	< − − −	[PC] + rel8
[PC(0)]	< − − −	0

BL
Branch Less than or equal to, branches to a location in the range of +254 bytes to −256 bytes if the last 'compare' instruction had a destination value that was less than or equal to the source value in an 'unsigned operation'.

Example:
BL rel8

[PC]	< − − −	[PC] + 2
If [Z-flag] OR [C-flag] = 1 then		
[PC]	< − − −	[PC] + rel8
[PC(0)]	< − − −	0

B.4 Data transfer

8051 INSTRUCTIONS

MOV MOVC MOVX

ADDITIONAL INSTRUCTIONS IN THE XA

MOVS (.b, .w)
MOVe Short, moves signed value (4-bit: +7 to −8) to destination.

Example:
MOVS Rd, # data4

[Rd]	< − − −	data4

LEA
Load Effective Address, adds the contents of the source register to the offset value (8/16-bit), and stores the result into destination register.

Example:
LEARd, Rs + offset8/16

[Rd] < – – – [Rs] + offset8/16

B.5 Miscellaneous

8051 INSTRUCTIONS

POP PUSH SWAP XCHD

ADDITIONAL INSTRUCTIONS FOR THE XA

POPU
POP User multiple, pops specified registers (one or more) from the stack (from 1 to 8 times). Any combination of bytes registers in group R0L to R3H or the group R4L to R7H may be popped in a single instruction. Also any combination of word registers in the group R0 to R7 may be popped in a single instruction.

Example:
POPU Rlist

[Ri] < – – – [SP(mem)]
[SP] < – – – [SP] + 2
Repeat for all selected Ri registers

PUSHU
PUSH User multiple, pushes specified registers (one or more) into the stack (from 1 to 8 times). Any combination of bytes registers in group R0L to R3H or the group R4L to R7H may be pushed in a single instruction. Also any combination of word registers in the group R0 to R7 may be pushed in a single instruction.

Example:
PUSHU Rlist

[SP] < – – – [SP] – 2
[SP(mem)] < – – – [Ri]
Repeat for all selected Ri registers

RESET
The chip is internally RESET without any external effects, when the RESET instruction is executed.

Example:
RESET

[PC]	$<---$	vector (0) bytes 2 and 3
[PSW]	$<---$	vector (0) bytes 0 and 1
[SFRs]	$<---$	reset values into SFRs

TRAP
This causes a specified software trap. The invoked routine is determined by branching to the specified vector entry point. The RETI, return from interrupt instruction, is used to resume execution after the trap routine has been completed.

Example:
RESET

[PC]	$<---$	[PC] + 2
[SSP]	$<---$	[SSP] – 6
[SSP(mem)]	$<---$	[PC]
[SSP(mem)]	$<---$	[PSW]
[PSW]	$<---$	trap vector
[PC(0–15)]	$<---$	trap vector
[PC(23–16, 0)]	$<---$	0

BKPT
This causes a BreaK PoinT trap. The break point trap acts like an immediate interrupt, using a vector call to a specific piece of code that will be executed in SM.

Example:
RESET

[PC]	$<---$	[PC] + 1
[SSP]	$<---$	[SSP] – 6
[SSP(mem)]	$<---$	[PC]
[SSP(mem)]	$<---$	[PSW]
[PSW]	$<---$	bkpt vector
[PC(0–15)]	$<---$	bkpt vector
[PC(23–16, 0)]	$<---$	0

Some examples of specific applications using the XA instructions are as follows:

```
add.b r0h, r0h                add.b rll, [ r2]              add.b rlh, [r3 + ]
add.b r21, [r4 + $44]         add.b r2h, [r5 + $5555]       add.b r31, $066
add.b r0h,#$11                add.b [r2],#22                add.b [r3 + ],#$33
add.b [r4 + $44],#$44         add.b [r5 + $5555]            add.b $066,#$66
```

add.w r9, r9
add.w R12, [R0 + $0C]
add.w r9,#$9999
add.w [R0 + $0C],#$CCCC

add.w R10, [R0]
add.w R13, [R0 + $0DDD]
add.w[R0],#$AAAA
add.w [R0 + $0DDD],#$0DDD

add.w Rll, [R0 +]
add.w R14, $EE
add.w[R0 +],#$BBBB
add.w $EE,#$EEEE

adds.b r0h,#$1
adds.b [r4 + $44],#$4

adds.b [r2],#$2
adds.b [r5 + $5555],#$5

adds.b [r3 +],#$3
adds.b $066,#$6

adds.w r9,#−7
adds.w [R0],#−6
adds.w [R0 + $0DDD],#−3

adds.w [R0 +],#−5
adds.w $EE,#−2

adds.w [R0 + $0C],#−4

addc.b R0h,R0h
addc.b R21, [R4 + $44]

addc.b R11, [R2]
addc.b R2h, [R5 + $5555]

addc.b Rlh, [R3 +]
addc.b R31, $66

addc.w R9, r9
addc.w rl2, [r0 + $0C]

addc.w rl0, [r0]
addc.w rl3, [r0 + $0DDD]

addc.w rll, [r0 +]
addc.w rl4, $EE

sub.b r0h, r0h
sub.b r21, [r4 + $44]

sub.b rll, [r2]
sub.b r2h, [r5 + $5555]

sub.b rlh, [r3 +]
sub.b r31, $066

sub.w r9, r9
sub.w Rl2, [R0 + $0C]

sub.w Rl0, [R0]
sub.w Rl3, [R0 + $0DDD]

sub.w Rll, [R0 +]
sub.w R14, $EE

subb.b r0h, r0h
subb.b r21, [r4 + $44]

subb.b rll, [r2]
subb.b r2h, [r5 + $5555]

subb.b rlh, [r3 +]
subb.b r31, $066

subb.w r9, r9
subb.w Rl2, [R0 + $0C]
movc.b r0l, [r0 +]
mov.b rlh, [r3 +]
mov.b r3l, $066
mov.b 00, [R0]

subb.w Rl0, [R0]
subb.w Rl3, [R0 + $0DDD]
mov.b r0h, r0h
mov.b r2l, [r4 + $44]
mov.b [r0 +], [r0 +]

subb.w Rll, [R0 +]
subb.w R14, $EE
mov.b rll, [r2]
mov.b r2h, [r5 + $5555]
mov.b [r0], 00

movc.w R8, [R0 +]
mov.w Rll, [R0 +]
mov.w R14, $EE
mov.w $88, [R0]

mov.w r9, r9
mov.w Rl2, [R0 + $0C]
mov.w [r0 +], [r0 +]

mov.w RI0, [R0]
mov.w Rl3, [R0 + $0DDD]
mov.w [r0], $88

movx.b r3h, [r7]

movx.w rl5, [r0]

movs.b r0h,#$1
movs.b [r4 + $44],#$4
movs.w r9,#−7
movs.w [R0 + $0C],#−4

movs.b [r2],#$2
movs.b [r5 + $5555],#$5
movs.w [R0],#−6
movs.w [R0 + $0DDD],#−3

movs.b [r3 +],#$3
movs.b $066,#$6
movs.w [R0 +],#−5
movs.w $EE,#−2

and.b r0h, r0h
and.b r21, [r4 + $44]

and.b rll, [r2]
and.b r2h, [r5 + $5555]

and.b rlh, [r3 +]
and.b r31, $066

and.w r9, r9
and.wR12, [R0 + $0C]

and.w RI0, [R0]
and.w Rl3, [R0 + $0DDD]

and.w Rll, [R0 +]
and.w R14, $EE

```
or.b r0h, r0h                or.b rll, [r2]               or.b rlh, [r3 + ]
or.b r21, [r4 + $44]         or.b r2h, [r5 + $5555]       or.b r31, $066

or.w r9, r9                  or.w RI0, [R0]               or.w Rll, [R0 + ]
or.wR12, [R0 + $0C]          or.w Rl3, [R0 + $0DDD]       or.w R14, $EE

xor.b r0h, r0h               xor.b r1l, [r2]              xor.b r1h, [r3 + ]
xor.b r2l, [r4 + $44]        xor.b r2h, [r5 + $5555]      xor.b r31, $066

xor.w r9, r9                 xor.w RI0, [R0]              xor.w Rll, [R0 + ]
xor.w Rl2, [R0 + $0C]        xor.w Rl3, [R0 + $0DDD]      xor.w R14, $EE

rr.b r01,#00                 rl.b rlh,#3                  rrc.b r4h,#$7
rlc.b R4h,#7                 sl.b r0h, r0h                asr.b rll, rll
asl.b r0h,#1                 asr.b r1l,#2                 lsr.b r0l, r0l
lsr.b r0l,#0

rr.w r8,#$8                  rl.w Rll,#11                 rrc.w rl5,#$f
rlc.w R7,#15                 asl.w R9, R01                asr.w R10, R01

asl.w R9,#9                  asr.w R10,#10                lsr.w R8,R01
lsr.w R8,#8

asl.d R3, R01                asr.d R5, R01                asl.d R3,#13
asr.d R5,#14                 lsr.d Rl,#12                 lsr.dRl, R0l

mulu.b R0l, R0l              mulu.w R4, R4                mul.w R6, R6
mulu.b R4l,#$88              mul.w R9,#$99                mul.w R9,#$99

divu.b R0h, R0h              divu.w R5, R2h               div.w R7, R3h
divu.b R4l,#$88              divu.w R8,#88                div.w R8,#88

divu.d R9,#$99               div.d R9,#$99                divu.dR13, R13
div.d R15,R15

clr my_bit                   setb my_bit                  mov C, my_bit
mov my_bit,C                 anl C, my_bit                anl C,/my_bit
orl C,my_bit                 orl C,/my_bit

xch.w r8,[R0]                xch.w R8, $88                xch.b r01, 0
xch.b R0l,[R0]               xch.b R0l, R0l               xch.w r8,R8

lea R0,R0 + 0                lea r0, R0 + $88

norm.b r1h,r1h               norm.w R11, R0l              norm.d R7, R01

fcall $443322                call $5566                   call [R6]

pushu.b R71,r6h,r61,r51,r4h,r41          push.b R31,r2h,r21,rll,r0h,r01
pushu.w R15,R14,R13,R12,R11,R10,R9,R8    push.w R7,R6,R5,R4,R3,R2,R1,R0

popu.b R71,r6h,r61,r51,r4h,r41           pop.b R71,r6h,r61,r51,r4h,r41
popu.w R15,R14,R13,R12,R11,R10,R9,R8     pop.w R15,R14,R13,R12,R11,R10,R9,R8

cmp.b r0h, r0h               cmp.b rl1, [r2]              cmp.b rlh, [r3 + ]
cmp.b r21, [r4 + $44]        cmp.b r2h, [r5 + $5555]      cmp.b r31, $066
```

```
cmp.w r9, r9                cmp.w R10, [R0]             cmp.w R13, [R0 + $0ddd]
cmp.w R 14, $EE             cmp.w Rl 1, [R0 + ]         cmp.w R12, [R0 + $OC]

cjne.b R1l, $22, dummy      cjne.b Rlh,#$33, dummyl     cjne.b [R3],#33, dummy1
cjne.w R10, $1AA, dummy1    djnz.w $1AA, dummyl         cjne.wRlI,#$BBBB, dummy1
cjne.w [R0], #$BBBB, dummyl

djnz.w rl5, dummy           djnz.b $222, dummyl

jz dummyl                   jnz dummyl                  jb my_bit, $            jnb my_bit,$
jbc my_bit, $

bcc dummy2                  bcs dummy2                  bne dummy2              beq dummy2
bnv dummy2                  bov dummy2                  bpl dummy2              bmi dummy2
bg dummy2                   bl dummy2                   bge dummy2              bit dummy2
bgt dummy2                  ble dummy2                  br dummy2

jmp [a + dptr]              jmp [[R0 + ]]               jmp [R6]                fjmp $443322
jmp $5566

trap #0                     trap #1                     trap #2                 trap #3
trap #4                     trap #5                     trap #6                 trap #7
trap #8                     trap #9                     trap #10                trap #11
trap #12                    trap #13                    trap #14                trap #15

da r0l                      cpl.b r0l                   neg.b r0l               sext r8
cpl.w r8                    neg.w r8                    ret                     reti
reset                       bkpt
```

Appendix C

8051 Microcontroller Structure

C.1 Introduction

There are a wide range of devices available in the 8051 family, differing in terms of memory type and capacity, number of counter/timers, types of serial interface, number of input/output ports, clock rates, frequency range, etc. However, there is a commonality among all devices in that they have been developed from the 'core' 8051 device with modifications to produce the particular attributes of a different family member. Each member of the 8051 microcontroller family has been designed with improved device specifications in mind and to provide the customer with a device to suit particular user requirements.

This appendix will consider the basic 8051 architecture and hardware arrangements in some detail. Three devices, each a member of the 8051 family, are considered in the chapters that make up the text and each are considered in the appendices that follow. The devices are the 8-bit 89C66x (Appendix D), the 89LPC932 (Appendix E) and the 16-bit XAG-49 (Appendix F). All are flash memory devices from the Philips Semiconductors microcontroller range and grateful acknowledgement is given to that company for permission to reproduce details of their devices.

The Philips 80C51 can be considered as the core device, and functions such as I/O ports, timer/counters, serial interfacing and interrupts will be discussed. Any variations that exist for a particular family member will be dealt with in the relevant appendix that covers a particular device.

The 80C51 is available in three different package types and is basically a 40-pin device (some packages have 44 pins but only 40 are internally connected) with the following architecture:

- 4 KB $\times$ 8 ROM;
- 128 $\times$ 8 RAM;
- four 8-bit I/O ports;

- full-duplex enhanced UART with framing error detection and automatic address recognition;
- three 16-bit counter/timers;
- a six-source four-priority level nested interrupt structure;
- on-chip oscillator.

The arrangement for the 80C51 device is shown in the block diagram of Figure C.1.

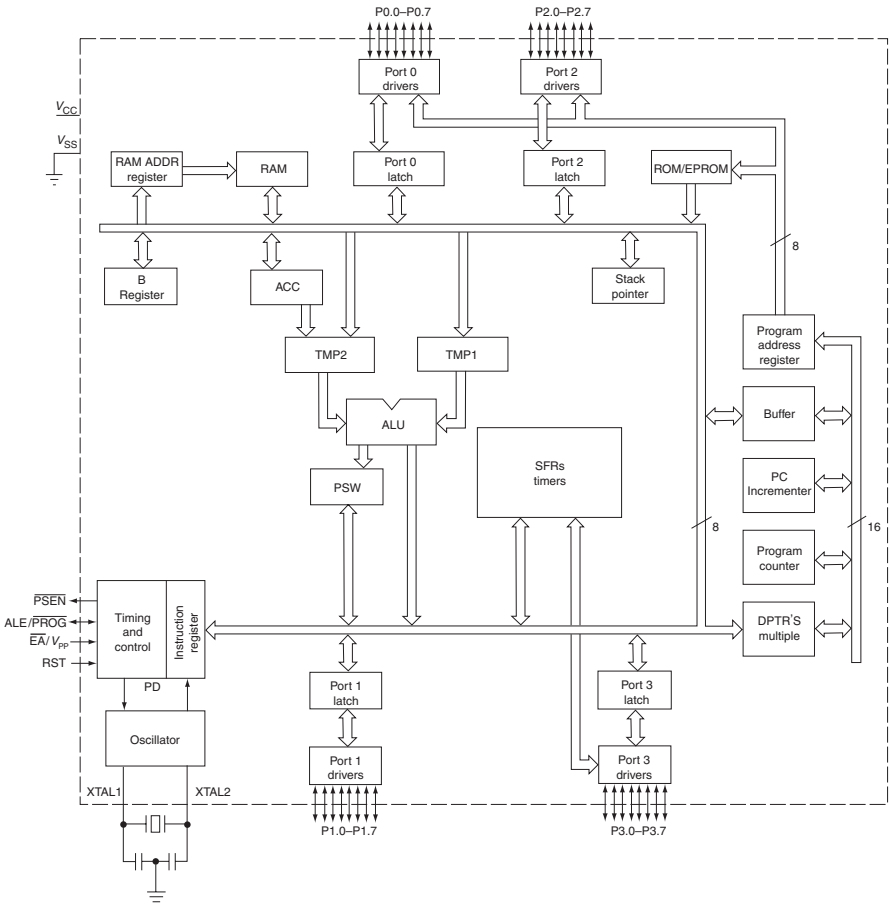


Figure C.1 80C51 block diagram (courtesy Philips Semiconductors)

Variations exist according to the family member, i.e. the on-chip program memory could be ROM or EPROM and the memory size could vary (the 80C52 has 8 KB ROM while the 87C52 has 8 KB EPROM). Also the on-chip data memory size could vary (both the 80C52 and 87C52 devices have 256 bytes of RAM).

C.2 Pin-out diagram for the 80C51

The 80C51 microcontroller is available in a 40-pin dual-in-line (DIL) package; the arrangement is shown in Figure C.2. Other packages are available and although the device pin functions are the same regardless of package configuration, pinout numbers vary. The pinout numbers referred to in the description that follows are valid only for the DIL package.

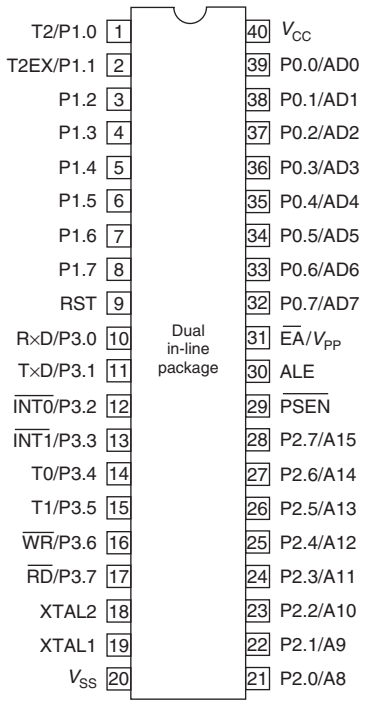


Figure C.2 80C51 pin-out layout (courtesy Philips Semiconductors)

A brief description of the function of each of the pins is as follows:

Supply voltage (V_{cc} and V_{SS}). The device operates from a single +5 V supply connected to pin 40 (V_{cc}) while pin 20 (V_{SS}) is grounded.

Input/output (I/O) ports. 32 of the pins are arranged as four 8-bit I/O ports P0–P3. Twenty-four of these pins are dual purpose (26 on the 80C52/80C58) with each capable of operating as a control line or part of the data/address bus in addition to the I/O functions. Details are as follows:

- Port 0. This is a dual-purpose port occupying pins 32 to 39 of the device. The port is an open-drain bidirectional I/O port with Schmitt trigger inputs. Pins

that have 1s written to them float and can be used as high-impedance inputs. The port may be used with external memory to provide a multiplexed address and data bus. In this application internal pull-ups are used when emitting 1s. The port also outputs the code bytes during EPROM programming. External pull-ups are necessary during program verification.

- Port 1. This is a dedicated I/O port occupying pins 1 to 8 of the device. The pins are connected via internal pull-ups and Schmitt trigger input. Pins that have 1s written to them are pulled high by the internal pull-ups and can be used as inputs; as inputs, pins that are externally pulled low will source current via the internal pull-ups. The port also receives the low-order address byte during program memory verification. Pins P1.0 and P1.1 could also function as external inputs for the third timer/counter i.e.:

  (P1.0) T2 Timer/counter 2 external count input/clockout
  (P1.1) T2EX Timer/counter 2 reload/capture/direction control

- Port 2. This is a dual-purpose port occupying pins 21 to 28 of the device. The specification is similar to that of port 1. The port may be used to provide the high-order byte of the address bus for external program memory or external data memory that uses 16-bit addresses. When accessing external data memory that uses 8-bit addresses, the port emits the contents of the P2 register. Some port 2 pins receive the high-order address bits during EPROM programming and verification.
- Port 3. This is a dual-purpose port occupying pins 10 to 17 of the device. The specification is similar to that of port 1. These pins, in addition to the I/O role, serve the special features of the 80C51 family; the alternate functions are summarised below:

P3.0	RxD	serial data input port
P3.1	TxD	serial data output port
P3.2	$\overline{\text{INT0}}$	external interrupt 0
P3.3	$\overline{\text{INT1}}$	external interrupt 1
P3.4	T0	timer/counter 0 external input
P3.5	T1	timer/counter 1 external input
P3.6	$\overline{\text{WR}}$	external data memory writes strobe
P3.7	$\overline{\text{RD}}$	external data memory read strobe.

Reset (pin 9). The 80C51 is reset by holding this input high for a minimum of two machine cycles before returning it low for normal running. An internal resistance connects to pin 20 (V_{SS}) allowing a power-on reset using an external capacitor connected to pin 40 (V_{cc}). The device internal registers are loaded with selected values prior to normal operation.

XTAL1 and XTAL2 (pins 19 and 18 respectively). The 80C51 on-chip oscillator is driven, usually, from an external crystal. The XTAL1 input also provides an input to the internal clock generator circuits.

$\overline{\text{PSEN}}$ (program store enable) (pin 29). This pin provides an output read strobe to external program memory. The output is active low during the fetch stage of an instruction. The signal is not activated during a fetch from internal memory.

ALE/$\overline{\text{PROG}}$ (address latch enable/program pulse) (pin 30). The ALE signal is an output pulse used to latch the low byte of an address during access to external memory. The signal rate is 1/6 the oscillator frequency and can be used as a general-purpose clock/timing pulse for the external circuitry. The pin also provides the program pulse input ($\overline{\text{PROG}}$) during EPROM programming. ALE can be disabled by setting SFR auxiliary.0. With this bit set ALE will be active only during a MOV X instruction.

$\overline{\text{EA}}/V_{\text{pp}}$ (external access/programming voltage) (pin 31). This pin is either tied high or low according to circuit requirements. If tied high the device will execute programs from internal memory provided the address is not higher than the last address in the internal ROM/OTP. When the $\overline{\text{EA}}$ pin is tied low, thus disabling the internal ROM, program code is accessed from external ROM. For a ROMless device the $\overline{\text{EA}}$ pin must be tied low permanently and the program code accessed from external ROM could be as much as 64 KB. EPROM versions of the device also use this pin for the supply voltage (V_{pp}) necessary for programming the internal EPROM. If security bit 1 is programmed, $\overline{\text{EA}}$ will be internally latched on reset.

C.3 80C51 family hardware

The 80C51 architecture is shown in Figure C.1. Although not numbered, the 40 pins and the pin functions as described earlier for the DIL package can be seen. The basic architecture is the same for all members of the 80C51 family although there are differences for devices, which may have more, or less, ports, comparators, ADC circuits, etc. Block diagrams for other relevant devices can be seen in those appendices that cover their specification.

Reference has already been made in general terms to the 80C51 ports, timer/counters, internal RAM and ROM/EPROM (where applicable). Specific features include:

- 8-bit CPU with registers A (accumulator) and B
- 16-bit program counter (PC)
- 16-bit data pointer register (DPTR)
- 8-bit program status word register (PSW)
- 8-bit stack pointer (SP).

It is clear from the above that the 80C51 has a collection of 8-bit and 16-bit registers and 8-bit memory locations. The internal memory of the 80C51 can be shown by the programming model of Figure C.3. In fact the 80C51 has more SFRs than shown in Figure C.3. Table C.1 lists the SFRs for the device

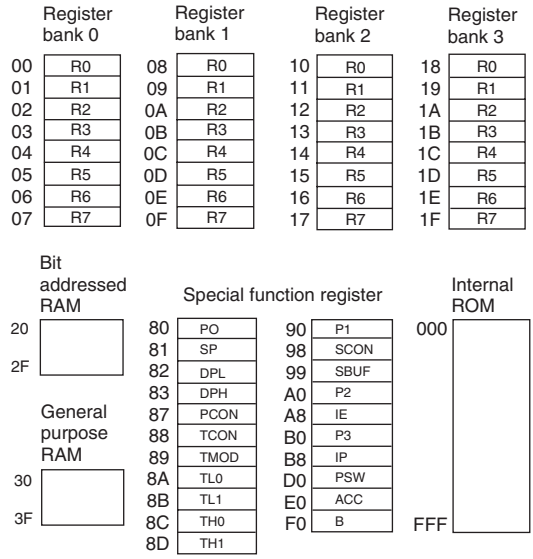


Figure C.3 80C51 programming model

and shows those SFRs that are modified versions or new additions to those shown in Figure C.3.

C.4 Memory organisation

INTERNAL RAM

The 80C51 has 128 bytes of on-chip RAM plus a number of SFRs. Including the SFR space gives 256 addressable locations but the addressing modes for internal RAM can accommodate 384 bytes by splitting the memory space into three blocks viz. the lower 128, the upper 128 and the SFR space. The lower 128 bytes use address locations 00H to 7FH and these can be accessed using direct and indirect addressing. The upper 128 bytes use address locations 80H to FFH and may be accessed using direct addressing only; locations in this space with addresses ending with 0H or 8H are also bit addressable. Some members of the 80C51 family have 256 bytes of on-chip RAM and the upper 128 bytes in this case would be accessible only using the indirect addressing mode.

For the 80C51 device, the internal RAM of 128 bytes is broken down into:

- Four register banks 0 to 3, each of which contains eight registers R0 to R7. The 32 bytes occupy addresses from 00H to 1FH. Each register can be addressed specifically when its bank is selected or an address can identify a particular register regardless of the bank, i.e. R2 of bank 2 can be specified if bank 2 is selected or the same location can be specified as address 12H. The register banks not selected can be used as general-purpose RAM. Bits 3

Table C.1 80C51/87C51/80C52/87C52 special function registers (SFRs)

Symbol	Description	Direct address	Bit address, symbol or alternative port function MSB							LSB	Reset value
ACC*	Accumulator	E0H	E7	E6	E5	E4	E3	E2	E1	E0	00H
AUXR#	Auxiliary	8EH	–	–	–	–	–	–	–	AO	xxxxxxx0B
AUXR1#	Auxiliary 1	A2H	–	–	–	LPEP[2]	WUPD	0	–	DPS	xxx000x0B
B*	B register	F0H	F7	F6	F5	F4	F3	F2	F1	F0	00H
DPTR:	Data pointer (2 bytes)										
DPH	Data pointer high	83H									00H
DPL	Data pointer low	82H									00H
			AF	AE	AD	AC	AB	AA	A9	A8	
IE*	Interrupt enable	A8H	EA	–	ET2	ES	ET1	EX1	ET0	EX0	0x000000B
			BF	BE	BD	BC	BB	BA	B9	B8	
IP*	Interrupt priority	B8H	–	–	PT2	PS	PT1	PX1	PT0	PX0	xx000000B
			B7	B6	B5	B4	B3	B2	B1	B0	
IPH#	Interrupt priority high	B7H	–	–	PT2H	PSH	PT1H	PX1H	PT0H	PX0H	xx000000B
			87	86	85	84	83	82	81	80	
P0*	Port 0	80H	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	FFH

			97	96	95	94	93	92	91	90	
P1*	Port 1	90H	–	–	–	–	–	–	T2EX	T2	FFH
			A7	A6	A5	A4	A3	A2	A1	A0	
P2*	Port 2	A0H	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8	FFH
			B7	B6	B5	B4	B3	B2	B1	B0	
P3*	Port 3	B0H	$\overline{RD}$	$\overline{WR}$	T1	T0	$\overline{INT1}$	$\overline{INT0}$	TxD	RxD	FFH
PCON#[1]	Power control	87H	SMOD1	SMOD0	–	POF	GF1	GF0	PD	IDL	00xx0000B
			D7	D6	D5	D4	D3	D2	D1	D0	
PSW*	Program status word	D0H	CY	AC	F0	RS1	RS0	OV	–	P	000000x0B
RACAP2H#	Timer 2 capture high	CBH									00H
RACAP2L#	Timer 2 capture low	CAH									00H
SADDR#	Slave address	A9H									00H
SADEN#	Slave address mask	B9H									00H
SBUF	Serial data buffer	99H									xxxxxxxxB
			9F	9E	9D	9C	9B	9A	99	98	
SCON*	Serial control	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	00H
SP	Stack pointer	81H									07H

Table C.1 Continued

Symbol	Description	Direct address	Bit address, symbol, or alternative port function MSB							LSB	Reset value
			8F	8E	8D	8C	8B	8A	89	88	
TCON*	Timer control	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00H
			CF	CE	CD	CC	CB	CA	C9	C8	
T2CON*	Timer 2 control	C8H	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/$\overline{RL2}$	00H
T2MOD#	Timer 2 mode control	C9H	–	–	–	–	–	–	T2OE	DCEN	xxxxxx00B
TH0	Timer high 0	8CH									00H
TH1	Timer high 1	8DH									00H
TH2#	Timer high 2	CDH									00H
TL0	Timer low 0	8AH									00H
TL1	Timer low 1	8BH									00H
TL2#	Timer low 2	CCH									00H
TMOD	Timer mode	89H	GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0	00H

Note: Unused register bits that are not defined should not be set by the user's program. If violated, the device could function incorrectly.
* SFRs are bit addressable.
# SFRs are modified from or added to the 80C51 SFRs.
– Reserved bits.
1. Reset value depends on reset source.
2. LPEP – Low-power EPROM operation (OTP/EPROM only).

and 4 of the PSW register determine which bank is selected when a program is running. Reset will cause bank 0 to be selected.

- Sixteen bytes that are bit addressable in the address range 20H to 2FH giving 128 addressable bits. The bits have individual addresses ranging from 00H to 07H for byte address 20H, to 78H to 7FH for byte address 2FH. Thus a bit may be addressed directly, say bit 78H, which is bit 7 of byte address 2F.
- A general-purpose memory range from 30H to 7FH, which is addressable as bytes.

In addition there are SFRs in the address range 80H to FFH. This address range actually gives 128 addresses but only 32 are defined for the 80C51; the number defined varies according to device, being much larger for some devices and less for others. Details of the SFRs for the devices referred to in the main body of the text can be found in Appendices D, E and F.

For the 80C51 the SFRs of the internal RAM are described in more detail as follows:

- **Accumulator**. This 8-bit register, usually referred to as register A, is the major register for data operations such as addition, subtraction, etc. and for Boolean bit manipulation. The register is also used for data transfers between the device and external memory, where applicable. The accumulator is both bit and byte addressable with the byte address at E0H and the bit addresses from E0H to E7H.
- **B register**. This 8-bit register is used for multiplication and division operations. For other instructions it can be considered another 'scratch pad' register. The B register is both bit and byte addressable with byte address at F0H and bit addresses from F0H to F7H.
- **Program status word (PSW)**. This 8-bit register at address D0H contains program status information as shown below:

MSB							LSB
CY	AC	F0	RS1	RS0	OV	–	P
D7H	D6H	D5H	D4H	D3H	D2H	D1H	D0H

with the bit functions defined in Table C.2.

- **Stack pointer (SP)**. This 8-bit register at address 81H is incremented before data is stored during PUSH and CALL executions. The SP is initialised to RAM address 07H after a reset, which causes the stack to commence at location 08H.
- **Data pointer (DP)**. This 16-bit register is intended to contain the two bytes that make a 16-bit address, with the high byte (DPH) at address 83H and the low byte (DPL) at address 82H. It may also be used as two independent 8-bit registers.

Table C.2 Program status word bit functions

Bit	Symbol	Function
PSW.7	CY	Carry flag
PSW.6	AC	Auxiliary carry flag (for BCD operations)
PSW.5	F0	Flag 0 (available for general-purpose use)
PSW.4	RS1	Register bank select control bit 1 set/cleared by software to determine working register bank (see Note)
PSW.3	RS0	Register bank select control bit set/cleared by software to determine working register bank (see Note)
PSW.2	OV	Overflow flag
PSW.1	–	User definable flag
PSW.0	P	Parity flag set/cleared by hardware each instruction cycle to indicate an odd/even number of 1 bits in the accumulator

Note: The contents of (RS1,RS0) enable the working register banks as follows: (0,0) – Bank 0 addresses 00H to 07H; (0,1) – Bank 1 addresses 08H to 0FH; (1,0) – Bank 2 addresses 10H to 17H; (1,1) – Bank 3 addresses 18H to 1FH.

- **Ports 0 to 3**. P0, P1, P2 and P3 are the 8-bit SFR latches of ports 0, 1, 2 and 3 respectively. The addresses are 80H, 90H, A0H and B0H respectively. Writing a '1' to any bit of any of the port SFRs causes the corresponding port output pin to go high; writing a '0' causes the corresponding port output pin to go low. When used as an input, the external state of any port pin will be held in the port SFR.
- **Serial data buffer (SBUF)**. This 8-bit register at address 99H is used for serial data in both transmit and receive modes. Moving data to SBUF loads the data ready for transmission while moving data from SBUF allows access to received data.
- **Timer registers**. The 80C51 contains three 16-bit timer/counters. Timer 0 has a low byte TL0 at address 8AH and a high byte TH0 at address 8CH while timer 1 has a low byte at address 8BH and a high byte at address 8DH. Timer 2 has a low byte at address CCH and a high byte at address CDH. Timer 2 can operate as an event timer or event counter. An extra SFR register, the T2CON register, at address C8H, controls this timer while a timer 2 mode control register T2MOD is at address C9H.
- **Control registers**. Certain control registers are required to provide control and status bits for the serial ports, timer/counters and the interrupt system. The 8-bit control registers are:

  TCON at address 88H
  TMOD at address 89H
  SCON at address 98H
  IE at address A8H
  IP at address B8H
  The effect of the control registers will be discussed later in this appendix.

INTERNAL ROM

As can be seen from Figure C.3, the 4 KB of ROM in the 80C51 occupy the address range 0000H to 0FFFH. The ROM in a microcontroller is provided so that the control program can be resident on-chip. If the control program can be accommodated within the 4 KB (or 8 KB in the case of the 80C52 device) then no external program memory is required; if however, the control program needs greater memory capacity external memory can be added up to 64 KB. The PC can access program memory in the range 0000H to FFFFH so that any program address higher than 0FFFH will have to be located in external program memory. As stated earlier, program memory can be exclusively external (and would have to be for the ROMless devices 80C31/80C32, etc.) by connecting to ground the external access pin $\overline{EA}$. The read strobe for external program memory is $\overline{PSEN}$ (see section on pinout functions).

On reset the CPU begins operations from memory location 0000H. Figure C.4 shows that for the 80C51 there are six interrupt sources located at memory addresses starting at 0003H, each consisting of eight bytes (Table C.3).

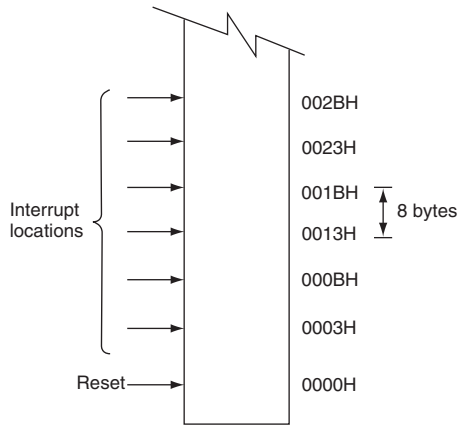


Figure C.4 80C51 program memory and interrupt locations

Table C.3

Interrupt source	Vector address
External 0	0003H
Timer 0 overflow	000BH
External 1	0013H
Timer 1 overflow	001BH
Serial port	0023H
Timer 2 overflow	002BH

For each of the interrupts the eight bytes may be sufficient to accommodate the interrupt servicing routine but if not the programmer should provide a

jump to the service routine. Whether or not an interrupt is enabled depends on the bit settings of the IE register. If no interrupts are used the programmer could establish the program from location 0000H; but with interrupts the programmer should enter a jump instruction from location 0000H to the starting address of the main program. Interrupts are dealt with later in this appendix.

EXTERNAL MEMORY

The 16-bit PC of the microcontroller will allow program memory addresses of up to 64 KB; similarly the 16-bit DP allows data memory addresses of up to 64 KB. Both address ranges are well beyond the capability of the microcontroller on its own but, if required, both data and program memory can be extended beyond the available on-chip values up to the 64 KB limit. Also involved with accessing external memory are certain control pins and input/output ports. In the sections that follow memory extension for program and data is dealt with separately although in practice they could occur simultaneously.

External program memory access

For the 80C51, if extra program memory is required a circuit arrangement as shown in Figure C.5 could be used. It can be seen from Figure C.5 that ports 0 and 2 are not available for I/O functions in this configuration but are used instead for bus functions during external memory fetches. Port 0 acts as a multiplexed address/data bus, sending the low byte of the PC (PCL) as an address and then waiting for the arrival of the code byte from external memory. The signal ALE clocks the PCL byte into the address latch during the period of time that the PCL byte is valid on port 0. The latch will hold the low address byte stable at the input to the external memory while the multiplexed bus is made ready to receive the code byte from the external memory. Port 2 sends the PC high byte (PCH) directly to the external memory; the signal $\overline{\text{PSEN}}$ then strobes the external memory allowing the code byte to be read by the microcontroller. The timing diagram for a program fetch from external memory is shown in Figure C.6.

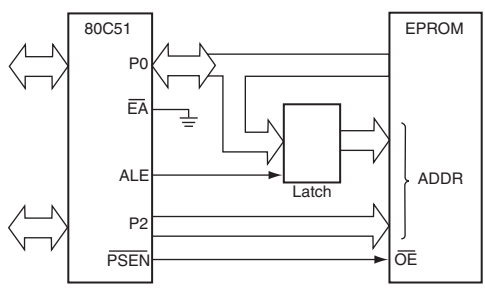


Figure C.5 Use of external program memory (courtesy Philips Semiconductors)

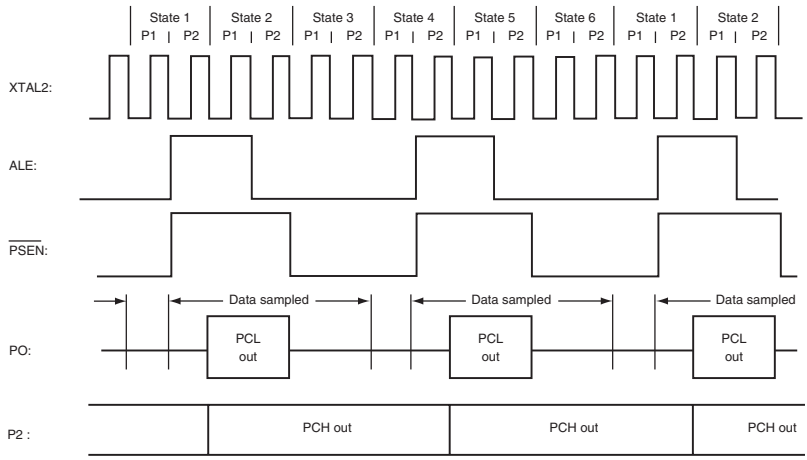


Figure C.6 External memory program fetches (courtesy Philips Semiconductors)

External data memory access

Up to 64 KB of read/write memory may be accessed by the 80C51 with the connections for the data and address lines the same as for program memory. The $\overline{RD}$ output from the microcontroller connects to the output enable ($\overline{OE}$) pin on the RAM while the $\overline{WR}$ output line connects to the RAM write enable ($\overline{WE}$) pin on the RAM. A possible arrangement is shown in Figure C.7.

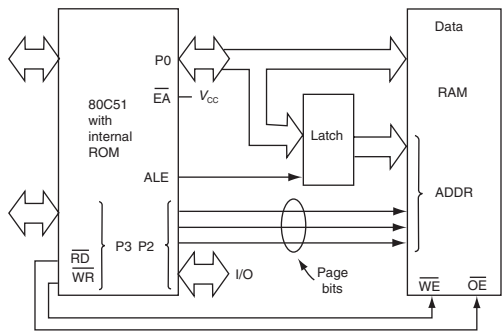


Figure C.7 Access of external data memory (courtesy Philips Semiconductors)

In this arrangement three lines of port 2 are being used to page the RAM. Memory addresses can be one or two bytes wide. One byte addresses are often used in conjunction with one or more other I/O lines to page the RAM as shown in Figure C.7. Using port lines to page RAM is an economical way to use external memory since any port lines not used for paging can be used for normal I/O functions. A page consists of 256 bytes of RAM so that two port lines are needed for accessing four pages and three port lines, as shown in Figure C.7, to access eight pages (which is 2 KB RAM). If two byte addresses

are used the high address byte is connected via port 2 in the same way as for accessing program memory. A typical timing diagram for a read cycle from external memory is shown in Figure C.8.

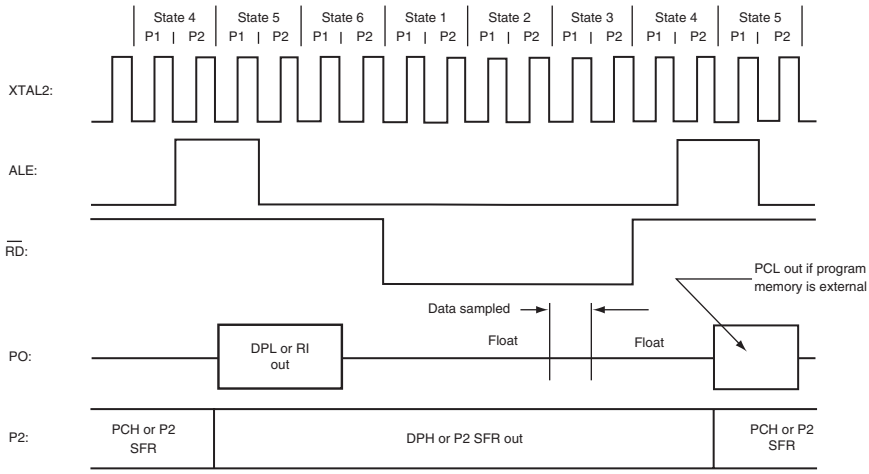


Figure C.8 External data memory read cycle (courtesy Philips Semiconductors)

The timing for a write cycle is similar except that the $\overline{WR}$ line pulses low, $\overline{RD}$ stays high and data is placed on port 0 lines as an input to the microcontroller.

C.5 I/O port configurations

As described elsewhere the four ports of the 80C51 differ slightly in that ports 0 and 2 may be used for address/data lines while port 3 has other functions. The structure of a port pin circuit varies according to the port but each port pin will have a bit latch and I/O buffer. The arrangement for a port 1 pin is shown in Figure C.9.

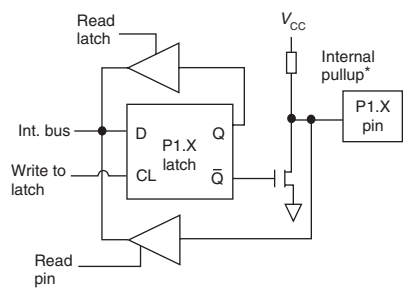


Figure C.9 80C51 port 1 bit latch (courtesy Philips Semiconductors)

The bit latch, shown as a D-type flip-flop, is one bit in the ports SFR. The latch will clock in a value from the internal bus in response to a write to latch

signal from the CPU or place its output level on the bus in response to a read signal from the CPU. The instructions that can be used to read a port can activate the 'read latch' signal or the 'read pin' signal. The requirement to read a latch rather than read a pin involves instructions known as 'read-modify-write'. These instructions would read the latch value, possibly modify the value and write it back to the latch. The reason for reading the latch rather than the pin under these circumstances is to avoid misinterpreting the pin voltage level when the pin is heavily loaded, as would be the case if driving the base of an external transistor. Suppose, for example, the port bit is connected to an external transistor base and a 1 is written to the bit turning the external transistor on; the CPU reading the pin would find the base-emitter voltage level of the on transistor and read this as 0 while reading the latch output would register the correct level 1.

Ports 1, 2 and 3 have internal pull-up resistances. If a 1 is placed on the internal bus and the write signal applied to the D-type clock input, Q goes low and the field-effect transistor (FET) goes off, allowing the pin value to go high via the pull-up resistor. Conversely a 0 on the bus latched into the flip-flop will switch the FET on and connect the output pin to ground.

For ports 1, 2 and 3, to read the signal level on the pin a 1 is written to the flip-flop which as before switches the FET off and connects the output pin via the pull-up resistor to V_{cc} i.e. logic 1; this level can be pulled low by an external source. For the output pin to go low the driving circuit must sink the current, which flows, via the pull-up resistor from V_{cc}; a read signal on the lower buffer will cause the pin signal to appear on the internal bus. The output buffers for port 1 (and ports 2 and 3) can each drive four low power Schottky (LS) TTL inputs. Port 0 has open drain outputs and each output buffer can drive eight LS TTL inputs.

For simplicity Figure C.9 does not show the alternate functions for ports 0, 2 and 3. The alternate functions are:

port 0 – Address/data
port 2 – Address
port 3 – Alternate I/O function.

The output drivers of ports 0 and 2 are switchable using an internal control signal. During external memory accesses, the P2 SFR remains unchanged but the P0 SFR has 1s written into it. Also if a P3 bit latch contains a 1 the output level is controlled by an 'alternate output function' signal while the actual port 3 pins level is always available to the pin's 'alternate input function' if any.

C.6 Timer/counters

The 80C51 has three 16-bit timer/counter registers known as timer 0, timer 1 and timer 2. Timers 0 and 1 are up-counters and may be programmed to count internal clock pulses (timer) or count external pulses (counter). Each counter is divided into two 8-bit registers to give timer low and timer high bytes i.e. TL0, TH0 and TL1, TH1.

TL0 is at address 8AH
TL1 is at address 8BH
TH0 is at address 8CH
TH1 is at address 8DH

None of these registers is bit addressable.

The operation of the timer/counters is controlled by the TMOD and TCON registers of the SFRs. TMOD is the timer SFR and is in effect two identical 4-bit registers, one each for the two timers. TCON consists of control bits and flags. Details of these two registers are shown below:

TMOD

MSB LSB

GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
89H							
---------- TIMER 1 ----------				---------- TIMER 0 ----------			

The bit functions are:

GATE – When set timer/counter x is enabled when INTx pin is high and TRx (see TCON) is set. When clear timer x is enabled when TRx bit set.

C/$\overline{T}$ – When clear, timer operation (input from internal clock). When set, counter operation (input from Tx input pin).

The M1 and M0 bit functions depend on the bit assignment as shown in Table C.4.

Table C.4

M1	M0	Operation
0	0	8048 8-bit timer TLx serves as 5-bit prescaler
0	1	16-bit timer/counter. THx and TLx are cascaded. No prescaler
1	0	8-bit autoreload timer/counter. THx contents loaded into TLx when it overflows
1	1	TL0 is 8-bit counter controlled by timer 0 control bits. TH0 is 8-bit timer controlled by timer 1 control bits
1	1	Timer 1 off

The TMOD byte is not bit addressable.

TCON

MSB LSB

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
8FH	8EH	8DH	8CH	8BH	8AH	89H	88H

The eight bits of the TCON register are duplicated pairs of four as shown in Table C.5.

Table C.5

Bit	Function
TF1/0	Timer 1/0 overflow flag. Set by hardware on timer/counter overflow. Cleared when CPU vectors to interrupt routine
TR1/0	Timer 1/0 run control bit. Set/cleared by software to turn timer/counter on/off
IE1/0	Interrupt 1/0 edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed
IT1/0	Interrupt 1/0 control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts

When the timer/counter is performing the counter function the register is incremented in response to a falling edge transition at its external input pin (T0 or T1). The TMOD bit C/$\overline{\text{T}}$ must be set to 1 to enable the pulses from the Tx pin to reach the control circuit. To count a certain number of internal or external pulses a number is put into one of the counters; the number inserted represents the maximum count, less the desired count, plus one. It takes two machine cycles (24 oscillator periods) to recognise a 1-to-0 transition, the maximum count rate is 1/24 of the oscillator frequency. There are no restrictions on the duty cycle of an external input cycle but to ensure a given level is sampled at least once before it changes it should be held for at least one full cycle. The counter will increment from the initial number to the maximum and then resets to zero on the last pulse, setting the timer flag. Testing the flag state allows confirmation of the completion of the count or, alternatively, the flag may be used to interrupt the program.

When the timer counter is performing the timer function the register is incremented every machine cycle. Thus with 12 oscillator periods per machine cycle the count rate is 1/12 of the oscillator frequency.

The timer/counters have four operating modes (modes 0, 1, 2 and 3), which are determined by the status of the bits M0 and M1 in the TMOD register. Modes 0, 1 and 2 are the same for both timer/counters but this is not the case for mode 3. Some information has already been shown in abridged form under the TMOD register description and is described below in more detail.

Mode 0

Setting the timer mode bits to 0 in the TMOD register provides an 8-bit counter (THx), preceded by 5 bits of (TLx) which gives a divide-by-32 prescaler. The pulse input is thus divided by 32 in TLx giving the oscillator frequency divided by 384. The arrangement is shown in Figure C.10 for timer 1. The arrangement for timer 0 is similar.

As the count rolls over from all 1s to all 0s the timer interrupt flag TFx is set. Figure C.10 shows that the input is passed to the timer when TRx = 1 AND GATE = 0 OR INT x = 1. TRx is a control bit in the TCON register while GATE is in the TMOD register. Setting the run flag TRx does not clear the registers.

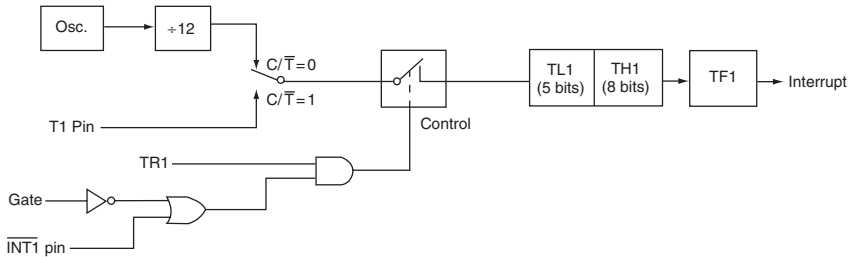


Figure C.10 Timer/counter mode 0 configuration (courtesy Philips Semiconductors)

Mode 1

This is provided when the TMOD register mode bits M1 = 0, M0 = 1 and gives the same effect as mode 0 except that the timer register runs using all 16 bits.

Mode 2

This occurs when TMOD register mode bits M1 = 1, M0 = 0 and configures the timer register as an 8-bit counter (TLx) with automatic reload. The arrangement is shown in Figure C.11 for timer 1. The arrangement for timer 0 is similar.

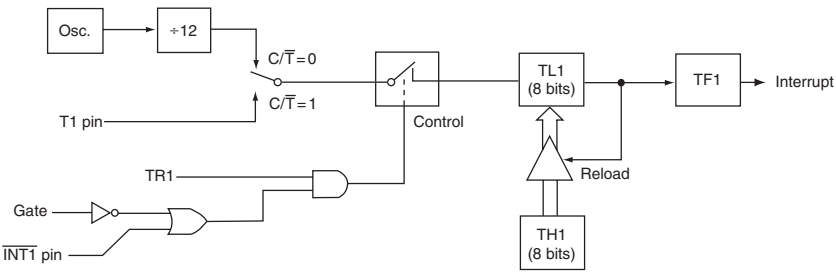


Figure C.11 Timer/counter mode 2 configuration (courtesy Philips Semiconductors)

Only the register TLx is used as an 8-bit counter while THx holds a value set by software that will be loaded into TLx every time TLx overflows. The overflow also sets the timer flag. This facility provides an initial count value for TLx that can be changed by software giving a predetermined time delay before overflow occurs.

Mode 3

This occurs when TMOD register mode bits M1 = 1, M0 = 1. Under these conditions timer 1 is off and its count is inhibited. The control bit TR1 and timer flag TF1 are then used by timer 0. Timer 0 has TL0 and TH0 as two separate counters with the arrangement shown in Figure C.12.

TL0 sets timer flag TF0 whenever overflow occurs while TH0 is controlled by TR1 and sets the timer flag TF1 whenever it overflows. Mode 3 is provided for applications that require an extra 8-bit timer on the counter. With timer 0 in

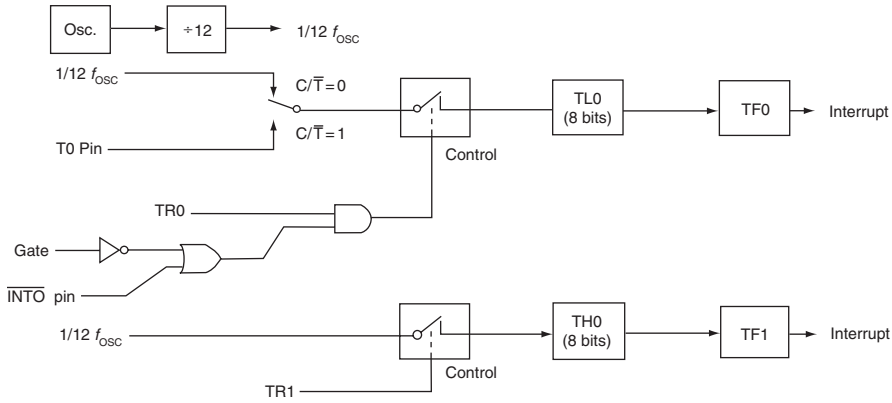


Figure C.12 Timer/counter 0 mode 3 configuration (courtesy Philips Semiconductors)

mode 3 the 80C51 can appear to have three timer/counters. When timer 0 is in mode 3, timer 1 can be switched in and out of its own mode 3 (switching timer 1 to mode 3 will hold whatever count it had reached prior to the switch) or it can be used by the serial port as a baud rate generator or any other mode 0, 1 or 2 application not requiring an interrupt (or any other use of the TF1 flag).

Timer 2 is a 16-bit timer consisting of two 8-bit registers TH2, which is the timer 2 high byte at address CDH, and TL2, which is the timer 2 low byte, situated at address CCH. Timer 2 can operate as an event timer or event counter as selected by bit C/$\overline{T2}$ in the SFR T2CON. Other bits, which affect timer 2 operation, are found in the SFR T2MOD. Details of the T2CON and T2MOD registers are shown below.

T2CON
address C8H.

MSB							LSB
TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/RL2
7	6	5	4	3	2	1	0

Bit	Symbol	Function
7	TF2	Timer 2 overflow flag set by timer 2 overflow; must be cleared by software. TF2 will not be set when either RCLK or TCLK = 1.
6	EXF2	Timer 2 external flag set when either a capture or reload is caused by a negative transition on T2EX and EXEN2 = 1. When timer 2 interrupt is enabled, EXF2 = 1 will cause the CPU to vector to the timer 2 interrupt routine. EXF2 must be cleared by software. EXF2 does not cause an interrupt in up/down counter mode (DCEN = 1).

5	RCLK	Receive clock flag. When set, causes the serial port to use timer 2 overflow pulses for its receive clock in modes 1 and 3. RCLK = 0 causes timer 1 overflow to be used for the receive clock.
4	TCLK	Transmit clock flag. When set, causes the serial port to use timer 2 overflow pulses for its transmit clock in modes 1 and 3. TCLK = 0 causes timer 1 overflow to be used for the transmit clock.
3	EXEN2	Timer 2 external enable flag. When set allows a capture or reload to occur as a result of a negative transition on T2EX if timer 2 is not being used to clock the serial port. EXEN2 = 0 causes timer 2 to ignore events at T2EX.
2	TR2	Start/stop control for timer 2. A logic 1 starts the timer.
1	C/$\overline{T2}$	Timer 2 timer or counter select: 0 = internal timer ($f_{osc}/12$) 1 = external counter (falling-edge triggered).
0	CP/$\overline{RL2}$	Capture/reload flag. When set, captures will occur on negative transitions at T2EX if EXEN2 = 1. When cleared, auto-reloads will occur either with timer 2 overflows or negative transitions at T2Ex when EXEN2 = 1. When either RCLK = 1 or TCLK = 1, this bit is ignored and the timer is forced to auto-reload on timer 2 overflow.

T2MOD

address 0C9H.
not bit addressable.

MSB							LSB
–	–	–	–	–	–	T2OE	DCEN
7	6	5	4	3	2	1	0

Bit	Symbol	Function
7, 6, 5, 4, 3, 2	–	Not implemented, reserved for future use. User software should not write to reserved bits. These bits may be used in future to invoke new features in which case the reset or inactive value of the new bit will be 0 while its active value will be 1. The value read from a reserved bit is indeterminate
1	T2OE	Timer 2 Output Enable bit
0	DCEN	Down Count ENable bit. When set, this allows timer 2 to be configured as an up/down counter

There are three operating modes for timer 2, namely:

1. capture
2. auto-reload (up/down counter)
3. baud rate generator.

The operating mode is selected by bits in the T2CON register as indicated in Table C.6.

Table C.6

RCLK + TCLK	CP/$\overline{RL2}$	TR2	Mode
0	0	1	16-bit auto-reload
0	1	1	16-bit capture
1	X	1	Baud rate generator
X	X	0	Off

CAPTURE MODE

In the capture mode there are two options each of which may be selected by bit EXEN2 in T2CON. If EXEN2 = 0, then timer 2 is a 16-bit timer or counter (as selected by C/$\overline{T2}$ in T2CON) which, upon overflowing sets bit TF2, the timer 2 overflow bit. This bit can be used to generate an interrupt (by enabling the timer 2 interrupt bit in the IE register). If EXEN2 = 1, timer 2 operates as described above, but with the added feature that a transition from 1 to 0 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, and EXF2, like TF2, can generate an interrupt which vectors to the same location as timer 2 overflow interrupt. The timer 2 interrupt service routine can interrogate TF2 and EXF2 to determine which event caused the interrupt. The capture mode is illustrated in Figure C.13. (There is no reload value for TL2 and TH2 in this mode. Even when a capture event occurs from T2EX, the counter keeps on counting T2EX pin transitions or $f_{osc}/12$ pulses.)

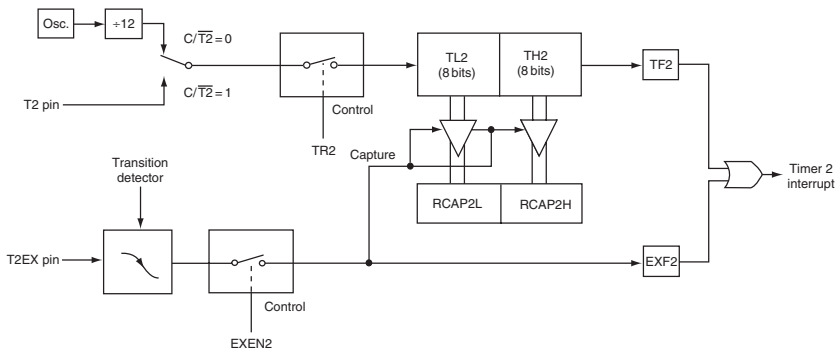


Figure C.13 Timer 2 in capture mode (courtesy Philips Semiconductors)

AUTO-RELOAD MODE (UP OR DOWN COUNTER)

In the 16-bit auto-reload mode, timer 2 can be configured (as either a timer or counter (C/$\overline{T2}$ in T2CON)) then programmed to count up or down. The

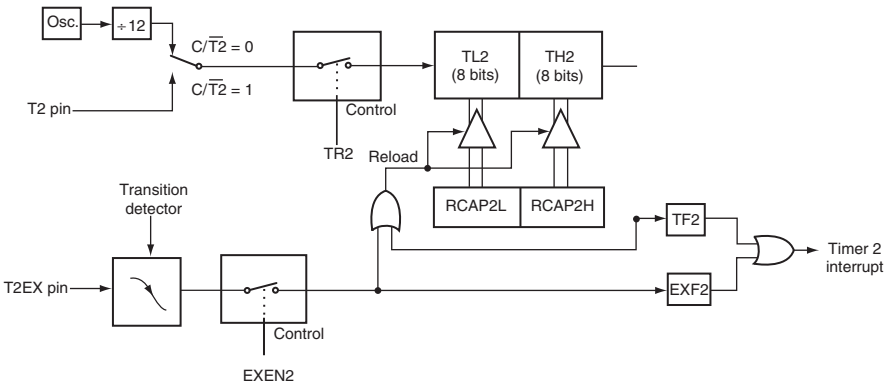


Figure C.14 Timer 2 in auto-reload mode (DCEN = 0) (courtesy Philips Semiconductors)

counting direction is determined by bit DCEN that is located in the T2MOD register. When reset is applied, DCEN = 0 which means timer 2 will default to counting up. If DCEN bit is set, timer 2 can count up or down depending on the value of the T2EX pin. Figure C.14 shows timer 2 which will count up automatically since DCEN = 0. In this mode there are two options selected by bit EXEN2 in T2CON register. If EXEN2 = 0, then timer 2 counts up to 0FFFFH and sets the TF2 (overflow flag) bit upon overflow. This causes the timer 2 registers to be reloaded with the 16-bit value in RCAP2L and RCAP2H. The values in RCAP2L and RCAP2H are preset by software means. If EXEN2 = 1, then 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. The timer 2 interrupt, if enabled, can be generated when either TF2 or EXF2 is 1. In Figure C.15 DCEN = 1 which enables timer 2 to count up or down. This mode allows pin T2EX to control the direction of count. When a logic 1 is applied at pin

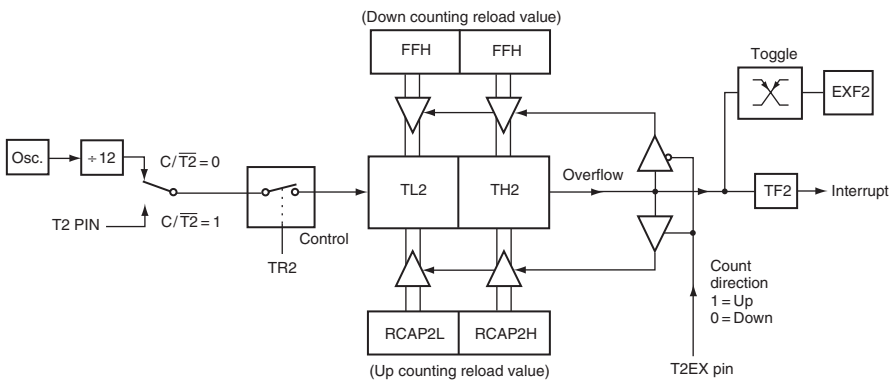


Figure C.15 Timer 2 in auto-reload mode (DCEN = 1) (courtesy Philips Semiconductors)

T2EX timer 2 will count up. Timer 2 will overflow at 0FFFFH and set the TF2 flag, which can then generate an interrupt, if the interrupt is enabled. This timer overflow also causes the 16-bit value in RCAP2L and RCAP2H to be reloaded into the timer registers TL2 and TH2. When a logic 0 is applied at pin T2EX this causes timer 2 to count down. The timer will underflow when TL2 and TH2 become equal to the value stored in RCAP2L and RCAP2H. Timer 2 underflow sets the TF2 flag and causes 0FFFFH to be reloaded into the timer registers TL2 and TH2. 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 of operation.

BAUD RATE GENERATOR MODE

Bits TCLK and/or RCLK in T2CON (Table C.6) allow the serial port transmit and receive baud rates to be derived from either timer 1 or timer 2. When TCLK = 0, timer 1 is used as the serial port transmit baud rate generator. When TCLK = 1, timer 2 is used as the serial port transmit baud rate generator. RCLK has the same effect for the serial port receive baud rate. With these two bits, the serial port can have different receive and transmit baud rates – one generated by timer 1, the other by timer 2.

Figure C.16 shows the timer 2 in baud rate generation mode. The baud rate generation mode is like the auto-reload mode, in that a rollover in TH2 causes the timer 2 registers to be reloaded with the 16-bit value in registers RCAP2H and RCAP2L, which are preset by software. The baud rates in modes 1 and 3 are determined by timer 2's overflow rate given below:

$$\text{Modes 1 and 3 baud rates} = \frac{\text{timer 2 overflow rate}}{16}$$

The timer can be configured for either 'timer' or 'counter' operation. In many applications, it is configured for 'timer' operation ($C/\overline{T2} = 0$). Timer operation is different for timer 2 when it is being used as a baud rate generator.

Usually, as a timer it would increment every machine cycle (i.e. 1/12 the oscillator frequency). As a baud rate generator, it increments every state time (i.e. 1/2 the oscillator frequency). Thus the baud rate formula is as follows:

$$\text{Modes 1 and 3 baud rates} = \frac{\text{oscillator frequency}}{32\,[65536 - (\text{RCAP2H},\ \text{RCAP2L})]}$$

Where RCAP2H, RCAP2L is the content of RCAP2H RCAP2L taken as a 16-bit unsigned integer.

The timer 2 as a baud rate generator mode shown in Figure C.16 is valid only if RCLK and/or TCLK = 1 in T2CON register. Note that a rollover in TH2 does not set TF2, and will not generate an interrupt. Thus, the timer 2 interrupt does not have to be disabled when timer 2 is in the baud rate generator mode. Also if the EXEN2 (T2 external enable flag) is set, a 1-to-0 transition in T2EX (timer/counter 2 trigger input) will set EXF2 (T2 external flag) but will not cause a reload from

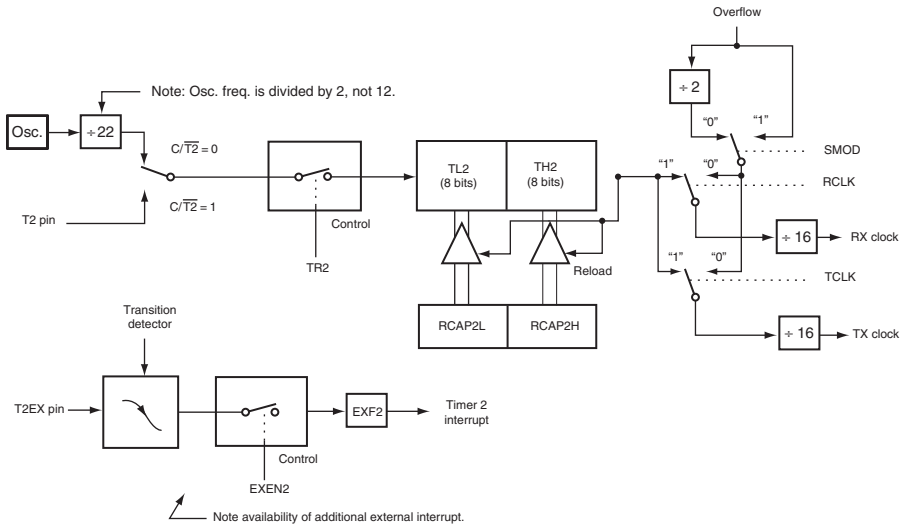


Figure C.16 Timer 2 in baud rate generator mode (courtesy Philips Semiconductors)

(RCAP2H, RCAP2L) to (TH2, TL2). Therefore when timer 2 is in use as a baud rate generator, T2EX can be used as an additional external interrupt, if needed.

When timer 2 is in the baud rate generator mode, one should not try to read or write TH2 and TL2. As a baud rate generator, timer 2 is incremented every state time ($f_{osc}/2$) or asynchronously from pin T2; under these conditions, a read or write of TH2 or TL2 may not be accurate. The RCAP2 registers may be read, but should not be written to, because a write might overlap a reload and cause write and/or reload errors. The timer should be turned off (clear TR2) before accessing the timer 2 or RCAP2 registers. Table C.7 shows commonly used baud rates and how they can be obtained from timer 2.

Table C.7 Timer 2 generated commonly used baud rates

Baud rate	Oscillator frequency (MHz)	Timer 2	
		RCAP2H	RCAP2L
375 k	12	FF	FF
9.6 k	12	FF	D9
2.8 k	12	FF	B2
2.4 k	12	FF	64
1.2 k	12	FE	C8
300	12	FB	1E
110	12	F2	AF
300	6	FD	8F
110	6	F9	57

Summary of baud rate equations

Timer 2 is in baud rate generating mode. If timer 2 is being clocked through pin T2 (P1.0) the baud rate is:

$$\text{Baud rate} = \frac{\text{Timer 2 overflow rate}}{16}$$

If timer 2 is being clocked internally, the baud rate is:

$$\frac{\text{Oscillator frequency}}{32\,[65536 - (\text{RCAP2H},\ \text{RCAP2L})]}$$

To obtain the reload value for RCAP2H and RCAP2L the above equation can be rewritten as:

$$\text{RCAP2H},\ \text{RCAP2L} = 65536 - [f_{\text{osc}}/(32 \times \text{baud rate})]$$

Timer/counter 2 set-up

Except for the baud rate generator mode, the values given for T2CON do not include the setting of the TR2 bit. Therefore, bit TR2 must be set, separately, to turn the timer on. See Table C.8 for set-up of timer 2 as a timer. Also see Table C.9 for set-up of timer 2 as a counter.

Table C.8 Timer 2 as a timer

	T2CON	
Mode	Internal control (see note 1)	External control (see note 2)
16-bit auto-reload	00H	08H
16-bit capture	01H	09H
Baud rate generator receive and transmit same baud rate	34H	36H
Receive only	24H	26H
Transmit only	14H	16H

Notes: (1) Capture/reload occurs only on timer/counter overflow; (2) Capture/reload occurs on timer/counter overflow and a 1-to-0 transition on T2EX (P1.1) pin except when timer 2 is used in the baud rate generator mode.

Programmable clock-out

A 50% duty cycle clock can be programmed to come out on P1.0. This pin, besides being a regular I/O pin, has two alternate functions. It can be programmed:

- to input the external clock for timer/counter 2, or
- to output a 50% duty cycle clock ranging from 61 Hz to 4 MHz at a 16 MHz operating frequency. To configure the timer/counter 2 as a clock generator,

Table C.9 Timer 2 as a counter

	TMOD	
Mode	Internal control (see Note 1)	External control (see Note 2)
16-bit	02H	0AH
Auto-reload	03H	0BH

Notes: See Notes of Table C.8.

bit C/$\overline{T2}$ (in T2CON) must be cleared and bit T2OE 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 (RCAP2H, RCAP2L) as shown:

$$\frac{\text{Oscillator frequency}}{4\,[65536 - (\text{RCAP2H},\ \text{RCAP2L})]}$$

where RCAP2H, RCAP2L is the contents of RCAP2H and RCAP2L taken as a 16-bit unsigned integer.

In the clock-out mode timer 2 rollovers 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 and the clock-out frequency will be the same.

C.7 Serial interface

The 80C51 possesses an on-chip serial port to enable serial data transmission between the device and external circuits. The serial port is full duplex so that it can receive and transmit data simultaneously. The port is also buffered in receive mode so that it can receive a second data byte before the first data byte has been read from the register.

The serial port register is SBUF at address 99H in the SFR. SBUF is actually two registers, one to handle receive data from the external source via RxD (P3.0) and one to hold transmit data for outward transmission via TxD (P3.1). Writing to SBUF loads data for transmission while reading SBUF accesses received data in the physically separate receive register.

Register SCON at address 98H controls data communication while register PCON at address 87H controls data rates. Details of the serial port control (SCON) register are as follows:

MSB							LSB
SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI
9FH	9EH	9DH	9CH	9BH	9AH	99H	98H

FE Framing Error bit. The receiver sets this bit when an invalid stop bit is detected. The FE bit is not cleared by valid frames but should be cleared by software. The SMOD0 bit (located at PCON.6) must be set to allow access to the FE bit.

SM0 Serial port mode bit 0 (SMOD0 must be equal to 0 to access bit SM0).

SM1 Serial port mode bit 1.

Bits SM0 and SM1 specify the serial port mode as shown in Table C.10.

Table C.10 Serial port mode options

SM0	SM1	Mode	Description	Baud rate
0	0	0	Shift register	$f_{osc}/12$
0	1	1	8-bit UART	Variable
1	0	2	9-bit UART	$f_{osc}/32$ or $f_{osc}/64$
1	1	3	9-bit UART	Variable

SM2 Enables the automatic address recognition feature in modes 2 and 3. If SM2 set to 1 then RI will not be set unless the received 9th data bit RB8 is 1, indicating an address, and the received byte is a given or broadcast address. In mode 1, if SM2 = 1 then RI will not be activated unless a valid stop bit was received and the received byte is a given or broadcast address. In mode 0, SM2 should be 0.

REN Set by software to enable serial reception. Clear by software to disable reception.

TB8 The 9th data bit that will be transmitted in modes 2 and 3. Set/clear by software.

RB8 In modes 2 and 3, is the 9th data bit received. In Mode 1 if SM2 = 0, RB8 is the stop bit that was received. In Mode 0, RB8 is not used.

TI Transmit interrupt flag. Set by hardware at the end of the 8th bit time in mode 0, or at the start of the stop bit in other modes, in any serial transmission. Must be cleared by software.

RI Receive interrupt flag. Set by hardware at the end of the 8th bit in mode 0, or halfway through the stop bit time in the other modes, in any serial reception (except see SM2). Must be cleared by software.

The serial port can operate in four modes:

Mode 0. Serial data enters and leaves via RXD. Pin TXD outputs the shift clock and this is used to synchronise data transmission/reception. Data is in the form of 8 bits with the LSB first. The rate of transmission (baud rate) is 1/12 of the oscillator frequency. Transmission is initiated by any instruction that uses SBUF as a destination register. The 'write to SBUF' signal will also load a 1 into the 9th position of the transmit shift register. Reception is initiated by the condition REN = 1 and RI = 0. A generalised diagram of the serial data format is shown in Figure C.17. This format is applicable to all modes with modification i.e. the 9th data bit is shown but is not present on modes 0 and 1.

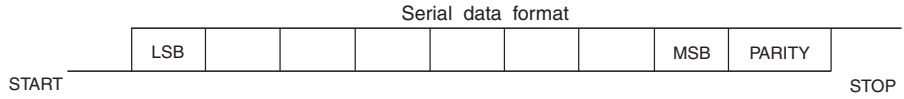


Figure C.17 Generalised serial data format

Mode 1. Ten bits are transmitted/received through TxD/RxD respectively. There is a start bit (low), 8 data bits (LSB first) followed by a stop bit (high). For transmission the interrupt flag TI is set after all 10 bits have been sent. The time for which each bit is at level 1 or 0 depends on the period set by the baud rate frequency. For received data, reception is initiated by the falling edge of the start bit and each bit is sampled in the centre of the bit time interval. The data word will be entered into the SBUF register provided:

RI = 0 AND
SM2 = 0 OR stop bit = 1.

If RI = 0 then the program has read any previous data and is ready for the next byte. The stop bit set to 1 will complete data transfer to the SBUF register regardless of the state of SM2. For SM2 = 0 the byte will be transferred to SBUF regardless of the stop bit level.

On receive the start bit is discarded, the data bits are in SBUF and the stop bit is placed in RB8 of the SCON register to indicate a data byte has been received. Note that if RI is set at the end of reception of a data byte, it suggests that the previously received data byte has not been read by the program; this would cause the new data to be lost since it will not be loaded. Transmission is again initiated by any instruction that uses SBUF as a destination register. In this mode however the bit times are synchronised to the divide-by-sixteen counter and not the 'write to SBUF' signal, as was the case for mode 0. Reception is initiated by the detection of a high-to-low transition at RxD.

Mode 2. Eleven bits are transmitted/received through TxD/RxD respectively. There is a start bit (low), 8 data bits (LSB first), a programmable 9th data bit and a stop bit (high). For transmission the 9th data bit (TB8 in SCON) can have the value 0 or 1 or the parity bit (P in the PSW) could be moved into TB8. On receive the 9th data bit goes into RB8 in the register SCON, while both the start and stop bits are ignored. The conditions for received data are similar to mode 1 i.e.:

RI = 0 AND
SM2 = 0 OR 9th data bit = 1.

If either of these conditions is not met the received frame is irretrievably lost.

Mode 3. This is identical to mode 2 in all respects except that the baud rate is not fixed (as it is for mode 2) but variable using timer 1 to provide the required communication frequencies.

ENHANCED UART

The UART operates in all of the usual modes that are described above. In addition the UART can perform framing error detect, by looking for missing stop bits, and automatic address recognition. The UART also fully supports multiprocessor communication. When used for framing error detect the UART looks for missing stop bits in the communication. A missing bit will set the FE bit in the SCON register. The FE bit shares the SCON.7 bit with SM0, and PCON.6 (SMOD0) determines the function of SCON.7. If SMOD0 is set then SCON.7 functions as FE. The SCON.7 functions as SM0 when SMOD0 is cleared. When used as FE, SCON.7 can only be cleared by software. Refer to Figure C.18.

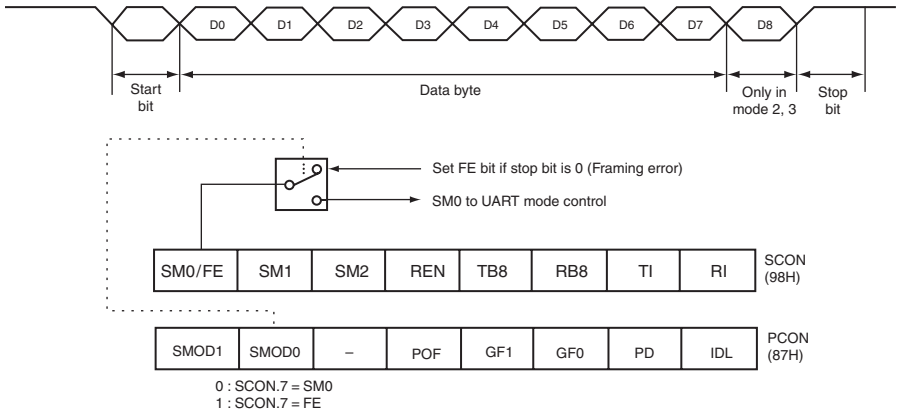


Figure C.18 UART framing error detection (courtesy Philips Semiconductors)

AUTOMATIC ADDRESS RECOGNITION

Automatic address recognition is a feature, which allows the UART to recognise 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 C.19.

The 8-bit mode is called mode 1. In this mode the RI flag will be set if SM2 is enabled and the information received has a valid stop bit following the 8 address bits and the information is either a given or broadcast address. Mode 0 is the shift register mode and SM2 is ignored. 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

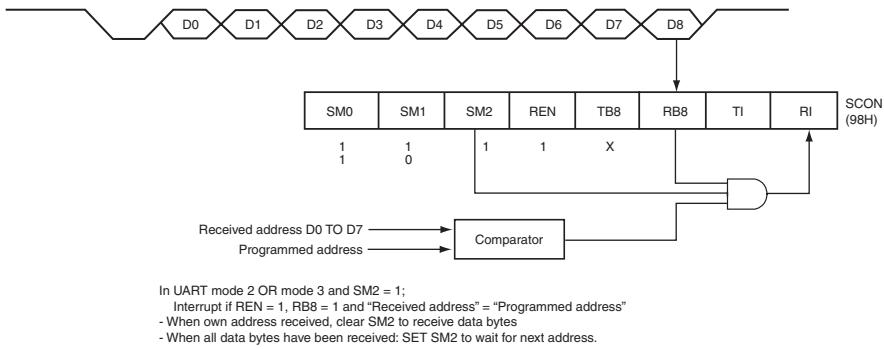


Figure C.19 UART multiprocessor communication, automatic address recognition (courtesy Philips Semiconductors)

may be contacted by using the broadcast address. Two SFRs 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 recognised while excluding others. The following examples will help to illustrate the point:

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 three 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. Taking the logical OR of SADDR and SADEN creates the broadcast address for each slave. Zeros in this result are treated as don't-cares. In most cases, interpreting the don't-cares as ones, the broadcast address will be FF hexadecimal.

Upon reset SADDR (SFR address 0A9H) and SADEN (SFR address 0B9H) 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 80C51 type UART drivers, which do not make use of this feature.

BAUD RATES

This has been described under the details of the SCON register. For mode 0 the baud rate is fixed at oscillator frequency/12. For mode 2 the baud rate depends on the value of the bit SMOD in the PCON SFR. If SMOD = 0 (which is the value on RESET), the baud rate is oscillator frequency/64. If SMOD = 1, the baud rate is oscillator frequency/32 i.e.: $f_{osc}/32$ or $f_{osc}/64$ (12 clock mode).

$$\text{Baud rate} = \frac{2^{\text{SMOD}}}{64} \times \text{Oscillator frequency}$$

In the 80C51 the baud rates in modes 1 and 3 are determined by the timer 1 overflow rate and the value of SMOD as follows:

$$\text{Baud rate} = \frac{2^{\text{SMOD}}}{32} \times \text{Timer 1 overflow rate}$$

The timer 1 interrupt should be disabled in this application. The timer can be configured for timer or counter mode and if used in timer operation in the auto-reload mode the baud rate is given by:

$$\text{Baud rate} = \frac{2^{\text{SMOD}}}{32} \times \frac{\text{Oscillator frequency}}{12 - [256 - (\text{TH1})]}$$

The oscillator frequency should be chosen to generate the required range of baud rates. Table C.11 shows variously commonly used baud rates and how they can be obtained from timer 1.

Applications involving baud rates and the use of timer 1 can be found in Chapter 4.

C.8 Interrupts

Whenever a computer program is running it can be forced to respond to external conditions either by software techniques or the use of hardware signals

Table C.11 Timer 1 generated commonly used baud rates

Baud rate	f_{osc}	SMOD1	Timer 1		
			C/$\overline{T}$	Mode	Reload value
Mode 0 Max; 1.67 MHz	20 MHz	X	X	X	X
Mode 2 Max; 625 k	20 MHz	1	X	X	X
Mode 1.3 Max; 104.2 k	20 MHz	1	0	2	FFH
19.2 k	11.059 MHz	1	0	2	FDH
9.6 k	11.059 MHz	0	0	2	FDH
4.8 k	11.059 MHz	0	0	2	FAH
2.4 k	11.059 MHz	0	0	2	F4H
1.2 k	11.059 MHz	0	0	2	E8H
137.5	11.986 MHz	0	0	2	1DH
110	6 MHz	0	0	2	72H
110	12 MHz	0	0	1	FEEBH

called interrupts. Software techniques involve checking flags or the status of port pins and could take up valuable processor time, while the interrupt signals only stop the main software program when necessary. Interrupts may be generated internally or externally; whatever the source of the interrupt request it causes the device to switch to an interrupt subroutine that is located at predetermined addresses in program memory (see under Internal ROM).

There are six interrupt sources provided by the 80C51 and these are shown in Table C.12 together with details of the polling priority assigned to each interrupt source.

Table C.12 Interrupt sources and polling priority levels

Source	Polling priority	Request bits	Hardware clear?	Vector address
X0	1	IE0	N(L)[1], Y(T)[2]	03H
T0	2	TF0	Y	0BH
X1	3	IE1	N(L)[1], Y(T)[2]	13H
T1	4	TF1	Y	1BH
SP	5	RI, TI	N	23H
T2	6	TF2, EXF2	N	2BH

Notes: 1. L = level activated; 2. T = transition activated.

There are three SFRs associated with interrupts. The registers are the interrupt enable (IE) at address A8H, interrupt priority (IP) at address B8H and the interrupt priority high (IPH) that provides a four-level interrupt routine. IPH is situated at address B7H.

Two of the interrupts are triggered by external signals via $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$ while the remaining interrupts are generated by internal operations; timer 0, timer 1, timer 2 and the ORed output of RI and TI. All of the bits that generate interrupts can be set or cleared by software. Each interrupt source can be

enabled/disabled by the setting/clearing of a bit in the SFR IE. This register, details of which are shown below, also has a global disable bit EA which disables all interrupts at once.

MSB							LSB
EA	X	ET2	ES	ET1	EX1	ET0	EX0
AFH	AEH	ADH	ACH	ABH	AAH	A9H	A8H

IE register bit functions and symbols are shown in Table C.13.

Table C.13 IE register bit functions

Bit	Symbol	Function
IE.7	EA	Disables ALL interrupts if EA = 0. If EA = 1 each interrupt source is individually enabled/disabled by setting/clearing its enable bit
IE.6	–	Reserved
IE.5	ET2	Enables/disables the timer 2 overflow interrupt. If ET2 = 0 the timer 1 interrupt is disabled
IE.4	ES	Enables/disables the serial port interrupt. If ES = 0, the serial port interrupt is disabled
IE.3	ET1	Enables/disables the timer 1 overflow interrupt. If ET1 = 0 the timer 1 interrupt is disabled
IE.2	EX1	Enables/disables external interrupt 1. If EX1 = 0, external interrupt 1 is disabled
IE.1	ET0	Enables/disables the timer 0 overflow interrupt. If ET0 = 0 the timer 0 interrupt is disabled
IE.0	EX0	Enables/disables external interrupt 0. If EX0 = 0, external interrupt 0 is disabled

Each interrupt source can also be individually programmed to one of two priority levels by setting/clearing a bit in SFR register IP. A low priority interrupt can be interrupted in turn by a high priority interrupt but not by another low priority interrupt. A high priority interrupt cannot be interrupted by any other interrupt source. If requests of different priority level are received simultaneously the higher priority level is serviced first. If requests of the same priority level are received simultaneously an internal polling sequence is invoked which determines a second priority level as shown in Table C.14.

The address given is the starting address of the relevant interrupt subroutine. If the routine cannot be fitted into the available 8 bytes a jump instruction should route the routine elsewhere in memory. The interrupt will cause the main program to stop while the interrupt is serviced, with the PC address being saved on the stack. A RETI instruction at the end of the service routine restores the address reached by the PC prior to the interrupt back to the PC and resets the interrupt logic so that another interrupt, should it occur, can be serviced.

Table C.14

Priority	Source	Address
1	IE0	0003H
2	TF0	000BH
3	IE1	0013H
4	TF1	001BH
5	RI/TI	0023H
6	TF2/EXF2	002BH

Details of the IP register are:

MSB LSB

X	X	PT2	PS	PT1	PX1	PT0	PX0
		BDH	BCH	BBH	BAH	B9H	B8H

IP register functions and symbols are shown in Table C.15.

Table C.15 IP register bit functions

Bit	Symbol	Function
IP.7	–	Reserved
IP.6	–	Reserved
IP.5	PT2	Defines timer 2 interrupt priority level. PT1 = 1 programs it to the higher priority level
IP.4	PS	Defines serial port interrupt priority level. PS = 1 programs it to higher priority level
IP.3	PT1	Defines timer 1 interrupt priority level. PT1 = 1 programs it to the higher priority level
IP.2	PX1	Defines external interrupt priority level. PX1 = 1 programs it to the higher priority level
IP.1	PT0	Enables/disables timer 0 interrupt priority level. PT0 = 1 programs it to the higher priority level
IP.0	PX0	Defines the external interrupt 0 priority level. PX0 = 1 programs it to the higher priority level

Details of the IPH register are:

MSB LSB

–	–	PT2H	PSH	PT1H	PX1H	PT0H	PX0H
B7H	B6H	B5H	B4H	B3H	B2H	B1H	B0H

IPH register functions and symbols are shown in Table C.16.

Table C.16 IPH register bit functions

Bit	Symbol	Function
IPH.7	–	Reserved for future use
IPH.6	–	Reserved for future use
IPH.5	PT2H	Timer 2 interrupt priority bit high
IPH.4	PSH	Serial Port interrupt priority bit high
IPH.3	PT1H	Timer 1 interrupt priority bit high
IPH.2	PX1H	External interrupt 1 priority bit high
IPH.1	PT0H	Timer 0 interrupt priority bit high
IPH.0	PX0H	External interrupt 0 priority bit high

Note: Priority bit = 1 assigns higher priority while priority bit = 0 assigns lower priority.

The combination of the IP and IPH registers determines the four-level interrupt structure as shown in Table C.17.

Table C.17

Priority bits		
IPH.x	IP.x	Interrupt priority level
0	0	Level 0 (lowest priority)
0	1	Level 1
1	0	Level 2
1	1	Level 3 (highest priority)

An interrupt will be serviced as long as an interrupt of equal or higher priority is not already being serviced. If an interrupt of equal or higher level priority is being serviced, the new interrupt will wait until the current interrupt is finished before it is serviced. If a lower priority level interrupt is being serviced, it will be stopped and the new interrupt serviced. When the new interrupt is finished the lower priority level interrupt that was stopped will be completed.

Internal interrupts

When a timer/counter overflows the corresponding timer flag TF0 or TF1 is set to 1. The flag is cleared by on-chip hardware when the service routine is vectored. The timer 2 flags TF2 and EXF2 must be cleared by software.

The serial port interrupt is generated by the logical OR of RI (set to 1 in the SCON register when a data byte is received) and TI (set to 1 in the SCON register when a data byte has been transmitted). Neither flag is cleared by hardware when vectoring to the service routine. In practice the service routine

will have to determine whether it was RI or TI that generated the interrupt and the bit cleared by software.

External interrupts

The external interrupts $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$ can be level activated or transition activated depending on the value of the bits IT0 and IT1 in the TCON register. The interrupts are actually generated by the flags IE0 and IE1 in TCON. When an external interrupt is generated the flag that caused it is only cleared by hardware when the service routine is vectored only if the interrupt was transition activated. Any level-activated interrupt must be reset by the programmer when the interrupt is serviced by the service subroutine. The low level must be removed from the external circuit before a RETI instruction is executed otherwise an immediate interrupt will occur after the execution of the RETI instruction.

Reset

The reset input is the RST pin and taking this pin high for at least two machine cycles while the oscillator is running will cause the CPU to generate an internal reset. Reset is a form of interrupt since the action of the RST pin overrides any software, which the 80C51 may be running at the time. Unlike other interrupts the value of the address on the PC is not saved but is reset to 0000H. In fact the internal reset algorithm writes 0s to all SFRs except the port latches, SP and SBUF. The 80C51 reset values are shown in Table C.1. Internal RAM is not affected by reset. However on power up the RAM values are indeterminate.

On-chip oscillators

The 8051 device is available in an NMOS version and a CMOS version, with the latter having lower power consumption. In either case, although the circuitry differs, the on-chip oscillator circuit is a positive reactance intended to provide crystal-controlled resonance with externally connected capacitors. The arrangement is shown in Figure C.20.

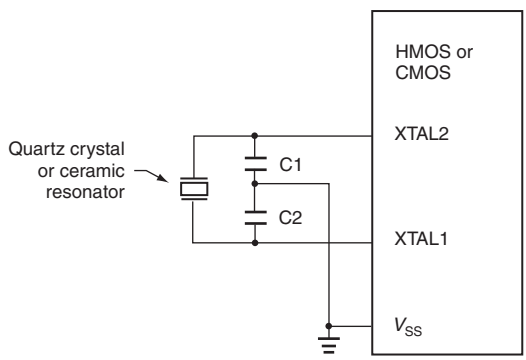


Figure C.20 80C51 on-chip oscillator (courtesy Philips Semiconductors)

The crystal specifications and capacitance values are not critical and 30 pF can be used at any frequency with good quality crystals. Where cost is critical ceramic resonators may be used and this case the capacitor values should be higher, typically 47 pF. To drive the device with an external clock source it is usual, for the CMOS device, to drive the XTAL1 input with the external clock and leave the input XTAL2 floating. This is shown in Figure C.21.

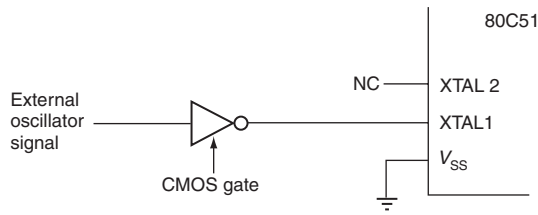


Figure C.21 Using external clock sources (courtesy Philips Semiconductors)

For the NMOS device the external clock is connected to XTAL2 input and XTAL1 is grounded. The reason for the difference is that in NMOS devices the internal timing circuits are driven by the signal at XTAL2 whereas in the CMOS devices they are driven by the signal at XTAL1.

Machine cycles

The oscillator formed by the crystal and associated circuit generates a pulse train at the crystal frequency f_{osc}. This pulse train sets the smallest interval of time P that exists within the microcontroller. The minimum time required by the microcontroller to complete a simple instruction, or part of a more complex instruction, is the machine cycle. The machine cycle consists of a sequence of six states, numbered S1 through to S6, with each state time lasting for two oscillator periods. Thus a machine cycle takes 12 oscillator periods. Each state is divided into a phase 1 half (P1) and a phase 2 half (P2). Figure C.22 shows the fetch/execute sequences in states and phases for various kinds of instructions.

Normally two program fetches are generated during each machine cycle even if the instruction being fetched does not require it. If the instruction being executed does not need extra code bytes the CPU ignores the extra fetch and the PC is not incremented.

Execution of a one-cycle instruction begins during state 1 of the machine cycle with the opcode latched into the instruction register. A second fetch occurs during S4 of the same machine cycle and execution is completed at the end of S6 of the machine cycle.

The MOV X instructions take two machine cycles to complete and no program fetch is generated during the second cycle of the instruction. This is the only time that program fetches are skipped. The sequences described are the same regardless of whether the program memory is internal or external to the chip since execution times do not depend on the location of code memory.

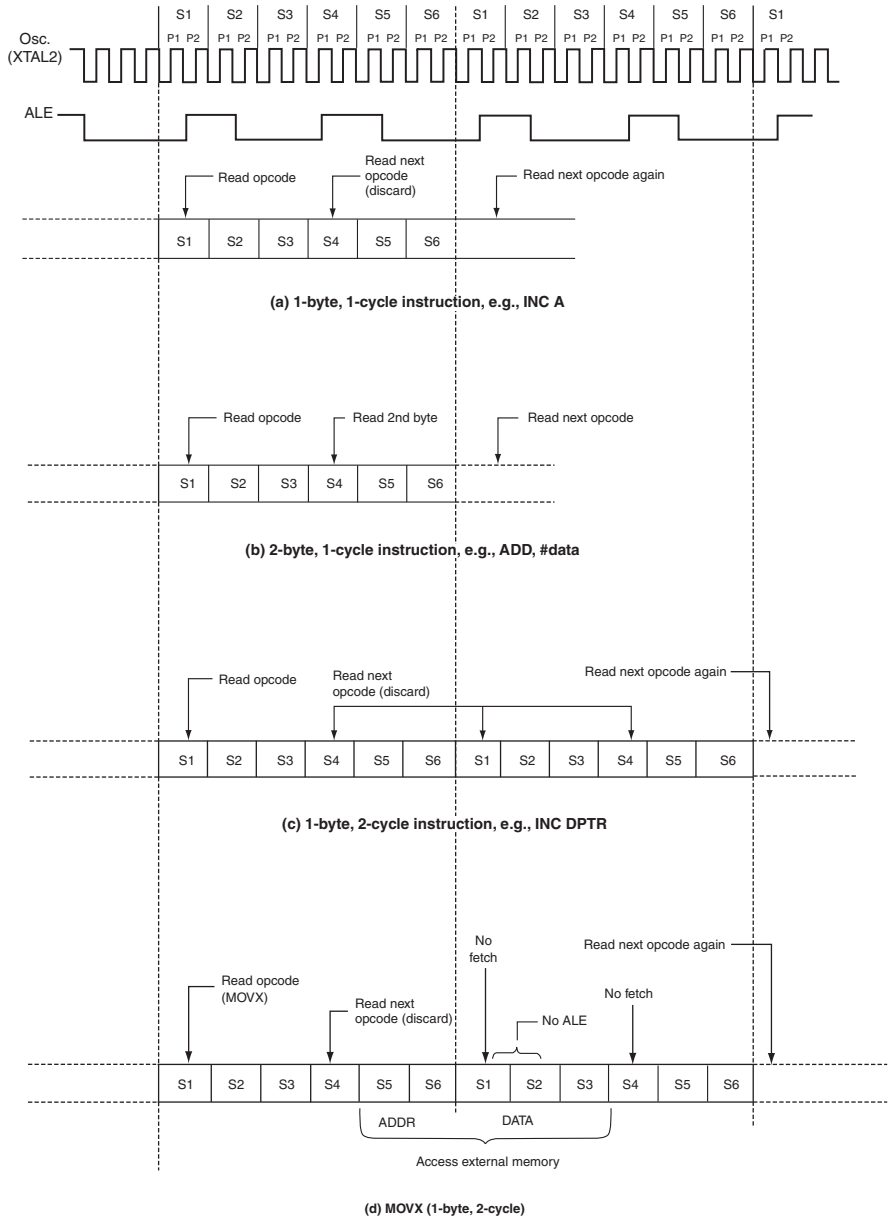


Figure C.22 80C51 family state sequences (courtesy Philips Semiconductors)

The above description defines the situation for those devices that operate using 12 oscillator periods. Some devices within the 8051 family are designed to operate on six oscillator periods while the LPC device described in Appendix E operates on two oscillator periods and is thus six times faster in operation than standard 8051 devices for a given clock frequency.

Appendix D

P89C66x Microcontroller

Details of this device are reproduced with kind permission of Philips Semiconductors. Data regarding the device may be found on the Philips website at www.semiconductors.philips.com. The P89C660/662/664/666 device contains a non-volatile 16 KB/32 KB/64 KB flash program memory (and 512 B/1 KB/2 KB/8 KB RAM) that is both parallel programmable and serial in-system and in-application programmable. In-system programming (ISP) allows the user to download new code while the microcontroller sits in the application. In-application programming (IAP) means that the microcontroller fetches new program code and reprograms itself while in the system. This allows for remote programming over a modem link. A default serial loader (boot loader) program in ROM allows serial ISP of the flash memory via the UART without the need for a loader in the flash code. For IAP, the user program erases and reprograms the flash memory by use of standard routines contained in ROM. This device executes one instruction in 6 clock cycles, hence providing twice the speed of a conventional 80C51. A one-time programmable (OTP) configuration bit gives the user the option to select conventional 12-clock timing. This device is a single-chip 8-bit microcontroller manufactured in advanced CMOS process and is a derivative of the 80C51 microcontroller family. The instruction set is 100% executing and timing compatible with the 80C51 instruction set. The device also has four 8-bit I/O ports, three 16-bit timer/event counters, a multi-source, four-priority-level, nested interrupt structure, an enhanced UART and on-chip oscillator and timing circuits. The added features of the P89C660/662/664/668 make it a powerful microcontroller for applications that require pulse width modulation, high-speed I/O an up/down counting capabilities such as motor control.

Features include:

- 80C51 central processing unit;
- on-chip flash program memory with ISP and IAP capability;
- boot ROM contains low-level flash programming routines for downloading via the UART;
- can be programmed by the end-user application (IAP);

- parallel programming with 87C51 compatible hardware interface to programmer;
- six clocks per machine cycle operation (standard);
- 12 clocks per machine cycle operation (optional);
- speed up to 20 MHz with 6 clock cycles per machine cycle (40 MHz equivalent performance); up to 33 MHz with 12 clocks per machine cycle;
- fully static operation;
- RAM externally expandable to 64 KB;
- four interrupt priority levels;
- eight interrupt sources;
- four 8-bit I/O ports;
- full-duplex enhanced UART
    1. framing error detection
    2. automatic address recognition;
- power control modes
    1. clock can be stopped and resumed
    2. idle mode
    3. power-down mode;
- programmable clock out;
- second DPTR register;
- asynchronous port reset;
- low EMI (Inhibit ALE);
- I^2C serial interface;
- programmable counter array (PCA)
    1. PWM
    2. capture/compare;
- well suited for IPMI applications.

The basic block diagram is shown in Figure D.1.

D.1 Pin-out diagram for the 89C66x

Packages include a 44-pin plastic leaded chip carrier (PLCC) package and a 44-pin low quad flat pack (LQFP) package. The PLCC package is illustrated in Figure D.2. Note that although both packages have 44 pins only 40 pins in each case are utilised since four pins have no internal connections.

A brief description of the function of each of the pins is given in the text that follows. Note that the pin number refers to the PLCC package. The functions of the LQFP package are the same as for the LPCC package but pin numbers vary between the packages.

Supply voltage (V_{cc} and V_{SS}). The device operates from a single supply connected to pin 44 (V_{cc}) while pin 22 (V_{SS}) is grounded.

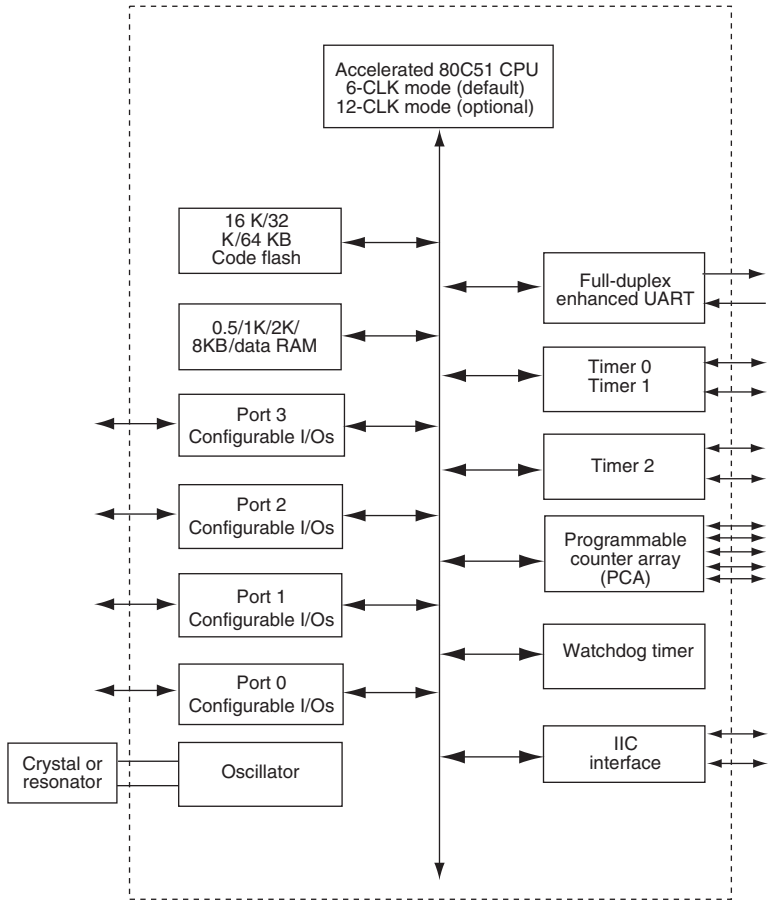
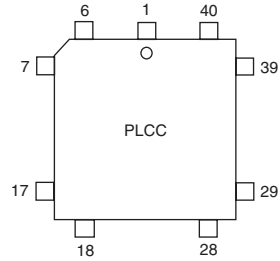


Figure D.1 89C66x block diagram (courtesy Philips Semiconductors)

Input/output (I/O) ports. Thirty-two of the pins are arranged as four 8-bit I/O ports P0–P3 with each capable of operating as a control line or part of the data/address bus in addition to the I/O functions. Details are as follows:

- **Port 0.** This is a dual-purpose port occupying pins 36 to 43 of the device. The port is an open-drain bidirectional I/O port. Pins that have 1s written to them float and can be used as high-impedance inputs. The port may be used with external memory to provide a multiplexed address and data bus. In this application internal pull-ups are used when emitting 1s.
- **Port 1.** This is an 8-bit bidirectional I/O port occupying pins 2 to 9 of the device with internal pull-ups on all pins except P1.6 and P1.7, which are open-drain. Pins that have 1s written to them are pulled high by the internal pull-ups and can be used as inputs; as inputs, pins that are externally pulled



Pin	Function	Pin	Function	Pin	Function
1	NIC*	16	P3.4/T0/CEX3	31	P2.7/A15
2	P1.0/T2	17	P3.5/T1/CEX4	32	$\overline{\text{PSEN}}$
3	P1.1/T2EX	18	P3.6/$\overline{\text{WR}}$	33	ALE
4	P1.2/ECI	19	P3.7/$\overline{\text{RD}}$	34	NIC*
5	P1.3/CEX0	20	XTAL2	35	$\overline{\text{EA}}/V_{pp}$
6	P1.4/CEX1	21	XTAL1	36	P0.7/AD7
7	P1.5/CEX2	22	V_{SS}	37	P0.6/AD6
8	P1.6/SCL	23	NIC*	38	P0.5/AD5
9	P1.7/SDA	24	P2.0/A8	39	P0.4/AD4
10	RST	25	P2.1/A9	40	P0.3/AD3
11	P3.0/R×D	26	P2.2/A10	41	P0.2/AD2
12	NIC*	27	P2.3/A11	42	P0.1/AD1
13	P3.1/T×D	28	P2.4/A12	43	P0.0/AD0
14	P3.2/$\overline{\text{INT0}}$	29	P2.5/A13	44	V_{CC}
15	P3.3/$\overline{\text{INT1}}$	30	P2.6/A14		

*No internal connection

Figure D.2 89C66x 44-pin PLCC package (courtesy Philips Semiconductors)

low will source current via the internal pull-ups. The port pins have alternate functions as follows:

(P1.0) T2 Timer/counter 2 external count input/clockout
(P1.1) T2EX Timer/counter 2 reload/capture/direction control
(P1.2) ECI External clock input to the PCA
(P1.3) CEX0 Capture/compare external I/O for PCA module 0
(P1.4) CEX1 Capture/compare external I/O for PCA module 1
(P1.5) CEX2 Capture/compare external I/O for PCA module 2
(P1.6) SCL I^2C bus clock line (open drain)
(P1.7) SDA I^2C bus data line (open drain)

- **Port 2.** This is an 8-bit bidirectional I/O port occupying pins 24 to 31 of the device with internal pull-ups. Pins that have 1s written to them are pulled high by the internal pull-ups and can be used as inputs; as inputs, pins that are externally pulled low will source current via the internal pull-ups. The port may be used to provide the high-order byte of the address bus for external program memory or external data memory that uses 16-bit addresses. When accessing external data memory that uses 8-bit addresses, the port emits the contents of the P2 register.
- **Port 3.** This is an 8-bit bidirectional I/O occupying pin 11 and pins 13 to 19 of the device with internal pull-ups. The specification is similar to that of

port 1. These pins, in addition to the I/O role, serve the special features of the 89C66x family; the alternate functions are summarised below:

P3.0 RxD	serial data input port
P3.1 TxD	serial data output port
P3.2 $\overline{\text{INT0}}$	external interrupt 0
P3.3 $\overline{\text{INT1}}$	external interrupt 1
P3.4 CEX3/T0	timer 0 external input; capture/compare external I/O for PCA module 3
P3.5 CEX4/T1	timer 1 external input; capture/compare external I/O for PCA module 4
P3.6 $\overline{\text{WR}}$	external data memory write strobe
P3.7 $\overline{\text{RD}}$	external data memory read strobe.

RESET (RST) (pin 10). The 89C66x is reset by holding this input high for a minimum of two machine cycles before returning it low for normal running. An internal resistance connects to pin 22 (V_{SS}) allowing a power-on reset using an external capacitor connected to pin 44 (V_{cc}).

XTAL1 and XTAL2 (pins 21 and 20 respectively). The 89C66x on-chip oscillator is driven, usually, from an external crystal. The XTAL1 input also provides an input to the internal clock generator circuits.

$\overline{\text{PSEN}}$ (program store enable) (pin 32). This pin provides an output read strobe to external program memory. When executing code from the external program memory, $\overline{\text{PSEN}}$ is activated twice each machine cycle, except that two $\overline{\text{PSEN}}$ activations are skipped during each access to external data memory. The signal is not activated during a fetch from internal memory.

ALE (address latch enable) (pin 33). The ALE signal is an output pulse used to latch the low byte of an address during access to external memory. In normal operation ALE is emitted twice every machine cycle and can be used for external timing or clocking. Note that one ALE pulse is skipped during each access to external data memory. ALE can be disabled by setting SFR auxiliary0. With this bit set ALE will be active only during a MOVX instruction.

$\overline{\text{EA}}/V_{pp}$ (external access/programming voltage) (pin 35). This pin is either held high or low according to circuit requirements. If held low the device will fetch code from external program memory locations. If held high the device executes programs from internal memory. The value on the pin is latched when RST is released and any subsequent changes have no effect. The pin also receives the programming supply voltage (V_{pp}) during flash programming.

D.2 Memory organisation

The P89C660/662/664/668 has internal data memory that is mapped into four separate segments: the lower 128 bytes of RAM, upper 128 bytes of RAM,

128 bytes SFR and 256 bytes expanded RAM (ERAM) (256 bytes for the '660; 768 bytes for the '662; 1792 bytes for the '664; 7936 bytes for the '668). The four segments are:

1. The lower 128 bytes of RAM (addresses 00H to 7FH), which are directly and indirectly addressable.
2. The upper 128 bytes of RAM (addresses 80H to FFH), which are indirectly addressable only.
3. The SFRs (addresses 80H to FFH), which are directly addressable only.
4. The 256/768/1792/7936-bytes expanded RAM (ERAM, 00H – FFH/2FFH/6FFH/1FFFH), which are indirectly accessed by move external instruction, MOVX. and with the EXTRAM bit cleared, see AUXR (Auxiliary Register), Table D.1.

AUXR (AUXILIARY REGISTER)

address 8EH

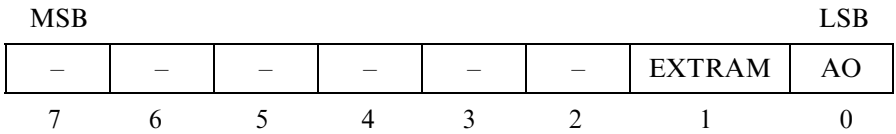


Table D.1 AUXR (auxiliary register)

Bit	Symbol	Function
7–2	–	Reserved for future use
1	EXTRAM	Internal/external RAM access using MOVX@Ri/@DPTR EXTRAM Operating mode 0 Internal ERAM access using MOVX@Ri/@DPTR 1 External data memory access
0	AO	Disable/enable ALE AO Operating mode 0 ALE is emitted at a constant rate of 1/3 the oscillator frequency (6 clock mode, 1/6 f_{osc} in 12 clock mode) 1 ALE is active only during off-chip memory access

The register is not bit addressable.

The lower 128 bytes can be accessed by either direct or indirect addressing. The upper 128 bytes can be accessed by indirect addressing only. The upper 128 bytes occupy the same address space as the SFR. That means they have the same address, but are physically separate from SFR space. When an instruction accesses an internal location above address 7FH, the CPU knows whether the access is to the upper 128 bytes of data RAM, or to SFR space by the addressing mode used in the instruction. Instructions that use direct addressing access SFR space. For example:

MOV 0A0H,A accesses the SFR at location 0A0H (which is P2)

Instructions that use indirect addressing, access the upper 128 bytes of data RAM. For example:

MOV @R0,A

where R0 contains 0A0H, accesses the data byte at address 0A0H, rather than P2 (whose address is 0A0H).

The ERAM can be accessed by indirect addressing, with EXTRAM bit cleared and MOVX instructions. This part of memory is physically located on-chip, logically occupies the first 256 bytes (660), 768 (662), 1792 (664); 7936 (668) of external data memory. With EXTRAM = 0, the ERAM is indirectly addressed, using the MOVX instruction in combination with any of the registers R0, R1 of the selected bank or DPTR. An access to ERAM will not affect ports P0, P3.6 (WR#) and P3.7 (RD#). The P2 SFR is in output state during external addressing. For example, with EXTRAM = 0,

MOVX @R0,A

where R0 contains 0A0H, accesses the ERAM at address 0A0H rather than external memory. An access to external data memory locations higher than the ERAM will be performed with the MOVX DPTR instructions in the same way as in the standard 80C51 (with P0 and P2 as data/address bus, and P3.6 and P3.7 as write and read timing signals; refer to Figure D.3).

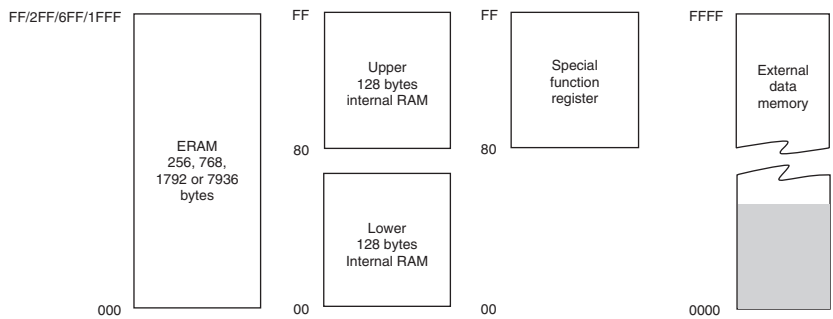


Figure D.3 89C66x internal/external data memory address spaces (courtesy Philips Semiconductors)

With EXTRAM = 1, MOVX @Ri and MOVX @DPTR will be similar to the standard 80C51. MOVX @ Rl will provide an 8-bit address multiplexed with data on port 0 and any output port pins can be used to output higher order address bits. This is to provide the external paging capability. MOVX @DPTR will generate a 16-bit address. Port 2 outputs the high-order eight address bits (the contents of DPH) while port 0 multiplexes the low-order eight address bits (the contents of DPL) with data: MOVX @RI and MOVX @DPTR will generate either read or write signals on P3.6 ($\overline{\text{WR}}$) and P3.7 ($\overline{\text{RD}}$). The SP may be located anywhere in the 256 bytes RAM (lower and upper RAM) internal data memory. The stack may not be located in the ERAM.

FLASH EPROM MEMORY

The P89C660/662/664/668 flash memory augments EPROM functionality with in-circuit electrical erasure and programming. The flash can be read and written as bytes. The chip erase operation will erase the entire program memory. The block erase function can erase any flash byte block; ISP and standard parallel programming are both available. On-chip erase and write timing generation contribute to a user-friendly programming interface. The P89C66x flash reliably stores memory contents even after 10 000 erase and program cycles. The cell is designed to optimise the erase and programming mechanisms. In addition, the combination of advanced tunnel oxide processing and low internal electric fields for erase and programming operations, produces reliable cycling. The P89C66x uses a +5 V V_{pp} supply to perform the program/erase algorithms.

ISP and IAP

- Flash EPROM internal program memory with block erase.
- Internal 1 KB fixed boot ROM, containing low-level ISP routines and a default serial loader. User program can call these routines to perform IAP. The Boot ROM can be turned off to provide access to the full 64 KB of flash memory.
- Boot vector allows user provided Flash loader code to reside anywhere in the Flash memory space. This configuration provides flexibility to the user.
- Default loader in boot ROM allows programming via the serial port without the need for a user provided loader.
- Up to 64 KB of external program memory if the internal program memory is disabled ($\overline{EA} = 0$).
- Programming and erase voltage +5 V (+12 V tolerant).
- Read/Programming/Erase using ISP/IAP:

    1. Byte programming (20 μs)
    2. Typical quick erase times:

        Block erase (8 KB or 16 KB) in 10 s.
        Full erase (64 KB) in 20 s.

- ISP
- Programmable security for the code in the flash
- 10 000 minimum erase/program cycles for each byte
- 10-year minimum data retention

Flash organisation

The P89C660/662/664/668 contains 16 KB/32 KB/64 KB of flash program memory. This memory is organised as five separate blocks. The first two blocks are 8 KB in size, filling the program memory space from address 0 through 3FFF hex. The final three blocks are 16 KB in size and occupy addresses from 4000 through FFFF hex. Figure D.4 illustrates the flash memory configurations.

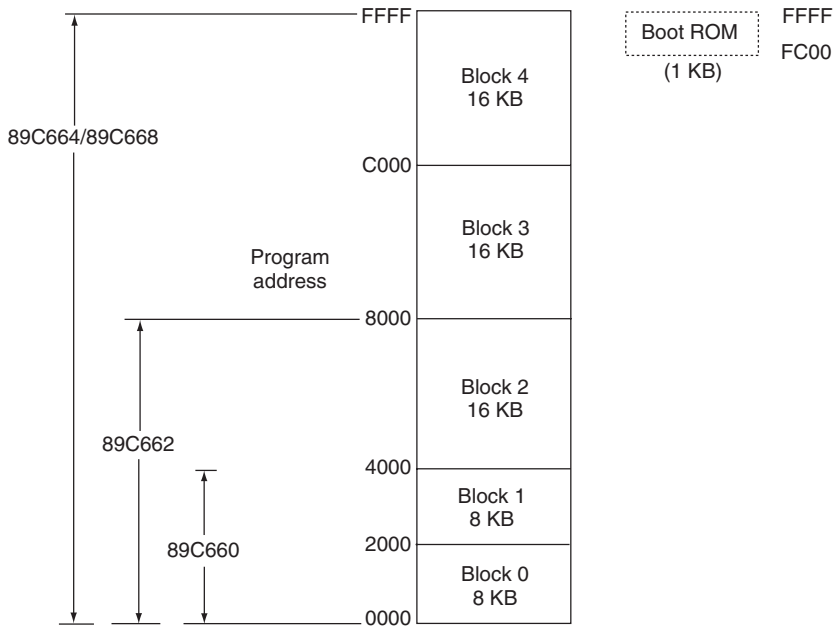


Figure D.4 Flash memory configurations (courtesy Philips Semiconductors)

Flash programming and erasure

There are three methods of erasing or programming of the flash memory that may be used. First, the flash may be programmed or erased in the end-user application by calling low-level routines through a common entry point in the boot ROM. The end-user application, though, must be executing code from a different block than the block that is being erased or programmed. Second, the on-chip ISP boot loader may be invoked. This ISP boot loader will, in turn, call low-level routines through the same common entry point in the boot ROM that can be used by the end-user application. Third, the flash may be programmed or erased using the parallel method by using a commercially available EPROM programmer. The parallel programming method used by these devices is similar to that used by EPROM 87C51, but it is not identical, and the commercially available programmer will need to have support for these devices.

BOOT ROM

When the microcontroller programs its own flash memory, all of the low-level details are handled by code that is permanently contained in a 1 KB 'Boot ROM' that is separate from the flash memory. A user program simply calls the common entry point with appropriate parameters in the boot ROM to accomplish the desired operation. Boot ROM operations include things like: erase block, program byte, verify byte, program security lock bit, etc. The boot ROM overlays the program memory space at the top of the address space from

FC00 to FFFF hex, when it is enabled. The boot ROM may be turned off so that the upper 1 KB of flash program memory is accessible for execution.

D.3 Special function registers (SFRs)

Details of the SFRs in the 89C66x family are shown in Table D.2. Appendix C deals with many of the SFRs that are common to the 80C51 and the 89C66x family and if necessary reference should be made to Appendix C if an SFR is not covered in detail in this appendix.

D.4 Timer/counters

Information regarding timers 0, 1 and 2 is discussed fully in Appendix C for the standard 80C51 device and the detail is no different for the 89C66x family. However, because the 89C66x devices can operate in the 6-clock mode, allowance should be made for this where relevant in the timer/counter section of Appendix C. For example, Figure C.13 of Appendix C shows timer 2 in capture mode, with the oscillator frequency shown as being divided by 12. This is correct for the 80C51 device and for the 89C66x device in 12-clock mode. For the 89C66x device in 6-clock mode the oscillator frequency is divided by 6.

Similarly, Table C.7 of Appendix C shows timer 2 generated commonly used baud rates. The baud rate shown in the Table for the 80C51 device is the same for the 89C66x device in 12-clock mode but the value is doubled for the 89C66x device 6-clock mode i.e. 375 k becomes 750 k, etc. Finally the formula for timer 2 baud rate when the timer is being clocked internally is shown in Appendix C as:

$$\frac{\text{Oscillator frequency}}{32\ [65536 - (\text{RCAP2H},\ \text{RCAP2L})]}$$

which can be written, more generally, as:

$$\frac{\text{Oscillator frequency}}{n\ [65536 - (\text{RCAP2H},\ \text{RCAP2L})]}$$

where $n = 16$ in 6-clock mode and $n = 32$ in 12-clock mode. (RCAP2H, RCAP2L) is the content of RCAP2H RCAP2L taken as a 16-bit unsigned integer.

To obtain the reload value for RCAP2H and RCAP2L, the above equation can be rewritten as:

$$\text{RCAP2H},\ \text{RCAP2L} = 65536 - [\,f_{\text{osc}}/(n \times \text{baud rate})]$$

where n has the same values as indicated above and f_{osc} = oscillator frequency. There is a 16-bit timer/counter used in the 89C66x that is not present in the standard 80C51 device. This is the programmable counter array (PCA) and its details follow.

Table D.2 89C66x special function registers (courtesy Philips Semiconductors)

Symbol	Description	Direct address	MSB			Bit address, symbol, or alternative port function				LSB	Reset value
ACC*	Accumulator	E0H	E7	E6	E5	E4	E3	E2	E1	E0	00H
AUXR#	Auxillary	8EH	–	–	–	–	–	–	EXTRAM	AO	xxxxxx10B
AUXR1#	Auxillary 1	A2H	–	–	ENBOOT	–	GF2	0	–	DPS	xxxxx0x0B
B*	B register	F0H	F7	F6	F5	F4	F3	F2	F1	F0	00H
CCAP0H#	Module 0 capture high	FAH									xxxxxxxxB
CCAP1H#	Module 1 capture high	FBH									xxxxxxxxB
CCAP2H#	Module 2 capture high	FCH									xxxxxxxxB
CCAP3H#	Module 3 capture high	FDH									xxxxxxxxB
CCAP4H#	Module 4 capture high	FEH									xxxxxxxxB
CCAP0L#	Module 0 capture low	EAH									xxxxxxxxB
CCAP1L#	Module 1 capture low	EBH									xxxxxxxxB
CCAP2L#	Module 2 capture low	ECH									xxxxxxxxB
CCAP3L#	Module 3 capture low	EDH									xxxxxxxxB
CCAP4L#	Module 4 capture low	EEH									xxxxxxxxB
CCAPM0#	Module 0 mode	C2H	–	ECOM	CAPP	CAPN	MAT	TOG	PWM	ECCF	x0000000B
CCAPM1#	Module 1 mode	C3H	–	ECOM	CAPP	CAPN	MAT	TOG	PWM	ECCF	x0000000B
CCAPM2#	Module 2 mode	C4H	–	ECOM	CAPP	CAPN	MAT	TOG	PWM	ECCF	x0000000B
CCAPM3#	Module 3 mode	C5H	–	ECOM	CAPP	CAPN	MAT	TOG	PWM	ECCF	x0000000B
CCAPM4#	Module 4 mode	C6H	–	ECOM	CAPP	CAPN	MAT	TOG	PWM	ECCF	x0000000B
			C7	C6	C5	C4	C3	C2	C1	C0	
CCON*#	PCA counter control	C0H	CF	CR	–	CCF4	CCF3	CCF2	CCF1	CCF0	00x00000B

Table D.2 Continued

Symbol	Description	Direct address	Bit address, symbol, or alternative port function MSB							LSB	Reset value
CH#	PCA counter high	F9H									00H
CL#	PCA counter low	E9H									00H
CMOD#	PCA counter mode	C1H	CIDL	WDTE	–	–	–	CPS1	CPS0	ECF	00xxx000B
DPTR:	Data pointer (2 bytes)										
DPH	Data pointer high	83H									00H
DPL	Data pointer low	82H									00H
			AF	AE	AD	AC	AB	AA	A9	A8	
IEN0*	Interrupt enable 0	A8H	EA	EC	ES1	ES0	ET1	EX1	ET0	EX0	00H
IEN1*	Interrupt enable 1	E8	–	–	–	–	–	–	–	ET2	xxxxxxx0B
			BF	BE	BD	BC	BB	BA	B9	B8	
IP*	Interrupt priority	B8H	PT2	PPC	PS1	PS0	PT1	PX1	PT0	PX0	x0000000B
IPH*	Interrupt priority high	B7H	PT2H	PPCH	PS1H	PS0H	PT1H	PX1H	PT0H	PX0H	x0000000B
			87	86	85	84	83	82	81	80	
P0*	Port 0	80H	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	FFH
			97	96	95	94	93	92	91	90	
P1*	Port 1	90H	SDA	SCL	CEX2	CEX1	CEX0	ECI	T2EX	T2	FFH
			A7	A6	A5	A4	A3	A2	A1	A0	
P2*	Port 2	A0H	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8	FFH
			B7	B6	B5	B4	B3	B2	B1	B0	
P3*	Port 3	B0H	$\overline{RD}$	$\overline{WR}$	T1/CEX4	T0/CEX3	$\overline{INT1}$	$\overline{INT0}$	TxD	RxD	FFH
PCON#[1]	Power control	87H	SMOD1	SMOD0	–	POF	GF1	GF0	PD	IDL	00xxx000B
			D7	D6	D5	D4	D3	D2	D1	D0	
PSW*	Program status word	D0H	CY	AC	F0	RS1	RS0	OV	F1	P	00000000B
RCAP2H#	Timer 2 capture high	CBH									00H
RCAP2L#	Timer 2 capture low	CAH									00H
SADDR#	Slave address	A9H									00H
SADEN#	Slave address mask	B9H									00H

S0BUF	Serial data buffer	99H									xxxxxxxxB
			9F	9E	9D	9C	9B	9A	99	98	
S0CON*	Serial control	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	00H
SP	Stack pointer	81H									07H
S1DAT#	Serial 1 data	DAH									00H
S1ADR#	Serial 1 address	DBH	Slave address							GC	00H
S1STA#	Serial 1 status	D9H	SC4	SC3	SC2	SC1	SC0	0	0	0	F8H
			DF	DE	DD	DC	DB	DA	D9	D8	
S1CON*#	Serial 1 control	D8H	CR2	ENS1	STA	ST0	SI	AA	CR1	CR0	00000000B
			8F	8E	8D	8C	8B	8A	89	88	
TCON*	Timer control	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00H
			CF	CE	CD	CC	CB	CA	C9	C8	
T2CON*	Timer 2 control	C8H	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/$\overline{RL2}$	00H
T2MOD#	Timer 2 mode control	C9H	–	–	–	–	–	–	T2OE	DCEN	xxxxxx00B
TH0	Timer high 0	8CH									00H
TH1	Timer high 1	8DH									00H
TH2#	Timer high 2	CDH									00H
TL0	Timer low 0	8AH									00H
TL1	Timer low 1	8BH									00H
TL2#	Timer low 2	CCH									00H
TMOD	Timer mode	89H	GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0	00H
WDTRST	Watchdog timer reset	A6H									

* SFRs are bit addressable.
\# SFRs are modified from or added to the 80C51 SFRs.
– Reserved bits.
1. Reset value depends on reset source.

PROGRAMMABLE COUNTER ARRAY (PCA)

The programmable counter array available on the 89C66x is a special 16-bit timer that has five 16-bit capture/compare modules associated with it. Each of the modules can be programmed to operate in one of four modes: rising and/or falling edge capture, software timer, high-speed output, or pulse width modulator. Each module has a pin associated with it in port 1. Module 0 is connected to P1.3 (CEXO), module 1 to P1.4 (CEX1), etc. The basic PCA configuration is shown in Figure D.5.

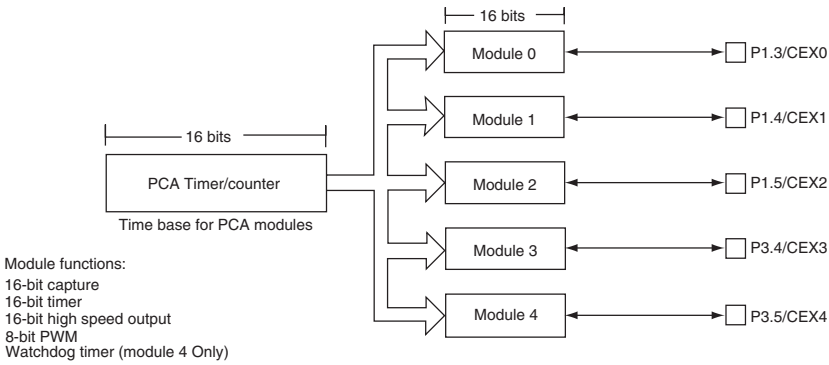


Figure D.5 Programmable counter array (PCA) (courtesy Philips Semiconductors)

The PCA timer is a common time base for all five modules and can be programmed to run at: 1/6 the oscillator frequency; 1/2 the oscillator frequency; the timer 0 overflow; or the input on the ECI pin (P1.2). The timer count source is determined from the CPS1 and CPS0 bits in the CMOD SFR as shown in Table D.3.

CMOD
address C1H

MSB							LSB
CIDL	WDTE	–	–	–	CPS1	CPS0	ECF
7	6	5	4	3	2	1	0

In the CMOD SFR, there are three additional bits associated with the PCA. They are CIDL which allows the PCA to stop during idle mode, WDTE which enables or disables the watchdog function on module 4, and ECF which, when set, causes an interrupt and the PCA overflow flag CF (in the CCON SFR) to be set when the PCA timer overflows. These functions are shown in Figure D.6.

The watchdog timer function is implemented in module 4 (see Figure D.12). The CCON SFR contains the run control bit for the PCA, and the flags for the PCA timer (CF) and each module. Details of the CCON SFR are shown in Table D.4.

Table D.3 PCA counter mode register (CMOD)

Bit	Symbol	Function
7	CIDL	Counter idle control. CIDL = 0 programs the PCA counter to continue functioning during idle mode. CIDL = 1 programs it to be gated off during idle
6	WDTE	Watchdog timer enable. WDTE = 0 disables watchdog timer function on PCA Module 4. WDTE = 1 enables it
5, 4, 3	–	Reserved for future use
2	CPS1	PCA count pulse select bit 1
1	CPS0	PCA count pulse select bit 0
	CPS1 CPSO	Selected PCA input
	0 0	Internal clock. 1/6 oscillator frequency (6 clock mode); 1/12 oscillator frequency (12 clock mode)
	0 1	Internal clock. 1/2 oscillator frequency (6 clock mode); 1/4 oscillator frequency (12 clock mode)
	1 0	Timer 0 overflow
	1 1	External clock at ECI pin (P1.2) (maximum rate = 1/4 oscillator frequency in 6-clock mode, 1/8 oscillator frequency in 12-clock mode)
0	ECF	PCA Enable Counter Overflow Interrupt. ECF = 1 enables CF bit in CCON to generate an interrupt. ECF = 0 disables that function of CF.

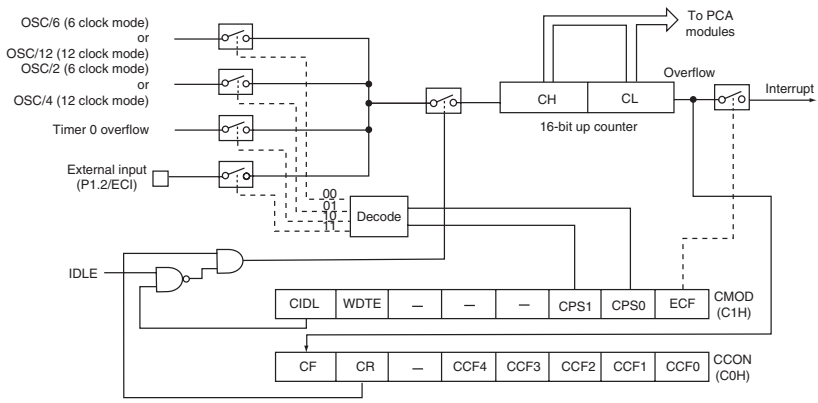


Figure D.6 PCA timer/counter (courtesy Philips Semiconductors)

CCON
address 0C0H
bit addressable

MSB							LSB
CF	CR	–	CCF4	CCF3	CCF2	CCF1	CCF0
7	6	5	4	3	2	1	0

Table D.4 PCA counter control register (CCON)

Bit	Symbol	Function
7	CF	PCA counter overflow flag. Set by hardware when the counter rolls over. CF flags an interrupt if bit ECF in CMOD is set. CF may be set either by hardware or software but can only be cleared by software
6	CR	PCA counter run control bit. Set by software to turn the PCA counter on. Must be cleared by software to turn the PCA counter off
5	–	Reserved for future use
4	CCF4	PCA module 4 interrupt flag. Set by hardware when a match or capture occurs. Must be cleared by software
3	CCF3	PCA module 3 interrupt flag. Set by hardware when a match or capture occurs. Must be cleared by software
2	CCF2	PCA module 2 interrupt flag. Set by hardware when a match or capture occurs. Must be cleared by software
1	CCF1	PCA module 1 interrupt flag. Set by hardware when a match or capture occurs. Must be cleared by software
0	CCF0	PCA module 0 interrupt flag. Set by hardware when a match or capture occurs. Must be cleared by software

To run the PCA the CR bit (CCON.6) must be set by software. The PCA is shut off by clearing this bit. The CF bit (CCON.7) is set when the PCA counter overflows and an interrupt will be generated if the ECF bit in the CMOD register is set. The CF bit can only be cleared by software. Bits 0 through 4 of the CCON register are the flags for the modules (bit 0 for module 0, bit 1 for module 1, etc.) and are set by hardware when either a match or a capture occurs. These flags also can only be cleared by software. The PCA interrupt system is shown in Figure D.7.

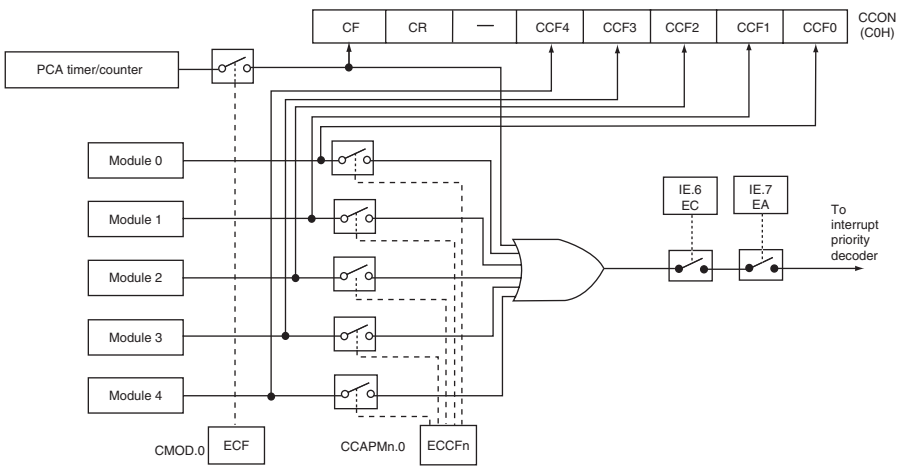


Figure D.7 PCA interrupt system (courtesy Philips Semiconductors)

Each module in the PCA has a SFR associated with it. These registers are: CCAPM0 for module 0, CCAPM1 for module 1, etc. Details are shown in Table D.5.

CCAPMN

address: CCAPM0 0C2H
CCAPM1 0C3H
CCAPM2 0C4H
CCAPM3 0C5H
CCAPM4 0C6H
not bit addressable

MSB							LSB
–	ECOMn	CAPPn	CAPNn	MATn	TOGn	PWMn	ECCFn
7	6	5	4	3	2	1	0

Table D.5 PCA modules compare/capture registers (CCAPMn)

Bit	Symbol	Function
7	–	Reserved for future use
6	ECOMn	Enable comparator. ECOMn = 1 enables the comparator function
5	CAPPn	Capture positive. CAPPn = 1 enables positive edge capture
4	CAPNn	Capture negative. CAPNn = 1 enables negative edge capture
3	MATn	Match. When MATn = 1, a match of the PCA counter with this module's compare/capture register causes the CCFn bit in CCON to be set, flagging an interrupt
2	TOGn	Toggle. When TOGn = 1, a match of the PCA counter with this module's compare/capture register causes the CEXn pin to toggle
1	PWMn	Pulse width modulation. PWMn = 1 enables the CEXn pin to be used as a pulse width modulated output
0	ECCFn	Enable CCF interrupt. Enables compare/capture flag CCFn in the CCON register to generate an interrupt

The registers contain the bits that control the mode that each module will operate in. The ECCF bit (CCAPMn.0 where n = 0,1,2,3 or 4 depending on the module) enables the CCF flag in the CCON SFR to generate an interrupt when a match or compare occurs in the associated module. PWM (CCAPMn.1) enables the pulse width modulation mode. The TOG bit (CCAPMn.2), when set, causes the CEX output associated with the module to toggle when there is a match between the PCA counter and the module's capture/compare register. The match bit MAT (CCAPMn.3), when set, will cause the CCFn bit in the CCON register to be set when there is a match between the PCA counter and the module's capture/compare register.

The next two bits CAPN (CCAPMn.4) and CAPP (CCAPMn.5) determine the edge that a capture input will be active on. The CAPN bit enables the negative edge, and the CAPP bit enables the positive edge. If both bits are set, both edges will be enabled and a capture will occur for either transition. The last bit ECOM (CCAPMn.6), when set, enables the comparator function. Table D.6 shows the CCAPMn settings for the various PCA functions.

Table D.6 PCA module modes (CCAPMn register)

–	ECOMn	CAPPn	CAPNn	MATn	TOGn	PWMn	ECCFn	Module function
X	0	0	0	0	0	0	0	No operation
X	X	1	0	0	0	0	X	16-bit capture by a positive-edge trigger on CEXn
X	X	0	1	0	0	0	X	16-bit capture by a negative-edge trigger on CEXn
X	X	1	1	0	0	0	X	16-bit capture by a transition on CEXn
X	1	0	0	1	0	0	X	16-bit software timer
X	1	0	0	1	1	0	X	16-bit high speed output
X	1	0	0	0	0	1	0	8-bit PWM
X	1	0	0	1	X	0	X	Watchdog timer

There are two additional registers associated with each of the PCA modules. They are CCAPnH and CCAPnL and these are the registers that store the 16-bit count when a capture occurs or a compare should occur. When a module is used in the PWM mode these registers are used to control the duty cycle of the output.

PCA CAPTURE MODE

To use one of the PCA modules in the capture mode, either one or both of the CCAPM bits, CAPN and CAPP, for that module must be set. The external CEX input for the module (on port 1) is sampled for a transition. When a valid transition occurs, the PCA hardware loads the value of the PCA counter registers (CH and CL) into the module's capture registers (CCAPnL and CCAPnH). If the CCFn bit for the module in the CCON SFR and the ECCFn bit in the CCAPMn SFR are set, then an interrupt will be generated. Figure D.8 shows the PCA capture mode.

16-BIT SOFTWARE TIMER MODE

The PCA modules can be used as software timers by setting both the ECOM and MAT bits in the modules CCAPMn register. The PCA timer will be

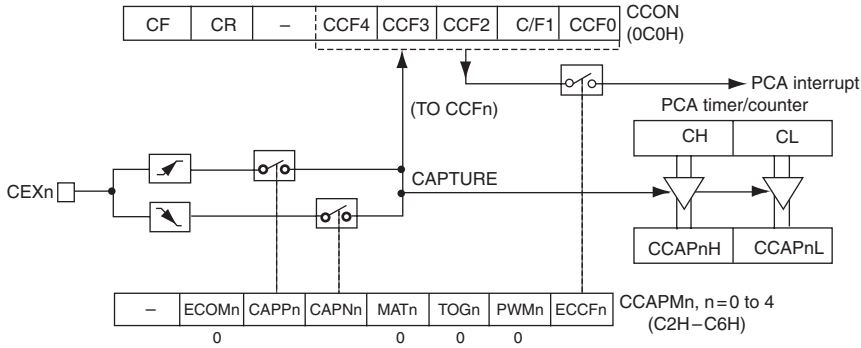


Figure D.8 PCA capture mode (courtesy Philips Semiconductors)

compared to the module's capture registers, and when a match occurs, an interrupt will occur if the CCFn (CCON SFR) and the ECCFn (CCAPMn SFR) bits for the module are both set, see Figure D.9.

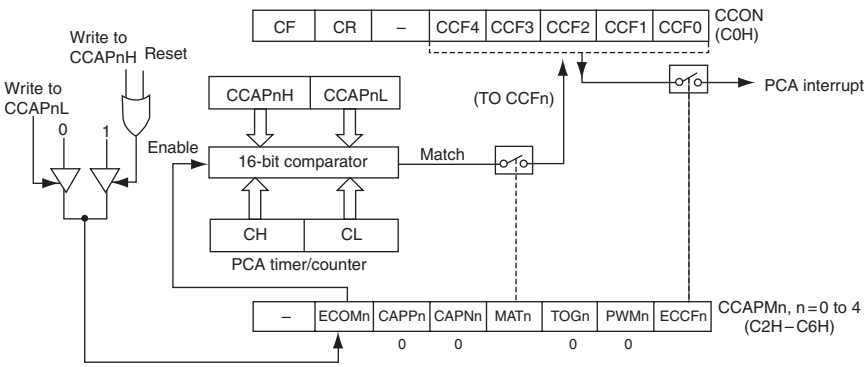


Figure D.9 PCA compare mode (courtesy Philips Semiconductors)

HIGH SPEED OUTPUT MODE

In this mode, the CEX output (on port 1) associated with the PCA module will toggle each time a match occurs between the PCA counter and the module's capture registers. To activate this mode, the TOG, MAT and ECOM bits in the module's CCAPMn SFR must be set, see Figure D.10.

PULSE WIDTH MODULATOR MODE

All of the PCA modules can be used as PWM outputs. Figure D.11 shows the PWM function. The frequency of the output depends on the source for the PCA timer. All the modules will have the same frequency of output because they all share the PCA timer. The duty cycle of each module is independently

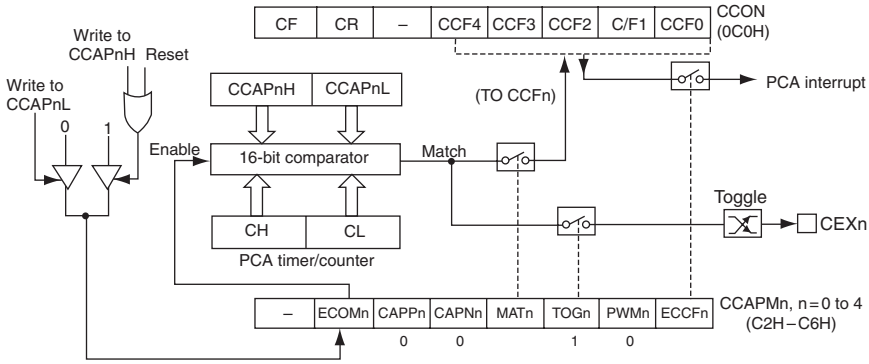


Figure D.10 PCA high-speed output mode (courtesy Philips Semiconductors)

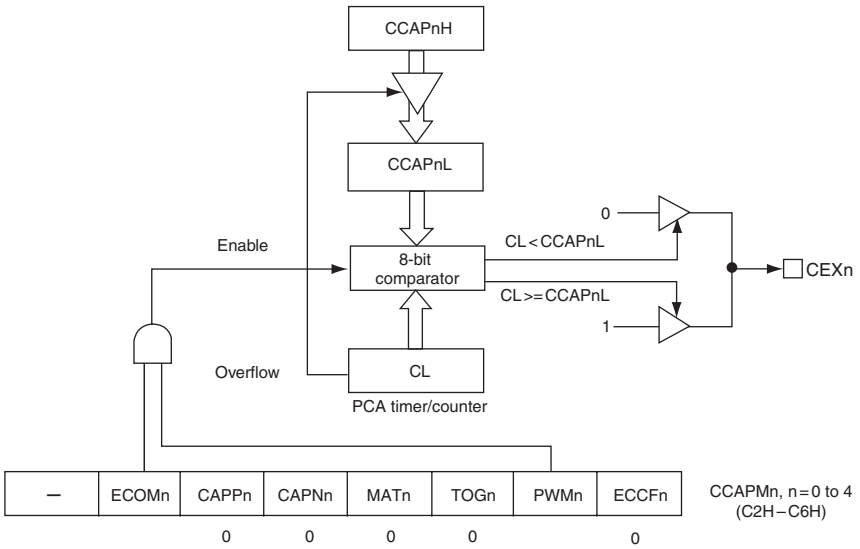


Figure D.11 PCA PWM mode (courtesy Philips Semiconductors)

variable by using the module's capture register CCAPLn. When the value of the PCA CL SFR is less than the value in the module's CCAPLn SFR, the output will be low. When it is equal to or greater than, the output will be high. When CL overflows from FFH to 00H, CCAPLn is reloaded with the value in CCAPHn; this allows PWM update without glitches. The PWM and ECOM bits in the module's CCAPMn register must be set to enable the PWM mode.

PCA WATCHDOG TIMER

An onboard watchdog timer is available with the PCA to improve the reliability of the system without increasing chip count. Watchdog timers are useful

for systems that are susceptible to noise, power glitches or electrostatic discharge. Module 4 is the only PCA module that can be programmed as a watchdog. However, this module can still be used for other modes, if the watchdog is not needed. Figure D.12 shows a diagram of how the watchdog works.

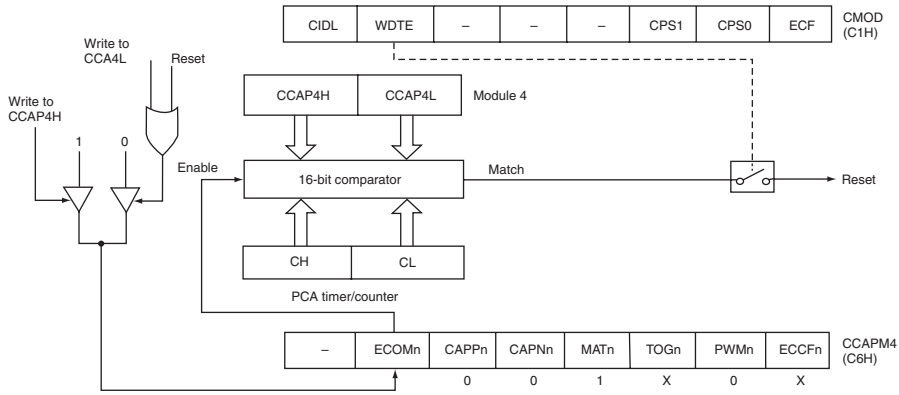


Figure D.12 PCA watchdog timer m (module 4 only) (courtesy Philips Semiconductors)

The user pre-loads a 16-bit value in the compare registers. Just like the other compare modes, this 16-bit value is compared to the PCA timer value. If a match is allowed to occur, an internal reset will be generated. This will not cause the RST pin to be driven high. In order to hold off the reset, the user has three options:

1. Periodically change the compare value, so it will never match the PCA timer.
2. Periodically change the PCA timer value, so it will never match the compare values.
3. Disable the watchdog by clearing the WDTE bit before a match occurs and then re-enable it.

The first two options are more reliable because the watchdog timer is never disabled as in option #3. If the PC ever goes astray, a match will eventually occur and cause an internal reset. The second option is also not recommended if other PCA modules are being used. The PCA timer is the time base for all modules; changing the time base for other modules would not be a good idea. Thus, in most applications the first solution is the best option. The watchdog timer requires initialising using a suitable WATCHDOG routine. Module 4 can be configured in either compare mode, and the WDTE bit in CMOD must also be set. The user's software must periodically change (CCAP4H,CCAP4L) to keep a match from occurring with the PCA timer (CH,CL). The WATCHDOG routine should not be part of an interrupt

service routine, because if the PC goes astray and gets stuck in an infinite loop, interrupts will still be serviced and the watchdog will keep getting reset. Thus, the purpose of the watchdog would be defeated. Instead, this subroutine should be called from the main program within 2^{16} count of the PCA timer.

D.5 Serial interface

The 89C66x device has two serial ports, which can operate independently of each other. The ports are SIO0, which is a full duplex UART port identical to the 80C51, and SIO1, which is used for the I^2C bus.

SIO0. This port operates in the same way as the 80C51 serial port and also uses timer 1 as a baud rate generator.

SIO1. The I^2C bus operates with two lines, SDA (serial data line) and SCL (serial clock line) in order to transfer data between the microcontroller and other devices connected to the bus. For this port to be enabled, the output latches of P1.6 (the SCL line) and P1.7 (the SDA line) must be set to logic 1.

FULL-DUPLEX ENHANCED UART

Standard UART operation

A full-duplex serial port can transmit and receive simultaneously. It is also receive buffered, meaning it can commence reception of a second byte before a previously received byte has been read from the register. (However, if the first byte still has not been read by the time reception of the second byte is complete, one of the bytes will be lost.) The serial port receive and transmit registers are both accessed at SFR SBUF. Writing to SBUF loads the transmit register, and reading SBUF accesses a physically separate receive register. The serial port can operate in four modes:

Mode 0. Serial data enters and exits through RxD. TxD outputs the shift clock. Eight bits are transmitted/received (LSB first). The baud rate is fixed at 1/12 the oscillator frequency in 12-clock mode or 1/6 the oscillator frequency in 6-clock mode.

Mode 1. Ten bits are transmitted (through TxD) or received (through RxD); a start bit (0), 8 data bits (LSB first) and a stop bit (1). On receive, the stop bit goes into RB8 in SFR SCON. The baud rate is variable.

Mode 2. Eleven bits are transmitted (through TxD) or received (through RxD): a 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 in SCON) can be assigned the value of 0 or 1. Or, for example, the parity bit (P, in the PSW) could be moved into TB8. On receive, the 9th data bit goes into RB8 in SFR SCON, while the stop bit is ignored. The baud rate is programmable to either 1/32 or 1/64 the oscillator frequency in 12-clock mode or 1/16 or 1/32 the oscillator frequency in 6-clock mode.

Mode 3. Eleven bits are transmitted (through TxD) or received (through RxD): 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 SBUF as a destination register. Reception is initiated in mode 0 by the condition RI = 0 and REN = 1. Reception is initiated in the other modes by the incoming start bit if REN = 1.

MULTIPROCESSOR COMMUNICATIONS

Modes 2 and 3 have a special provision for multiprocessor communications. In these modes, 9 data bits are received. The 9th bit 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 were not being addressed leave their SM2s set and go on about their business, ignoring the coming data bytes.

SM2 has no effect in mode 0. In mode 1, it can be used to check the validity of the stop bit. In a mode 1 reception, if SM2 = 1, the receive interrupt will not be activated unless a valid stop bit is received.

SERIAL PORT CONTROL REGISTER

The serial port control and status register is the SFR S0CON. This register contains not only the mode selection bits, but also the 9th data bit for transmit and receive (TB8 and RB8), and the serial port interrupt bits (TI and RI). Register S0CON at address 98H controls data communication while register PCON at address 87H controls data rates.

S0CON
address 98H
bit addressable

MSB							LSB
SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI
7	6	5	4	3	2	1	0

Bit	Symbol	Function
7	FE	Framing error bit. This bit is set by the receiver when an invalid stop bit is detected. The FE bit is not cleared by valid frames but should be cleared by software. The SMOD0 bit (located at PCON.6) must be set to allow access to the FE bit
7	SM0	Serial port mode bit 0
6	SM1	Serial port mode bit 1

Bits SM0 and SM1 specify the serial port mode as shown in Table D.7

Table D.7 Serial port mode options

SM0	SM1	Mode	Description	Baud rate
0	0	0	Shift register	$f_{osc}/12$ (12-clock mode) $f_{osc}/6$ (6-clock mode)
0	1	1	8-bit UART	Variable
1	0	2	9-bit UART	$f_{osc}/32$ or $f_{osc}/64$ (12-clock mode) $f_{osc}/32$ or $f_{osc}/16$ (6-clock mode)
1	1	3	9-bit UART	Variable

SM2	Enables the multiprocessor feature in modes 2 and 3. In modes 2 or 3, if SM2 is set to 1 then RI will not be activated if the received 9th data bit RB8 is 0, indicating an address and the received byte is a given or broadcast address. 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
REN	Set by software to enable serial reception. Clear by software to disable reception
TB8	The 9th data bit that will be transmitted in modes 2 and 3. Set/clear by software
RB8	In modes 2 and 3, is the 9th data bit received In mode 1 if SM2 = 0, RB8 is the stop bit that was received In mode 0, RB8 is not used
TI	Transmit interrupt flag. Set by hardware at the end of the 8th bit time in mode 0, or at the start of the stop bit in other modes, in any serial transmission. Must be cleared by software
RI	Receive interrupt flag. Set by hardware at the end of the 8th bit in mode 0, or halfway through the stop bit time in the other modes, in any serial reception (except see SM2). Must be cleared by software.

BAUD RATES

The baud rate in mode 0 is fixed i.e. mode 0 baud rate = oscillator frequency/12 (12-clock mode) or /6 (6-clock mode). The baud rate in mode 2 depends

on the value of bit SMOD in SFR PCON. If SMOD = 0 (which is the value on reset), and the port pins in 12-clock mode, the baud rate is 1/64 the oscillator frequency. If SMOD = 1, the baud rate is 1/32 the oscillator frequency. In 6-clock mode, the baud rate is 1/32 or 1/16 the oscillator frequency, respectively.

$$\text{Mode 2 baud rate} = \text{baud rate} = \frac{2^{SMOD}}{n} \times \text{oscillator frequency}$$

where $n = 64$ in 12-clock mode, 32 in 6-clock mode.

The baud rates in modes 1 and 3 are determined by the timer 1 or timer 2 overflow rate.

Using timer 1 to generate baud rates. When timer 1 is used as the baud rate generator (T2CON.5 = 0, T2CON.4 = 0), the baud rates in modes 1 and 3 are determined by the timer 1 overflow rate and the value of SMOD as follows:

$$\text{baud rate} = \frac{2^{SMOD}}{n} \times (\text{timer 1 overflow rate})$$

where $n = 32$ in 12-clock mode, 16 in 6-clock mode.

The timer 1 interrupt should be disabled in this application. The timer itself can be configured for either 'timer' or 'counter' operation, and in any of its 3 running modes. In the most typical applications, it is configured for 'timer' operation, in the auto-reload mode (high nibble of TMOD = 0010B). In that case the baud rate is given by the formula:

$$\text{baud rate} = \frac{2^{SMOD}}{n} \times \frac{\text{oscillator frequency}}{12[1256 - (\text{TH1})]}$$

where $n = 32$ in 12-clock mode, 16 in 6-clock mode.

Very low baud rates with timer 1 can be achieved by leaving the timer 1 interrupt enabled, and configuring the timer to run as a 16-bit timer (high nibble of TMOD = 0001B), and using the timer 1 interrupt to do a 16-bit software reload. Table D.8 lists various commonly used baud rates and how they can be obtained from timer 1.

ENHANCED UART

The UART operates in all of the usual modes that are described above. In addition the UART can perform framing error detect, by looking for missing stop bits, and automatic address recognition. The UART also fully supports multiprocessor communication. When used for framing error detect the UART looks for missing stop bits in the communication. A missing bit will set the FE bit in the S0CON register. The FE bit shares the S0CON.7 bit with SM0 and the function of S0CON.7 is determined by PCON.6 (SMOD0). If SMOD0 is set then S0CON.7 functions as FE. S0CON.7 functions as SM0 when SMOD0

Table D.8 Timer 1 generated commonly used baud rates

Baud rate					Timer 1		
Mode	12-clock mode	6-clock mode	f_{osc}	SMOD	C/$\overline{T}$	Mode	Reload value
Mode 0 max	1.67 MHz	3.34 MHz	20 MHz	X	X	X	X
Mode 2 max	625 k	1250 k	20 MHz	1	X	X	X
Mode 1,3 max	104.2 k	208.4 k	20 MHz	1	0	2	FFH
	19.2 k	38.4 k	11.059 MHz	1	0	2	FDH
	9.6 k	19.2 k	11.059 MHz	0	0	2	FDH
	4.8 k	9.6 k	11.059 MHz	0	0	2	FAH
	2.4 k	4.8 k	11.059 MHz	0	0	2	F4H
	1.2 k	2.4 k	11.059 MHz	0	0	2	E8H
	137.5	275	11.059 MHz	0	0	2	1DH
	110	220	6 MHz	0	0	2	72H
	110	220	6 MHz	0	0	1	FEEBH

is cleared. When used as FE S0CON.7 can only be cleared by software. Refer to Figure D.13.

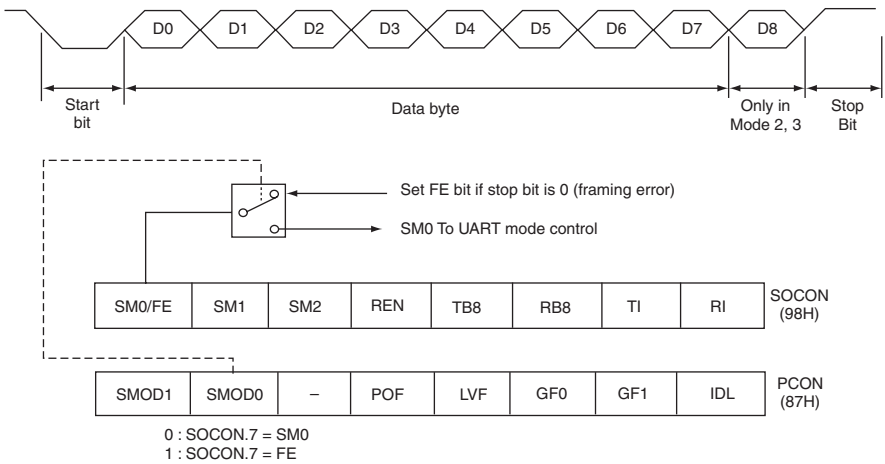


Figure D.13 UART framing error detection (courtesy Philips Semiconductors)

AUTOMATIC ADDRESS RECOGNITION

Automatic address recognition is a feature, which allows the UART to recognise 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 S0CON. In the 9-bit UART modes, modes 2 and 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 1 to indicate that the received information is an address and not data. Automatic address recognition is shown in Figure D.14.

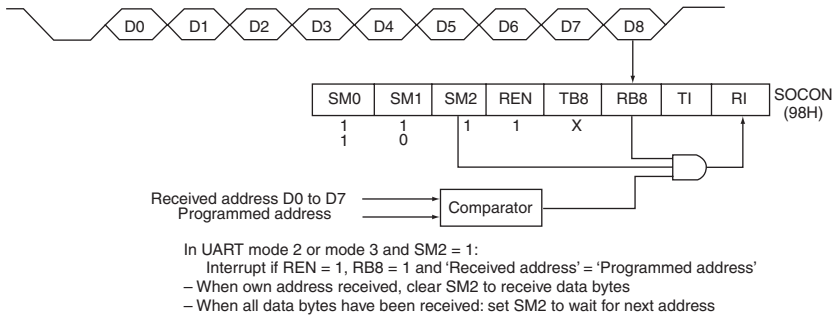


Figure D.14 UART multiprocessor communication, automatic address recognition (courtesy Philips Semiconductors)

The 8-bit mode is called mode 1. In this mode the RI flag will be set if SM2 is enabled and the information received has a valid stop bit following the 8 address bits and the information is either a given or broadcast address. Mode 0 is the shift register mode and SM2 is ignored.

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 SFRs 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 recognised while excluding others. The following examples will help to illustrate the point:

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 that 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 three 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 treated as don't-cares. In most cases, interpreting the don't-cares as ones, the broadcast address will be FF hexadecimal.

Upon reset SADDR (SFR address 0A9H) and SADEN (SFR address 0B9H) 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 80C51 type UART drivers that do not make use of this feature.

I^2C SERIAL COMMUNICATION

The P89C660/662/664/668 PC pins are alternate functions to port pins P1.6 and P1.7. Because of this, P1.6 and P1.7 on these ports do not have a pull-up structure as found on the 80C51. Therefore P1.6 and P1.7 have open drain outputs on the P89C660/662/664/668. The I^2C bus uses two wires (SDA and SCL) to transfer information between devices connected to the bus. The main features of the bus are:

- bidirectional data transfer between masters and slaves;
- multimaster bus (no central master);
- arbitration between simultaneously transmitting masters without corruption of serial data on the bus;
- serial clock synchronisation allows devices with different bit rates to communicate via one serial bus;

- serial clock synchronisation can be used as a handshake mechanism to suspend and resume serial transfer;
- the I^2C bus may be used for test and diagnostic purposes.

The output latches of P1.6 and P1.7 must be set to logic 1 in order to enable SIO1. The P89C66x on-chip I^2C logic provides a serial interface that meets the I^2C bus specification and supports all transfer modes (other than the low-speed mode) from and to the I^2C bus. The SIO1 logic handles bytes transfer autonomously. It also keeps track of serial transfers, and a status register (S1STA) reflects the status of SIO1 and the I^2C bus. The CPU interfaces to the I^2C logic via the following four SFRs: S1CON (SIO1 control register), S1STA (SIO1 status register), S1DAT (SIO1 data register) and S1ADR (SIO1 slave address register). The SIO1 logic interfaces to the external I^2C bus via two port 1 pins: P1.6/SCL (serial clock line) and P1.7/SDA (serial data line). A typical PC bus configuration is shown in Figure D.15.

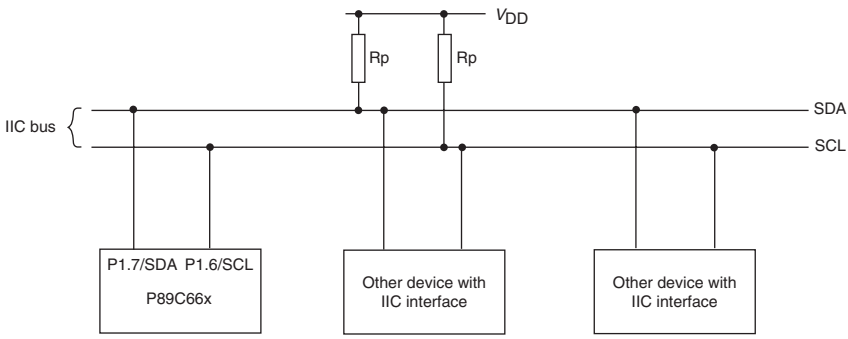


Figure D.15 Typical I^2C bus configuration (courtesy Philips Semiconductors)

Figure D.16 shows how a data transfer is accomplished on the bus. Depending on the state of the direction bit ($R/\overline{W}$), two types of data transfers are possible on the I^2C bus:

1. Data transfer from a master transmitter to a slave receiver. The first byte transmitted by the master is the slave address. Next follows a number of data bytes. The slave returns an acknowledge bit after each received byte.
2. Data transfer from a slave transmitter to a master receiver. The first byte (the slave address) is transmitted by the master. The slave then returns an acknowledge bit. Next follows the data bytes transmitted by the slave to the master. The master returns an acknowledge bit after all received bytes other than the last byte. At the end of the last received byte, a 'not acknowledge' is returned.

The master device generates all of the serial clock pulses and the START and STOP conditions. A transfer is ended with a STOP condition or with a repeated

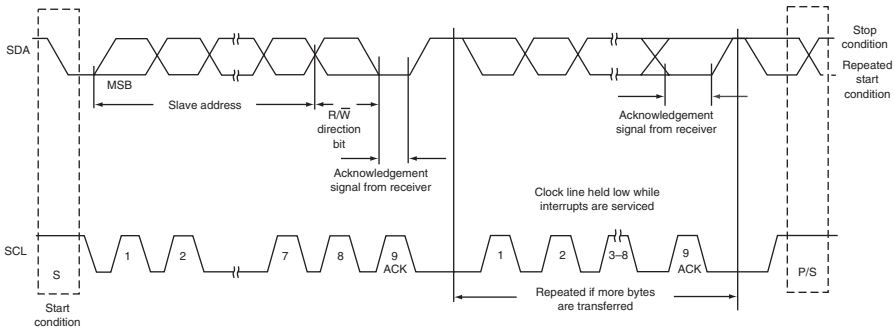


Figure D.16 Data transfer on the I^2C bus (courtesy Philips Semiconductors)

START condition. Since a repeated START condition is also the beginning of the next serial transfer, the PC bus will not be released.

MODES OF OPERATION

The on-chip SIO1 logic may operate in the following four modes:

1. Master transmitter mode. Pins P1.6 and P1.7 are outputs (serial data on P1.7 (SDA) and serial clock on P1.6 (SCL)). The first byte transmitted contains the address of the slave receiving device (7 bits) and the data direction bit, which in this case is logic 0 (data direction bit R/$\overline{W}$ determines data direction and is logic 1 for read and logic 0 for write). Serial data is transmitted 8 bits at a time and after each byte an acknowledge bit is received. START and STOP conditions are also output to indicate the beginning and end of a serial transfer.
2. Master receiver mode. Similar to 1 above except that P1.6 (SCL) and P1.7 (SDA) are inputs and the data direction bit is logic 1 (for read). Serial data is received through SDA while SCL outputs the serial clock. Serial data is received a byte at a time and after each byte an acknowledge bit is transmitted. START and STOP conditions are also output to indicate the beginning and end of a serial transfer.
3. Slave receiver mode. Serial data and serial clock are received through pins P1.7 (SDA) and P1.6 (SCL) respectively. After each byte is received an acknowledge bit is transmitted. START and STOP are recognised as the beginning and end respectively of the serial transfer. Recognition of the address is performed by hardware after reception of the slave address and direction bit.
4. Slave transmitter mode. The first byte is received and handled as in the slave receiver mode. However, in this mode, the direction bit will show that the direction of data transfer is reversed. As before serial data is through P1.7 (SDA) and serial clock through P1.6 (SCL). START and STOP conditions are also output to indicate the beginning and end of a serial transfer.

SIO1 can operate as a master or a slave. As a slave the SIO1 hardware looks for its own slave address and the general call address and if one of these addresses is detected, an interrupt is requested. If the microcontroller wishes to become the master, the hardware waits until the bus is free before the master mode is entered in order not to interrupt a possible slave action. If bus arbitration is lost in the master mode, SIO1 switches to the slave mode immediately and can detect its own slave address in the same serial transfer.

SIO1 IMPLEMENTATION AND OPERATION

Figure D.17 shows how the on-chip I^2C bus interface is implemented, and the following text describes the individual blocks.

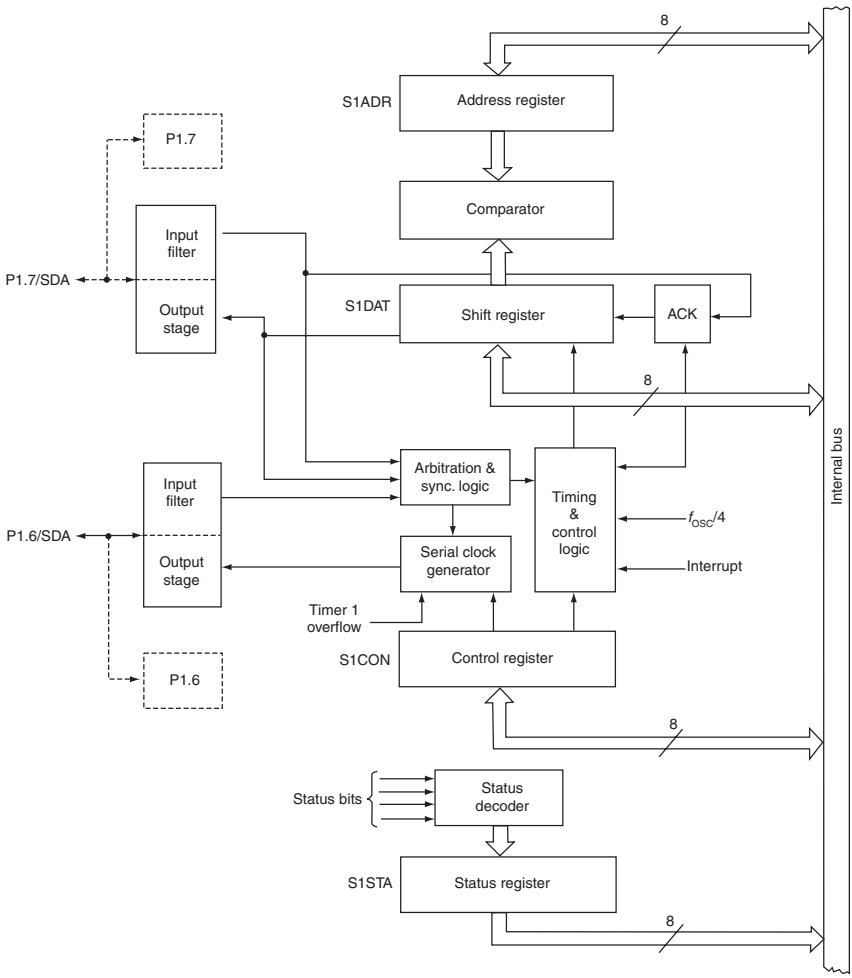


Figure D.17 I^2C bus serial interface block diagram (courtesy Philips Semiconductors)

INPUT FILTERS AND OUTPUT STAGES

The input filters have I^2C compatible input levels. If the input voltage is less than 1.5 V, the input logic level is interpreted as 0; if the input voltage is greater than 3.0 V, the input logic level is interpreted as 1. Input signals are synchronised with the internal clock ($f_{osc}/4$), and spikes shorter than three oscillator periods are filtered out.

The output stages consist of open drain transistors that can sink 3 mA at $V_{OUT} < 0-4$ V. These open drain outputs do not have damping diodes to V_{DD}; thus, if the device is connected to the I^2C bus and V_{DD} is switched off, the I^2C bus is not affected.

ADDRESS REGISTER, S1ADR

This 8-bit SFR may be loaded with the 7-bit slave address (7 most significant bits) to which SIO1 will respond when programmed as a slave transmitter or receiver. The LSB (GC) is used to enable general call address (00H) recognition.

COMPARATOR

The comparator compares the received 7-bit slave address with its own slave address (7 most significant bits in S1ADR). It also compares the first received 8-bit byte with the general call address (00H). If equality is found, the appropriate status bits are set and an interrupt is requested.

SHIFT REGISTER, S1DAT

This 8-bit SFR contains a byte of serial data to be transmitted or a byte that has just been received. Data in S1DAT is always shifted from right to left; the first bit to be transmitted is the MSB (bit 7) and, after a byte has been received, the first bit of received data is located at the MSB of S1DAT. While data is being shifted out, data on the bus is simultaneously being shifted in; S1DAT always contains the last byte present on the bus. Thus, in the event of lost arbitration, the transition from master transmitter to slave receiver is made with the correct data in S1DAT.

ARBITRATION AND SYNCHRONISATION LOGIC

In the master transmitter mode, the arbitration logic checks that every transmitted logic 1 actually appears as a logic 1 on the I^2C bus. If another device on the bus overrules a logic 1 and pulls the SDA line low, arbitration is lost, and SIO1 immediately changes from master transmitter to slave receiver. SIO1 will continue to output clock pulses (on SCL) until transmission of the current serial byte is complete. Arbitration may also be lost in the master receiver mode. Loss of arbitration in this mode can only occur

while SIO1 is returning a 'not acknowledge' (logic 1) to the bus. Arbitration is lost when another device on the bus pulls this signal low. Since this can occur only at the end of a serial byte, SIO1 generates no further clock pulses.

The synchronisation logic will synchronise the serial clock generator with the clock pulses on the SCL line from another device. If two or more master devices generate clock pulses, the 'mark' duration is determined by the device that generates the shortest 'marks', and the 'space' duration is determined by the device that generates the longest 'spaces'. A slave may stretch the space duration to slow down the bus master. The space duration may also be stretched for handshaking purposes. This can be done after each bit or after a complete byte transfer. SIO1 will stretch the SCL space duration after a byte has been transmitted or received and the acknowledge bit has been transferred. The serial interrupt flag (SI) is set, and the stretching continues until the serial interrupt flag is cleared.

SERIAL CLOCK GENERATOR

This programmable clock pulse generator provides the SCL clock pulses when SIO1 is in the master transmitter or master receiver mode. It is switched off when SIO1 is in a slave mode. The programmable output clock frequencies are: f_{osc} /120, f_{osc} /9600 (12-clock mode) or f_{osc} /60, f_{osc} /4800 (6-clock mode) and the timer 1 overflow rate divided by eight. The output clock pulses have a 50% duty cycle unless the clock generator is synchronised with other SCL clock sources as described above.

TIMING AND CONTROL

The timing and control logic generates the timing and control signals for serial byte handling. This logic block provides the shift pulses for S1DAT, enables the comparator, generates and detects start and stop conditions, receives and transmits acknowledge bits, controls the master and slave modes, contains interrupt request logic, and monitors the I^2C bus status.

CONTROL REGISTER, S1CON

This 7-bit SFR is used by the microcontroller to control the following SIO1 functions; start and restart of a serial transfer, termination of a serial transfer, bit rate, address recognition and acknowledgement.

STATUS DECODER AND STATUS REGISTER

The status decoder takes all of the internal status bits and compresses them into a 5-bit code. This code is unique for each I^2C bus status. The 5-bit code may be used to generate vector addresses for fast processing of the various service routines. Each service routine processes a particular bus status. There are

26 possible bus states if all four modes of SIO1 are used. The 5-bit status code is latched into the five most significant bits of the status register when the serial interrupt flag is set (by hardware) and remains stable until the interrupt flag is cleared by software. The three least significant bits of the status register are always zero. If the status code is used as a vector to service routines, then the routines are displaced by eight address locations. Eight bytes of code are sufficient for most of the service routines.

THE FOUR SIO1 SPECIAL FUNCTION REGISTERS

The microcontroller interfaces to SIO1 via four SFRs. These four SFRs (S1ADR, S1DAT, S1CON and S1STA) are described individually in the following sections. The address of each of the SFRs together with their reset values can be seen in Table D.2.

The address register, S1ADR

The CPU can read from and write to this 8-bit, directly addressable SFR. S1ADR is not affected by the SIO1 hardware. The contents of this register are irrelevant when SIO1 is in a master mode. In the slave modes, the seven most significant bits must be loaded with the microcontroller's own slave address, and, if the least significant bit is set, the general call address (00H) is recognised; otherwise it is ignored.

X	X	X	X	X	X	X	GC
7	6	5	4	3	2	1	0

< ---------- Own slave address ---------- >

The most significant bit corresponds to the first bit received from the I^2C bus after a start condition. A logic 1 in S1ADR corresponds to a high level on the I^2C bus, and a logic 0 corresponds to a low level on the bus.

The data register, S1DAT

SD7	SD6	SD5	SD4	SD3	SD2	SD1	SD0
7	6	5	4	3	2	1	0

< ---------- Shift direction ----------

S1DAT contains a byte of serial data to be transmitted or a byte that has just been received. The CPU can read from and write to this 8-bit, directly addressable SPR while it is not in the process of shifting a byte. This occurs when SIO1 is in a defined state and the serial interrupt flag is set. Data in

S1DAT remains stable as long as SI is set. Data in S1DAT is always shifted from right to left: the first bit to be transmitted is the MSB (bit 7), and, after a byte has been received, the first bit of received data is located at the MSB of S1DAT. While data is being shifted out, data on the bus is simultaneously being shifted in. S1DAT always contains the last data byte present on the bus. Thus, in the event of lost arbitration, the transition from master transmitter to slave receiver is made with the correct data in S1DAT.

SD7–SD0 are the 8 bits to be transmitted or just received. A logic 1 in S1DAT corresponds to a high level on the I^2C bus, and a logic 0 corresponds to a low level on the bus. Serial data shifts through S1DAT from right to left. Figure D.18 shows how data in S1DAT is serially transferred to and from the SDA line.

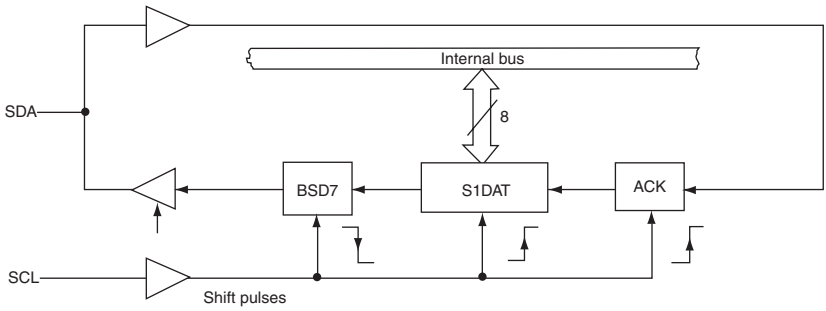


Figure D.18 Serial input/output configuration (courtesy Philips Semiconductors)

S1DAT and the ACK flag form a 9-bit shift register which shifts in or shifts out an 8-bit byte, followed by an acknowledge bit. The ACK flag is controlled by the SIO1 hardware and cannot be accessed by the CPU. Serial data is shifted through the ACK flag into S1DAT on the rising edges of serial clock pulses on the SCL line. When a byte has been shifted into S1DAT the serial data is available in S1DAT and the acknowledge bit is returned by the control logic during the ninth clock pulse. Serial data is shifted out from S1DAT via a buffer (BSD7) on the falling edges of clock pulses on the SCL line. When the CPU writes to S1DAT BSD7 is loaded with the content of S1DAT.7, which is the first bit to be transmitted to the SDA line (see Figure D.19). After nine serial clock pulses, the 8 bits in S1DAT will have been transmitted to the SDA line, and the acknowledge bit will be present in ACK. Note that the eight transmitted bits are shifted back into S1DAT.

The control register, S1CON

The CPU can read from and write to this 8-bit, directly addressable SFR. Two bits are affected by the SIO1 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 I^2C bus. The STO bit is also cleared when ENS1=0.

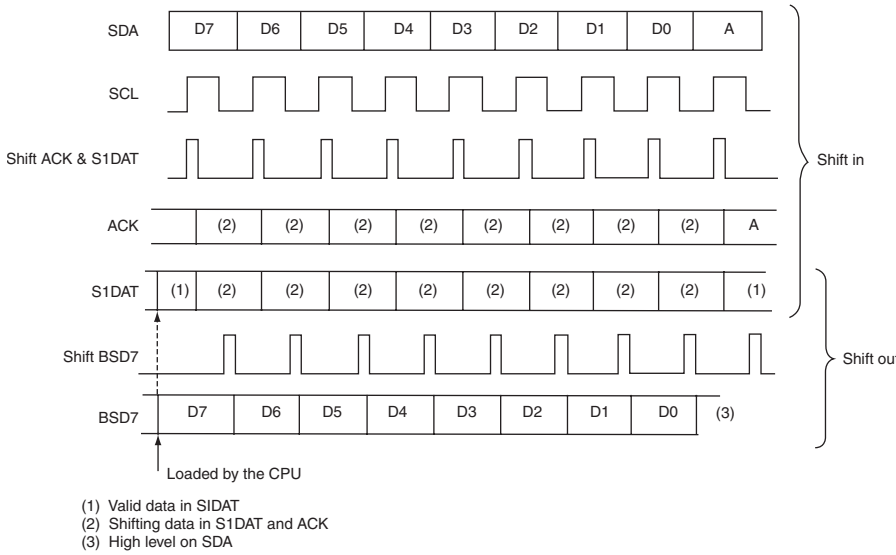


Figure D.19 Shift-in and shift-out timing (courtesy Philips Semiconductors)

CR2	ENS1	STA	STO	SI	AA	CR1	CR0
7	6	5	4	3	2	1	0

ENS1, the SIO1 Enable Bit

*ENS*1=0. When ENS1 is 0, the SDA and SCL outputs are in a high impedance state. SDA and SCL input signals are ignored, SIO1 is in the 'not addressed' slave state, and the STO bit in S1CON is forced to 0. No other bits are affected. P1.6 and PI .7 may be used as open drain I/O ports.

*ENS*1=1. When ENS1 is 1, SIO1 is enabled. The P1.6 and P1.7 port latches must be set to logic 1. ENS1 should not be used to temporarily release SIO1 from the I^2C bus since, when ENS1 is reset, the I^2C bus status is lost. The AA flag should be used instead.

In the description that follows, it is assumed that ENS1= 1.

The 'START' Flag STA

STA = 1. When the STA bit is set to enter a master mode, the SIO1 hardware checks the status of the I^2C bus and generates a START condition if the bus is free. If the bus is not free, then SIO1 waits for a STOP condition (which will free the bus) and generates a START condition after a delay of half a clock period of the internal serial clock generator. If STA is set while SIO1 is already in a master mode and one or more bytes are transmitted or received, SIO1 transmits a repeated START condition. STA may be set at any time. STA may also be set when SIO1 is an addressed slave.

STA=0. When the STA bit is reset, no START condition or repeated START condition will be generated.

The STOP Flag STO

STO=1. When the STO bit is set while SIO1 is in a master mode, a STOP condition is transmitted to the I^2C bus. When the STOP condition is detected on the bus, the SIO1 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 I^2C bus. However, the SIO1 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, the STOP condition is transmitted to the I^2C bus if SIO1 is in a master mode (in a Slave mode SIO1 generates an internal STOP condition which is not transmitted). SIO1 then transmits a START condition.

STO=0. When the STO bit is reset, no STOP condition will be generated.

The serial interrupt flag, SI

SI=1. When the SI flag is set, then, if the EA and ES1 (interrupt enable register) bits are also set, a serial interrupt is requested. SI is set by hardware when one of 25 of the 26 possible SIO1 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.

The assert acknowledge flag, AA

$AA = 1$. 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:

1. the 'own slave address' has been received;
2. the general call address has been received while the general call bit (GC) in S1ADR is set;
3. a data byte has been received while SIO1 is in the master receiver mode;
4. a data byte has been received while SIO1 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 SCL when:

1. a data byte has been received while SIO1 is in the master receiver mode;
2. a data byte has been received while SIO1 is in the addressed slave receiver mode.

When SIO1 is in the addressed slave transmitter mode, state C8H will be entered after the last serial data is transmitted.

When Sl is cleared, SIO1 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 SIO1 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, SIO1 can be temporarily released from the I^2C bus while the bus status is monitored. While SIO1 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 recognised at the end of the byte transmission.

The clock rate bits CR0, CR1 and CR2

These three bits determine the serial clock frequency when SIO1 is in a master mode. The various serial rates are shown in Table D.9.

A 12.5 kHz bit rate may be used by devices that interface to the I^2C bus via standard I/O port lines which are software driven and slow. 100 kHz is usually the maximum bit rate and can be derived from a 16 MHz, 12 MHz or a 6 MHz oscillator. A variable bit rate (0.5 kHz to 62.5 kHz) may also be used if timer 1 is not required for any other purpose while SIO1 is in a master mode. The frequencies shown in Table D.9 are unimportant when SIO1 is in a slave mode. In the slave modes, SIO1 will automatically synchronise with any clock frequency up to 100 kHz.

The status register, S1STA

S1STA is an 8-bit read-only SFR. The three least significant bits are always zero. The five most significant bits contain the status code. There are 26 possible status codes. When S1STA contains F8H, no relevant state information is available and no serial interrupt is requested. All other S1STA values correspond to defined SIO1 states. When each of these states is entered, a serial interrupt is requested (SI = 1). A valid status code is present in S1STA one machine cycle after SI is set by hardware and is still present one machine cycle after SI has been reset by software.

D.6 Interrupt priority structure

The P89C660/662/664/668 has an 8-source four-level interrupt structure (see Table D.10). There are four SFRs associated with the four-level interrupt. They are the IEN0, IP, IPH and IEN1 registers. Details of these registers are shown in Tables D.11, D.12, D.13 and D.14 respectively. The IPH (interrupt priority high) register makes the four-level interrupt structure possible. The SFR addresses and their reset values are shown in Table D.2. The function of the IPH SFR, when combined with the IP SFR, determines the priority of each interrupt. The priority of each interrupt is determined as shown in Table D.15.

Table D.9 Serial clock rates

6-clock mode			Bit frequency (kHz) at f_{osc}					
CR2	CR1	CR0	3 MHz	6 MHz	8 MHz	12 MHz[2]	15 MHz[2]	f_{osc} divided by
0	0	0	23	47	62.5	94	117[1]	128
0	0	1	27	54	71	107[1]	134[1]	112
0	1	0	31	63	83.3	125[1]	156[1]	96
0	1	1	37	75	100	150[1]	188[1]	80
1	0	0	6.25	12.5	17	25	31	480
1	0	1	50	100	133[1]	200[1]	250[1]	60
1	1	0	100	200	267[1]	400[1]	500[1]	30
1	1	1	0.24 < 62.5 0 < 255	0.49 < 62.5 0 < 254	0.65 < 55.6 0 < 253	0.98 < 50.0 0 < 251	1.22 < 52.1 0 < 250	48 × (256− (reload value timer1)) Reload value timer 1 in mode 2

12-clock mode			Bit frequency (kHz) at f_{osc}					
CR2	CR1	CR0	6 MHz	12 MHz	16 MHz	24 MHz[2]	30 MHz[2]	f_{osc} divided by
0	0	0	23	47	62.5	94	117[1]	256
0	0	1	27	54	71	107[1]	134[1]	224
0	1	0	31	63	83.3	125[1]	156[1]	192
0	1	1	37	75	100	150[1]	188[1]	160
1	0	0	6.25	12.5	17	25	31	960
1	0	1	50	100	133[1]	200[1]	250[1]	120
1	1	0	100	200	267[1]	400[1]	500[1]	60
1	1	1	0.24 < 62.5 0 < 255	0.49 < 62.5 0 < 254	0.65 < 55.6 0 < 253	0.98 < 50.0 0 < 251	1.22 < 52.1 0 < 250	96 × (256− (reload value timer 1)) Reload value timer 1 in mode 2

1. These frequencies exceed the upper limit of 100 kHz of the I^2C-bus specification and cannot be used in an I^2C-bus application.
2. At f_{osc} = 12 MHz/15 MHz the maximum I^2C bus rate of 100 kHz cannot be realised due to the fixed divider rates.
3. At f_{osc} = 24 MHz/30 MHz the maximum I^2C bus rate of 100 kHz cannot be realised due to the fixed divider rates.

Table D.10 Interrupt table

Source	Polling priority	Request bits	Hardware clear?	Vector address
X0	1	IE0	N(L)[1] Y(T)[2]	03H
S101(I^2C)	2	–	N	2BH
T0	3	TP0	Y	0BH
X1	4	IE1	N(L) Y(T)	13H
T1	5	TF1	Y	1BH
SP	6	Rl.,TI	N	23H
T2	7	TF2, EXF2	N	3BH
PCA	8	CF, CCFn n = 0−4	N	33H

Notes: 1. L = Level activated; 2. T = Transition activated.

Table D.11 Details of the IEN0 register

MSB							LSB
EA	EC	ES1	ES0	ET1	EX1	ET0	EX0
7	6	5	4	3	2	1	0

Enable Bit = 1 enables the interrupt
Enable Bit = 0 disables it.

Bit	Symbol	Function
7	EA	Global disable bit. If EA = 0, all interrupts are disabled. If EA = 1, each interrupt can be individually enabled or disabled by setting or clearing its enable bit
6	EC	PCA interrupt enable bit
5	ES1	I^2C interrupt enable bit
4	ESO	Serial port interrupt enable bit
3	ET1	Timer 1 interrupt enable bit
2	EX1	External interrupt 1 enable bit
1	ET0	Timer 0 Interrupt enable bit
0	EX0	External interrupt 0 enable bit

Table D.12 Details of the IP register

MSB							LSB
PT2	PPC	PS1	PS0	PT1	PX1	PT0	PX0
7	6	5	4	3	2	1	0

Priority Bit = 1 assigns high priority
Priority Bit = 0 assigns low priority

Bit	Symbol	Function
7	PT2	Timer 2 interrupt priority bit
6	PPC	PCA interrupt priority bit
5	PS1	Serial 1/01 (I^2C) interrupt priority bit
4	PSO	Serial port interrupt priority bit
3	PT1	Timer 1 interrupt priority bit
2	PX1	External interrupt 1 priority bit
1	PTO	Timer 0 interrupt priority bit

Table D.13 Details of the IPH register

MSB							LSB
PT2H	PPCH	PS1H	PS0H	PT1H	PX1H	PT0H	PX0H
7	6	5	4	3	2	1	0

Priority bit = 1 assigns higher priority
Priority bit = 0 assigns lower priority

Bit	Symbol	Function
7	PT2H	Timer 2 interrupt priority bit high
6	PPCH	PCA interrupt priority bit
5	PS1H	Serial I/O (I^2C) interrupt priority bit high
4	PS0H	Serial port interrupt priority bit high
3	PT1H	Timer 1 interrupt priority bit high
2	PX1H	External interrupt 1 priority bit high
1	PT0H	Timer 0 interrupt priority bit high
0	PX0H	External interrupt 0 priority bit high

Table D.14 Details of the IEN1 register

MSB							LSB
–	–	–	–	–	–	–	ET2
7	6	5	4	3	2	1	0

Enable Bit = 1 enables the interrupt
Enable Bit = 0 disables the Interrupt

Bit	Symbol	Function
7	–	
6	–	
5	–	
4	–	
3	–	
2	–	
1	–	
0	ET2	Timer 2 interrupt enable bit

The priority scheme for servicing the interrupts is the same as that for the 80C51, except that there are four interrupt levels rather than two (as on the 80C51). An interrupt will be serviced as long as an interrupt of equal or higher

Table D.15 89C66x interrupt priority levels

Priority bits		
IPH.x	IP.x	Interrupt priority level
0	0	Level 0 (lowest priority)
0	1	Level 1
1	0	Level 2
1	1	Level 3 (highest priority)

priority is not already being serviced. If an interrupt of equal or higher level priority is being serviced, the new interrupt will wait until it is finished before being serviced. If a lower priority level interrupt is being serviced, it will be stopped and the new interrupt serviced. When the new interrupt is finished, the lower priority level interrupt that was stopped will be completed.

Appendix E

P89LPC932 Microcontroller

Details of this device are reproduced with kind permission of Philips Semiconductors. Data regarding the device may be found on the Philips website at www.semiconductors.philips.com. This device is a development of the 80C51 device and offers high-integration with low cost; additionally operation is at an improved speed compared to 80C51 devices operating at the same frequency. Features include a low pincount and a 2.4 V to 3.6 V operating range for V_{DD}. The basic block diagram is shown in Figure E.1.

The LPC932 uses an enhanced 80C51 CPU which runs at 6 times the speed of standard 80C51 devices. A machine code consists of two CPU clock cycles and most instructions execute in one or two machine cycles.

Other device features include:

- Two 16-bit counter/timers with each timer able to be configured to toggle a port output or to become a PWM (pulse width modulation) output.
- CCU (capture/compare unit), which provides PWM, input capture and output compare functions.
- Two analogue comparators with selectable inputs and reference sources.
- UART with fractional baud rate generator, framing error detection, versatile interrupt capabilities, etc.
- I^2C and SPI communication ports.
- Four interrupt priority levels.
- Watchdog timer with separate on-chip oscillator, requiring no external components. The watchdog timeout is selectable from 8 values.
- LED drive capability of 20 mA on all port pins, subject to a maximum limit for the entire chip.
- A minimum of 23 I/O pins. If using on-chip oscillator and reset options this may be increased to 26 I/O pins.
- Serial flash programming allows simple in-circuit production coding.

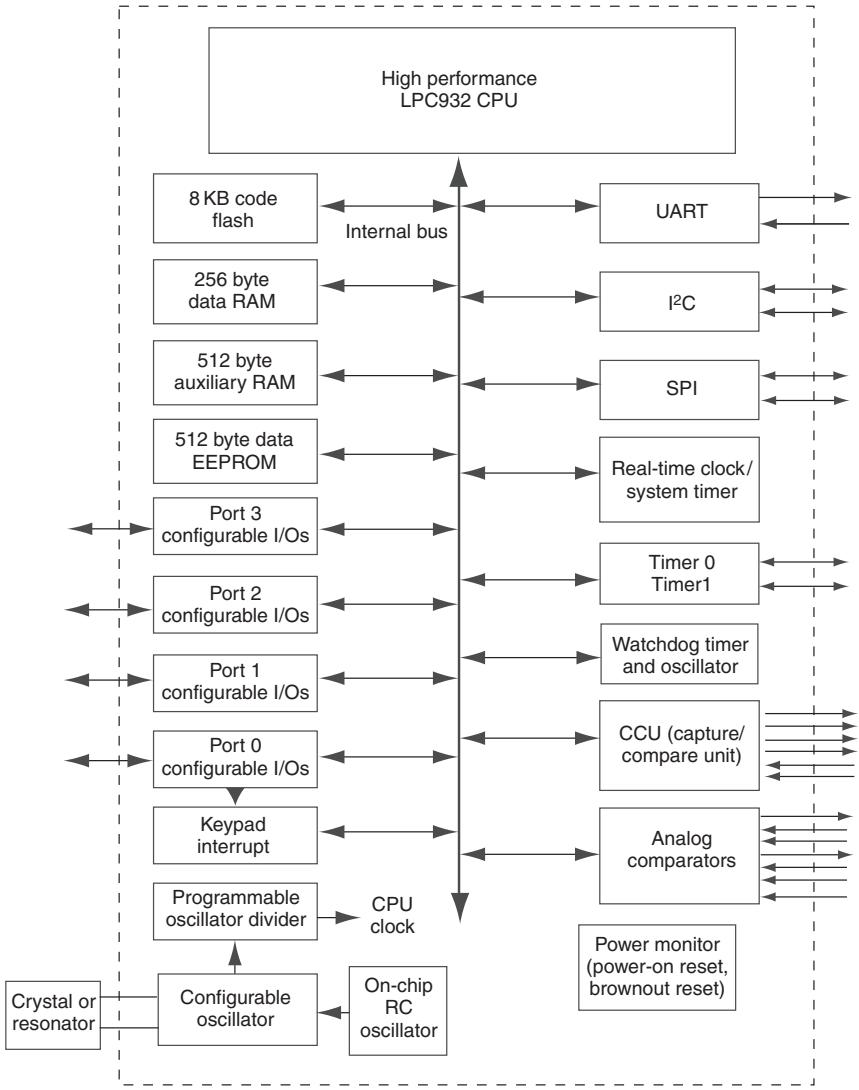


Figure E.1 LPC932 block diagram (courtesy Philips Semiconductors)

Packages include a 28-pin TSSOP package and a 28-pin PLCC package. The latter package is illustrated in Figure E.2.

E.1 Device pin functions

SUPPLY VOLTAGE (V_{DD} AND V_{SS})

The device operates from a single supply connected to pin 21 (V_{DD}) while pin 7 (V_{SS}) is grounded.

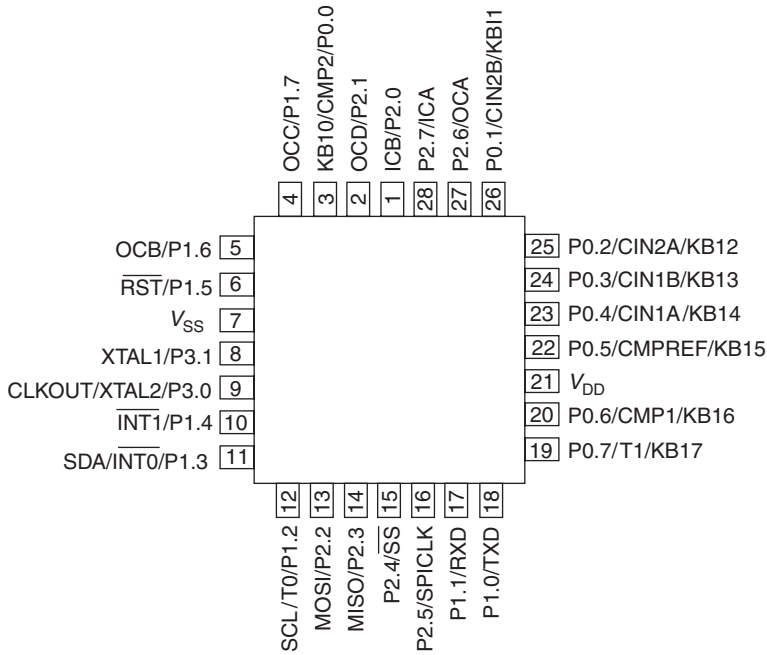


Figure E.2 PLCC 28-pin package (courtesy Philips Semiconductors)

INPUT/OUTPUT (I/O) PORTS

Details are as follows:

Port 0. This is an 8-bit I/O port with user configurable output type. During reset port 0 latches are configured in the input only mode with the internal pull-up disabled. The operation of the port pins as inputs or outputs depends on the port configuration selected. The keypad interrupt feature operates with port 0 pins. Special functions of port 0 pins are as follows:

Pin number	Input/output	Function
3	O	Comparator 2 output (CMP2)
	I	Keyboard input 0 (KBI0)
26	I	Comparator 2 positive input B (CIN2B)
	I	Keyboard input 1 (KBI1)
25	I	Comparator 2 positive input A (CIN2A)
	I	Keyboard input 2 (KBI2)
24	I	Comparator 1 positive input B (CIN1B)
	I	Keyboard input 3 (KBI3)
23	I	Comparator 2 positive input A (CIN2A)
	I	Keyboard input 4 (KBI4)

22	I	Comparator reference (negative) input
	I	Keyboard input 5 (KBI5)
20	O	Comparator 1 output (CMP1)
	I	Keyboard input 6 (KBI6)
19	I/O	Timer/counter 1 external count input or overflow output
	I	Keyboard input 7 (KBI7)

Pin numbers 3, 26, 25, 24, 23, 22, 20 and 19 also function for port 0 as bit 0, bit 1, bit 2, bit 3, bit 4, bit 5, bit 6 and bit 7 respectively. In this case each pin is an I/O pin.

Port 1. This is an 8-bit I/O port with user configurable output type (except for three pins, noted below). During reset port 1 latches are configured in the input only mode with the internal pull-up disabled. The operation of the port pins as inputs or outputs depends on the port configuration selected. P1.2 and P1.3 are open-drain when used as outputs while P1.5 is input only. Special functions of port 1 pins are as follows:

Pin number	Input/output	Function
18	O	Transmitter output for the serial port (TxD)
17	I	Receiver input for the serial port (RxD)
12	I × O	Timer/counter 0 external count input or overflow output (open-drain when used as outputs)
	I × O	I^2C serial clock input/output (SCL)
11	I	External interrupt 0 input ($\overline{\text{INT0}}$)
10	I	External interrupt 1 input ($\overline{\text{INT1}}$)
6	I	External reset during power-on or if selected via UCFG1. When functioning as a reset input a low on this pin resets the microcontroller, causing I/O ports and peripherals to take on their default states, and the processor begins execution at address 0. Also used during a power-on sequence to force ISP mode.
5	O	Output compare B (OCB)
4	O	Output compare C (OCC)

Pin numbers 18, 17, 12, 5, and 4 also function for port 1 as bit 0, bit 1, bit 2, bit 6 and bit 7 respectively. In this case each pin is an I/O pin. Pin numbers 11, 10 and 6 also function for port 1 as input pins for bit 3, bit 4 and bit 5 respectively.

Port 2. This is an 8-bit I/O port with user configurable output type. During reset port 2 latches are configured in the input only mode with the internal pull-up disabled. The operation of the port pins as inputs or outputs depends on the port configuration selected. Special functions of port 2 pins are as follows:

Pin number	Input/output	Function
1	I	Input capture B (ICB)
2	O	Output compare D (OCD)
13	I/O	SPI master out slave in. When configured as master, this pin is output; when configured as slave, this pin is input (MOSI)
14	I/O	SPI master in slave out. When configured as master, this pin is input; when configured as slave, this pin is output (MISO)
15	I	SPI slave select ($\overline{SS}$)
16	I/O	SPI clock. When configured as master, this pin is output; when configured as slave, this pin is input (SPICLK)
27	O	Output compare A (OCA)
28	I	Input capture A (ICA)

Pin numbers 1, 2, 13, 14, 15, 16, 27 and 28 also function for port 2 as bit 0, bit 1, bit 2, bit 3, bit 4, bit 5, bit 6 and bit 7 respectively. In this case each pin is an I/O pin.

Port 3. This is a 2-bit I/O port with user configurable output type. During reset port 3 latches are configured in the input only mode with the internal pull-up disabled. The operation of the port pins as inputs or outputs depends on the port configuration selected. Special functions of port 3 pins are as follows:

Pin number	Input/output	Function
9	O	Output from the oscillator amplifier (when a crystal oscillator option is selected via the FLASH configuration) (XTAL2)
	O	CPU clock divided by 2 when enabled via SFR bit (ENCLK – TRIM.6). It can be used if the CPU clock is the internal RC oscillator, watchdog oscillator or external clock input, except when XTAL1/XTAL2 is used to generate clock source for the real-time clock/system timer (CLKOUT)
8	I	input to the oscillator circuit and internal clock generator circuits (when selected via the FLASH configuration). It can be a port pin if internal RC oscillator or watchdog oscillator is used as the CPU clock source, AND if XTAL1/XTAL2 are not used to generate the clock for the Real-Time clock/system timer (XTAL1)

Pin numbers 9 and 8 also function for port 3 as bit 0 and bit 1 respectively. In this case each pin is an I/O pin. The device has SFRs as shown in Table E.1.

Table E.1 Special function registers (courtesy Philips Semiconductors)

Name	Description	SFR address	Bit functions and addresses MSB							LSB	Reset value Hex	Binary
			E7	E6	E5	E4	E3	E2	E1	E0		
ACC*	Accumulator	E0H									00H	00000000
AUXR1#	Auxiliary function register	A2H	CLKLP	EBRR	ENT1	ENT0	SRST	0	–	DPS	00H[1]	000000x0
			F7	F6	F5	F4	F3	F2	F1	F0		
B*	B register	F0H									00H	00000000
BRGR0#§	Baud rate generator rate low	BEH									00H	00000000
BRGR1#§	Baud rate generator rate high	BFH									00H	00000000
BRGCON#	Baud rate generator control	BDH	–	–	–	–	–	–	SBRGS	BRGEN	00H%	xxxxxx00
CCCRA#	Capture compare A control register	EAH	ICECA2	ICECA1	ICECA0	ICESA	ICNFA	FCOA	OCMA1	OCMA0	00H	00000000
CCCRB#	Capture compare B control register	EBH	ICECB2	ICECB1	ICECB0	ICESB	ICNFB	FCOB	OCMB1	OCMB0	00H	00000000
CCCRC#	Capture compare C control register	ECH	–	–	–	–	–	FCOC	OCMC1	OCMC0	00H	xxxxx000
CCCRD#	Capture compare D control register	EDH	–	–	–	–	–	FCOD	OCMD1	OCMD0	00H	xxxxx000
CMP1#	Comparator 1 control register	ACH	–	–	CE1	CP1	CN1	OE1	CO1	CMF1	00H[1]	xx000000
CMP2#	Comparator 2 control register	ADH	–	–	CE2	CP2	CN2	OE2	CO2	CMF2	00H[1]	xx000000

DEECON#	Data EEPROM control register	F1H	EEIF	HVERR	ECTL1	ECTL0	–	–	–	EADR8	0EH	00001110
DEEDAT#	Data EEPROM data register	F2H									00H	00000000
DEEADR#	Data EEPROM address register	F3H									00H	00000000
DIVM#	CPU clock divide-by-M control	95H									00H	00000000
DPTR	Data pointer (2 bytes)											
DPH	Data pointer high	83H									00H	00000000
DPL	Data pointer low	82H									00H	00000000
12ADR#	I^2C slave address register	DBH	12ADR.6	12ADR.5	12ADR.4	12ADR.3	12ADR.2	12ADR.1	12ADR.0	GC	00H	00000000
			DF	DE	DD	DC	DB	DA	D9	D8		
12CON*#	I^2C control register	D8H	–	12EN	STA	STO	SI	AA	–	CRSEL	00H	x00000x0
12DAT#	I^2C data register	DAH										
12SCLH#	Serial clock generator/ SCL duty cycle register high	DDH									00H	00000000
12SCLL#	Serial clock generator/ SCL duty cycle register low	DCH									00H	00000000
12STAT#	I^2C status register	D9H	STA.4	STA.3	STA.2	STA.1	STA.0	0	0	0	F8H	11111000
ICRAH#	Input capture A register high	ABH									00H	
ICRAL#	Input capture A register low	AAH									00H	00000000
ICRBH#	Input capture B register high	AFH									00H	00000000

Table E.1 Continued

Name	Description	SFR address	Bit functions and addresses								Reset value	
			MSB							LSB	Hex	Binary
ICRBL#	Input capture B register low	AEH									00H	00000000
			AF	AE	AD	AC	AB	AA	A9	A8		
IEN0*	Interrupt enable 0	A8H	EA	EWDRT	EBO	ES/ESR	ET1	EX1	ET0	EX0	00H	00000000
			EF	EE	ED	EC	EB	EA	E9	E8		
IEN1*#	Interrupt enable 1	E8H	EIEE	EST	–	ECCU	ESPI	EC	EKBI	EI2C	00H[1]	00x00000
			BF	BE	BD	BC	BB	BA	B9	B8		
IP0*	Interrupt priority 0	B8H	–	PWDRT	PBO	PS/PSR	PT1	PX1	PT0	PX0	00H[1]	x0000000
IP0H#	Interrupt priority 0 high	B7H	–	PWDRTH	PBOH	PSH/PSRH	PT1H	PX1H	PT0H	PX0H	00H[1]	x0000000
			FF	FE	FD	FC	FB	FA	F9	F8		
IP1*#	Interrupt priority 1	F8H	PIEE	PST	–	PCCU	PSPI	PC	PKBI	PI2C	00H[1]	00x00000
IP1H#	Interrupt priority 1 high	F7H	PIEEH	PSTH	–	PCCUH	PSPIH	PCH	PKBIH	PI2CH	00H[1]	00x00000
KBCON#	Keypad control register	94H	–	–	–	–	–	–	PATN_SEL	KBIF	00H[1]	xxxxxx00
KBMASK#	Keypad interrupt mask register	86H									00H	00000000
KBPATN#	Keypad pattern register	93H									FFH	11111111
OCRAH#	Output compare A register high	EFH									00H	00000000
OCRAL#	Output compare A register low	EEH									00H	00000000
OCRBH#	Output compare B register high	FBH									00H	00000000
OCRBL#	Output compare B register low	FAH									00H	00000000

OCRCH#	Output compare C register high	FDH								00H	00000000
OCRCL#	Output compare C register low	FCH								00H	00000000
OCRDH#	Output compare D register high	FFH								00H	00000000
OCRDL#	Output compare D register low	FEH								00H	00000000
			87	86	85	84	83	82	81	80	
P0*	Port 0	80H	T1/KB7	CMP1/ KB6	CMPREF/ KB5	CIN1A/ KB4	CIN1B/ KB3	CIN2A/ KB2	CIN2B/ KB1	CMP2/ KB0	Note 1
			97	96	95	94	93	92	91	90	
P1*	Port 1	90H	OCC	OCB	$\overline{\text{RST}}$	$\overline{\text{INT1}}$	$\overline{\text{INT0}}$/ $\overline{\text{SDA}}$	T0/SCL	RxD	TxD	Note 1
			A7	A6	A5	A4	A3	A2	A1	A0	
P2*	Port 2	A0H	ICA	OCA	SPICLK	$\overline{\text{SS}}$	MISO	MOSI	OCD	ICB	Note 1
			B7	B6	B5	B4	B3	B2	B1	B0	
P3*	Port 3	B0H	_	_	_	_	_	_	XTAL1	XTAL2	Note 1
POM1#	Port 0 output mode 1	84H	(POM1.7)	(POM1.6)	(POM1.5)	(POM1.4)	(POM1.3)	(POM1.2)	(POM1.1)	(POM1.0)	FFH 11111111
POM2#	Port 0 output mode 2	85H	(POM2.7)	(POM2.6)	(POM2.5)	(POM2.4)	(POM2.3)	(POM2.2)	(POM2.1)	(POM2.0)	00H 00000000
P1M1#	Port 1 output mode 1	91H	(P1M1.7)	(P1M1.6)	–	(P1M1.4)	(P1M1.3)	(P1M1.2)	(P1M1.1)	(P1M1.0)	D3H[1] 11x1xx11
P1M2#	Port 1 output mode 2	92H	(P1M2.7)	(P1M2.6)	–	(P1M2.4)	(P1M2.3)	(P1M2.2)	(P1M2.1)	(P1M2.0)	00H[1] 00x0xx00
P2M1#	Port 2 output mode 1	A4H	(P2M1.7)	(P2M1.6)	(P2M1.5)	(P2M1.4)	(P2M1.3)	(P2M1.2)	(P2M1.1)	(P2M1.0)	FFH 11111111
P2M2#	Port 2 output mode 2	A5H	(P2M2.7)	(P2M2.6)	(P2M2.5)	(P2M2.4)	(P2M2.3)	(P2M2.2)	(P2M2.1)	(P2M2.0)	00H 00000000
P3M1#	Port 3 output mode 1	B1H	_	_	_	_	_	_	(P3M1.1)	(P3M1.0)	03H[1] xxxxxx11
P3M2#	Port 3 output mode 2	B2H	_	_	_	_	_	_	(P3M2.1)	(P3M2.0)	00H[1] xxxxxx00
PCON#	Power control register	87H	SMOD1	SMOD0	BOPD	BOI	GF1	GF0	PMOD1	PMOD0	00H 00000000
PCONA#	Power control register A	B5H	RTCPD	DEEPD	VCPD		12PD	SPPD	SPD	CCUPD	00H[1] 00000000

Table E.1 Continued

Name	Description	SFR address	Bit functions and addresses								Reset value	
			MSB							LSB	Hex	Binary
			D7	D6	D5	D4	D3	D2	D1	D0		
PSW*	Program status word	D0H	CY	AC	F0	RS1	RS0	OV	F1	P	00H	00000000
PT0AD#	Port 0 digital input disable	F6H	–	–	PT0AD.5	PT0AD.4	PT0AD.3	PT0AD.2	PT0AD.1	–	00H	xx00000x
RSTSRC#	Reset source register	DFH	–	–	BOF	POF	R_BK	R_WD	R_SF	R_EX	Note 2	
RTCCON#	Real-time clock control	D1H	RTCF	RTCS1	RTCS0	–	–	–	ERTC	RTCEN	60H[1,5]	011xxx00
RTCH#	Real-time clock register high	D2H									00H[5]	00000000
RTCL#	Real-time clock register low	D3H									00H[5]	00000000
SADDR#	Serial port address register	A9H									00H	00000000
SADEN#	Serial port address enable	B9H									00H	00000000
SBUF#	Serial port data buffer register	99H									xxH	xxxxxxxx
			9F	9E	9D	9C	9B	9A	99	98		
SCON*	Serial port control	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	00H	00000000
SSTAT#	Serial port extended status register	BAH	DBMOD	INTLO	CIDIS	DBISEL	FE	BR	OE	STINT	00H	00000000
SP	Stack pointer	81H									07H	00000111
SPCTL#	SPI control register	E2H	SSIG	SPEN	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	04H	00000100
SPSTAT#	SPI status register	E1H	SPIF	WCOL	–	–	–	–	–	–	00H	00xxxxx
SPDAT#	SPI data register	E3H									00H	00000000

TAMOD#	Timer 0 and 1 auxiliary mode	8FH	–	–	–	T1M2	–	–	–	T0M2	00H	xxx0xxx0
			8F	8E	8D	8C	8B	8A	89	88		
TCON*	Timer 0 and 1 control	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00H	00000000
TCR20*#	CCU control register 0	C8H	PLEEN	HLTRN	HLTEN	ALTCD	ALTAB	TDIR2	TMOD21	TMOD20	00H	00000000
TCR21#	CCU control register 1	F9H	TCOU2	–	–	–	PLLDV.3	PLLDV.2	PLLDV.1	PLLDV.0	00H	0xxx0000
TH0	Timer 0 high	8CH									00H	00000000
TH1	Timer 1 high	8DH									00H	00000000
TH2#	CCU timer high	CDH									00H	00000000
TICR2#	CCU interrupt control register	C9H	TOIE2	TOCIE2D	TOCIE2C	TOCIE2B	TOCIE2A	–	TICIE2B	TICIE2A	00H	00000x00
TIFR2#	CCU interrupt flag register	E9H	TOIF2	TOCF2D	TOCF2C	TOCF2B	TOCF2A	–	TICF2B	TICF2A	00H	00000x00
TISE2#	CCU interrupt status encode register	DEH	–	–	–	–	–	ENCINT.2	ENCINT.1	ENCINT.0	00H	xxxxx000
TL0	Timer 0 low	8AH									00H	00000000
TL1	Timer 1 low	8BH									00H	00000000
TL2#	CCU timer low	CCH									00H	00000000
TMOD	Timer 0 and 1 mode	89H	T1GATE	T1C/T	T1M1	T1M0	T0GATE	T0C/T	T0M1	T0M0	00H	00000000
TOR2H#	CCU reload register high	CFH									00H	00000000
TOR2L#	CCU reload register low	CEH									00H	00000000
TPCR2H#	Prescaler control register high	CBH	–	–	–	–	–	–	TPCR2H.1	TPCR2H.0	00H	xxxxxx00
TPCR2L#	Prescaler control register low	CAH	TPCR2L.7	TPCR2L.6	TPCR2L.5	TPCR2L.4	TPCR2L.3	TPCR2L.2	TPCR2L.1	TPCR2L.0	00H	00000000
TRIM#	Internal oscillator trim register	96H	–	ENCLK	TRIM.5	TRIM.4	TRIM.3	TRIM.2	TRIM.1	TRIM.0	Notes 4,5	
WDCON#	Watchdog control register	A7H	PRE2	PRE1	PRE0	–	–	WDRUN	WDTOF	WDCLK	Notes 3,5	

Table E.1 Continued

Name	Description	SFR address	Bit functions and addresses MSB	LSB	Reset value Hex	Binary
WDL#	Watchdog load	C1H			FFH	11111111
WFEED1#	Watchdog feed 1	C2H				
WFEED2#	Watchdog feed 2	C3H				

Notes:
* SFRs are bit addressable.
# SFRs are modified from or added to the 80C51 SFRs.
˜ Reserved bits, must be written with 0 s.
§ BRGR1 and BRGR0 must only be written if BRGEN in BRGCON SFR is '0'. If any of them is written if BRGEN = 1, result is unpredictable.
Unimplemented bits in SFRs (labeled '−') are X (unknown) at all times. Unless otherwise specified, ones should not be written to these bits since they may be used for other purposes in future derivatives. The reset values shown for these bits are '0's although they are unknown when read.
1. All ports are in input only (high impendance) state after power-up.
2. The RSTSRC register reflects the cause of the LPC932 reset. Upon a power-up reset, all reset source flags are cleared except POF and BOF-the power-on reset value is xx110000.
3. After reset, the value is 111001x1, i.e., PRE2-PRE0 are all 1, WDRUN = 1 and WDCLK = 1. WDTOF bit is 1 after watchdog reset and is 0 after power-on reset. Other resets will not affect WDTOF.
4. On power-on reset, TRIM SFR is initialised with a factory preprogrammed value. Other resets will not cause initialisation of the TRIM register.
5. The only reset source that affects these SFRs is power-on reset.

E.2 Memory organisation

The LPC932 memory map is shown in Figure E.3. The various LPC932 memory spaces are as follows:

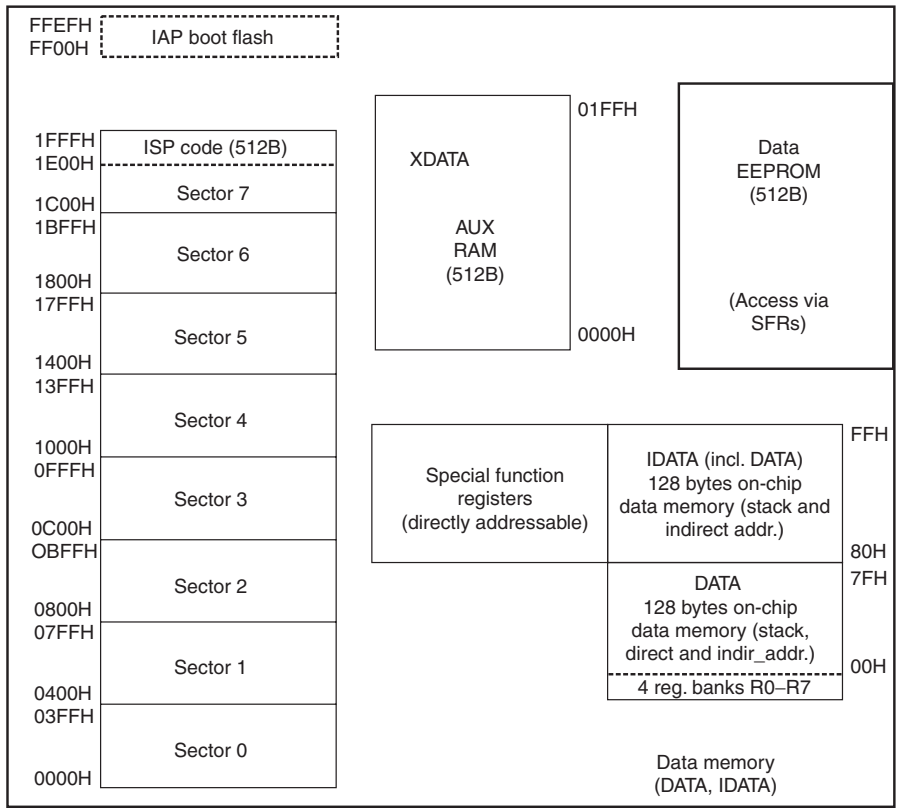


Figure E.3 LPC932 memory map (courtesy Philips Semiconductors)

- DATA 128 bytes of internal data memory space (00H–7FH) accessed via direct or indirect addressing, using instructions other than MOVX and MOVC. All or part of the stack may be in this area.
- IDATA (indirect data) 256 bytes of internal data memory space (00H–FFH) accessed via indirect addressing using instructions other than MOVX and MOVC. All or part of the stack may be in this area. This area includes the DATA area and the 128 bytes immediately above it.
- SFR (special function registers) Selected CPU registers and peripheral control and status registers, accessible only via direct addressing.
- XDATA ('External' data or auxiliary RAM) Duplicates the 80C51 64 KB memory space addressed via the MOVX instruction using the DPTR, R0

or R1. All or part of this space could be implemented on-chip. The LPC932 has 512 bytes of on-chip XDATA memory.
- CODE 64 KB of code memory space, accessed as part of program execution and via the MOVC instruction. The LPC932 has 8 KB of on-chip code memory.

The LPC932 also has 512 bytes of on-chip data EEPROM that is SFR based, byte readable, byte writeable and erasable (via row fill and sector fill). The user can read, write and fill the memory via SFRs and one interrupt. This data EEPROM provides 100 000 minimum erase/program cycles for each byte.

- Byte mode: In this mode, data can be read and written one byte at a time.
- Row fill: In this mode, the addressed row (64 bytes) is filled with a single value. The entire row can be erased by writing 00H.
- Sector fill: In this mode, all 512 bytes are filled with a single value. Writing 00H can erase the entire sector.

After the operation finishes, the hardware will set the EEIF bit, which if enabled will generate an interrupt. The flag is cleared by software.

DATA RAM ARRANGEMENT

The 768 bytes of on-chip RAM organised as shown in Table E.2.

Table E.2 On-chip data memory usages (courtesy Philips Semiconductors)

Type	Data RAM	Size (bytes)
DATA	Memory that can be addressed directly and indirectly	128
IDATA	Memory that can be addressed indirectly	256
XDATA	Auxiliary ('external data') on-chip memory that is accessed using the MOVX instructions	512

FLASH PROGRAM MEMORY

The LPC932 flash memory provides in-circuit electrical erasure and programming. The flash can be read and written as bytes. The sector and page erase functions can erase any flash sector (1 KB) or page (64 bytes). The chip erase operation will erase the entire program memory. In-system programming and standard parallel programming are both available. On-chip erase and write timing generation contribute to a user-friendly programming interface. The LPC932 flash reliably stores memory contents even after 10 000 erase and program cycles. The cell is designed to optimise the erase and programming mechanisms. The LPC932 uses V_{DD} as the supply voltage to perform the program/erase algorithms.

Features:

- Internal fixed boot ROM, containing low-level in-application programming (IAP) routines.
- User programs can call these routines to perform IAP.
- Default loader providing ISP via the serial port, located in upper end of user program memory.
- Boot vector allows user provided flash loader code to reside anywhere in the flash memory space, providing flexibility to the user.
- Programming and erase over the full operating voltage range.
- Read/programming/erase using ISP/IAP.
- Any flash program/erase operation in 2 ms.
- Parallel programming with industry-standard commercial programmers.
- Programmable security for the code in the flash for each sector.
- 10 000 minimum erase/program cycles for each byte.
- 10-year minimum data retention.

E.3 I/O ports

The LPC932 has four I/O ports: port 0, port 1, port 2 and port 3. Ports 0,1 and 2 are 8-bit ports and port 3 is a 2-bit port. The exact number of pins available for I/O depends upon the clock and reset options chosen. Table E.3 shows the number of I/O pins available depending on the clock source.

Table E.3 Number of I/O pins available for the LPC932 28-pin package (courtesy Philips Semiconductors)

Clock source	Reset option	Number of I/O pins
On-chip oscillator or watchdog oscillator	No external reset (except during power-up)	26
	External RST pin supported	25
External clock input	No external reset (except during power-up)	25
	External RST pin supported	24
Low/medium/high speed oscillator (external crystal or resonator)	No external reset (except during power-up)	24
	External RST pin supported	23

PORT CONFIGURATIONS

All but three I/O port pins on the LPC932 may be configured by software to one of four types on a bit-by-bit basis. These are: quasi-bidirectional (standard

80C51 port outputs), push-pull, open drain and input-only. Two configuration registers for each port select the output type for each port pin.

- P1.5 ($\overline{\text{RST}}$) can only be an input and cannot be configured.
- P1.2 (SCL/T0) and P1.3 (SDA/$\overline{\text{INT0}}$) may only be configured to be either input-only or open drain.

Quasi-bidirectional output configuration

Quasi-bidirectional 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 driven 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 pull-up transistors in the quasi-bidirectional output that serve different purposes. LPC932 is a 3 V device, but the pins are 5 V-tolerant. In quasi-bidirectional mode, if a user applies 5 V on the pin, there will be a current flowing from the pin to V_{DD}, causing extra power consumption. Therefore, applying 5 V in quasi-bidirectional mode is discouraged. A quasi-bidirectional port pin has a Schmitt-triggered input that also has a glitch suppression circuit.

Open drain output configuration

The open drain output configuration turns off all pull-ups and only drives the pull-down 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 pull-up, typically a resistor tied to V_{DD}. An open drain port pin has a Schmitt-triggered input that also has a glitch suppression circuit.

Input-only configuration

The input-only port configuration has no output drivers. It is a Schmitt-triggered input that also has a glitch suppression circuit.

Push-pull output configuration

The push-pull output configuration has the same pull-down structure as both the open drain and the quasi-bidirectional output modes, but provides a continuous strong pull-up when the port latch contains a logic 1. The push-pull mode may be used when more source current is needed from a port output. A push-pull port pin has a Schmitt-triggered input that also has a glitch suppression circuit.

PORT 0 ANALOG FUNCTIONS

The LPC932 incorporates two analog comparators. In order to give the best analog function performance and to minimise power consumption, pins that are being used for analog functions must have the digital outputs and digital inputs disabled. Digital outputs are disabled by putting the port output into the input-only (high impedance) mode as described earlier. Digital inputs on port 0 may be disabled through the use of the PT0AD register, bits 1:5. On any reset, PT0AD1.5 defaults to 0s to enable digital functions.

ADDITIONAL PORT FEATURES

After power-up, all pins are in input-only mode. After power-up, all I/O pins except P1.5, may be configured by software. Pin P1.5 is input only. Pins P1.2 and P1.3 are configurable for either input-only or open drain. Every output on the LPC932 has been designed to sink typical LED drive current. However, there is a maximum total output current for all ports, which must not be exceeded. All ports pins that can function as an output have slew rate controlled outputs to limit noise generated by quickly switching output signals. The slew rate is factory-set to approximately 10 ns rise and fall times.

E.4 Timers/counters 0 and 1

The LPC932 has two general-purpose counter/timers which are upward compatible with the standard 80C51 timer 0 and timer 1. Both can be configured to operate either as timers or event counter. An option to automatically toggle the T0 and/or T1 pins upon timer overflow has been added. In the 'Timer' function, the register is incremented every machine cycle. In the 'Counter' function, the register is incremented in response to a 1-to-0 transition at its corresponding external input pin, T0 or T1. In this function, the external input is sampled once during every machine cycle. Timers 0 and 1 have five operating modes (modes 0, 1, 2, 3 and 6). Modes 0,1, 2 and 6 are the same for both timers/counters. Mode 3 is different.

Mode 0. Putting either timer into mode 0 makes it look like an 8048 timer, which is an 8-bit counter with a divide-by-32 prescaler. In this mode, the timer register is configured as a 13-bit register. Mode 0 operation is the same for timer 0 and timer 1.

Mode 1. This mode is the same as mode 0, except that all 16 bits of the timer register are used.

Mode 2. Configures the timer register as an 8-bit counter with automatic reload. Mode 2 operation is the same for timer 0 and timer 1.

Mode 3. When timer 1 is in mode 3 it is stopped. Timer 0 in mode 3 forms two separate 8-bit counters and is provided for applications that require an extra 8-bit timer. When timer 1 is in mode 3 it can still be used by the serial port as a baud rate generator.

Mode 6. In this mode, the corresponding timer can be changed to a PWM with a full period of 256 timer clocks.

TIMER OVERFLOW TOGGLE OUTPUT

Timers 0 and 1 can be configured to automatically toggle a port output whenever a timer overflow occurs. The same device pins that are used for the T0 and T1 count inputs are also used for the timer toggle outputs. The port outputs will be a logic 1 prior to the first timer overflow when this mode is turned on.

REAL-TIME CLOCK/SYSTEM TIMER

The LPC932 has a simple real-time clock that allows a user to continue running an accurate timer while the rest of the device is powered down. The real-time clock can be a wake-up or an interrupt source. The real-time clock is a 23-bit down counter comprised of a 7-bit prescaler and a 16-bit loadable down counter. When it reaches all 0s, the counter will be reloaded again and the RTCF flag will be set. The clock source for this counter can be either the CPU clock (CCLK) or the XTAL oscillator, provided that the XTAL oscillator is not being used as the CPU clock. If the XTAL oscillator is used as the CPU clock, then the RTC will use CCLK as its clock source. Only power-on reset will reset the real-time clock and its associated SFRs to the default state.

E.5 Capture/compare unit (CCU)

This unit features:

- a 16-bit timer with 16-bit reload on overflow;
- selectable dock, with prescaler to divide clock source by any integral number between 1 and 1024;
- 4 compare/PWM outputs with selectable polarity;
- symmetrical/asymmetrical PWM selection;
- 2 capture inputs with event counter and digital noise rejection filter;
- 7 interrupts with common interrupt vector (one overflow, 2Xcapture, 4Xcompare);
- safe 16-bit read/write via shadow registers.

CCU CLOCK (CCUCLK)

The CCU runs on the CCUCLK, which is either PCLK in basic timer mode, or the output of a phase-locked loop (PLL). The PLL is designed to use a clock source between 0.5 MHz and 1 MHz that is multiplied by 32 to produce a CCUCLK between 16 MHz and 32 MHz in PWM mode (asymmetrical or symmetrical). The PLL contains a 4-bit divider to help divide PCLK into a frequency between 0.5 MHz and 1 MHz.

CCU CLOCK PRESCALING

This CCUCLK can be further divided down by a prescaler. The prescaler is implemented as a 10-bit free-running counter with programmable reload at overflow.

BASIC TIMER OPERATION

The timer is a free-running up/down counter with a direction control bit. If the timer counting direction is changed while the counter is running, the count sequence will be reversed. The timer can be written or read at any time. When a reload occurs, the CCU timer overflow interrupt flag will be set, and an interrupt generated if enabled. The 16-bit CCU timer may also be used as an 8-bit up/down timer.

OUTPUT COMPARE

There are four output compare channels A, B, C and D. Each output compare channel needs to be enabled in order to operate and the user will have to set the associated I/O pin to the desired output mode to connect the pin. When the contents of the timer matches that of a capture compare control register, the timer output compare interrupt flag – TOCFx becomes set. An interrupt will occur if enabled.

INPUT CAPTURE

Input capture is always enabled. Each time a capture event occurs on one of the two input capture pins, the contents of the timer are transferred to the corresponding 16-bit input capture register. The capture event can be programmed to be either rising or falling edge triggered. A simple noise filter can be enabled on the input capture by enabling the input capture noise filter bit. If set, the capture logic needs to see four consecutive samples of the same value in order to recognise an edge as a capture event. An event counter can be set to delay a capture by a number of capture events.

PWM OPERATION

PWM operation has two main modes, symmetrical and asymmetrical. In asymmetrical PWM operation the CCU timer operates in down-counting mode regardless of the direction control bit. In symmetrical mode, the timer counts up/down alternately. The main difference from basic timer operation is the operation of the compare module, which in PWM mode is used for PWM waveform generation. As with basic timer operation, when the PWM (compare) pins are connected to the compare logic, their logic state remains unchanged. However, since bit FCO is used to hold the halt value, only a compare event can change the state of the pin. An example of symmetrical PWM waveform generation is shown in Figure E.4.

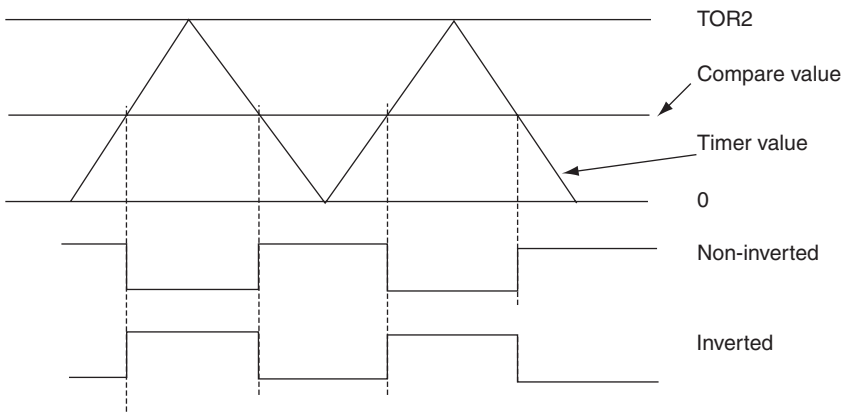


Figure E.4 Symmetrical PWM waveform (courtesy Philips Semiconductors)

Alternating output mode

In asymmetrical mode, the user can set up PWM channels A/B and C/D as alternating pairs for bridge drive control. In this mode the output of these PWM channels is alternately gated on every counter cycle.

PLL OPERATION

The PWM module features a phase locked loop that can be used to generate a CCUCLK frequency between 16 MHz and 32 MHz. At this frequency the PWM module provides ultrasonic PWM frequency with 10-bit resolution provided that the crystal frequency is 1 MHz or higher. The PLL is fed an input signal of 0.5 – 1 MHz and generates an output signal of 32 times the input frequency. This signal is used to clock the timer. The user will have to set a divider that scales PCLK by a factor of 1–16. This divider is found in the SFR register TCR21. The PLL frequency can be expressed as follows:

$$\text{PLL frequency} = \frac{\text{PCLK}}{N+1}$$

where N is the value of PLLDV3:0. Since N ranges in 0–15, the CCLK frequency can be in the range of PCLK to PCLK/16.

WATCHDOG TIMER

The watchdog timer causes a system reset when it underflows as a result of a failure to feed the timer prior to the timer reaching its terminal count. It consists of a programmable 12-bit prescaler and an 8-bit down counter. The down counter is decremented by a tap taken from the prescaler. The clock source for the prescaler is either the PCLK or the nominal 40 kHz watchdog oscillator. The watchdog timer can only be reset by a power-on reset. When the

watchdog feature is disabled, it can be used as an interval timer and may generate an interrupt. Figure E.5 shows the watchdog timer in watchdog mode. Feeding the watchdog requires a two-byte sequence. If PCLK is selected as the watchdog clock and the CPU is powered down, the watchdog is disabled. The watchdog timer has a timeout period that ranges from a few microseconds to a few seconds.

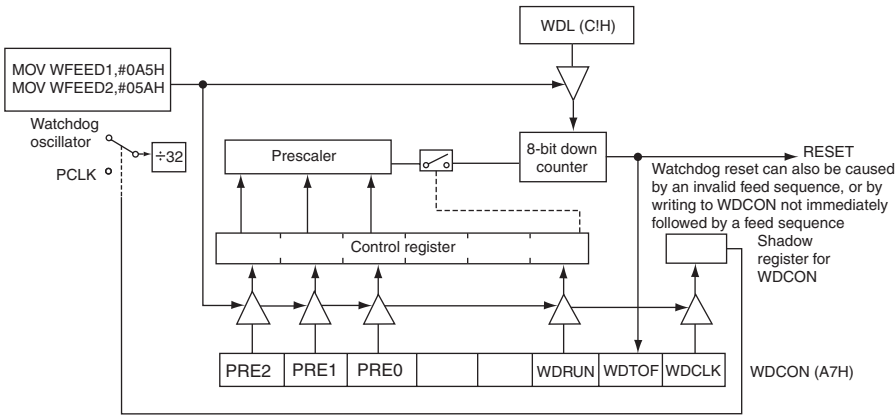


Figure E.5 Watchdog timer in watchdog mode (WDTE = 1) (courtesy Philips Semiconductors)

E.6 Serial interface

UART

The LPC932 has an enhanced UART that is compatible with the conventional 80C51 UART except that Timer 2 overflow cannot be used as a baud rate source. The LPC932 does include an independent baud rate generator. The baud rate can be selected from the oscillator (divided by a constant), timer 1 overflow or the independent baud rate generator. In addition to the baud rate generation, enhancements over the standard 80C51 UART include framing error detection, automatic address recognition, selectable double buffering and several interrupt options. The UART can be operated in four modes: shift register, 8-bit UART, 9-bit UART and CPU clock/32 or CPU clock/16.

Mode 0. Serial data enters and exits through RxD. TxD outputs the shift clock. Eight bits are transmitted or received, LSB first. The baud rate is fixed at 1/16 of the CPU clock frequency.

Mode 1. Ten bits are transmitted (through TxD) or received (through RxD): a start bit (logical 0), 8 data bits (LSB first) and a stop bit (logical 1). When data is received, the stop bit is stored in RB8 in SFR SCON. The baud rate is variable and is determined by the timer 1 overflow rate or the baud rate generator.

Mode 2. Eleven bits are transmitted (through TxD) or received (through RxD): start bit (logical 0), 8 data bits (LSB first), a programmable 9th data bit and a stop bit (logical 1). When data is transmitted, the 9th data bit (TB8 in SCON) can be assigned the value of 0 or 1. Or, for example, the parity bit (P, in the PSW) could be moved into TB8. When data is received, the 9th data bit goes into RB8 in SFR SCON, while the stop bit is not saved. The baud rate is programmable to either 1/16 or 1/32 of the CPU dock frequency, as determined by the SMOD1 bit in PCON.

Mode 3. Eleven bits are transmitted (through TxD) or received (through RxD): a start bit (logical 0), 8 data bits (LSB first), a programmable 9th data bit and a stop bit (logical 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 and is determined by the timer 1 overflow rate or the baud rate generator.

SFR space

The UART SFRs are at the locations shown in Table E.4.

Table E.4 SFR locations for UARTs (courtesy Philips Semiconductors)

Register	Description	SFR location
PCON	Power control	87H
SCON	Serial port (UART) control	98H
SBUF	Serial port (UART) data buffer	99H
SADDR	Serial port (UART) address	A9H
SADEN	Serial port (UART) address enable	B9H
SSTAT	Serial port (UART) status	BAH
BRGR1	Baud rate generator rate high byte	BFH
BRGRO	Baud rate generator rate low byte	BEH
BRGCON	Baud rate generator control	BDH

Baud rate generator and selection

The LPC932 enhanced UART has an independent baud rate generator. The baud rate is determined by a baud rate preprogrammed into the BRGR1 and BRGR0 SFRs, which together form a 16-bit baud rate divisor value that works in a similar manner as timer 1. If the baud rate generator is used, timer 1 can be used for other timing functions. The UART can use either timer 1 or the baud rate generator output (see Figure E.6). Note that timer T1 is further divided by 2 if the SMOD1 bit (PCON.7) is set. The independent baud rate generator uses OSCCLK.

Framing error

Framing error is reported in the status register (SSTAT). In addition, if SMOD0 (PCON.6) is 1, framing errors can be made available in SCON.7

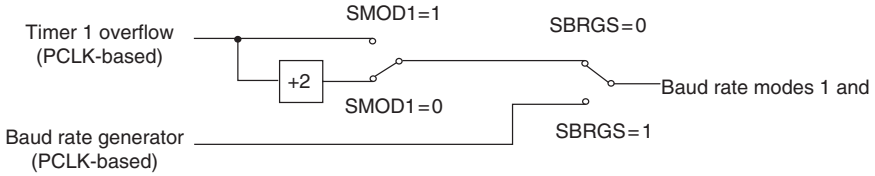


Figure E.6 Baud rate sources for UART (modes 1, 3) (courtesy Philips Semiconductors)

respectively. If SMOD0 is 0, SCON.7 is SM0. It is recommended that SM0 and SM1 (SCON.7-6) are set up when SMOD0 is '0'.

Break detect

Break detect is reported in the status register (SSTAT). A break is detected when 11 consecutive bits are sensed low. The break detect can be used to reset the device and force the device into ISP mode.

Double buffering

The UART has a transmit double buffer that allows buffering of the next character to be written to SBUF while the first character is being transmitted. Double buffering allows transmission of a string of characters with only one stop bit between any two characters, as long as the next character is written between the start bit and the stop bit of the previous character. Double buffering can be disabled. If disabled (DBMOB, i.e. SSTAT.7 = 0), the UART is compatible with the conventional 80C51 UART. If enabled, the UART allows writing to SnBUF while the previous data is being shifted out. Double buffering is only allowed in modes 1, 2 and 3. When operated in mode 0, double buffering must be disabled (DBMOD = 0).

Transmit interrupts with double buffering enabled (Modes 1, 2 and 3)

Unlike the conventional UART, in double buffering mode, the Tx interrupt is generated when the double buffer is ready to receive new data.

The 9th bit (Bit 8) in double buffering (Modes 1, 2 and 3)

If double buffering is disabled TB8 can be written before or after SBUF is written, as long as TB8 is updated some time before that bit is shifted out. TB8 must not be changed until the bit is shifted out, as indicated by the Tx interrupt. If double buffering is enabled, TB8 MUST be updated before SBUF is written, as TB8 will be double-buffered together with SBUF data.

I²C SERIAL INTERFACE

The I²C bus uses two wires (SDA and SCL) to transfer information between devices connected to the bus and has the following features:

- bidirectional data transfer between masters and slaves;
- multimaster bus (no central master);
- arbitration between simultaneously transmitting masters without corruption of serial data on the bus;
- serial dock synchronisation allows devices with different bit rates to communicate via one serial bus;
- serial clock synchronisation can be used as a handshake mechanism to suspend and resume serial transfer;
- the I²C bus may be used for test and diagnostic purposes.

A typical I²C bus configuration is shown in Figure E.7. The LPC932 device provides a byte-oriented I²C interface that supports data transfers up to 400 kHz.

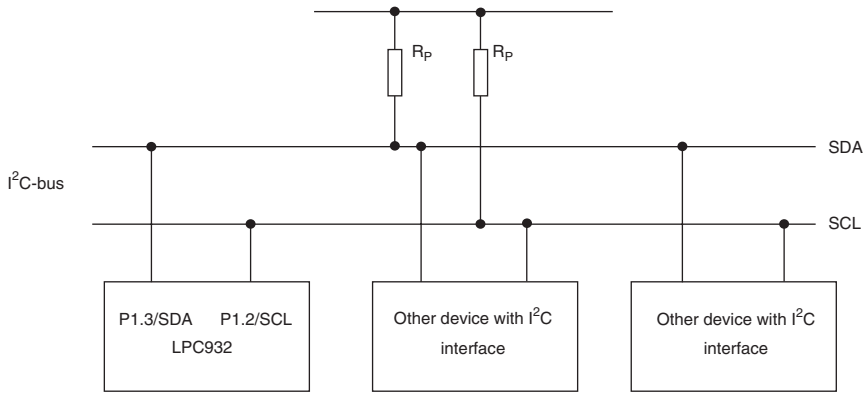


Figure E.7 I²C bus configuration (courtesy Philips Semiconductors)

SERIAL PERIPHERAL INTERFACE (SPI)

The LPC932 device provides another high-speed serial communication interface – the SPI interface. SPI is a full-duplex, high-speed, synchronous communication bus with two operation modes: master mode and slave mode. Up to 3 Mbit/s can be supported in either master or slave mode. It has a transfer completion flag and write collision flag protection.

The SPI interface has four pins; SPICLK, MOSI, MISO and $\overline{SS}$.

- SPICLK, MOSI and MISO are typically tied together between two or more SPI devices. Data flows from master to slave on MOSI (master out slave in) pin and flows from slave to master on MISO (master in slave out) pin. The

SPICLK signal is output in the master mode and is input in the slave mode. If the SPI system is disabled, i.e. SPEN (SPCTL.6) = 0 (reset value), these pins are configured for port functions.

- $\overline{SS}$ is the optional slave select pin. In a typical configuration, an SPI master asserts one of its port pins to select one SPI device as the current slave. An SPI slave device uses its $\overline{SS}$ pin to determine whether it is selected. Typical connections are shown in Figure E.8(a)–(c).

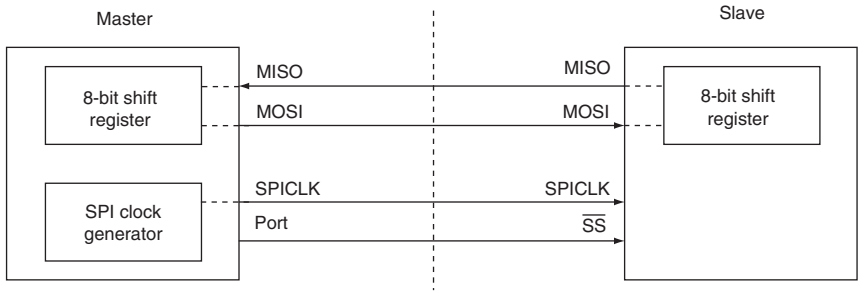


Figure E.8(a) SPI single master, single slave configuration (courtesy Philips Semiconductors)

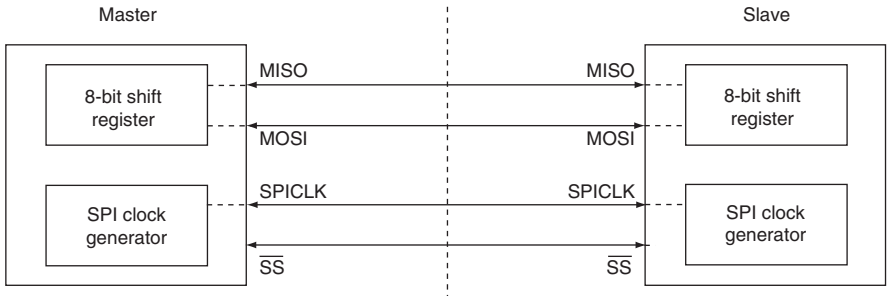


Figure E.8(b) SPI dual device configuration, where either device can be master or slave (courtesy Philips Semiconductors)

E.7 Interrupts

The LPC932 uses a four-priority level interrupt structure. This allows great flexibility in controlling the handling of the LPC932's 15 interrupt sources. Each interrupt source can be individually enabled or disabled by setting or clearing a bit in the IE registers IEN0 or IEN1. The IEN0 register also contains a global enable bit, EA, which enables all interrupts. Each interrupt source can be individually programmed to one of four priority levels by setting or clearing bits in the interrupt priority registers IP0, IP0H, IP1 and IP1H. An interrupt service routine in progress can be interrupted by a higher priority interrupt, but not by another interrupt of the same or lower priority. The highest priority interrupt service cannot be interrupted by any other interrupt source. If two

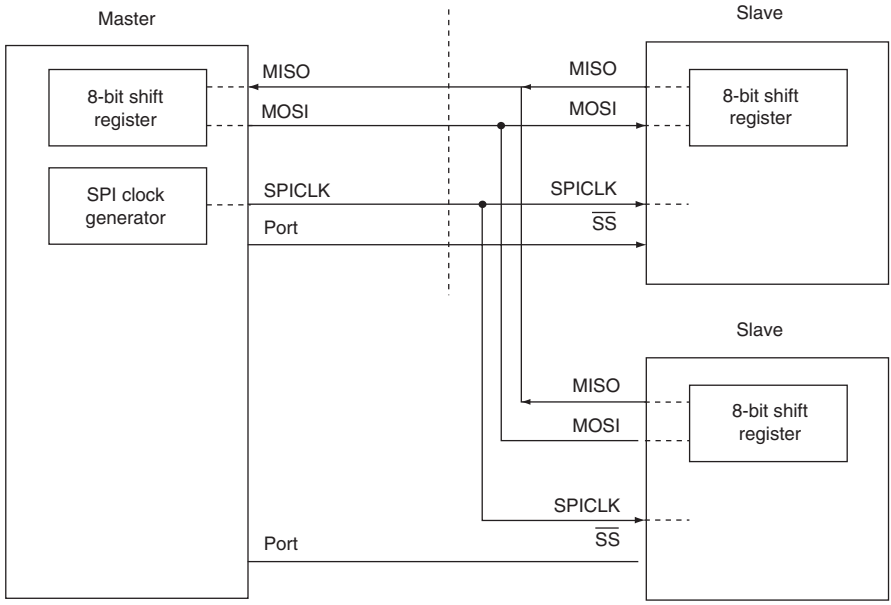


Figure E.8(c) SPI single master, multiple slaves configuration (courtesy Philips Semiconductors)

requests of different priority levels are received simultaneously, the request of higher priority level is serviced. If requests of the same priority level are pending at the start of an instruction cycle, an internal polling sequence determines which request is serviced. This is called the arbitration ranking. The arbitration ranking is only used for pending requests of the same priority level. Table E.5 summarises the interrupt sources, flag bits, vector addresses, enable bits, priority bits, arbitration ranking and whether each interrupt may wake up the CPU from a power down mode.

Interrupt priority structure

There are four SFRs associated with the four interrupt levels: IP0, IP0H, IP1, IP1H. Every interrupt has two bits in IPx and IPxH (x = 0,1) and can therefore be assigned to one of four levels, as shown in Table E.6.

External interrupt inputs

The LPC932 has two external interrupt inputs in addition to the keypad interrupt function. The two interrupt inputs are identical to those present on the standard 80C51 microcontrollers. These external interrupts can be programmed to be level triggered or edge triggered by clearing or setting bit IT1 or IT0 in register TCON. If ITn = 0, external interrupt n is triggered by a low level detected at the INTn pin. If ITn = 1, external interrupt n is edge triggered.

Table E.5 Summary of interrupts (courtesy Philips Semiconductors)

Description	Interrupt flag bit(s)	Vector address	Interrupt enable bit(s)	Interrupt priority	Arbitration ranking	Power down wakeup
External interrupt 0	IE0	0003H	EX0 (IEN0.0)	IP0H.0,IP0.0	1 (highest)	Yes
Timer 0 interrupt	TF0	000BH	ET0 (IEN0.1)	IP0H.1,IP0.1	4	No
External interrupt 1	IE1	0013H	EX1 (IEN0.2)	IP0H.2,IP0.2	7	Yes
Timer 1 interrupt	TF1	001BH	ET1 (IEN0.3)	IP0H.3,IP0.3	10	No
Serial port Tx/Rx[1.4]	TI & RI	0023H	ES/ESR (IEN0.4)	IP0H.4,IP0.4	13	No
Serial port Rx[1.4]	RI					
Brownout detect	BOF	002BH	EB0 (IEN0.5)	IP0H.5,IP0.5	2	Yes
Watchdog timer/real-time clock	WDOVF/ RTCF	0053H	EWDRT (IEN0.6)	IP0H.6,IP0.6	3	Yes
I^2C interrupt	SI	0033H	EI2C (IEN1.0)	IP1H.0,IP1.0	5	No
KBI interrupt	KBIF	003BH	EKBI (IEN1.1)	IP1H.1,IP1.1	8	Yes
Comparators 1/2 interrupt	CMF1/ CMF2	0043H	EC (IEN1.2)	IP1H.2,IP1.2	11	Yes
SPI interrupt	SPIF	004BH	ESPI (IEN1.3)	IP1H.3,IP1.3	14	No
Capture/ compare unit[2]	See Note 2	005BH	ECCU (IEN1.4)	IP1H.4,IP1.4	6	No
Reserved		0063H	(EN1.5)	IP1H.5,IP1.5	9	Yes
Serial port Tx[3]	TI	006BH	EST (IEN1.6)	IP1H.6,IP1.6	12	No
Data EEPROM write completed	EEPROM	0073H	IIEE (IEN1.7)	IP1H.7,IP1.7	15 (lowest)	No

Notes:
1. SSTAT.5 = 0 selects combined serial port (UART) Tx and Rx interrupt; SSTAT.5 = 1 selects serial port Rx interrupt only (Tx interrupt will be different, see Note 3 below).
2. CCU interrupt has multiple sources. Any source in the TIFR2 SFR can cause a CCU interrupt.
3. This interrupt is used as serial port (UART) Tx interrupt if and only if SSTAT.5 = 1, and is disabled otherwise.
4. If SSTAT.O = 1, the following serial port additional flag bits can cause this interrupt: FE, BR, OE.

Table E.6 Interrupt priority level (courtesy Philips Semiconductors)

Priority bits		
IPxH	IPx	Interrupt priority level
0	0	Level 0 (lowest priority)
0	1	Level 1
1	0	Level 2
1	1	Level 3 (highest priority)

In this mode if consecutive samples of the INTn pin show a high level in one cycle and a low level in the next cycle, interrupt request flag IEn in TCON is set, causing an interrupt request. Since the external interrupt pins are sampled once each machine cycle, an input high or low level should be held for at least one machine cycle to ensure proper sampling. If the external interrupt is edge triggered, the external source has to hold the request pin high for at least one machine cycle, and then hold it low for at least one machine cycle. This is to ensure that the transition is detected and that interrupt request flag IEn is set. IEn is automatically cleared by the CPU when the service routine is called. If the external interrupt is level triggered, the external source must hold the request active until the requested interrupt is generated. If the external interrupt is still asserted when the interrupt service routine is completed, another interrupt will be generated. It is not necessary to clear the interrupt flag IEn when the interrupt is level sensitive, it simply tracks the input pin level. If an external interrupt is enabled when the LPC932 is put into power down or idle mode, the interrupt occurrence will cause the processor to wake up and resume operation.

External interrupt pin glitch suppression

Most of the LPC932 pins have glitch suppression circuits to reject short glitches. However, pins SDA/$\overline{\text{INT0}}$/P1.3 and SCL/T0/P1.2 do not have the glitch suppression circuits. Therefore, $\overline{\text{INT1}}$ has glitch suppression while $\overline{\text{INT0}}$ does not.

Keypad interrupt (KBI)

The Keypad interrupt function is intended primarily to allow a single interrupt to be generated when port 0 is equal to or not equal to a certain pattern. This function can be used for bus address recognition or keypad recognition. The user can configure the port via SFRs for different tasks. The keypad interrupt mask register (KBMASK) is used to define which input pins connected to port 0 can trigger the interrupt. The keypad pattern register (KBPATN) is used to define a pattern that is compared to the value of port 0. The keypad interrupt flag (KBIF) in the keypad interrupt control register (KBCON) is set when the condition is matched while the keypad interrupt function is active. An interrupt will be generated if enabled. The PATN_SEL bit in the keypad interrupt

control register (KBCON) is used to define equal or not-equal for the comparison. In order to use the keypad interrupt as an original KBI function the user needs to set KBPATN = 0FFH and PATN_SEL = 1 (not equal), then any key connected to port 0 which is enabled by the KBMASK register will cause the hardware to set KBIF and generate an interrupt if it has been enabled. The interrupt may be used to wake up the CPU from idle or power down modes. This feature is particularly useful in handheld, battery-powered systems that need to carefully manage power consumption yet also need to be convenient to use. In order to set the flag and cause an interrupt, the pattern on port 0 must be held longer than 6 CCLKs.

E.8 Analog comparators

Two analog comparators are provided on the LPC932. Input and output options allow use of the comparators in a number of different configurations. Comparator operation is such that the output is a logical one (which may be read in a register and/or routed to a pin) when the positive input (one of two selectable pins) is greater than the negative input (selectable from a pin or an internal reference voltage). Otherwise the output is a zero. Each comparator may be configured to cause an interrupt when the output value changes.

Comparator configuration

Each comparator has a control register, CMP1 for comparator 1 and CMP2 for comparator 2. The control registers are identical. The overall connections to both comparators are shown in Figure E.9.

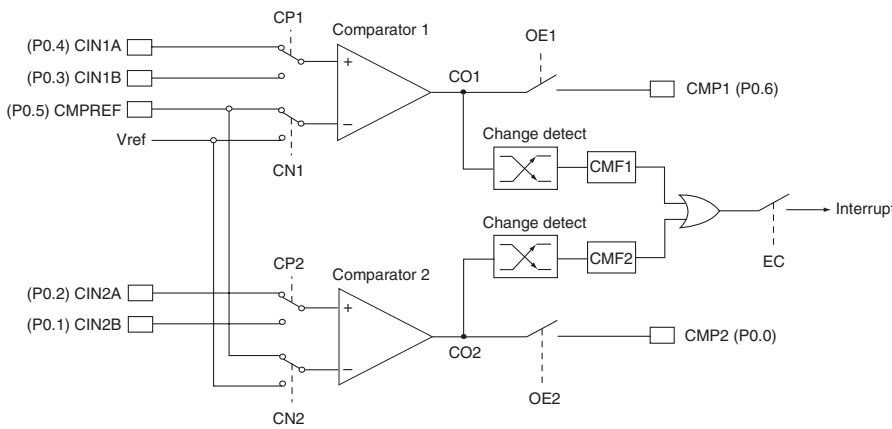


Figure E.9 Comparator input and output connections (courtesy Philips Semiconductors)

There are eight possible configurations for each comparator, as determined by the control bits in the corresponding CMPn register: CPn, CNn and OEn. When

each comparator is first enabled, the comparator output and interrupt flag are not guaranteed to be stable for 10 μs. The corresponding comparator interrupt should not be enabled during that time, and the comparator interrupt flag must be cleared before the interrupt is enabled in order to prevent an immediate interrupt service.

CMPn (Comparator control registers (CMP1 and CMP2))

Address: ACh for CMP1, ADh for CMP2

7	6	5	4	3	2	1	0
–	–	CEn	CPn	CNn	OEn	COn	CMFn

Not bit addressable
Reset source(s): any reset
Reset value: xx000000B

Bit	Symbol	Function
CMPn.7, 6	–	Reserved for future use. Should not be set to 1 by user programs
CMPn.5	CEn	Comparator enable. When set, the corresponding comparator function is enabled. Comparator output is stable 10 μs after CEn is set
CMPn.4	CPn	Comparator positive input select. When 0, CINnA is selected as the positive comparator input. When 1, CINnB is selected as the positive comparator input
CMPn.3	CNn	Comparator negative input select. When 0, the comparator reference pin CMPREF is selected as the negative comparator input. When 1, the internal comparator reference, Vref, is selected as the negative comparator input
CMPn.2	OEn	Output enable. When 1, the comparator output is connected to the CMPn pin if the comparator is enabled (CEn = 1). This output is asynchronous to the CPU clock
CMPn.1	COn	Comparator output, synchronised to the CPU clock to allow reading by software
CMPn.0	CMFn	Comparator interrupt flag. This bit is set by hardware whenever the comparator output COn changes state. This bit will cause a hardware interrupt if enabled. Cleared by software

Internal reference voltage

An internal reference voltage, Vref, may supply a default reference when a single comparator input pin is used.

Comparator interrupt

Each comparator has an interrupt flag CMFn contained in its configuration register. This flag is set whenever the comparator output changes state. The flag

may be polled by software or may be used to generate an interrupt. The two comparators use one common interrupt vector. The interrupt will be generated when the IE bit EC in the IEN1 register is set and the interrupt system is enabled via the EA bit in the IEN0 register. If both comparators enable interrupts, after entering the interrupt service routine, the user will need to read the flags to determine which comparator caused the interrupt.

Comparators and power reduction modes

Either or both comparators may remain enabled when power down or idle mode is activated, but both comparators are disabled automatically in total power down mode. If a comparator interrupt is enabled (except in total power down mode), a change of the comparator output state will generate an interrupt and wake up the processor. If the comparator output to a pin is enabled, the pin should be configured in the push-pull mode in order to obtain fast switching times while in power down mode. The reason is that with the oscillator stopped, the temporary strong pull-up that normally occurs during switching on a quasi-bidirectional port pin does not take place. Comparators consume power in power down and idle modes, as well as in the normal operating mode. This should be taken into consideration when system power consumption is an issue. To minimise power consumption, the user can power down the comparators by disabling the comparators and setting PCONA.5 to '1', or simply putting the device in total power down mode.

E.9 Clocks

Clock definitions

The LPC932 device has several internal clocks defined as:

- OSCCLK – Input to the DIVM clock divider. OSCCLK is selected from one of four clock sources and can also be optionally divided to a slower frequency.

  Note: f_{osc} is defined as the OSCCLK frequency.

- CCLK – CPU clock; output of the clock divider. There are two CCLK cycles per machine cycle, and most instructions are executed in one to two machine cycles (two or four CCLK cycles).
- RCCLK – The internal 7.373 MHz RC oscillator output.
- PCLK – Clock for the various peripheral devices and is CCLK/2.

CPU clock (OSCCLK)

The LPC932 provides several user-selectable oscillator options in generating the CPU clock. This allows optimisation for a range of needs from high

precision to lowest possible cost. These options are configured when the FLASH is programmed and include an on-chip watchdog oscillator, an on-chip RC oscillator, an oscillator using an external crystal, or an external clock source. The crystal oscillator can be optimised for low, medium or high frequency crystals covering a range from 20 kHz to 12 MHz.

Low speed oscillator option. This option supports an external crystal in the range of 20 kHz to 100 kHz. Ceramic resonators are also supported in this configuration.

Medium speed oscillator option. This option supports an external crystal in the range of 100 kHz to 4 MHz. Ceramic resonators are also supported in this configuration.

High speed oscillator option. This option supports an external crystal in the range of 4 MHz to 12 MHz. Ceramic resonators are also supported in this configuration.

Figure E.10 shows the connections for a crystal oscillator.

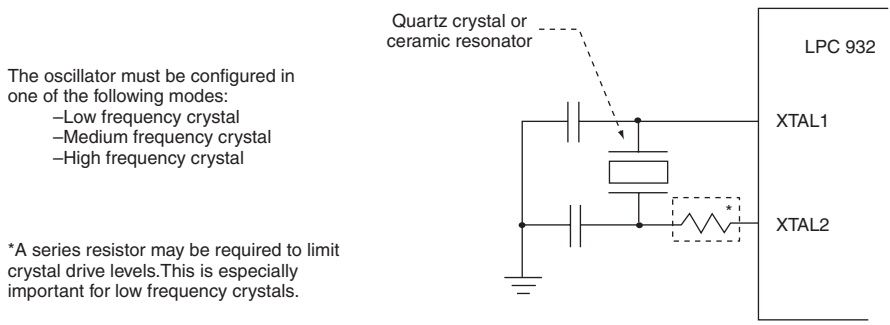


Figure E.10 Using the crystal oscillator (courtesy Philips Semiconductors)

Clock output

The LPC932 supports a user selectable dock output function on the XTAL2/CLKOUT pin when crystal oscillator is not being used. This condition occurs if another clock source has been selected (on-chip RC oscillator, watchdog oscillator, external clock input on X1) and if the real-time clock is not using the crystal oscillator as its clock source. This allows external devices to synchronise to the LPC932. This output is enabled by the ENCLK bit in the TRIM register. The frequency of this clock output is half that of the CCLK. If the clock output is not needed in idle mode, it may be turned off prior to entering idle, saving additional power.

On-chip RC oscillator option. The LPC932 has a 6-bit TRIM register that can be used to tune the frequency of the RC oscillator. During reset, the TRIM value is initialised to a factory preprogrammed value to adjust the oscillator frequency to 7.373 MHz, $\pm 2.5\%$. End user applications can write to the trim register to adjust the on-chip RC oscillator to other frequencies.

TRIM register

7	6	5	4	3	2	1	0
–	ENCLK	TRIM.5	TRIM.4	TRIM.3	TRIM.2	TRIM.1	TRIM.O

Address: 96H
Not bit addressable
Reset source(s): power-up only
Reset value: on power-up reset, ENCLK = 0 and TRIM.5-0 are loaded with the factory programmed value.

Bit	Symbol	Function
TRIM.7	–	Reserved
TRIM.6	ENCLK	When ENCLK = 1, CCLK/2 is output on the XTAL2 pin (P3.0) provided that the crystal oscillator is not being used. When ENCLK = 0, no clock output is enabled
TRIM.5–0		Trim value

Note: On reset, the TRIM SFR is initialised with a factory preprogrammed value. When setting or clearing the ENCLK bit, the user should retain the contents of bits 5:0 of the TRIM register. This can be done by reading the contents of the TRIM register (into the ACC for example), modifying bit 6 and writing this result back into the TRIM register. Alternatively, the 'ANL direct' or 'ORL direct' instructions can be used to clear or set bit 6 of the TRIM register.

Watchdog oscillator option. The watchdog has a separate oscillator, which has a frequency of 400 kHz. This oscillator can be used to save power when a high clock frequency is not needed.

External clock input option. In this configuration, the processor clock is derived from an external source driving the XTAL1/P3.1 pin. The rate may be from 0 Hz up to 12 MHz. The XTAL2/P3.0 pin may be used as a standard port pin or a clock output.

Appendix F

XAG49 Microcontroller

F.1 Introduction

Details of this device are reproduced with kind permission of Philips Semiconductors. Data regarding the device may be found on the Philips website at www.semiconductors.philips.com.

GENERAL DESCRIPTION

The XAG49 is a member of Philips' 80C51 XA (extended architecture) family of high performance 16-bit single-chip microcontrollers. The XA range offers compatibility with the 80C51, giving the user enhanced performance with increased memory capacity. The XAG49 contains 64 KB 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 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 (in-application programming).

FEATURES

- 64 KB of on-chip flash program memory with ISP capability;
- five flash blocks; two 8 KB blocks and three 16 KB blocks;
- nearly identical to XA-G3, except for double the program and RAM memories;
- single supply voltage ISP of the flash memory ($V_{pp} = V_{DD}$ or $V_{pp} = 12\,V$ if desired);
- boot ROM contains low-level flash programming routines for in-application programming 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 MB (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 four programmable output configurations for each pin;
- 30 MHz operating frequency at 5 V;
- 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.

The basic block diagram is shown in Figure F.1.

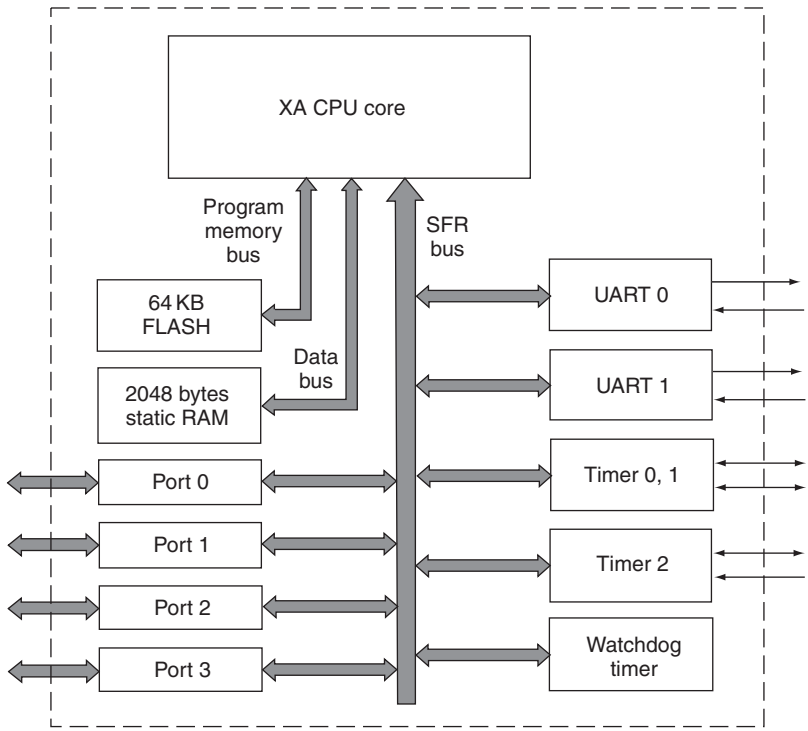
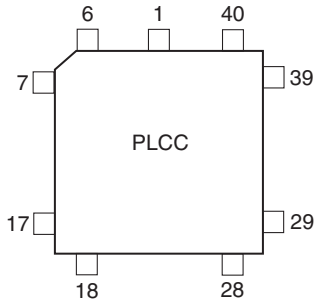


Figure F.1 XAG49 block diagram (courtesy Philips Semiconductors)

F.2 Pin-out diagram for the XAG49

Packages include a 44-pin PLCC package and a 44-pin low quad flat pack (LQFP) package. The PLCC package is illustrated in Figure F.2. Note that although both packages have 44 pins only 42 pins in each case are utilised since 2 pins have no internal connections (NC).



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
12	NC	34	$\overline{NC}$
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}

Figure F.2 XAG49 PLCC package pin-out layout (courtesy Philips Semiconductors)

A brief description of the function of each of the pins, as applicable to the PLCC package, is as follows:

SUPPLY VOLTAGE (V_{DD} and V_{SS})

The device operates from a single +5 V supply connected to pins 23 and 44 (V_{DD}) while pins 1 and 22 (V_{SS}) are grounded.

INPUT/OUTPUT (I/O) PORTS

Thirty-two of the pins are arranged as four 8-bit I/O ports P0–P3. These pins are dual purpose with each capable of operating as a control line or part of the

data/address bus in addition to the I/O functions. Each port operates as an 8-bit I/O port with a user-configurable output type. Port latches have 1s written to them and are configured in the quasi-bidirectional mode during reset. Operation of the port pins as inputs or outputs depends on the selected port configuration. Each port pin may be configured independently. Details of each port are as follows:

- Port 0 (pins 36 to 43). When the external program/data bus is used, the port becomes the multiplexed low data/instruction byte and address lines 4 to 11.
- Port 1 (pins 2 to 9). The port pins also serve special functions as follows:

P1.0	A0/$\overline{\text{WRH}}$	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 16-bit width, this pin becomes the high byte write strobe
P1.1	A1	Address bit 1 of the external address bus
P1.2	A2	Address bit 2 of the external address bus
P1.3	A3	Address bit 3 of the external address bus
P1.4	RxD1	Receiver input for serial port 1
P1.5	TxD1	Transmitter output for serial port 1
P1.6	T2	Timer/counter 2 external count input/clockout
P1.7	T2EX	Timer/counter 2 reload/capture/direction control

- Port 2 (pins 24 to 31). When the external program/data bus is used in 8-bit mode the number of address lines that appear on the port is user programmable; when used in 16-bit mode the port becomes the multiplexed low data/instruction byte and address lines 12 to 19.
- Port 3 (pins 11 and pins 13 to 19). These pins, in addition to the I/O role, serve the special functions summarised below:

P3.0	RxD0	Receiver serial data input port
P3.1	TxD0	Transmitter serial data output port
P3.2	$\overline{\text{INT0}}$	External interrupt 0 input
P3.3	$\overline{\text{INT1}}$	External interrupt 1 input
P3.4	T0	Timer/counter 0 external input, or timer 0 overflow output
P3.5	T1/BUSW	Timer/counter 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) where 0 = 8-bit bus and 1 = 16-bit bus
P3.6	$\overline{\text{WRL}}$	External data memory low byte write strobe
P3.7	$\overline{\text{RD}}$	External data memory read strobe

$\overline{RST}$ (RESET) (PIN 10)

When this input goes low the microcontroller is reset, causing the I/O ports and peripherals to assume their default values. The processor also begins execution at the address contained in the reset vector.

XTAL1 AND XTAL2 (PINS 21 AND 20 RESPECTIVELY)

The XTAL1 input provides an input to the inverting amplifier used in the oscillator circuit and an input to the internal clock generator circuits. The XTAL2 pin provides an output from the oscillator amplifier.

$\overline{PSEN}$ (PROGRAM STORE ENABLE) (PIN 32)

This pin provides an output read strobe to external program memory. The output is active low during the fetch stage of an instruction. The signal is not activated during a fetch from internal memory.

ALE (ADDRESS LATCH ENABLE) (PIN 33)

A high ALE signal is an output pulse used to latch the address portion of the multiplexed address/data bus. The signal only occurs when it is needed in order to process a bus cycle.

$\overline{EA}$/WAIT/V_{PP} (EXTERNAL ACCESS/WAIT/PROGRAMMING VOLTAGE) (PIN 35)

The $\overline{\text{EA}}$ input determines whether the internal program memory of the microcontroller is used for code execution. The value on the $\overline{\text{EA}}$ 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, with 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, the cycle will be extended until Wait is released. During EPROM programming this pin is also the programming supply voltage input.

F.3 Memory organisation

INTRODUCTION

The memory space of XA is configured such that code and data memory (including SFRs) are organised in separate address spaces. The XA architecture supports 16 MB (24-bit address) of both code and data space. The size and type of memory are specific to an XA derivative. The XAG49 has only 20 address

lines with a limit of 1 MB of memory. 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. The XAG49 has flash code memory and RAM data memory.

THE XA REGISTER FILE

The XA architecture is optimised for arithmetic, logical and address-computation operations on the contents of one or more registers in the XA register file. The register file (see Figure F.3) allows access to eight words of data at any one time; the eight words are also addressable as 16 bytes.

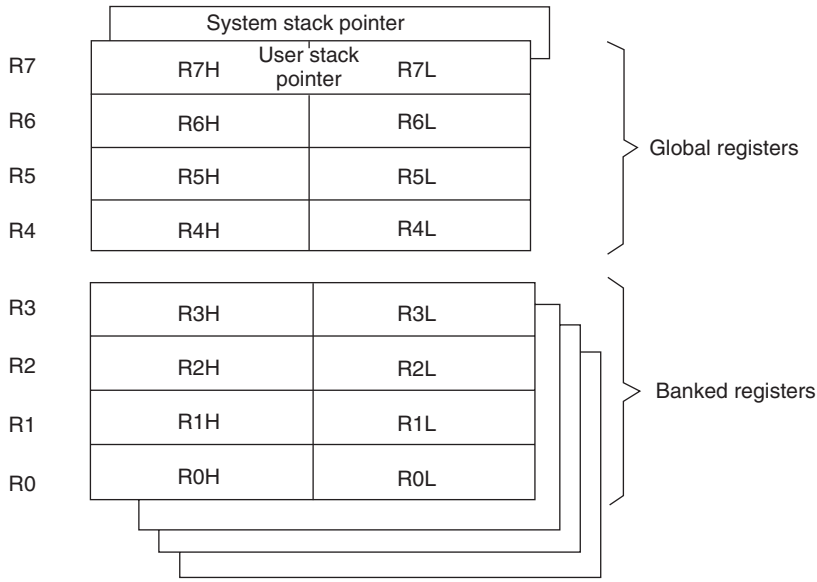


Figure F.3 XA register file diagram (courtesy Philips Semiconductors)

The bottom four word registers are 'banked'. That is, there are four groups of registers, any one of which may occupy the bottom four words of the register file at any one time. This feature may be used to minimise 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 SP, 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 three unselected banks of the bottom four 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 eight 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.

There are two stack pointers, one for user mode and another for SM. 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 SM, and SSP is in SP (R7). (Note that 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 one set of the four 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

BIT ACCESS TO REGISTERS

The XA registers are all bit addressable. Figure F.4 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.

THE XA MEMORY SPACES

The XA divides physical memory into program and data memory spaces. Twenty-four address bits, corresponding to a 16 MB address space, are defined in the XA architecture. In any given XA implementation, fewer than all 24 address bits may actually be used, and there is provision for a small-memory mode which uses only 16-bit addresses. 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. As mentioned earlier the XAG49 utilises only 20 address lines with a 1 MB address space.

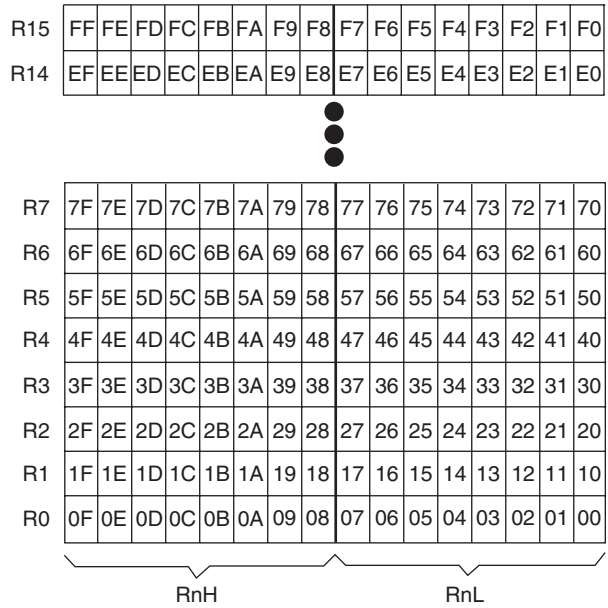


Figure F.4 XA bit address to registers (courtesy Philips Semiconductors)

DATA MEMORY

The XA architecture supports a 16 MB 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 64 KB segments (Figure F.5) to provide an intrinsic protection mechanism for multi-tasking systems and to improve performance.

Segment registers provide the upper eight address bits needed to obtain a complete 24-bit address in applications that require large data memories. The XA provides two 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. Address generation using the segment select register is shown in Figure F.6.

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 64 KB each.

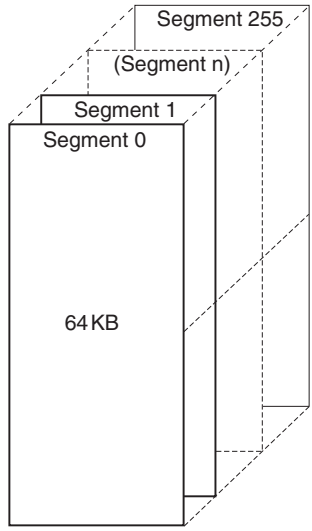


Figure F.5 XA data memory segments (courtesy Philips Semiconductors)

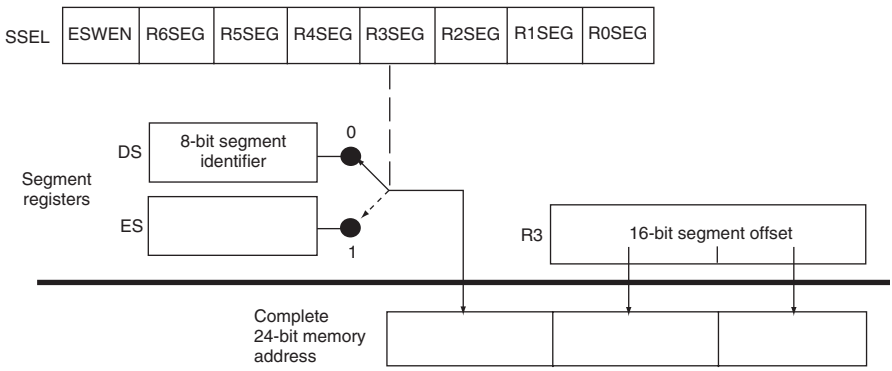


Figure F.6 Use of segment registers for XA address generation (courtesy Philips Semiconductors)

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

Direct. The first 1 KB 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. The memory map for the XAG49 is shown in Figure F.7.

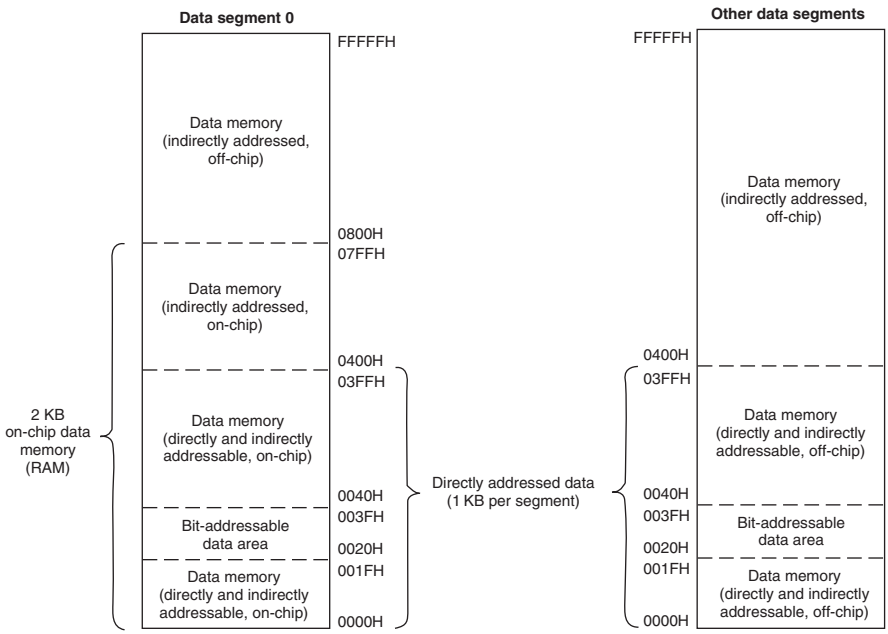
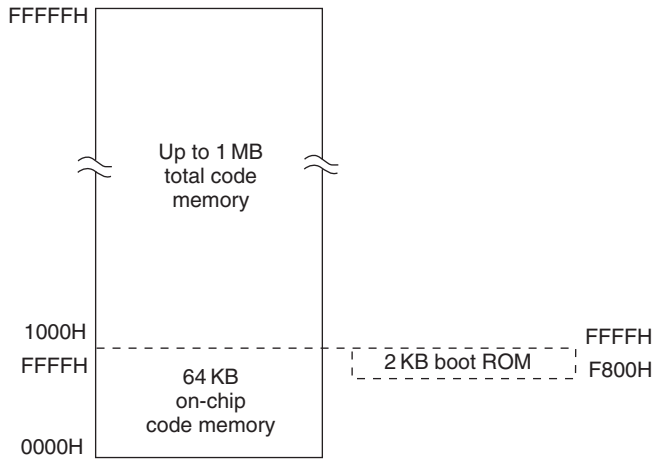


Figure F.7 XAG49 data memory map (courtesy Philips Semiconductors)

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. In XA derivatives with on-chip ROM or EPROM code memory, the on-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 PC may be used to identify the portion of code memory referenced by the pointer. The arrangement for the XAG49 device is shown in Figure F.8.



Note: The boot ROM replaces the top 2 KB of Flash memory when it is enabled.

Figure F.8 XAG49 program memory map (courtesy Philips Semiconductors)

FLASH EPROM MEMORY

The XAG49 flash memory augments EPROM functionality with in-circuit electrical erasure and programming. The flash can be read and written as bytes. The chip erase operation will erase the entire program memory. The block erase function can erase any single flash block. In-circuit programming and standard parallel programming are both available. On-chip erase and write timing generation contribute to a user-friendly programming interface. The XAG49 flash reliably stores memory contents even after 10 000 erase and program cycles. The cell is designed to optimise the erase and programming mechanisms. In addition, the combination of advanced tunnel oxide processing and low internal electric fields for erase and programming operations produces reliable cycling. For ISP, the XAG49 can use a single +5 V power supply. Faster ISP may be obtained, if required, through the use of a +12 V V_{pp} supply. Parallel programming (using separate programming hardware) uses a +12 V V_{pp} supply.

Features

- Flash EPROM internal program memory with single voltage programming and block erase capability.
- Internal 2 KB fixed boot ROM, containing low-level programming routines and a default loader. The boot ROM can be turned off to provide access to the full 64 KB flash memory.
- Boot vector allows user provided flash loader code to reside anywhere in the flash memory space. This configuration provides flexibility to the user.
- Default loader in boot ROM allows programming via the serial port without the need for a user provided loader.
- Up to 1 MB external program memory if the internal program memory is disabled ($\overline{EA} = 0$) .
- Programming and erase voltage: $V_{pp} = V_{DD}$ (5 V power supply), or 12 V $\pm 5\%$ for ISP. Using 12 V V_{pp} for ISP improves programming and erase time.
- Read/programming/erase in ISP:
  1. Byte-wise read (60 ns access time).
  2. Byte programming (3–4 min for 64 KB flash, depending on clock frequency).
- In-circuit programming via user-selected method, typically RS232 or parallel I/O port interface.
- Programmable security for the code in the flash.
- 10 000 minimum erase/program cycles each byte over operating temperature range.
- 10-year minimum data retention.

Flash memory organisation

The XAG49 contains 64 KB of flash program memory. This memory is organised as five separate blocks. The first two blocks are 8 KB in size, filling the program memory space from address 0 through 3FFF hex. The final three blocks are 16 KB in size and occupy addresses from 4000 through FFFF hex. Figure F.9 shows the flash memory configuration.

Flash programming and erasure

The XAG49 flash microcontroller supports a number of programming possibilities for the on-chip flash memory. The flash memory may be programmed in a parallel fashion on standard programming equipment in a manner similar to an EPROM microcontroller. The XAG49 microcontroller is able to program its own flash memory while the application code is running. Also, a default loader built into a boot ROM allows programming blank devices serially through the UART.

Using any of these types of programming, any of the individual blocks may be erased separately, or the entire chip may be erased. Programming of the flash memory is accomplished one byte at a time.

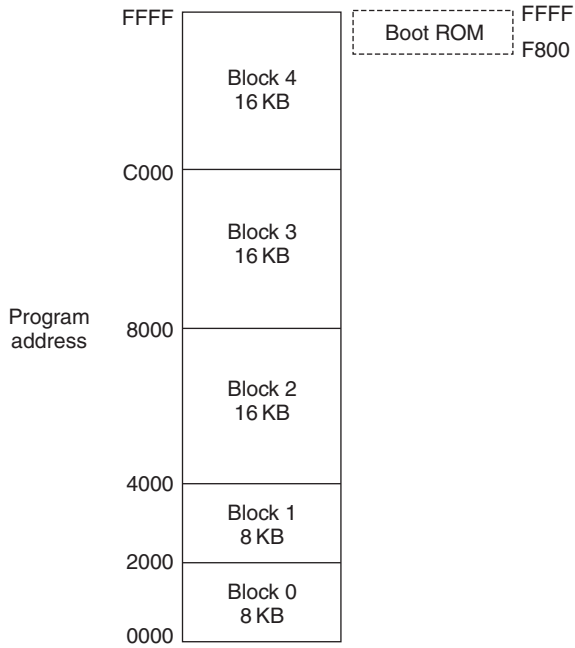


Figure F.9 XAG49 flash memory configuration (courtesy Philips Semiconductors)

BOOT ROM

When the microcontroller programs its own flash memory, all of the low level details are handled by code that is permanently contained in a 2 KB 'Boot ROM' that is separate from the flash memory. A user program simply calls the entry point with the appropriate parameters to accomplish the desired operation. Boot ROM operations include things like: erase block, program byte, verify byte program security lock bit, etc. The boot ROM overlays the program memory space at the top of the address space from F800 to FFFF hex, when it is enabled by setting the ENBOOT bit at AUXR1.7. The boot ROM may be turned off so that the upper 2 KB of flash program memory are accessible for execution.

ENBOOT AND PWR.VLD

Setting the ENBOOT bit in the AUXR register enables the boot ROM and activates the on-chip V_{pp} generator if V_{pp} is connected to V_{DD} rather than 12 V externally. The PWR_VLD flag indicates that V_{pp} is available for programming and erase operations. This flag should be checked prior to calling the boot ROM for programming and erase services. When ENBOOT is set, it typically takes 5 μs for the internal programming voltage to be ready.

The ENBOOT bit will automatically be set if the status byte is non-zero during reset, or when $\overline{\text{PSEN}}$ is low, ALE is high and $\overline{\text{EA}}$ is high at the falling

edge of reset. Otherwise, ENBOOT will be cleared during reset. When programming functions are not needed, ENBOOT may be cleared. This enables access to the 2 KB of flash code memory that is overlaid by the boot ROM, allowing a full 64 KB of flash code memory.

F.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. The core registers that are accessed as SFRs include PCON, SCR, SSEL, PSWH, PSWL, CS, ES and DS. Communication with these registers as well as on-chip peripheral devices is via the dedicated SFR bus, which is shown in Figure F.1.

SFR ADDRESS SPACE

The total SFR space is 1 KB 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 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. The complete list of SFRs available for the XAG49 device is illustrated in Table F.1.

I/O port output configuration. Each I/O port pin can be user configured to one of four output types. The types are:

- quasi-bidirectional (essentially the same as standard 80C51 family I/O ports)
- open-drain
- push-pull
- 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 two SFRs for each port, called PnCFGA and PnCFGB, where 'n' is the port

Table F.1 XAG49 special function registers (courtesy Philips Semiconductors)
Special function registers

Name	Description	SFR address	MSB			Bit functions and addresses				LSB	Reset value
AUXR	Auxiliary function register	44C	ENBOOT	FMIDLE	PWR_VLD	–	–	–	–	–	
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
			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

Table F.1 Continued

Name	Description	SFR address	Bit functions and addresses								Reset value
			MSB							LSB	
			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

Table F.1 Continued

Name	Description	SFR address	Bit functions and addresses								Reset value
			MSB							LSB	
			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-G49 has 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-G49 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.
8. The AUXR reset value is typically 00H. If the Boot Loader is activated at reset because the Flash status byte is non-zero or because the Boot Vector has been forced (by $\overline{\text{PSEN}} = 0$, ALE = 1, $\overline{\text{EA}} = 1$ at reset), the AUXR reset value will be 1x00 0000B. Bit 6 will be a 1 if the on-chip V_{pp} generator is running and ready, otherwise it will be a 0.

number. One bit in each of the two SFRs relates to the output setting for the corresponding port pin, allowing any combination of the two output types to be mixed on those port pins. For example, the output type of port 1 pin 3 is controlled by the setting of bit 3 in the SFRs P1CFGA and P1CFGB. Table F.2 shows the configuration register settings for the four port output types.

Table F.2 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.

F.5 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 (timers 0, 1 and 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). Details of the SCR register are shown in Table F.3.

The frequency of TCLK may be selected to be the oscillator input divided by 4 ($f_{osc}/4$), the oscillator input divided by 16 ($f_{osc}/16$), or the oscillator input divided by 64 ($f_{osc}/64$). 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 two oscillator cycles, so it can take as many as four oscillator cycles to detect a transition. Thus the maximum count rate that can be

Table F.3 SCR register bit functions
SCR.
address 440H
not bit addressable

MSB							LSB
–	–	–	–	PT1	PT0	CM	PZ
7	6	5	4	3	2	1	0

Bit	Symbol	Function
7, 6, 5, 4	–	Reserved for future use
3, 2	PT1, PT0	Sets operating conditions as follows: PT1 PT0 Prescaler selection 0 0 $f_{osc}/4$ 0 1 $f_{osc}/16$ 1 0 $f_{osc}/64$ 1 1 reserved
1	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
0	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 64 KB

supported is $f_{osc}/4$. 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 two 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 SFR 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 Table F.4.

Table F.4 TMOD register bit functions
TMOD.
address 45CH
not bit addressable

MSB LSB

GATE	C/T	M1	M0	GATE	C/T	M1	M0
---------- TIMER 1 ----------				---------- TIMER 0 ----------			

The bit functions are:

GATE When set timer/counter x is enabled when INTx pin is high and TRx (see TCON) is set. When clear timer x is enabled when TRx bit set

C/T When clear, timer operation (input from internal clock) When set, counter operation (input from Tx input pin)

The M1 and M0 bit functions depend on the bit assignment as shown below:

M1	M0	Operation
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)

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 new enhanced 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:

$$\text{Timer\_rate} = \frac{f_{\text{osc}}}{N^{*}(65536 - \text{Timer\_reload\_value})}$$

where N = the TCLK prescaler value: 4 (default), 16, or 64.

Mode l

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:

$$\text{Timer\_rate} = \frac{f_{\text{osc}}}{N(256 - \text{Timer\_Reload\_value})}$$

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, $\overline{\text{INT0}}$ and TF0. TH0 is locked into a timer function and takes over the use of TR1 and TFI 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. Details of the TCON register are shown in Table F.5.

Table F.5 TCON register bit functions

TCON.
address 410H
bit addressable

MSB							LSB
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
7	6	5	4	3	2	1	0

Bit	Symbol	Function
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
6	TR1	Timer 1 run control bit. Set/cleared by software to turn counter/timer 1 on/off
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
4	TR0	Timer 0 run control bit. Set/cleared by software to turn counter/timer 0 on/off
3	IE1	Interrupt 1 edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed
2	IT1	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupts
1	IE0	Interrupt 0 edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed
0	IT0	Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupts

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 ($f_{osc}/8$ to $f_{osc}/8{,}388{,}608$) outputs could be achieved by adjusting the prescaler along with the auto-reload register values. With a 30.0 MHz oscillator, this range would be 3.58 Hz to 3.75 MHz. Details of the SFR register TSTAT are shown in Table F.6.

Table F.6 TSTAT register bit functions

TSTAT
address 411H
bit addressable

MSB							LSB
–	–	–	–	–	T1OE	–	T0OE
7	6	5	4	3	2	1	0

Bit	Symbol	Function
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
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

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 SFR 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 F.7. Details of the T2MOD and T2CON registers are shown in Tables F.8 and F.9 respectively.

Table F.7 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/down depending on T2EX pin
1	1	0	X	16-bit capture
	X	1	X	Baud rate generator

Table F.8 Details of the T2MOD register bit functions

T2MOD
address 419H
bit addressable

MSB							LSB
–	–	RCLK1	TCLK1	–	–	T2OE	DCEN
7	6	5	4	3	2	1	0

Bit	Symbol	Function
7, 6, 3, 2	–	Not implemented, reserved for future use
5	RCLK1	Receive clock flag
4	TCLK1	Transmit clock flag. RCLK1 and TCLK1 are used to select timer 2 overflow rate as a clock source for UART 1 instead of timer 1
1	T2OE	Timer 2 output enable bit
0	DCEN	Down count enable bit. When set, this allows timer 2 to be configured as an up/down counter

Table F.9 Details of the T2CON register bit functions

T2CON
address 418H
bit addressable

MSB							LSB
TF2	EXF2	RCLK0	TCLK0	EXEN2	TR2	C/T2	CP/RL2
7	6	5	4	3	2	1	0

Bit	Symbol	Function
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, TCLK0, RCLK1, TCLK1 or T2OE = 1
6	EXF2	Timer 2 external flag set when either a capture or reload is caused by a negative transition on T2EX and EXEN2 = 1. This flag will cause a timer 2 interrupt when this interrupt is enabled. EXF2 is cleared by software
5	RCLK0	Receive clock flag
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 1
3	EXEN2	Timer 2 external enable flag. Allows a capture or reload to occur as a result of a negative transition on T2EX
2	TR2	Start/stop control for timer 2. A logic 1 starts the timer

Table F.9 Continued

Bit	Symbol	Function
1	C/T2	Timer 2 timer or counter select: 0 = internal timer 1 = external event counter (falling-edge triggered).
0	CP/RL2	Capture/reload flag. If CP/RL2 and 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

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 F.10.

AUTO-RELOAD MODE (UP/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 initialised 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 F.11.

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 SFR (see Table F.7). 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 F.11 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 F.12, 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

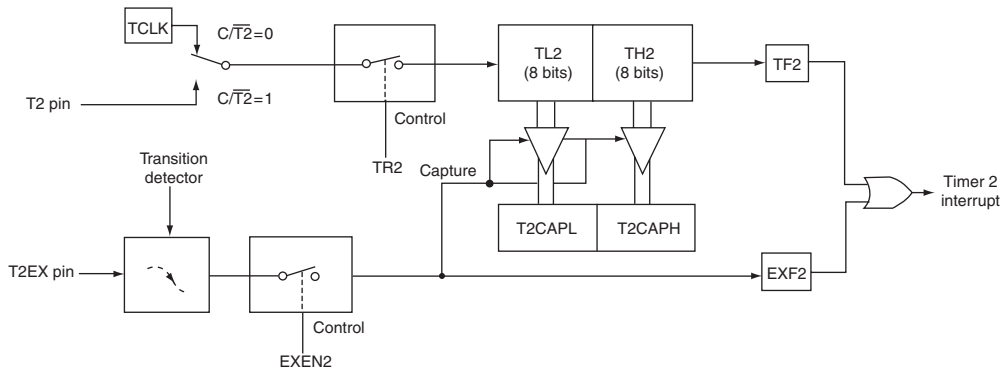


Figure F.10 Timer 2 in capture mode (courtesy Philips Semiconductors)

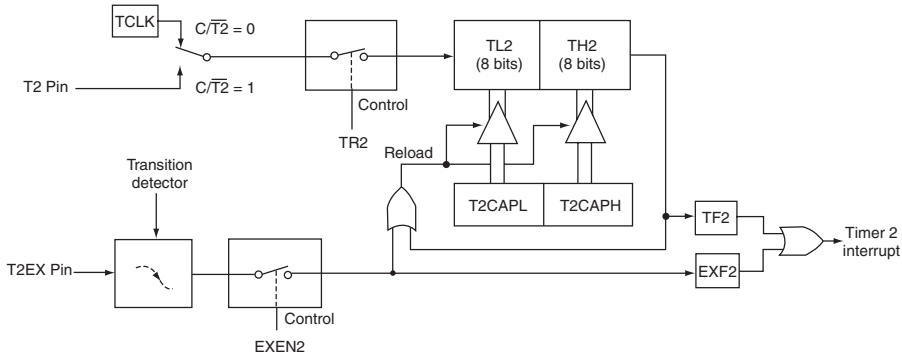


Figure F.11 Timer 2 in auto-reload mode (DCEN = 0) (courtesy Philips Semiconductors)

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 pin P1.6. This pin, besides being a regular I/O pin, has two alternative 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 3.58 Hz to 3.75 MHz at a 30 MHz 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

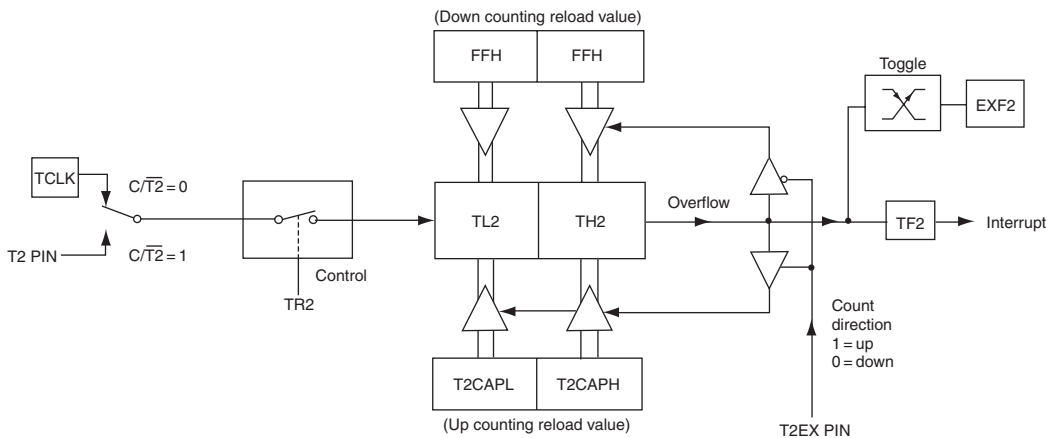


Figure F.12 Timer 2 auto-reload mode (DCEN =1) (courtesy Philips Semiconductors)

frequency and the reload value of timer 2 capture registers (TCAP2H, TCAP2L) as shown in this equation:

$$\frac{\text{TCLK}}{2(65536 - \text{TCAP2H},\ \text{TCAP2L})}$$

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.

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 F.13.

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 32 TCLKs (see Table F.10).

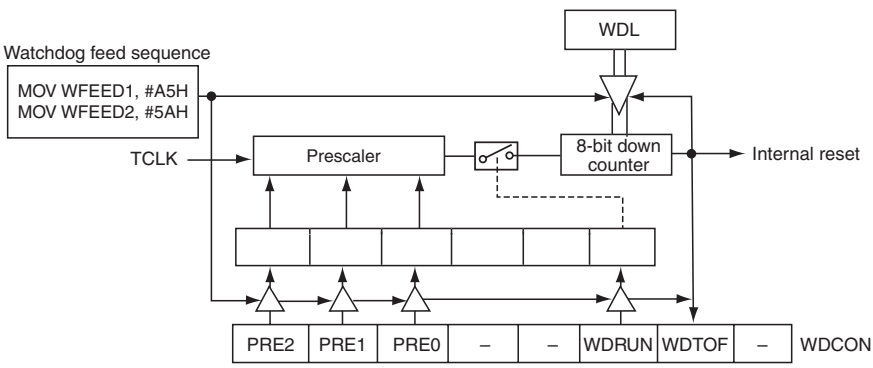


Figure F.13 XAG49 watchdog timer arrangement (courtesy Philips Semiconductors)

Table F.10 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

The watchdog generates an underflow signal (and is auto-loaded 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 auto-load value can range from 0 to FFH. (The auto-load value of 0 is permissible since the prescaler is cleared upon auto-load.) This leads to the following user design equations:

$$t_{\min} = t_{\text{osc}} \times 4 \times 32 \ (W = 0, N = 4)$$
$$t_{\max} = t_{\text{osc}} \times 64 \times 4096 \times 256 \ (W = 255, \ N = 64)$$
$$t_{\text{D}} = t_{\text{osc}} \times \text{N} \times \text{P} \times (W + 1)$$

where t_{osc} is the oscillator period, N is the selected prescaler tap value, W is the main counter auto-load value, P is the prescaler value from Table F.10, $t_{\min}$ is the minimum watchdog time-out value (when the auto-load value is 0), $t_{\max}$ is the maximum time-out value (when the auto-load value is FFH), t_{D} is the design time-out value.

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 were 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 initialisation 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.

Watchdog detailed operation

When external RESET is applied, the following takes place:

- watchdog run control bit set to ON (1)
- auto-load register WDL set to 00 (min. count)
- watchdog time-out flag cleared
- prescaler is cleared
- prescaler tap set to the highest divide
- auto-load takes place.

When coming out of a hardware reset, the software should load the auto-load register and then feed the watchdog (cause an auto-load). 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.

When the watchdog underflows, the following action takes place (see Figure F.13):

- 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 PC 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. The watchdog control register (WDCON) bit definitions are shown in Table F.11.

Table F.11 WDCON register bit definitions

Bit	Symbol	Function
7	PRE2	Prescaler select 2, reset to 1
6	PRE1	Prescaler select 1, reset to 1
5	PREO	Prescaler select 0, reset to 1
4	–	
3	–	
2	WDRUN	Watchdog run control bit, reset to 1
1	WDTOF	Timeout flag
0	–	

F.6 UARTS

The XAG49 includes two 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. The UART transmitter has been double buffered, allowing packed transmission of data with no gaps between bytes and less critical interrupt service routine timing. 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. The double-buffered UART transmitter may require some software changes in code written for the original XAG49 single-buffered UART.

Each UART baud 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 SFR SnBUF. Writing to SnBUF loads the transmit register, and reading SnBUF accesses a physically separate receive register.

The serial port can operate in four 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. Ten 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 SFR SnCON. The baud rate is variable.

Mode 2. Fixed rate 9-bit UART mode. Eleven 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 SFR 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. Eleven 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 SFR SnCON, shown in Table F.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 (Tl_n and Rl_n).

TI flag

In order to allow easy use of the double-buffered UART transmitter feature, the TI_n flag is set by the UART hardware under two conditions. The first condition is the completion of any byte transmission. This occurs at the end of the stop bit in modes 1, 2 or 3, or at the end of the eighth data bit in mode 0. The second condition is when SnBUF is written while the UART transmitter is idle. In this case, the TI_n flag is set in order to indicate that the second UART transmitter buffer is still available.

Table F.12 SnCON register bit functions

SnCON

address: S0CON 420H
S1CON 424H

bit addressable

MSB							LSB
SM0	SM1	SM2	REN	TB8	RB8	TI	RI
7	6	5	4	3	2	1	0

Bit	Symbol	Function
7, 6	SM0, SM1	Specify the serial port mode as follows:
	SM0 SM1	Mode Description Baud rate
	0 0	0 shift register $f_{osc}/16$
	0 1	1 8-bit UART variable
	1 0	2 9-bit UART $f_{osc}/32$
	1 1	3 9-bit UART variable
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 ninth 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
4	REN	Set by software to enable reception. Clear by software to disable reception
3	TB8	The ninth data bit that will be transmitted in modes 2 and 3. Set or cleared by software as desired. The TB8 bit is not double buffered
2	RB8	This is the ninth data bit received in modes 2 and 3. In mode 1, if SM2 = 0, RB8 is the stop bit that was received. In mode 0, RB8 is not used
1	TI	Transmit interrupt flag. Set when another byte may be written to the UART transmitter. Must be cleared by software
0	RI	Receiver interrupt flag. Set by hardware at the end of the eighth 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

Typically, UART transmitters generate one interrupt per byte transmitted. In the case of the XA UART, one additional interrupt is generated as defined by the stated conditions for setting the TI_n flag. This additional interrupt does not occur if double buffering is bypassed as explained below. Note that if a character-oriented approach is used to transmit data through the UART, there could be a second interrupt for each character transmitted, depending

on the timing of the writes to SBUF. For this reason, it is generally better to bypass double buffering when the UART transmitter is used in character-oriented mode. This is also true if the UART is polled rather than interrupt driven, and when transmission is character oriented rather than message or string oriented. The interrupt occurs at the end of the last byte transmitted when the UART becomes idle. Among other things, this allows a program to determine when a message has been transmitted completely. The interrupt service routine should handle this additional interrupt. The recommended method of using the double buffering in the application program is to have the interrupt service routine handle a single byte for each interrupt occurrence. In this manner the program essentially does not require any special considerations for double buffering. Unless higher priority interrupts cause delays in the servicing of the UART transmitter interrupt, the double buffering will result in transmitted bytes being tightly packed with no intervening gaps.

9-bit mode

Note that the ninth data bit (TB8) is not double buffered and care must be taken to ensure that the TB8 bit contains the intended data at the point where it is transmitted. Double buffering of the UART transmitter may be bypassed as a simple means of synchronising TB8 to the rest of the data stream.

Bypassing double buffering

The UART transmitter may be used as if it is single buffered. The recommended UART transmitter interrupt service routine (ISR) technique to bypass double buffering first clears the Tl_n flag upon entry into the ISR, as in standard practice. This clears the interrupt that activated the ISR. Secondly, the Tl_n flag is cleared immediately following each write to SnBUF. This clears the interrupt flag that would otherwise direct the program to write to the second transmitter buffer. If there is any possibility that a higher priority interrupt might become active between the write to SnBUF and the clearing of the Tl_n flag, the interrupt system may have to be temporarily disabled during that sequence by clearing, then setting the EA bit in the IEL register.

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, then since maximum speed of timers/baud clock is $f_{osc}/4$, the maximum baud rate is timer overflow divided by 16 i.e. $f_{osc}/64$. In mode 0, it is fixed at $f_{osc}/16$. In mode 2, however, the fixed rate is $f_{osc}/32$.

The prescaler for timers 0, 1 and 2 is controlled by bits PT1 and PT0 in the SCR register (see Table F.3).
Baud rate for UART mode 0:

$$\text{Baud rate} = \frac{f_{\text{osc}}}{16}$$

Baud rate calculation for UART modes 1 and 3:

$$\text{Baud rate} = \frac{\text{Timer rate}}{16}$$

$$\text{Timer rate} = \frac{f_{\text{osc}}}{N(\text{Timer range} - \text{Timer reload value})}$$

where N = the TCLK prescaler value: 4, 16 or 64 and timer range is equal to 256 for timer 1 in mode 2 and 65536 for timer 1 in mode 0 and timer 2 in count up mode.
The timer reload value may be calculated as follows:

$$\text{Timer reload value} = \text{Timer range} - \left(\frac{f_{\text{osc}}}{\text{Baud rate} \times N \times 16}\right)$$

Notes:

1. The maximum baud rate for a UART in mode 1 or 3 is $f_{\text{osc}}/64$.
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:

$$\text{Baud rate} = \frac{f_{\text{osc}}}{32}$$

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 f_{osc}/N where $N = 4$, 16 or 64 depending on TCLK as programmed in the SCR bits PT1 and PT0. 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. Details of the T2MOD and T2CON registers can be found in Tables F.8 and F.9 respectively.

F.7 Interrupt scheme

There are separate interrupt vectors for each UART'S transmit and receive functions and these are shown in Table F.13.

Table F.13 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 an 80C51 interrupt scheme.

ERROR HANDLING, STATUS FLAGS AND BREAK DETECT

The UARTs in XA have error flags as shown in the serial port extended status register SnSTAT which is described in Table F.14.

MULTIPROCESSOR COMMUNICATIONS AND AUTOMATIC ADDRESS RECOGNITION

This is discussed fully in Appendix D and will not be repeated here.

INTERRUPTS

The XAG49 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:

1. *Exception interrupts* – These are system level errors and other very important occurrences, which include stack overflow, divide-by-0 and reset.
2. *Event interrupts* – These are peripheral interrupts from devices such as UARTs, timers and external interrupt inputs.
3. *Software interrupts* – These are equivalent of hardware interrupt, but are requested only under software control.
4. *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 while event interrupts tend to be different on different XA derivatives.

Table F.14 SnSTAT register bit functions

SnSTAT.
address S0STAT 421H
S1STAT 425H
not bit addressable.

MSB							LSB
–	–	–	–	FEn	BRn	OEn	STINTn
7	6	5	4	3	2	1	0

Bit	Symbol	Function
7, 6, 5, 4	–	Not implemented, reserved for future use
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
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
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
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

The XAG49 supports a total of 9 maskable event interrupt sources (for the various XA peripherals), 7 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 IE bit (in the IEL or IEH registers). Only three bits of the IPA register values are used on the XAG49. 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 XAG49, including all 4 interrupt types, is shown in Table F.15. The table includes the address of the vector for

Table F.15 XA 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(PRI0)	7
Serial port 0 Tx	Tl.0	00A4 – 00A7	ETI0	IPA4.6 – 4(PTI0)	8
Serial port 1 Rx	RI.1	00A8 – 00AB	ERI1	IPA5,2 – 0(PRI1)	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)

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.

Appendix G

P89C66x and XAG49 Microcontroller PCB Board Layouts

Details are given in the introduction to Chapter 2 regarding application notes, produced by Philips Semiconductor engineers, which describe the in-circuit programming of the P89C66x and XAG49 devices; the data included suggested schematic circuits.

The authors adapted the schematic designs to produce PCBs suitable for the P89C664 and XAG49 devices. Each design was based on the 44 pin PLCC package. The schematic circuit diagrams are shown in Chapter 2. Comparison of the schematic circuit diagram for each device shows many similarities but some differences. For example the reset of the P89C664 is active high, same as all the standard 8051 devices, whereas the XA reset is active low; also the XAG49 does not have an I^2C peripheral and so has no need for pull-up resistors on pins 6 and 7 of port 1.

This appendix includes a full-size PCB board layout for both the P89C66x and the XAG49 devices that could be used to produce boards similar to those used by the authors. The design utilises single-sided copper faced PCB material that is readily available from electronic retailers.

G.1 P89C66x board

The artwork for the connection pads and wiring is shown in Figure G.1 while Figure G.2 shows the arrangement for the layout of the components required to complete the circuit. The numbering of the components in Figure G.2 matches the numbering in the schematic circuit diagram of Chapter 2. The latter diagram also indicates the values required for the passive components.

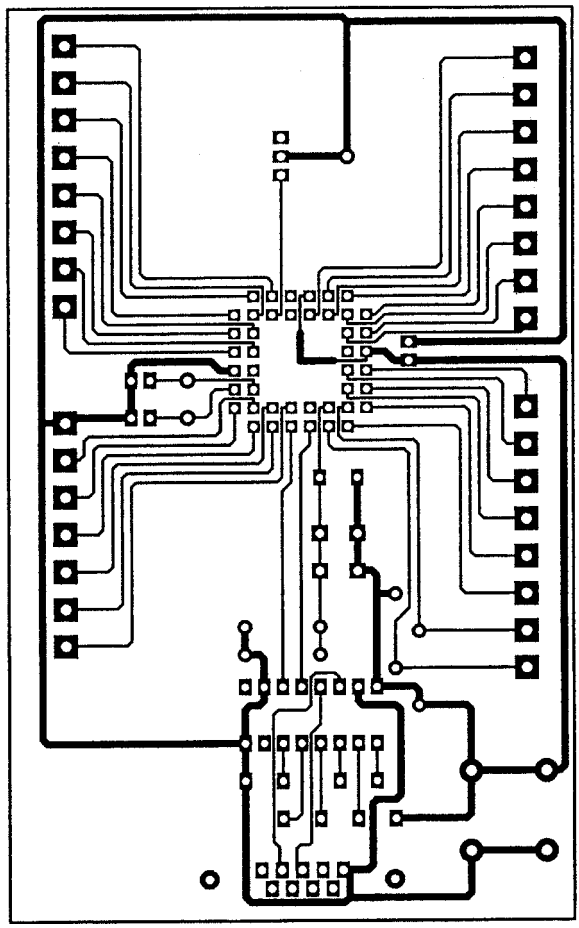


Figure G.1 Full-size single-sided artwork for the P89C66x microcontroller board

G.2 XAG49 board

The artwork for the connection pads and wiring is shown in Figure G.3 while Figure G.4 shows the arrangement for the layout of the components required to complete the circuit. The numbering of the components in G.2 matches the numbering in the schematic circuit diagram of Chapter 2. The latter diagram also indicates the values required for the passive components.

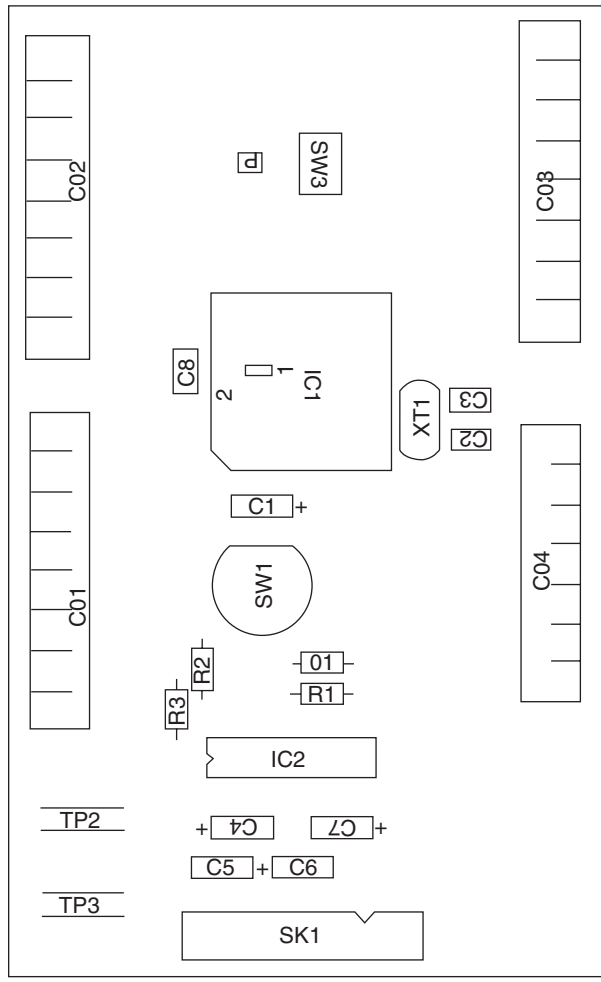


Figure G.2 Component layout for the P89C66x microcontroller board

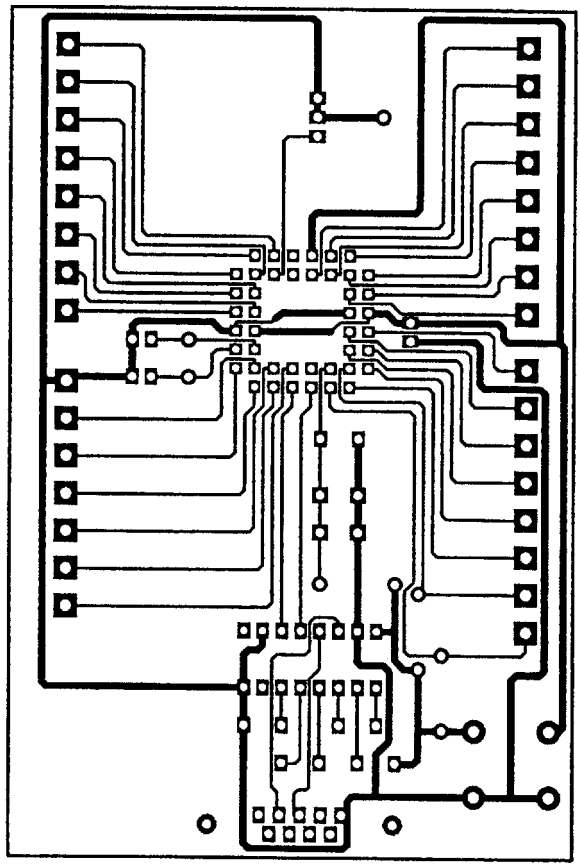


Figure G.3 Full-size single-sided artwork for the XAG49 microcontroller board

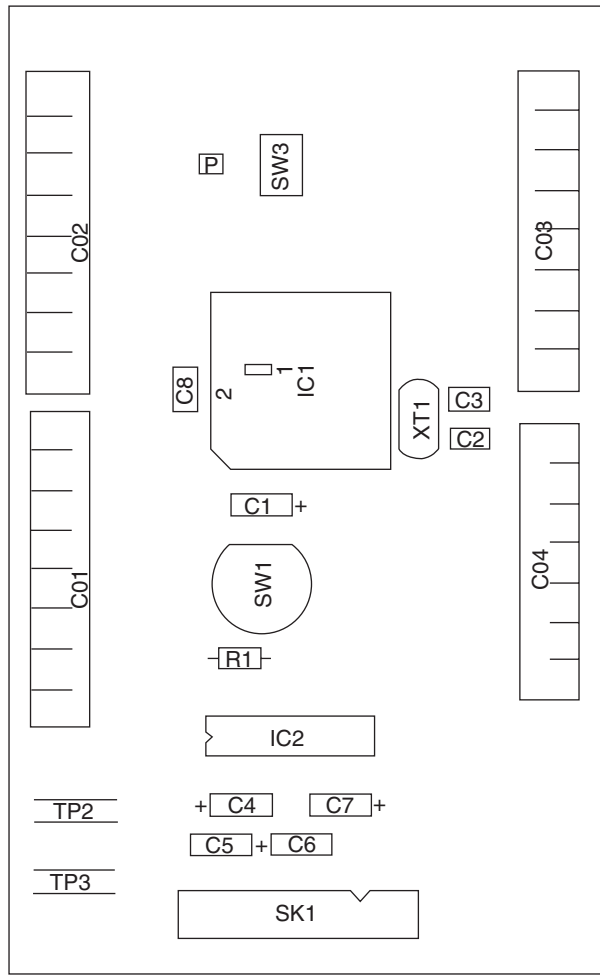


Figure G.4 Component layout for the XAG49 microcontroller board

This page intentionally left blank

Index