PaulOS: Part I

An 8051 Real-Time Operating System

Paul P. Debono

bookboon
The eBook company

Paul P. Debono

PaulOS

An 8051 Real-Time Operating System

Part I

Download free eBooks at bookboon.com

PaulOS: An 8051 Real-Time Operating System
Part I
1st edition
© 2015 Paul P. Debono & bookboon.com
ISBN 978-87-403-0449-7

Download free eBooks at bookboon.com

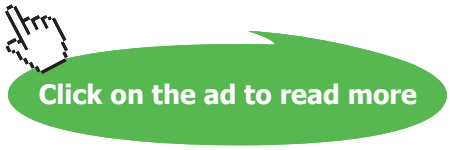
Contents

	Preface	**10**
	Acknowledgements	**13**
	Dedications	**14**
	List of Figures	**15**
	List of Tables	**18**
1	**8051 Basics**	**21**
1.1	Introduction	21
1.2	Memory Types	22
1.3	Code Memory	25
1.4	External RAM	25
1.5	Register Banks	28
1.6	Bit Memory	29

Click on the ad to read more

Download free eBooks at bookboon.com

1.7	Special Function Register (SFR) Memory	31
1.8	SFR Descriptions	34
2	**Basic Registers**	**55**
2.1	The Accumulator, Address E0H, Bit-addressable	55
2.2	The R registers	56
2.3	The B Register, address F0H, Bit-addressable	57
2.4	The Data Pointer (DPTR)	57
2.5	The Program Counter (PC)	57
2.6	The Stack Pointer (SP), address 81H	58
2.7	Addressing Modes	59
2.8	Program Flow	64
2.9	Low-Level Information	68
2.10	Timers	70
2.11	Serial Port Operation	99
2.12	Interrupts	111



Download free eBooks at bookboon.com

3	**A51 Examples**	**128**
3.1	Template.a51	128
3.2	Serial Port Example Program	131
3.3	Traffic Lights A51 Program	135
4	**8032 Differences**	**140**
4.1	8032 Extras	140
4.2	256 Bytes of Internal RAM	141
4.3	Additional Timer 2	143
5	**Evaluation Boards**	**152**
5.1	FLITE-32 Development Board	152
5.2	Typical Settings for KEIL uV2	159
5.3	The NMIY-0031 Board	160
5.4	C8051F020TB	165
6	**Programming in C with KEIL µV2 IDE**	**167**
6.1	Byte Ordering – BIG ENDIAN and LITTLE ENDIAN	168
6.2	Explicitly Declared Memory Types	180

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

Click on the ad to read more

Download free eBooks at bookboon.com

6.3	Data types	180
6.4	Interrupt routines	183
7	**Real-Time Operating System**	**185**
7.1	What is a Real-Time Operating System	185
7.2	Types of RTOSs	187
8	**SanctOS – a Round-Robin RTOS**	**190**
8.1	SanctOS System Commands	190
8.2	Variations from the A51 version	191
8.3	SanctOS example program	194
9	**PaulOS – a Co-operative RTOS**	**200**
9.1	Description of the RTOS Operation	201
9.2	PaulOS.C System Commands	204
9.3	Descriptions of the commands	206
9.4	PaulOS parameters header file	218
9.5	Example using PaulOS RTOS	219

Click on the ad to read more

Download free eBooks at bookboon.com

10	**MagnOS – a Pre-Emptive RTOS**	**224**
10.1	MagnOS System Commands	224
10.2	Detailed description of commands	226
11	**Interfacing**	**245**
11.1	Interfacing add-ons to the 8051	245
11.2	LEDs	246
11.3	Input Switches	257
11.4	Keypad	260
10.5	LCD Display	263
11.6	LCD Command Set	265
11.7	DC Motor	274
11.8	DC motor using H-Bridge	276
11.9	Model Servo Control	284
11.10	Stepper Motor	285
	Index for Part I	**287**
	Index for Part II	**290**

Click on the ad to read more

Download free eBooks at bookboon.com

12	**Programming Tips and Pitfalls**	**Part II**
12.1	RAM size	Part II
12.2	SP setting	Part II
12.3	SFRs	Part II
12.4	Port usage	Part II
12.5	DPTR	Part II
12.6	Serial port (UART)	Part II
12.7	Interrupts	Part II
12.8	RTOSs pitfalls	Part II
12.8	C Tips	Part II
	Appendix A ParrOS.a51	**Part II**
	Appendix B PaulOS A51 version	**Part II**
	Appendix C SanctOS.C	**Part II**
	Appendix D PaulOS.C	**Part II**
	Appendix E MagnOS.C	**Part II**
	Appendix F Further Examples	**Part II**
	Appendix G 8086 PaulOS RTOS	**Part II**
	Appendix H 8051 Instruction Set	**Part II**
	Bibliography	**Part II**
	Index	**Part II**
	End Notes	**Part II**

Download free eBooks at bookboon.com

Preface

This text book is intended to be used either as a stand-alone text for an 8051-based course on micro controllers or as a reference book for those whose work requires familiarity with micro controllers and real-time operating systems.

The strong emphasis of this book is on interfacing and programming the 8051 to typical real-world devices such as switches, displays, motors, and A/D converters through both assembly language and C language programming. In particular, a variety of Real-Time Operating Systems (RTOS) are well explained and working programs are actually implemented (and well documented). Many programming examples, in both assembly language and C, are also included in order to help the students (and anyone interested in the topic) better understand the RTOS principle.

It would be helpful if the reader has already got some familiarity with personal computers and has taken introductory courses in digital devices and some experience with assembly language programming. It is assumed that the reader is familiar with binary and hexadecimal numbers.

Learning to write programs is like learning to ride a bicycle in that reading alone is not enough. Hands-on practical experience is essential. Therefore, to enhance the usefulness of this book as a learning tool, the reader is encouraged to test some of the example programs given throughout this book using easily available free software, such as the latest demo version of the KEIL IDE (http://www.keil.com).

The book is structured into 12 chapters and apendecies with the full source code. A brief outline of the contents of each chapter is given below:

Chapter 1:
This chapter describes the basic 8051 micro-controller and explains its internal organization and uses of the internal special function registers.

Chapter 2:
This chapter deals with the controller addressing mode, interrupts and internal peripherals (timers, serial and parallel input/output ports) of the basic 8051 and goes into more detail on the actual internal special function registers and how they are use in order to program and control the peripherals.

Chapter 3:
In this chapter we present the first simple user progam using assembly language. A template is provided which can be used in other user developed programs.

Download free eBooks at bookboon.com

Chapter 4:
The 8051 is the very basic micro-controller. In this chapter we present one of the first improved versions, namely the 8032/8052 micro-controller, with an enhanced internal memory and with an additional timer. An explanation of the new special function registers associated with the new internal peripheral is also given.

Chapter 5:
Here we discuss just a few of the many development boards which are widely available for the 8051 family of micro-controllers. These evaluation boards can be used to develop and test the program on the actual hardware and are especially useful for students whilst gaining experience on the micro-controller. Actual add-on hardware (such as LCD displays, dc motors, LEDs, keyboards) can also be connected to these boards in order to implement the required project. We discuss and explain the main features of the Flite-32 from Flite Electronics International Limited (http://www.flite.co.uk) 8032 board, the NMIY-0032 8051 board from New Micros, Inc. (http://www.newmicros.com) and the high performance C8051F020TB from Silicon Labs (http://www.silabs.com).

Chapter 6:
This chapter explains the use of the KEIL IDE and how we can set it up to reflect the actual hardware which we intend to use for our particular project. Example programs written in C are given to help the reader grasp the basic principles involved when programming micro-controllers.

Chapter 7:
We now come to the Real-Time Operating System (RTOS) itself and we start by giving the general principles behind the RTOS concept. An explanation of the three main variations of RTOSs which we will deal with is given, namely the round-robin, the co-operative and the pre-emptive versions of the RTOS.

Chapter 8:
This chapter explains the very simple round-robin RTOS called SanctOS, where each task (or function) works for a specified amount of time before passing on the processor time to the next task.

Chapter 9:
The PaulOS co-operative RTOS is described here. This is the 'flagship' RTOS which we regularly use during the year with our students. It is heavily used also for their final year theses and it has been regularly refined to reflect the changes and upgrading requested by the students as they became more and more familiar with the performance and limitations of this co-operative RTOS. In this RTOS, each task is free to run for as long as it wishes. The task itself can control when to give up the processor time to allow other tasks to run.

Download free eBooks at bookboon.com

Chapter 10:
The final RTOS which we discuss is the MagnOS, which gives a demonstration of a pre-emptive RTOS. In this system, each task is given a priority, and the basic control logic of this RTOS is that the highest priority task runs for as long as necessary, until a higher priority task becomes ready to execute. The trick here is to learn to decide what priority to give to the individual tasks so as to avoid having a single task taking over completely the processor time, without giving a chance for other tasks to run.

Chapter 11:
This chapter deals with interfacing various devices to the 8051 family of micro-controllers. The list here is endless but the basic add-ons such as simple LEDs, switches, keypads, LCDs, DC motors (including servos and stepper motors) are all well covered with example programs.

Chapter 12: (Part II)
In this final chapter we discuss some programming tips and common pitfalls which should be avoided when programming such micro-controllers. It would be a good idea to read this chapter first before attempting to write the first program.

Appendices: (Part II) Finally in the appendices we can find program source listings of all the various types of RTOSs found in earlier chapters, both in assembly language and in C language format. A complete list of the basic 8051 intruction set is also given at the end.

Whilst hoping that you will find this book useful, please feel free to contact me on pawlu.debono@yahoo.co.uk if you have any queries or suggestions. A version of PaulOS RTOS for the C8051F020 device is also available and I would be glad to email it to anyone who is genuinely interested.

Download free eBooks at bookboon.com

Acknowledgements

I would like to acknowledge the assistance given by my students who helped me test some of the examples and pointed out some mistakes and omissions.

I am also very grateful for the contributions made by my brother Egidio who tested some of the example programs and to my other brother Albert who proof read the first draft. I would also like to thank my nephew Conrad Micallef for his suggestions and constructive comments.

Finally I am deeply greatful to Prof. Ing. Victor Buttigieg who kindly reviewed the final version of the book and put forward valuable and much appreciated suggestions.

Download free eBooks at bookboon.com

Dedications

To

My wife Maria for being so supportive and patient with me
and my two sons Neil and Luke for their continuous encouragement.

Download free eBooks at bookboon.com

List of Figures

Figure 1-1 Basic 8051
Figure 1-2 Pull-Up resistors
Figure 1-3 8051 Port 0 Structure
Figure 1-4 Setting 8051 Port 0.X as an input pin
Figure 1-5 8051 Port 1 Structure, with internal load
Figure 1-6 Writing '0' to a port
Figure 1-7 Never connect an input port pin directly to Vcc
Figure 1-8 Input switch with no Vcc
Figure 1-9 Input switch with pull-up resistor on Port 0
Figure 1-10 Input switch with pull-resistor on Port 1
Figure 1-11 Buffering input switch connected directly to Vcc
Figure 1-12 Reading the latch

Figure 2-1 Timer/Counter 1 Mode 0 and Mode 1 operation
Figure 2-2 Timer 1 Mode 1
Figure 2-3 Timer 1 Mode 2
Figure 2-4 Timer 0 16-bit pulse-duration mode

Click on the ad to read more

Download free eBooks at bookboon.com

Figure 2-5 Timer 1 16-bit pulse duration mode
Figure 2-6 Master-slaves connection
Figure 2-7 Tri-state buffers

Figure 4-1 Timer 2 Auto-reload Mode
Figure 4-2 Timer 2 in 16-bit capture mode

Figure 5-1 Flite-32 Board
Figure 5-2 NMIY-0031 Board
Figure 5-3 C8051F020 Board

Figure 6-1 The KEIL µV2 environment
Figure 6-2 KEIL µV2 CPU type selection
Figure 6-3 KEIL µV2 Target setup
Figure 6-4 KEIL µV2 Target Output options
Figure 6-5 KEIL µV2 Target listing options
Figure 6-6 KEIL µV2 C51 options

Figure 7-1 RTOS Task states diagram

Figure B-1 Keil Screen shot using PaulOS RTOS

Figure 9-1 RTOS Task states diagram

Figure 11-1 Port Driving LED (pin High = LED on)
Figure 11-2 Port Sinking LED (pin Low = LED on)
Figure 11-3 7-segmnet LED displays
Figure 11-4 Multiplexing displays
Figure 11-5 LED BCD driver
Figure 11-6 Multiplexing 4511s
Figure 11-7 Switch (normally open, high on port pin)
Figure 11-8 Switch (normally open, low on port pin)
Figure 11-9 Switch bounce
Figure 11-10 Keypad switch matrix
Figure 11-11 Interrupt keypad interface
Figure 11-12 Standard LCD connections
Figure 11-13 DC Motor interfacing
Figure 11-14 PWM used to control DC motor speed

Download free eBooks at bookboon.com

Figure 11-15 Motor Off
Figure 11-16 Motor Clockwise Rotation
Figure 11-17 Motor Anti-Clockwise Rotation
Figure 11-18 H-Bridge circuit with discrete devices
Figure 11-19 L293D H-bridge connection
Figure 11-20 RC Servo (www.parallaxinc.com)
Figure 11-21 RC Servo connection
Figure 11-22 Typical Stepper Motors
Figure 11-23 Stepper Motor sequence (zone.ni.com)

Download free eBooks at bookboon.com

List of Tables

Table 1-1 8051 memory space
Table 1-2 8032 memory map (Development System)
Table 1-3 8051 Total Internal RAM organisation
Table 1-4 8051 Internal RAM organisation
Table 1-5 8051 Special Function Registers (SFRs)-DIRECT addressing ONLY
Table 1-6 P0
Table 1-7 P1
Table 1-8 Read-Modify-Write Instructions
Table 1-9 P2
Table 1-10 P3
Table 1-11 PCON
Table 1-12 TCON
Table 1-13 TMOD
Table 1-14 SCON
Table 1-15 IE
Table 1-16 IP
Table 1-17 PSW

Click on the ad to read more

Download free eBooks at bookboon.com

Table 1-18 Register Bank Selection bits
Table 1-19 ACC
Table 1-20 B

Table 2-1 ACC
Table 2-2 B
Table 2-3 Timer-related SFRs
Table 2-4 TMOD (89H) SFR
Table 2-5 Timer Mode Control bits
Table 2-6 Timer counters registers
Table 2-7 TCON (88H) SFR
Table 2-8 TCON (88H) SFR
Table 2-9 SCON (99H) SFR
Table 2-10 Serial Mode selection bits
Table 2-11 Baud Rate calculation
Table 2-12 Crystal Divisor
Table 2-13 Tri-state truth table
Table 2-14 8051 Interrupt Vector Table location
Table 2-15 IE (A8H) SFR
Table 2-16 Polling Sequence Order
Table 2-17 IP (B8H) SFR

Table 4-1 8032 Total Internal RAM organisation
Table 4-2 8032 Internal RAM organisation
Table 4-3 8032 Special Function Registers (SFRs)-DIRECT addressing ONLY
Table 4-4 T2CON (C8H) SFR
Table 4-5 8032 Interrupt Vector Table location

Table 5-1 FLT-32 Memory map
Table 5-2 FLT-32 Interrupt Vector Table
Table 5-3 RAM Size Dec-Hex Conversion
Table 5-4 External Memory (Link Settings)
Table 5-5 NMIY J4 Pinouts
Table 5-6 NMIY J5 Pinouts
Table 5-7 NMIY J6 Pinouts

Table 6-1 8032 Interrupt Vector Table location
Table 6-2 FLITE-32 Interrupt Vector Table location
Table 6-3 Locations of Variables
Table 6-4 C51 compiler data types

Download free eBooks at bookboon.com

Table 8-1 IEMASK Parameter (SanctOS)

Table 9-1 IEMASK Parameter (PaulOS)

Table 10-1 IEMASK values

Table 11-1 LED 7 segment connections
Table 11-2 LCD 8-bit write sequence
Table 11-3 LCD 4-bit write sequence
Table 11-4 LCD Command set
Table 11-5 LCD 8-bit mode initialisation
Table 11-6 LCD 4-bit mode initialisation

Table A-1 PARROS.A51 Variables setup, with 20 tasks. (NOOFTSKS=20)

Table B-2 System Calls without any parameters
Table B-3 System calls needing some parameters
Table B-4 IEMASK parameter
Table B-5 PaulOS.A51 Variables setup, with 18 (12H) tasks. (NOOFTSKS=12H)

Click on the ad to read more

Download free eBooks at bookboon.com

1 8051 Basics

This chapter describes the basic 8051 micro-controller and explains its internal organization and uses of the internal special function registers. Many web pages, books (see bibliography list [1], [2], [3], [8], [9], [13], [14], [15], [18]), and tools are available for the 8051 developer, and many of them are free!. This chapter will assist the reader in mastering basic 8051 programming (using both assembly language and C language) and should eliminate the need to have an additional book specifically on the 8051.

1.1 Introduction

Despite its relatively old age, the 8051 (developed by Intel Corporation in the early 1980s) is one of the most popular micro-controllers in use today. Many derivative micro controllers have since been developed that are based on and compatible with the 8051. Thus, the ability to program an 8051 is an important skill for anyone who plans to develop products that will take advantage of most micro controllers.

The various sections of the first two chapters will explain the 8051micro-controller step by step. The sections in these chapters are targeted at students who are attempting to learn the 8051 assembly language programming and are also useful to those who prefer using C. The appendices are a useful reference tool that will assist both the novice programmer as well as the experienced professional developer, since they provide a wide range of programs complete with source code.

No knowledge of the 8051 is assumed; however, it is assumed some amount of programming has been done before with a basic understanding of the hardware and a firm grasp on the three numbering systems mentioned above. The concept of converting a number from decimal to hexadecimal and/or to binary is not within the scope of this book, and familiarity with these types of conversions would help in understanding some concepts.

This chapter attempts to address the need of the typical programmer. For example, there are certain features that are nifty and in some cases very useful, but 95% of the programmers will never use these features. Those already familiar with the 8051 may skim over some details described in this chapter.

The basic 8051 is a 40-pin IC as shown in Figure 1-1.

Download free eBooks at bookboon.com

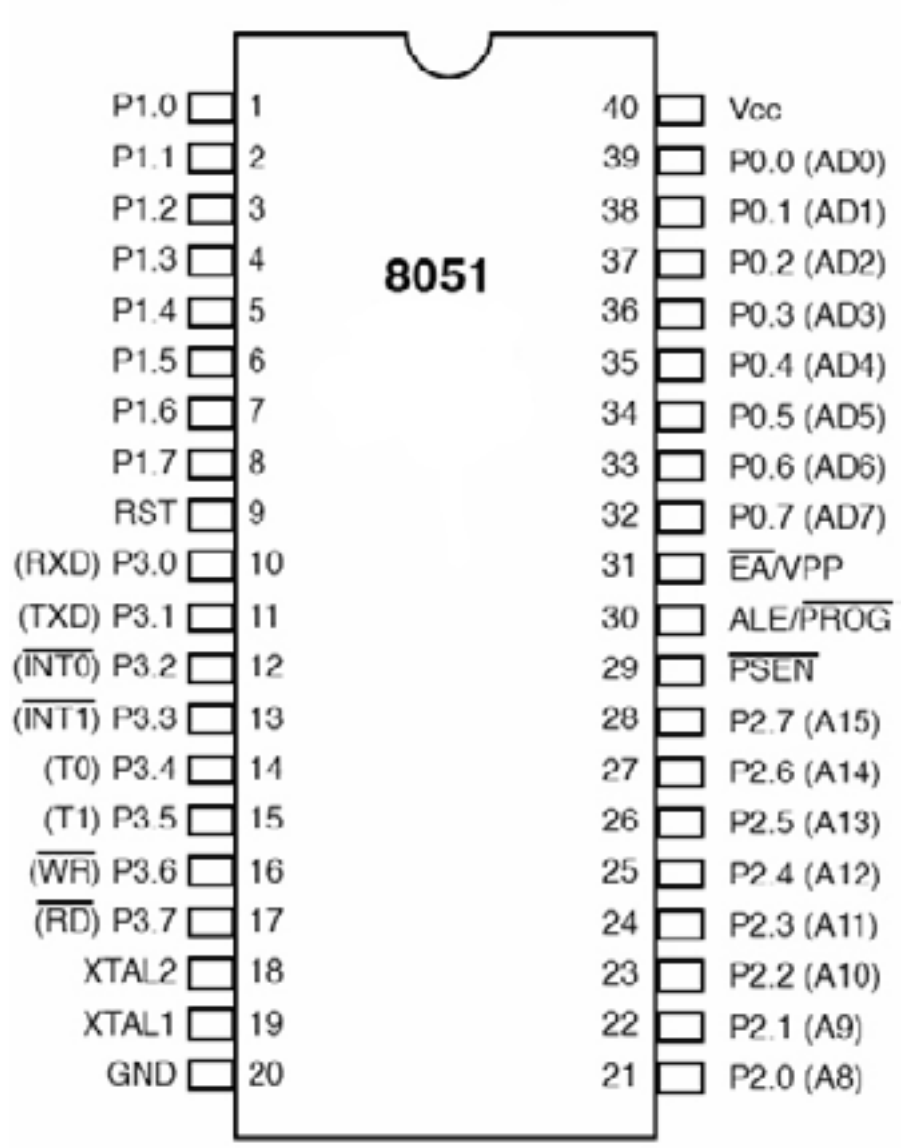


Figure 1-1 Basic 8051

We shall now deal with the internal organisation of the 8051 micro-controller.

1.2 Memory Types

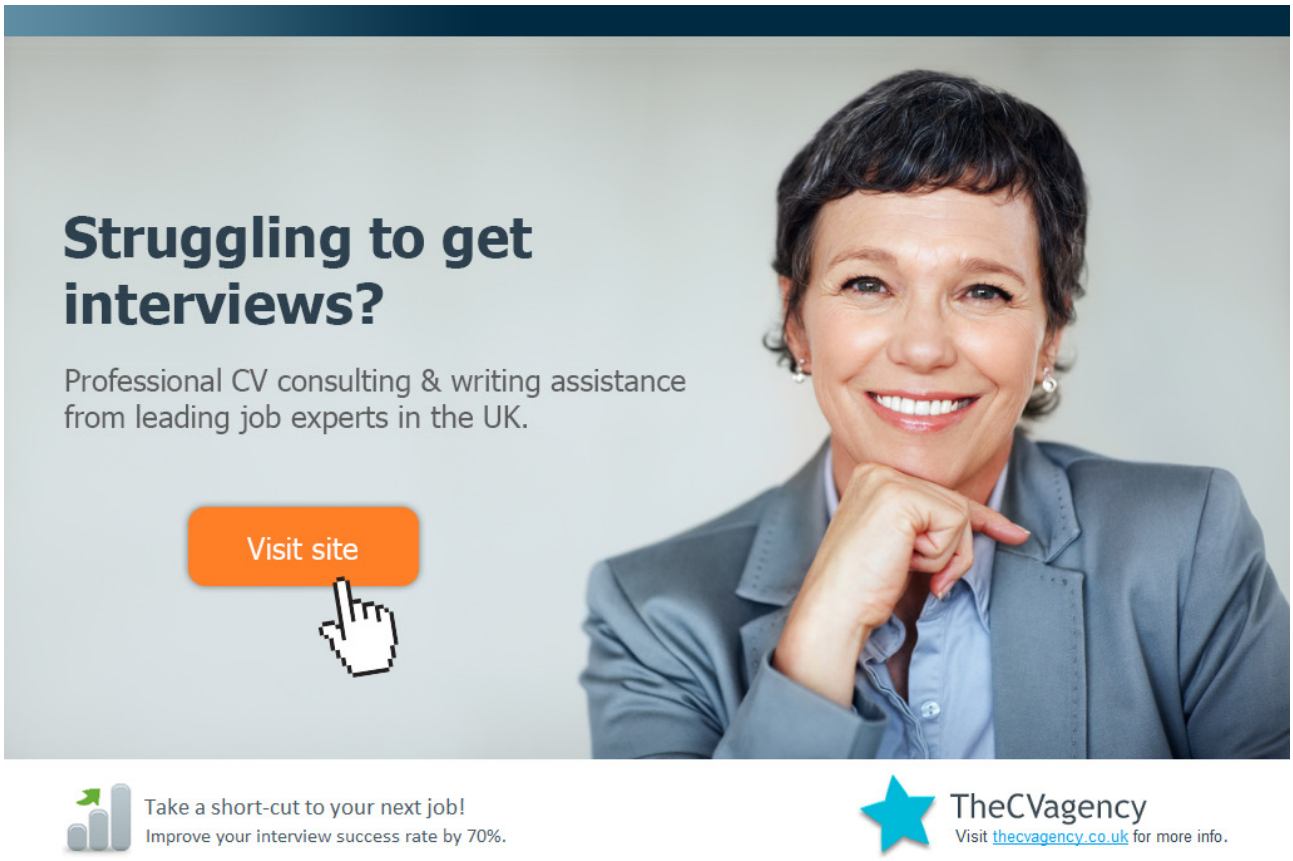
The 8051 has three very general types of memory and each type has to be addressed in a different way. To effectively program the 8051 it is necessary to have a basic understanding of these memory types and how to address them, especially when programming directly in assembly language. The memory types found on the 8051 are illustrated in Table 11 namely the On-Chip Memory, the External Code Memory and External Data RAM. Addresses throughout this book are shown suffixed either with a lower case h (i.e. 0Fh) or with a upper case H (i.e. 0FH) to signify that they are hexadecimal numbers.

Download free eBooks at bookboon.com

Higher memory non existent	FFFFH	FFFFH
0FFH Not Available on basic 8051 80H	External Code Memory	External Data Memory
7FH Internal On-Chip Memory 00H	0000H	0000H

Table 1-1 8051 memory space

It is also very common, especially in many development boards, that the external ram is organised as a contiguous memory map, made up as shown in Table 1-2. Generally, the EEPROM (or ROM) would occupy the lower address area, since the 8051 starts executing instructions from location number 0000H.



Download free eBooks at bookboon.com

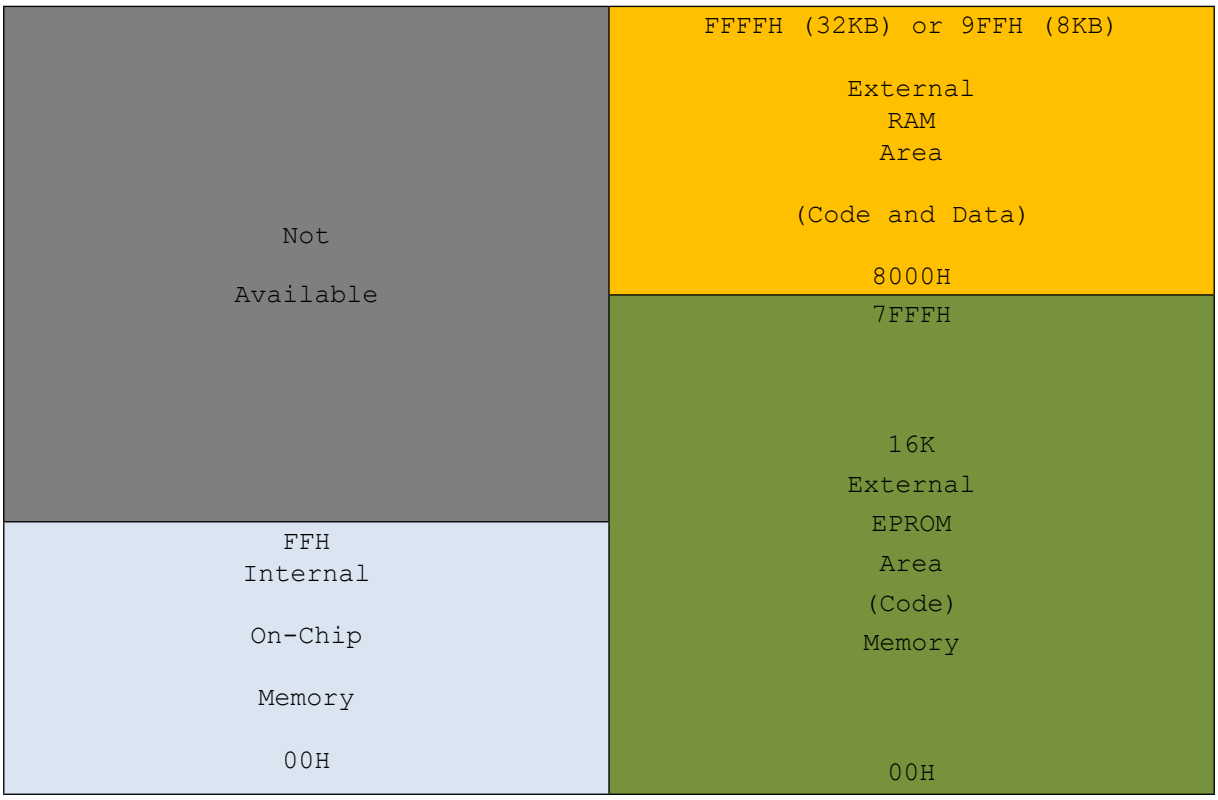


Table 1-2 8032 memory map (Development System)

The EEPROM would generally contain the monitor program so that the user can communicate with the board via the RS232, and also he would be able to transfer his own program into the upper RAM area, where it would be executed for testing and prototyping. The monitor program usually sets the serial port, perhaps under interrupt control (see section 2.9). It would also map the interrupt vector table into the RAM area so that the user application can make use of interrupts by having access to the interrupt vector table. If the interrupt vector table is left in the ROM, the user would not be able to write the address of his Interrupt Service Routines (ISRs) in the EEPROM directly and easily, (he would have to burn a new EEPROM each time! Recent versions with Flash memory have eliminated this problem.). The monitor system must at least have enough commands to be able to transfer and run the program. More commands are usually available depending on the sophistication required. Some have built-in assemblers, dis-assemblers, step-by-step execution and trace facilities for de-bugging purposes.

Most common memory set-ups involve an 8KB or a 16KB EPROM and at least 8KB RAM, both of which can be expanded. The address range would normally be selected by means of shorting some links. The internal memory on chip is only 128 bytes for the normal 8051 (see Table 1-1) but is doubled on the 8032 to 256 bytes as shown in Table 1-2.

On-Chip Memory refers to any memory (Code, RAM, or other) that physically exists on the micro controller itself. On-chip memory can be of several types, but we'll get into that shortly.

External Code Memory is code (or program) memory that resides off-chip. This is often in the form of an external EEPROM.

Download free eBooks at bookboon.com

External RAM is RAM memory that resides off-chip. This is often in the form of standard static RAM or flash RAM.

1.3 Code Memory

Code memory is the memory that holds the actual 8051 program that is to be executed. This memory is normally limited to 64KB although it comes in many shapes and sizes. Since there are many variants of the basic 8051 the Code memory may be found in various forms depending on the device. It can either be burned into the micro-controller as ROM or as an EEPROM and it may also be stored completely off-chip in an external ROM or, more commonly in basic versions, as an external EEPROM. Flash memory is also another popular method of storing a program or code. Various combinations of these memory types may also be used, that is to say, it is possible to have 4KB of code memory on-chip and 64KB of code memory off-chip in an EEPROM.

When the program is stored on-chip, the 64KB maximum value is often reduced to 4KB, 8KB, or 16KB. This varies depending on the version of the micro-controller that is being used. Each version offers specific capabilities and one of the distinguishing factors from one chip to another is how much ROM/EEPROM space the chip has. 64KB and even 128KB flash eprom devices are now available, such as the Silicon Labs C8051F020.

However, code memory is most commonly implemented as off-chip EEPROM, in low-cost development systems and in systems developed by students.

Speeds (and hence performance) are also rapidly increasing with improved architecture and now we have high-speed devices running at 40MHz and using only one clock cycle per instruction instead of the original twelve clock cycles found on the early devices.

1.4 External RAM

As an obvious opposite of Internal RAM, the 8051 also supports what is called External RAM.

As the name suggests, External RAM is any random access memory which is found off-chip. Since the memory is off-chip it is not as flexible in terms of accessing, and is also slower. For example, to increment an Internal RAM location by 1 (such as INC R1) requires only one instruction which is executed in one instruction cycle. To increment a 1-byte value stored in External RAM requires four instructions which are executed in seven instruction cycles. In this case, external memory is seven times slower!

```
MOV DPTR, #address      (2 instruction cycles)
MOVX A, @DPTR           (2 instruction cycles)
INC A                   (1 instruction cycle)
MOVX @DPTR, A           (2 instruction cycles)
```


Download free eBooks at bookboon.com

What External RAM loses in speed and flexibility it gains in quantity. While the Internal RAM is limited to 128 bytes (256 bytes with an 8032/8052), the 8051 supports an External RAM of up to 64KB.

Modern devices now also have this so-called external RAM, physically residing on the same chip, but it is still referred to as external (or XDATA) and all the information listed in this book still holds.

1.4.1 On-Chip Memory

As mentioned at the beginning of this chapter, the 8051 includes a certain amount of on-chip memory. On-chip memory is really one of two types: Internal RAM usually used to store variable and Special Function Register (SFR) memory, used to store the registers which control the built-in peripherals. The layout of the 8051's internal memory is presented in the memory map shown in Table 1-3.

The 8051 has a bank of 128 bytes of Internal RAM. This Internal RAM is found on-chip on the 8051 so it is the fastest RAM available, and it is also the most flexible in terms of reading, writing, and modifying its contents. Internal RAM is volatile, so that when the 8051 is reset this memory is cleared.

Hex Byte Address	Notes		Hex Byte Address
Not Available On the Basic 8051	(8032 ONLY) Accessible By Indirect Addressing only	SFR area Accessible By Direct Addressing only	FFH 80H
7FH Lower 128 bytes 00H	Accessible By Direct And Indirect Addressing.		

Table 1-3 8051 Total Internal RAM organisation

The 128 bytes of internal ram is subdivided as shown on the memory map in Table 1-4. The first eight bytes (00h – 07h) are referred to as register bank 0. By manipulating certain SFR bits (in the PSW special function register), a program may choose to use register banks 1, 2, or 3. These alternative register banks are located in internal RAM, occupying addresses 08h through 1Fh. We will discuss register banks in more detail in section 1.5. For now it is sufficient to know that they are part of the internal RAM.

Bit Memory is also another part of internal RAM, which as the name implies is able to store and manipulate bit variables. We will say more about the bit memory area later (see section 1.6), but for now we just have to keep in mind that the bit memory actually resides in internal RAM, ranging from address 20h through address 2Fh.

Download free eBooks at bookboon.com

The 80 bytes that remain in Internal RAM, from address 30h through address 7Fh, may be used to store any user variables that need to be accessed frequently or at high-speed during the execution of the program. This area is also utilised by the micro-controller as a storage area for the operating stack.

Hex Byte Address	Hex Bit Address								Notes
7FH 30H	Directly and Indirectly Addressable General Purpose RAM								Used as a STACK Area and to store user variables
2FH	7F	7E	7D	7C	7B	7A	79	78	Bit Addressable Section (Bit Addresses shown are in hex)
2EH	77	76	75	74	73	72	71	70	
2DH	6F	6E	6D	6C	6B	6A	69	68	
2CH	67	66	65	64	63	62	61	60	
2BH	5F	5E	5D	5C	5B	5A	59	58	
2AH	57	56	55	54	53	52	51	50	
29H	4F	4E	4D	4C	4B	4A	49	48	
28H	47	46	45	44	43	42	41	40	
27H	3F	3E	3D	3C	3B	3A	39	38	
26H	37	36	35	34	33	32	31	30	
25H	2F	2E	2D	2C	2B	2A	29	28	
24H	27	26	25	24	23	22	21	20	
23H	1F	1E	1D	1C	1B	1A	19	18	
22H	17	16	15	14	13	12	11	10	
21H	0F	0E	0D	0C	0B	0A	09	08	
20H	07	06	05	04	03	02	01	00	
1FH 18H	Register Bank 3 (R0 – R7)								Bank is Selected Using RS0 and RS1 In the PSW Register. See SFRs.
17H 10H	Register Bank 2 (R0 – R7)								
0FH 08H	Register Bank 1 (R0 – R7)								
07H 00H	Register Bank 0 (R0 – R7)								

Table 1-4 8051 Internal RAM organisation

Download free eBooks at bookboon.com

The stack is used to save return addresses when calling functions or subroutines. It is also used to store some values temporarily until they are retrieved again when needed. The fact that the stack size is rather small severely limits the 8051's stack use since, as illustrated in the memory map of Table 1-4, the area reserved for the stack is only 80 bytes, and usually it is effectively a little bit less since these 80 bytes have to be shared between the stack and user variables.

1.5 Register Banks

The 8051 uses eight so-called R registers which are used in many of its instructions. These R registers are numbered from 0 through 7 (R0, R1, R2, R3, R4, R5, R6, and R7) and are generally used to assist in manipulating values and moving data from one memory location to another. For example, to add the value of R4 to the Accumulator, we would execute the following instruction:

```
ADD A,R4
```

Thus if the Accumulator (A) contained the value 6 and R4 contained the value 3, the Accumulator would contain the value 9 after this instruction was executed.

However, as the memory map of Table 1-4 shows, register R4 is really part of Internal RAM. Specifically, R4 (of bank 0) is located at address 04h. Thus the above instruction accomplishes the same thing as the following operation:

Click on the ad to read more

Download free eBooks at bookboon.com

```
ADD A,04h
```

This instruction adds the value found in Internal RAM address 04h (the contents of location 04h) to the value of the Accumulator, leaving the result in the Accumulator. Since R4 is really residing in Internal RAM address 04h, the above instruction has therefore effectively accomplished the same thing as the ADD A,R4 instruction.

But we must be careful since as the memory map shows, the 8051 has four distinct register banks. When the 8051 is first booted up, register bank 0 (addresses 00h through 07h) is used by default. However, our program may instruct the 8051 to use one of the alternate register banks; i.e., register banks 1, 2, or 3. In this case, R4 will no longer be in Internal RAM address 04h but somewhere else. For example, if our program instructs the 8051 to use register bank 3, register R4 will now be located at Internal RAM address 1Ch (see Table 1-4).

The concept of register banks adds a great level of flexibility to the 8051, especially when dealing with interrupts, where we can allocate a specific register bank to a particular interrupt, so as not to corrupt other main program information stored in another bank of registers. (we shall cover interrupts in more detail later, see section 2.9). However we must always remember that the register banks really reside in the first 32 bytes of Internal RAM.

1.6 Bit Memory

The 8051, being a communications-oriented micro-controller, gives the user the ability to access a number of bit variables. These variables may only take the value of either 1 or 0.

There are 128 bit variables available to the user (see Table 1-4), individually numbered 00h through 7Fh. We may make use of these variables with assembly language commands such as *SETB bit address* and *CLR bit address*. For example, to set bit number 24 (hex) to 1 we would execute the instruction:

```
SETB 24h
```

It is important to note that the Bit Memory area is really a part of the Internal RAM. In fact, the 128 bit variables occupy the 16 bytes of Internal RAM from address 20h through address 2Fh. Thus, if we write the value FFh to Internal RAM address 20h we have effectively set bits 00h through 07h to 1 with just one instruction. That is to say that:

```
MOV 20h, #0FFh
```


Download free eBooks at bookboon.com

is equivalent to the following 8 instructions, where we are setting the bits one at a time:

```
SETB 00h
SETB 01h
SETB 02h
SETB 03h
SETB 04h
SETB 05h
SETB 06h
SETB 07h
```

As illustrated in Table 1-4, the bit memory is not a new type of memory but it is just a subset of Internal RAM. Since the 8051 provides special instructions to access these 16 bytes (or 128 bits) of memory on a bit by bit basis it is useful to think of it as a separate type of memory. However, since it is just a subset of Internal RAM then we must remember that any operations performed on the Internal RAM can change the values of these bit variables.

Bit variables 00h through 7Fh are for user-defined variables used in the program. These are not the only bit variables available on the 8051. Other bits in certain Special Function Registers (SFRs) can also be addressed individually as explained in the next section. These bits variables have an address of 80h or higher and are actually used to access certain Special Function Registers (SFRs) on a bit-by-bit basis so as to program and control certain peripherals of the 8051. For example, if output lines P0.0 through P0.7 are all cleared (0) and we want to turn on the P0.0 output line (set bit 0 of port 0 to logic 1) we may either execute:

```
MOV P0,#01h
```

or

```
ORL P0,#01h             ; logically OR P0 with 00000001 binary
```

or

```
SETB 80h
```

or even

```
SETB P0.0               ; the assembler knows that P0.0 = 80h
```


Download free eBooks at bookboon.com

All these instructions listed above accomplish the same thing, although there are some slight differences. Using the SETB or the ORL command will turn on (set to 1) the P0.0 line without affecting the status of any of the other P0 output lines. The MOV command effectively would indeed turn on (1) the P0.0 line but it would also turn off (0) all the other seven output lines (P0.1 to P0.7) which in some cases, may not be what is actually required. Hence caution has to be taken to ensure that we use the correct and most efficient method when setting or clearing bits.

1.7 Special Function Register (SFR) Memory

Special Function Registers (SFRs) reside in areas of internal memory that control specific functionality of the 8051 processor. For example, four SFRs permit access to the 8051's 32 input/output lines. Another SFR allows a program to read or write to the 8051's serial port. Other SFRs allow the user to set the serial baud rate, control and access timers and configure the 8051's interrupt system.

When programming, we may get the illusion that the SFRs are Internal Memory. This is because they are directly addressable. For example, if we want to write the value 1 to Internal RAM location 50 hex we would execute the instruction:

```
MOV 50h, #01h
```

Click on the ad to read more

Download free eBooks at bookboon.com

Similarly, if we want to write the value 1 to the 8051's serial port we would write this value to the SBUF SFR, which has an SFR address of 99 Hex. Thus, to write the value 1 to the serial port we would execute the instruction:

```
MOV 99h,#01h or MOV SBUF,#01h
```

When using this method of memory access (called direct addressing mode), any instruction that has an address of 00h through 7Fh refers to an Internal RAM memory address while any instruction with an address of 80h through FFh refers to an SFR control register.

1.7.1 SFR Addresses

The 8051 is a flexible micro-controller with a relatively large number of modes of operations. In order to be able to make full use of these different modes or ways of using the built in peripherals of this versatile micro-control, our program may inspect and/or change the operating mode of the 8051 by manipulating the values of some specific 8051's SFRs.

They are accessed as if they were normal Internal RAM. The only difference is that Internal RAM for the 8051 resides from address 00h through 7Fh whereas the SFR registers exist in the address range of 80h through FFh. Each SFR has an address (80h through FFh) and a name.

Table 1-5a and 1-5b provide a graphical representation of the 8051's SFRs, their name, and their address in hexadecimal. Although the address range is from 80h through FFh, thus offering 128 possible addresses, there are only 21 SFRs in a standard 8051. The free locations are reserved for future enhanced and upgraded versions of the 8051 family, such as the 8032 discussed in Chapter 4. Moreover, reading data from these empty addresses will in general return some meaningless random data while writing data to these addresses will have no effect. In fact the actual memory cell of these free locations might not be physically present.

Hex Byte Address	Hex Bit Address								Symbol
FF – F9	Not implemented on chip								-
* F8 *	Not implemented on chip								-
F7 – F1	Not implemented on chip								-
* F0 *	F7	F6	F5	F4	F3	F2	F1	F0	B
EF – E9	Not implemented on chip								-
* E8 *	Not implemented on chip								-
E7 – E1	Not implemented on chip								-
* E0 *	E7	E6	E5	E4	E3	E2	E1	E0	ACC
DF – D9	Not implemented on chip								-
* D8 *	Not implemented on chip								-

Table 1-5a 8051 Special Function Registers (SFRs)-DIRECT addressing ONLY

Download free eBooks at bookboon.com

Hex Byte Address	Hex Bit Address								Symbol
D7 – D1	Not implemented on chip								-
* D0 *	D7	D6	D5	D4	D3	D2	D1	D0	PSW
CF – C9	Not implemented on chip								-
* C8 *	Not implemented on chip								-
C7 – C1	Not implemented on chip								-
* C0 *	Not implemented on chip								-
BF – B9	Not implemented on chip								-
* B8 *	-	-	-	BC	BB	BA	B9	B8	IP
B7 – B1	Not implemented on chip								-
* B0 *	B7	B6	B5	B4	B3	B2	B1	B0	P3
AF – A9	Not implemented on chip								-
* A8 *	AF	-	-	AC	AB	AA	A9	A8	IE
A7 – A1	Not implemented on chip								-
* A0 *	A7	A6	A5	A4	A3	A2	A1	A0	P2
9F – 9A	Not implemented on chip								-
99									SBUF
* 98 *	9F	9E	9D	9C	9B	9A	99	98	SCON
97 – 91	Not implemented on chip								-
* 90 *	97	96	95	94	93	92	91	90	P1
8F –8E	Not implemented on chip								-
8D									TH1
8C									TH0
8B									TL1
8A									TL0
89									TMOD
* 88 *	8F	8E	8D	8C	8B	8A	89	88	TCON
87									PCON
86 – 84	Not implemented on chip								-
83									DPH
82									DPL
81									SP
* 80 *	87	86	85	84	83	82	81	80	P0

Hex addresses shown within asterisks are bit-addressable locations.

Table 1-5b 8051 Special Function Registers (SFRs)-DIRECT addressing ONLY

Download free eBooks at bookboon.com

We should therefore stick to the rule that any user developed software should not write anything to these unimplemented locations, since they may be used in future products to invoke new features. All unimplemented addresses in the SFR range (80h through0 FFh) are considered invalid and writing to or reading from these non-existent register locations may produce undefined values or behaviour.

1.7.2 SFR Types

As mentioned in Table 1-5 itself, some SFRs (P0, P1, P2 and P3) are SFRs related to the I/O ports. The 8051 has four I/O ports of 8 bits, for a total of 32 I/O lines. Whether a given I/O line is high or low and the value read from the line are controlled by these SFRs. It should be noted that all of these ports are Bit-addressable. This means that we can read from or write to a single bit of any port.

Some other SFRs are used to control the operation or the configuration of some aspect of the 8051. For example, TCON and TMOD control the timers while SCON controls serial port operations.

The other remaining SFRs can be thought of as auxillary SFRs in the sense that they do not directly configure the 8051 but obviously the 8051 cannot operate without them. For example, once the serial port has been configured using SCON, the program may read or write data characters or bytes to the serial port using the SBUF register.

The SFRs whose address has an asterisk (*) in the Table 1-5 above, are special SFRs that may also be accessed via bit operations (i.e., using the SETB and CLR instructions). The other SFRs cannot be accessed using bit operations but have to be handled using byte operations. As we can see, all SFRs whose addresses are divisible by 8 (having an address ending with a 0h or 8h) can be accessed with bit operations, meaning that they are bit-addressable.

1.8 SFR Descriptions

This section will endeavour to quickly overview each of the standard SFRs found in the above SFR chart map (Table 1-5). It is not the intention of this section to fully explain the functionality of each SFR, as this information will be covered in separate dedicated sections of this chapter.

1.8.1 P0 (Port 0, Address 80h, Bit-Addressable)

All four ports P0, P1, P2 and P3 each use 8 pins, making them 8-bit ports. All the ports upon RESET are configured as output ports. To use any bit of these ports as an input port bit, it must be programmed to do so, by writing a 1 to that particular bit. The operation of the ports is well explained in ([9] Mazidi & Mazidi 2000, pp. 384–390), and is being reproduced with some added comments in the following paragraphs dealing with the ports.

Download free eBooks at bookboon.com

The structure of input/output port 0 or P0, is shown in Figure 1-2. Each bit of this SFR corresponds to one of the port pins on the micro-controller. For example, bit 0 of port 0 is pin P0.0, bit 7 is pin P0.7. Writing a value of 1 to a bit (SETB P0.7) of this SFR will send a high level on the corresponding I/O pin whereas writing a value of 0 (CLR P0.7) will bring it to a low level. If used as an input, the status of a bit can be checked by the program by using for example:

```
JB P0.7,Label ; (Jump to Label if bit P0.7 is 1).
```

or in C, assuming that Port0_bit7 was declared by ujsing the **sbit** bit variable declaration

```
sbit Port0_bit7 = P0^7;
```

then we can write

……

```
if (Port0_bit7 == 1) { …… }
```

or

```
if (Port0_bit7) { …… }
```

Click on the ad to read more

Download free eBooks at bookboon.com

The **sbit** variable declaration enables us to gain access to the port pin by giving the pin a name. The caret symbol (^) is used in C instead of the dot (.) when referring to bits, since the dot is used in C when referring to union members. Although the caret symbol is also used in C for the bitwise XOR (exclusive OR) operator, no XOR operation is involved here. The sbit keyword helps the compiler to sort this out.

To use the pins of port 0 as both input and output ports, each pin must be connected externally to a +5V rail via a 10k ohm pull-up resistor. This is due to the fact that P0 uses an open drain configuration, unlike P1, P2 and P3. *Open drain* is a term used for MOS chips in the same way that an *open collector* is used for TTL chips. With external pull-up resistors connected, upon reset port 0 is configured as an output port (default mode). This setup is also shown in Figure 1-2 with the pull-up resistor shown in a shaded box to highlight the fact that this is an additional external connection.

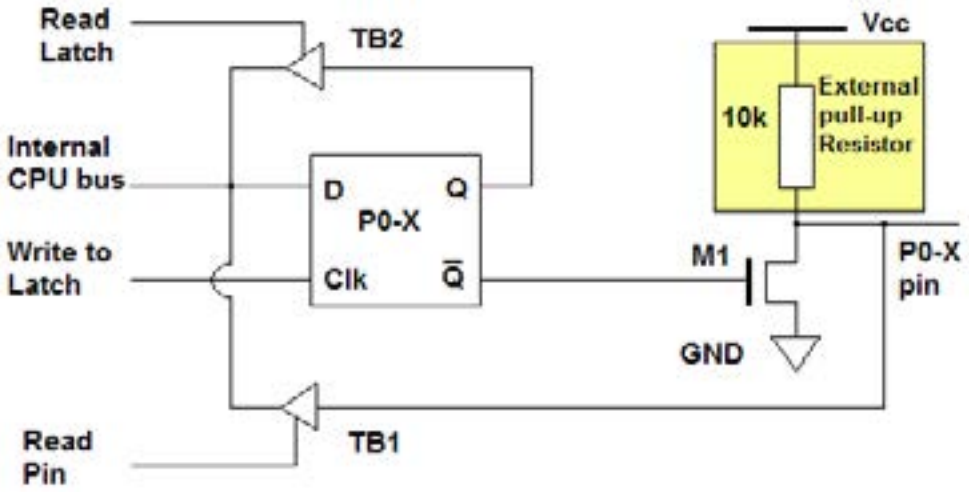


Figure 1-2 Pull-Up resistors

With resistors connected, in order to make it an input port, the port must first be programmed to the input mopde. This is achieved by writing a 1 to all the bits required to act as an input. For example to make all port 0 act as an input port, we must first use:

```
MOV P0, #0FFH
```

or in C

```
P0 = 0xFF;
```

And then we can read data from the port into the accumulator by using

```
MOV A, P0
```

or in C, assuming Inputdata was previously declared as an 8-bit variable

Download free eBooks at bookboon.com

```
Inputdata = P0;
```

Hex Byte Address	Bit-addressable								Symbol
80	87	86	85	84	83	82	81	80	P0
	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0	Bit – ASM
	P0^7	P0^6	P0^5	P0^4	P0^3	P0^2	P0^1	P0^0	Bit – KEIL C

Table 1-6 P0

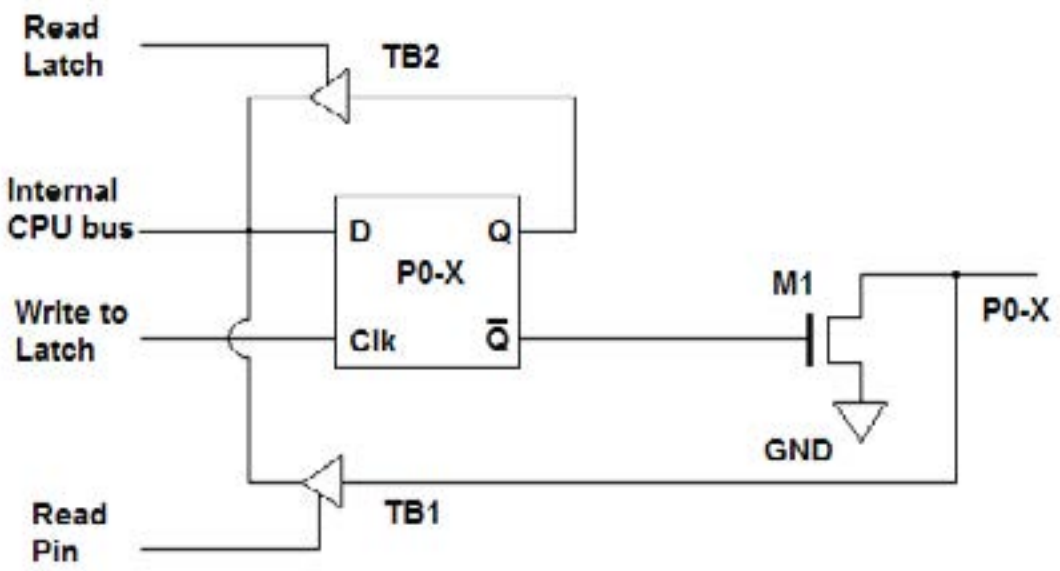


Figure 1-3 8051 Port 0 Structure

Port 0 has a dual role, allowing it to be used for both address and data transfers. When connecting the 8051 to an external memory, port 0 provides both the address and the data signals. The 8051 multiplexes address and data functions through port 0 in order to save on the number of pins on the IC, the Address Latch Enable (ALE) pin providing the necessary control function. If ALE = 0, port 0 provides data and when ALE = 1 it carries address bits A0 to A7.

Since all the ports of the 8051 are bi-directional, they all have the following three basic components:

- D latch
- Output driver
- Input buffer

Download free eBooks at bookboon.com

Now the question arises when reading the port, are we reading the status of the input pin or are we reading the status of the latch? That is an extremely important question and its answer depends on the type of instruction we are using to address the port. The instruction itself would dictate which tri-state input buffer is to be activated, whether TB1(pin) or TB2 (latch). The explanation is given in section 1.8.5.

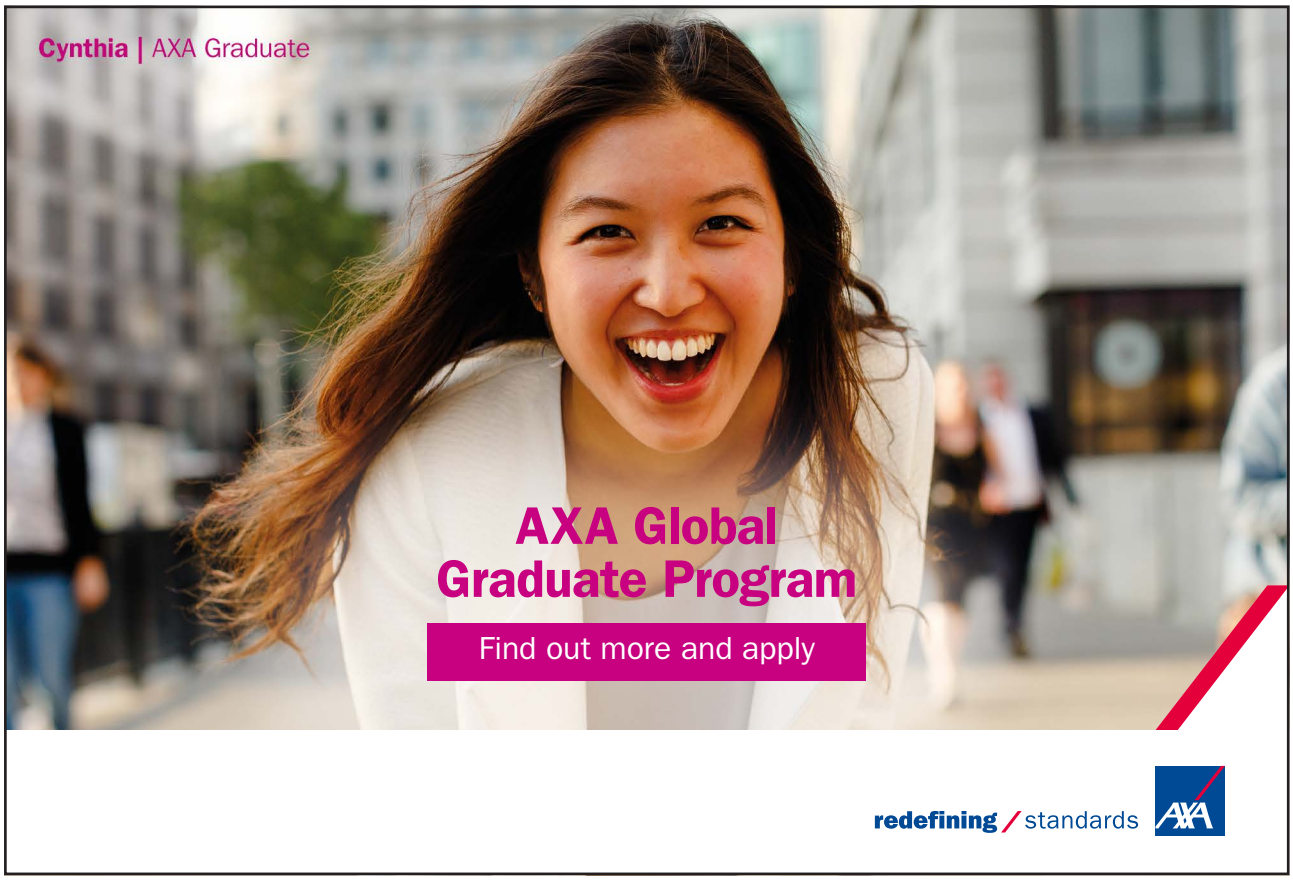
1.8.2 Reading the input pin

As stated earlier, in order to make any bit of any port of the 8051 an input port, we first must write a 1 (logic high) to that port bit. With reference to Figure 1-4, since we have chosen port 0, the load R1 would be an externally connected pull-up resistor of say 10kΩ (shown in a shaded box to denote an additional external component).

Writing a 1 to the port bit, causes a 1 to be written to the latch and the D latch therefore has a logic high on its pin. Therefore Q = 1 and $\overline{Q} = 0$.

Consequently the transistor M1 gate is 0 or at a low level and the transistor is therefore turned off.

M1 therefore blocks the path to ground for any signal connected to the input pin P0.X and the input signal is therefore directed to the tri-state buffer TB1.



Click on the ad to read more

Download free eBooks at bookboon.com

When reading the input port with an instruction such as using MOV A, P1 we are therefore actually reading directly the data present at the pin since this instruction activates the read pin of TB1 and lets the data flow into the CPU's internal bus.

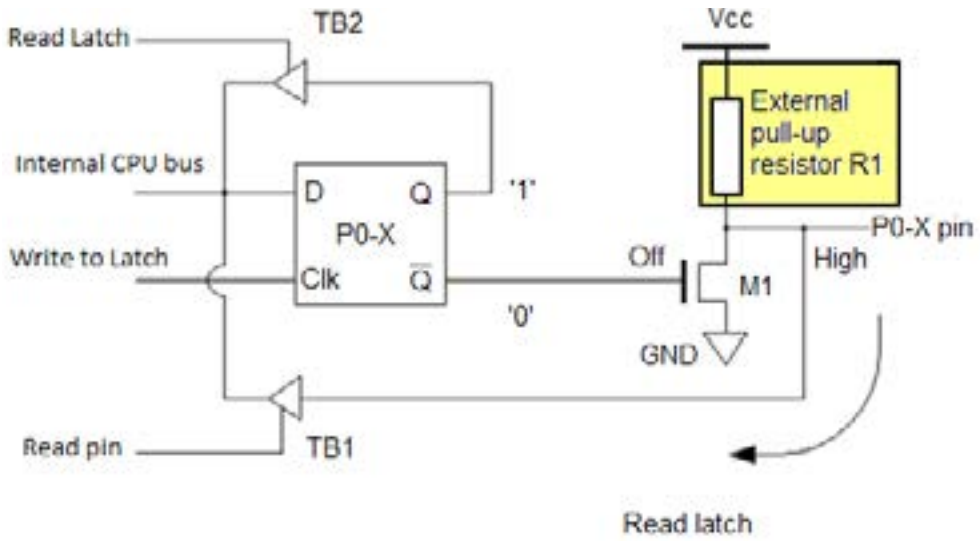


Figure 1-4 Setting 8051 Port 0.X as an input pin

Writing to a pin which was previously set to the input mode can have serious repercussions on the port and this is dealt with in section 1.8.4 where we explain how we can damage the port. This particular situation can easily occur if we are not careful when programming the device.

1.8.3 P1 (Port 1, Address 90h, Bit-Addressable)

This is input/output port 1 as shown in Figure 1-5. Each bit of this SFR corresponds to one of the pins on the micro-controller. For example, bit 0 of port 1 is pin P1.0, bit 7 is pin P1.7. Writing a value of 1 to a bit (SETB P1.7) of this SFR will send a high logic level (say 3.5–5 V) on the corresponding I/O pin whereas a value of 0 (CLR P1.7) will bring it to a low logic level (0V). If used as an input, the status of a bit can be checked by the program by using for example

```
JB P1.7,Label ;(Jump to Label if bit P1.7 is 1).
```

or in C, again assuming the sbit variable declaration

```
sbit Port1_bit7 = P1^7;
```

….

```
If (Port1_bit7 == 1) { … … }
```

As seen in Figure 1-5, in contrast to port 0, this port does not need any pull-up resistors since they are already built-in internally. However, once again as in port 0, in order to make it an input port, the port must first be programmed by writing a 1 to all the bits required to act as an input.

Download free eBooks at bookboon.com

Hex Byte Address	Bit-addressable								Symbol
90	97	96	95	94	93	92	91	90	P1
	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0	Bit – ASM
	P1^7	P1^6	P1^5	P1^4	P1^3	P1^2	P1^1	P1^0	Bit – KEIL C

Table 1-7 P1

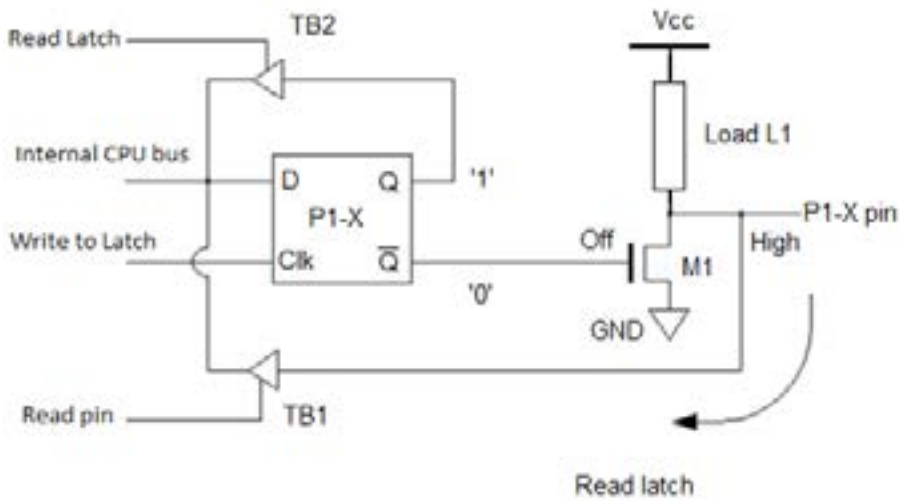


Figure 1-5 8051 Port 1 Structure, with internal load

1.8.4 Damaging the port

Looking at Figure 1-6 we can see that if we write a 0 (low) to a port bit, then $Q = 0$ and $\overline{Q} = 1$. As a result transistor M1 is now ON and therefore provides a path to ground for load L1 and the pin P1.X is effectively grounded.

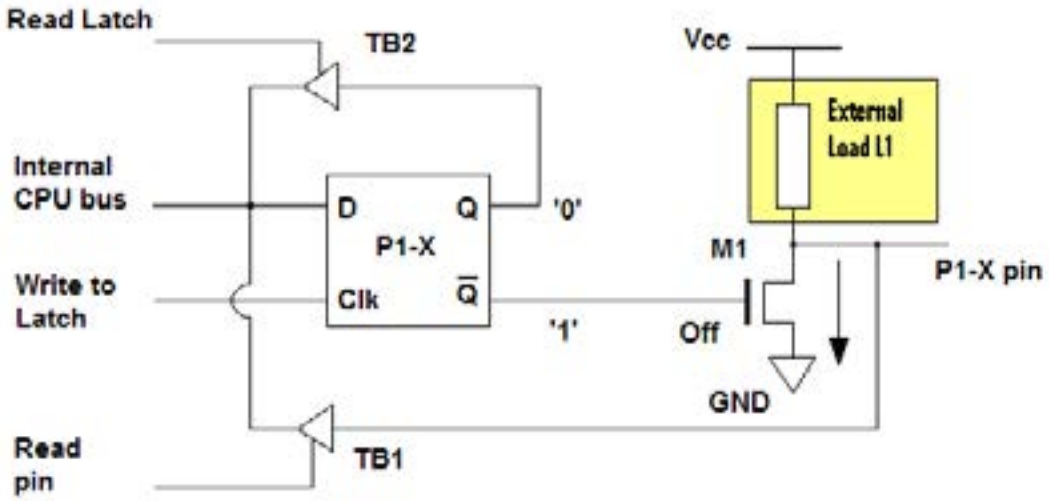


Figure 1-6 Writing '0' to a port

Download free eBooks at bookboon.com

This is normal and correct, since when we write a zero to an output port, we expect to have 0V at the output pin. If however the port was originally intended to be used as an input port and we had an external connection as shown in Figure 1-7 the effect of inadvertently writing a '0' to an input configured port could have a very damaging effect. With the transistor switched on and if a two-way switch between supply Vcc and ground is connected directly to the pin as shown, then the transistor will sink current from both the internal load L1 and the external Vcc via the switch. This will cause too much current to flow in M1 and thus damaging permanently the port bit. In order to avoid damaging the port even if we use the wrong instruction by mistake, the correct kind of connection should be used when using switches or when supplying signals to an input port.

Some examples of the correct type of connection are shown in Figure 1-8 to Figure 1-11.

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

Click on the ad to read more

Download free eBooks at bookboon.com

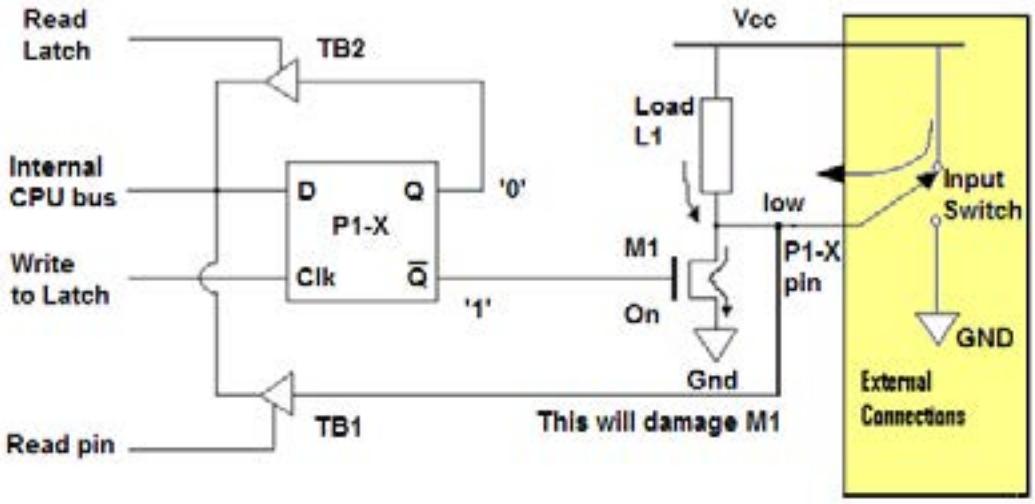


Figure 1-7 Never connect an input port pin directly to Vcc

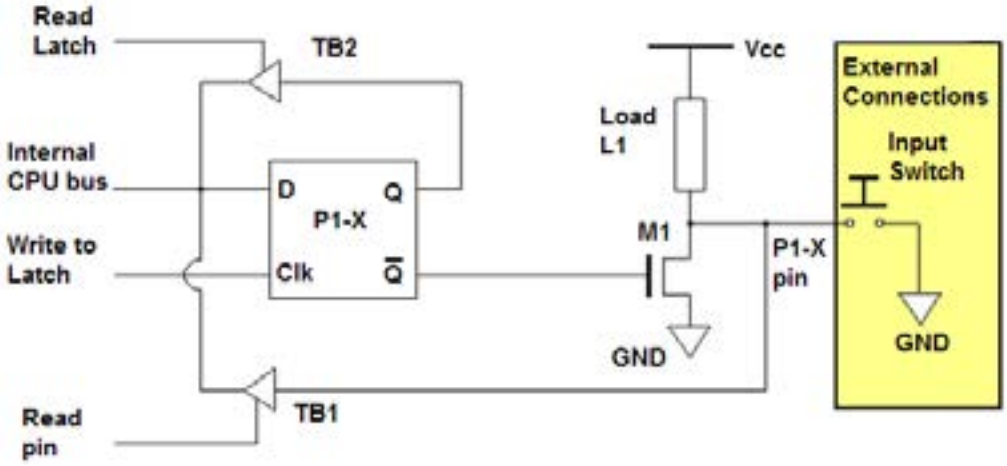


Figure 1-8 Input switch with no Vcc

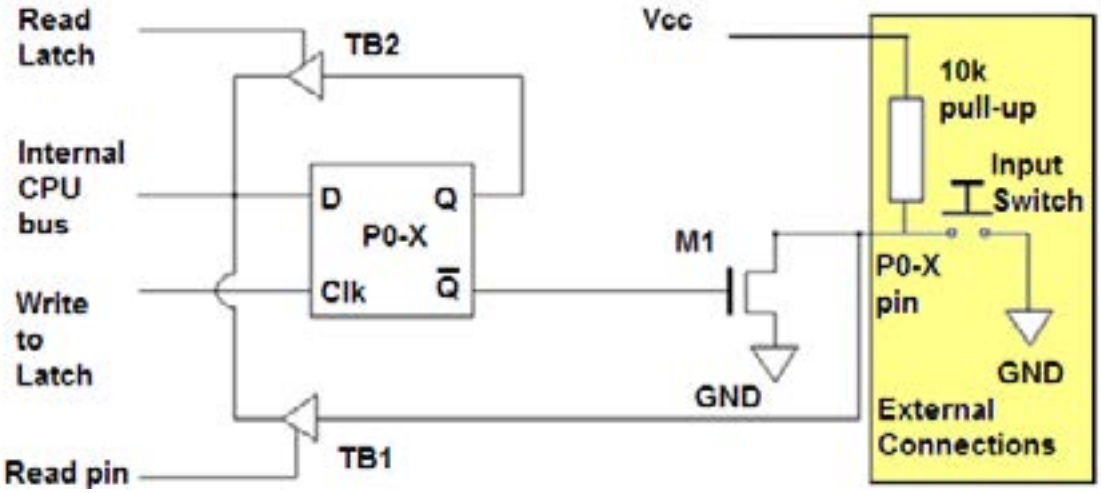


Figure 1-9 Input switch with pull-up resistor on Port 0

Download free eBooks at bookboon.com

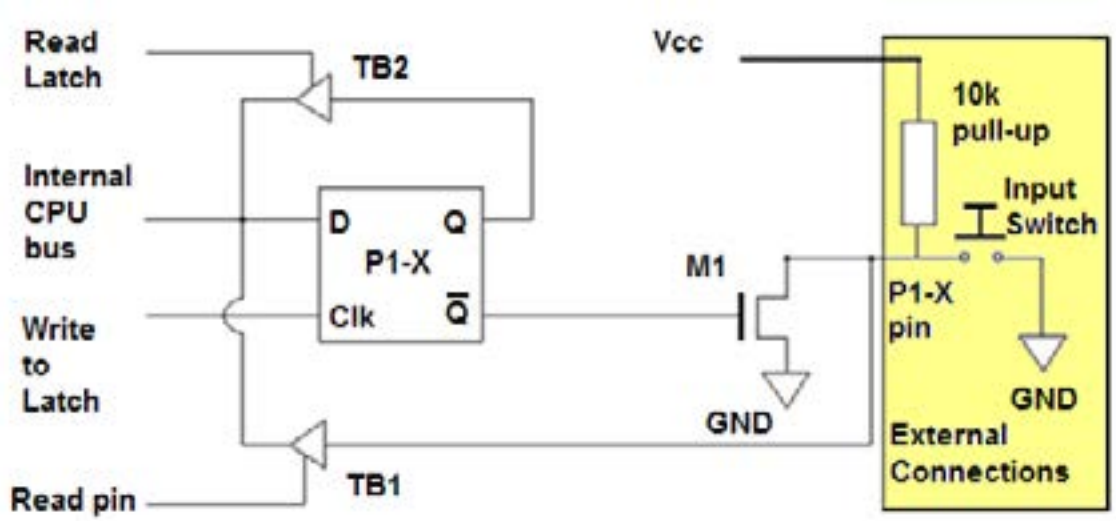


Figure 1-10 Input switch with pull-resistor on Port 1

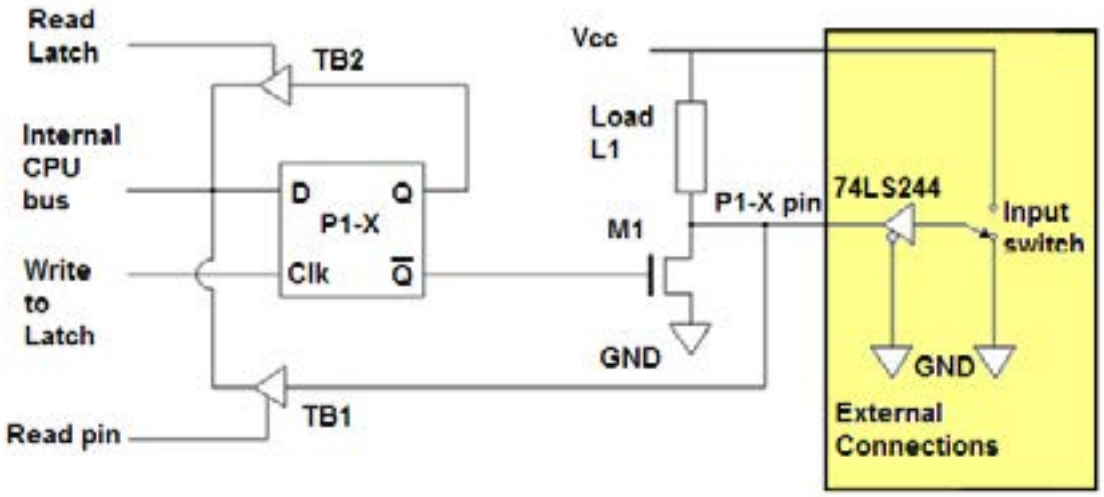


Figure 1-11 Buffering input switch connected directly to Vcc

1.8.5 Reading the latch

In reading the port, (see Figure 1-12) we may be reading the latch instead of the actual port pin. Consider the case of the logical AND instruction

```
ANL P1,A
```


Download free eBooks at bookboon.com

which is actually a READ-MODIFY-WRITE instruction. This is typical for **bit-wise** operations such as ANL, ORL and XRL. There are usually no side-effects of READ-MODIFY-WRITE instructions when accessing registers that have the same values when read or when written (because they act like RAM). However, READ-MODIFY-WRITE instructions can cause problems when the register being accessed is write-only or reads a different value than what was written.

A characteristic of the I/O ports of the 8051 is that the value you write may not be the value you read (since reading the port returns the state of the port pins). However, these registers are specially treated for READ-MODIFY-WRITE instructions.

Looking at the sequence of actions taking place when this instruction (ANL P1,A) is executed, we shall see exactly why and what we are reading:

- The read latch in this case activates the tri-state buffer TB2 and brings the data from the Q latch (not the input pins via TB1 as in previous examples) into the CPU.
- This data is ANDed with the contents of register A.
- The result is re-written to the latch.

Click on the ad to read more

Download free eBooks at bookboon.com

After re-writing the result to the latch there are two possibilities:

If $Q = 0$, then $\overline{Q} = 1$ and M1 is ON. The output pin has 0, the same as the status of the Q latch.

If $Q = 1$, then $\overline{Q} = 0$ and M1 is OFF. The output pin has a 1, the same as the status of the Q latch.

From the above discussion, we conclude that the instruction that reads the latch normally reads a value, performs an operation (possibly changing the value), and re-writes it to the latch. Hence this is often called a READ-MODIFY-WRITE instruction. Table 1-8 provides a list of examples for such instructions, which ALL use the port as the destination operand.

- ANL P1,A
- ORL P1,A
- XRL P1,A
- JBC P1.1, LABEL
- CPL P1.2
- INC P1
- DEC P1
- DJNZ P1, LABEL
- MOV P1.2,C
- CLR P1.3
- SETB P1.4

Table 1-8 Read-Modify-Write Instructions

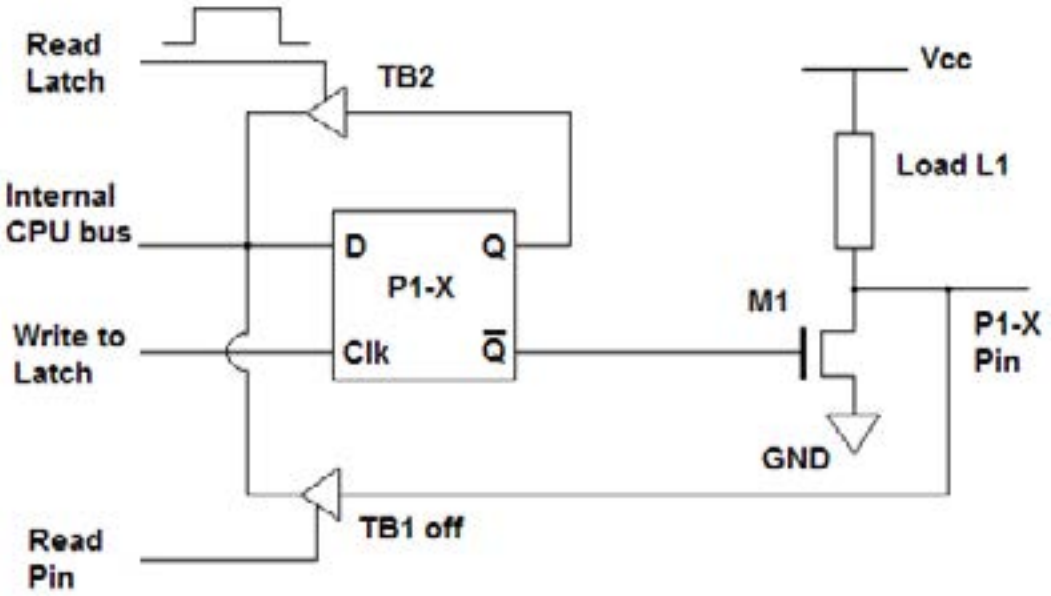


Figure 1-12 Reading the latch

Download free eBooks at bookboon.com

1.8.6 P2 (Port 2, Address A0h, Bit-addressable)

This is input/output port 2. Each bit of this SFR corresponds to one of the pins on the micro-controller. For example, bit 0 of port 2 is pin P2.0, bit 7 is pin P2.7. Writing a value of 1 to a bit (SETB P2.7) of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 (CLR P2.7) will bring it to a low level. If used as an input, the status of a bit can be checked by the program by using for example:

```
JB P2.7, Label ; (Jump to Label if bit P2.7 is 1).
```

or in C, with a previous sbit declaration of the variable Port2_bit

```
if (Port2_bit7) { ……}
```

Same as port 1, this port does not need any pull-up resistors since they are already built-in internally. Also as in port 1, in order to make it an input port, the port must first be programmed by writing a 1 to all the bits required to act as an input.

Hex Byte Address	Bit-addressable								Symbol
A0	A7	A6	A5	A4	A3	A2	A1	A0	P2
	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0	Bit – ASM
	P2^7	P2^6	P2^5	P2^4	P2^3	P2^2	P2^1	P2^0	Bit – KEIL C

Table 1-9 P2

Port 2 as port 0, also has a dual role, allowing it to be used to provide the higher 8-bit address when connecting the 8051 to external memory. Used in conjunction with port 0 provides the address bits A8 to A15 thus making the 8051 capable of addressing up to 64KB (16-bit) of external memory.

1.8.7 P3 (Port 3, Address B0h, Bit-addressable)

This is input/output port 3 and each bit of this SFR corresponds to one of the pins on the micro-controller. For example, bit 0 of port 3 is pin P3.0, bit 7 is pin P3.7.

Hex Byte Address	Bit-addressable								Symbol
B0	B7	B6	B5	B4	B3	B2	B1	B0	P3
	RD	WR	T1	T0	INT1	INT0	TXD	RXD	Other use
	P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0	Bit – ASM
	P3^7	P3^6	P3^5	P3^4	P3^3	P3^2	P3^1	P3^0	Bit – KEIL C

Table 1-10 P3

Download free eBooks at bookboon.com

Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas writing a value of 0 will bring it down to a low level.

P3 is also used for interrupts as well as other signals as shown in Table 1-10. Port 3 too, does not need any pull-up resistors, same as P1 and P2. Although port 3 is configured as an output port upon reset, this is not the way it is commonly used.

- Bits 0 and 1 are used for the RxD (input data) and TxD (output data) serial communications signals.
- Bits 2 and 3 are set aside for external interrupt input signals.
- Bits 4 and 5 can be used as input signals for the timers and
- Bits 6 and 7 can provide the write and read signals for any external memories connected to the 8051.

Thus P3 has some pins dedicated for specific jobs which restrict its use for other purposes.

Click on the ad to read more

Download free eBooks at bookboon.com

1.8.8 SP (Stack Pointer, Address 81h)

This is the stack pointer of the micro-controller. This SFR indicates where the next value to be taken from the stack will be read from in Internal RAM. If we push a value onto the stack, the value will be written to the address of SP + 1. That is to say, if SP holds the value 07h (this is the default reset value), a PUSH instruction will push the value onto the stack at address 08h. This SFR is modified by all instructions which modify the stack, such as PUSH, POP, LCALL, RET, RETI, and whenever interrupts are provoked by the micro-controller.

1.8.9 DPL/DPH (Data Pointer Low/High, Addresses 82h/83h)

The SFRs DPL and DPH work together to represent a 16-bit value called the Data Pointer (DPTR) and is used in operations regarding external RAM and some instructions involving code memory. Having 16-bits it can represent values from 0000h to FFFFh (0 through 65,535 decimal).

1.8.10 PCON (Power Control, Address 87h)

The Power Control SFR is used to control the 8051's power control modes. Certain operating modes of the 8051 allow the 8051 to go into a sort of sleep mode which requires much less power. These modes of operation are controlled through specific bits in PCON.Additionally, one of the bits in PCON (PCON.7 also known as SMOD) is used to double the effective baud rate of the 8051's serial port. Other bits are not implemented (-). Note that this SFR is not Bit-addressable, and hence in order to set SMOD to 1, without altering the other bits in the SFR, we should use:

```
ORL PCON, #80h
```

or in C

```
PCON |= 0x80;
```

Hex Byte Address	Not Bit-addressable								Symbol
	7	6	5	4	3	2	1	0	
87	-	-	-	-	-	-	-	-	PCON
	SMOD	-	-	-	GF1	GF2	PD	IDL	Bit

Table 1-11 PCON

1.8.11 TCON (Timer Control, Addresses 88h, Bit-addressable)

The Timer Control (TCON) SFR is used to configure and modify the way in which the 8051's two timers operate.

Download free eBooks at bookboon.com

Hex Byte Address	Bit-addressable								Symbol
88	8F	8E	8D	8C	8B	8A	89	88	TCON
	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	Bit Symbol
	TCON.7	TCON.6	TCON.5	TCON.4	TCON.3	TCON.2	TCON.1	TCON.0	Bit – ASM
	TCON^7	TCON^6	TCON^5	TCON^4	TCON^3	TCON^2	TCON^1	TCON^0	Bit – KEIL C

Table 1-12 TCON

This SFR controls whether each of the two timers is running or stopped and contains a flag to indicate that the timer has overflowed. Additionally, some non-timer related bits are also located in the TCON SFR. These bits are used to configure the way in which the external interrupts are activated and also contain the external interrupt flags which are set when an external interrupt has occured.

1.8.12 TMOD (Timer Mode, Address 89h)

The Timer Mode SFR is used to configure the mode of operation of each of the two timers. Using this SFR, (see Table 113) our program may configure each timer to be a 16-bit timer, an 8-bit auto-reload timer, a 13-bit timer, or two separate timers. The timer can be used to count pulses from the internal clock ($C/\overline{T} = 0$) or to count events ($C/\overline{T} = 1$) connected to an external pin (P3.5 T1 for timer 1 or P3.4 T0 for timer 0). Additionally, in order to facilitate pulse-width measurements, we may configure the timers (setting the GATE bit to 1) to only start counting when an external pin (P3.3 INT1 for timer 1 or P3.2 INT0 for timer 0) is high. This is further explained in section 2.11.15.

Hex Byte Address	Not Bit-addressable								Symbol
89	–	–	–	–	–	–	–	–	TMOD
	GATE	C/	M1	M0	GATE	C/	M1	M0	Bit
	Timer 1				Timer0				

Table 1-13 TMOD

Download free eBooks at bookboon.com

1.8.13 TL0/TH0 (Timer 0 Low/High, Addresses 8Ah/8Ch)

These two SFRs, taken together, represent the timer 0 counting registers. Their exact behaviour depends on how the timer is configured in the TMOD SFR; however, these timers always count up. What is configurable is how and when they increment in value. Note that these two registers do not occupy consecutive address locations, and hence cannot be loaded together say by using an SFR16 data type variable in KEIL C. (see note on Little Endian / Big Endian in section 6.1).

1.8.14 TL1/TH1 (Timer 1 Low/High, Addresses 8Bh/8Dh)

These two SFRs, taken together, represent the timer 1 counting registers. As for timer 0, their exact behaviour depends on how the timer is configured in the TMOD SFR; however, these timers always count up. What is configurable is how and when they increment in value. Note that these two registers, same as TL0 and TH0, do not occupy consecutive address locations, and hence once again they cannot be loaded together say by using an SFR16 data type variable in KEIL C.

1.8.15 SCON (Serial Control, Address 98h, Bit-Addressable)

The Serial Control SFR is used to configure the behaviour of the 8051's on-board serial port. This SFR controls the baud rate of the serial port, whether the serial port is activated to receive data, and also contains flags (TI and RI) that are set when a byte is successfully sent or received. These in turn can also be programmed to generate interrupts, thus providing the capability to have an interrupt controlled serial reception and/or transmission.

Hex Byte Address	Bit-addressable								Symbol
98	9F	9E	9D	9C	9B	9A	99	98	SCON
	SM0	SM1	SM2	REN	TB8	RB8	TI	RI	Bit Symbol
	SCON.7	SCON.6	SCON.5	SCON.4	SCON.3	SCON.2	SCON.1	SCON.0	Bit – ASM
	SCON^7	SCON^6	SCON^5	SCON^4	SCON^3	SCON^2	SCON^1	SCON^0	Bit – KEIL C

Table 1-14 SCON

1.8.16 SBUF (Serial Buffer Address 99h)

The Serial Buffer SFR is used to send and receive data via the on-board serial port. Any value written to SBUF will be sent out the serial port's TXD pin (which is actually pin P3.1). Likewise, any value which the 8051 receives via the serial port's RXD pin (which is actually pin P3.0) will be delivered to the user program via SBUF. In other words, SBUF serves as the output port when written to and as an input port when read from. Although SBUF has just one address, it is actually two separate registers, one activated by a READ instruction (to read a character which has been received) and the other activated by a WRITE instruction used to send the data which has to be transmitted. Simultaneous transmit and receive operations (full-duplex) can thus be handled.

Download free eBooks at bookboon.com

Moreover, when data is received into SBUF, RI flag (bit SCON.0) is set. This may in turn be programmed to generate an interrupt, signaling that a character has been received which can then be read by the program. Similarly, when a character has been sent by the device, TI flag (bit SCON.1) is set which can also be programmed to trigger an interrupt. This would indicate that a character has been transmitted and thus SBUF can be loaded again which a fresh character for subsequent transmission. Both RI and TI trigger the same serial interrupt and therefore the Interrupt Service Routine would have to check which flag caused the interrupt (it may even be both of them at the same time!) and branch accordingly.

1.8.17 IE (Interrupt Enable, Addresses A8h)

The Interrupt Enable SFR is used to enable and disable specific interrupts. The low 7 bits of the SFR are used to enable/disable the specific interrupts, whereas the most significant bit (msb) is used to enable or disable ALL the interrupts. Thus, if the msb of IE is 0 all interrupts are disabled regardless of whether an individual interrupt is enabled by setting a lower bit.

Hex Byte Address	Bit-addressable								Symbol
A8	AF	AE	AD	AC	AB	AA	A9	A8	IE
	EA	-	ET2	ES	ET1	EX1	ET0	EX0	Bit Symbol
	IE.7	IE.6	IE.5	IE.4	IE.3	IE.2	IE.1	IE.0	Bit – ASM
	IE^7	IE^6	IE^5	IE^4	IE^3	IE^2	IE^1	IE^0	Bit – KEIL C

Table 1-15 IE

This e-book
is made with
SetaPDF

SETASIGN

PDF components for **PHP** developers

www.setasign.com

Click on the ad to read more

Download free eBooks at bookboon.com

1.8.18 IP (Interrupt Priority, Address B8h, Bit-Addressable)

The Interrupt Priority SFR is used to specify the relative priority of each interrupt. On the 8051, an interrupt can be of any one of two types. It may either be of a low (0) priority or a high (1) priority.

Hex Byte Address	Bit-addressable								Symbol
B8	BF	BE	BD	BC	BB	BA	B9	B8	IP
	-	-	PT2	PS	PT1	PX1	PT0	PX0	Bit Symbol
	IP.7	IP.6	IP.5	IP.4	IP.3	IP.2	IP.1	IP.0	Bit – ASM
	IP^7	IP^6	IP^5	IP^4	IP^3	IP^2	IP^1	IP^0	Bit – KEIL C

Table 1-16 IP

An interrupt may only interrupt other interrupts of lower priority. For example, if we configure the 8051 so that all interrupts are of low priority except the serial interrupt, the serial interrupt will always be able to interrupt the system, even if another interrupt is currently executing its service routine. However, if a serial interrupt service routine is executing then no other interrupt will be able to interfere with the serial interrupt service routine since the serial interrupt has the highest priority.

1.8.19 PSW (Program Status Word, Address D0h, Bit-Addressable)

The Program Status Word is used to store a number of important bits that are set and cleared by some of the 8051 instructions. The PSW SFR contains the carry flag, the auxiliary carry flag, the overflow flag, and the parity flag. Additionally, the PSW register contains the register bank select flags (RS1 and RS0) which are used to select which of the register banks is currently selected. Bits 3 and 4 of the PSW SFR determine which register bank is currently being used as shown in Table 1-18 Register Bank Selection bits. The default (at switch-on) reset value is bank 0 (RS0 = RS1 = 0).

Hex Byte Address	Bit-addressable								Symbol
D0	D7	D6	D5	D4	D3	D2	D1	D0	PSW
	CY	AC	F0	RS1	RS0	OV	-	P	Bit Symbol
	PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0	Bit – ASM
	PSW^7	PSW^6	PSW^5	PSW^4	PSW^3	PSW^2	PSW^1	PSW^0	Bit – KEIL C

Table 1-17 PSW

RS1	RS0	Register Bank	Address Range
0	0	0	00H – 07H
0	1	1	08H – 0FH
1	0	2	10H – 17H
1	1	3	18H – 1FH

Table 1-18 Register Bank Selection bits

Download free eBooks at bookboon.com

1.8.20 ACC (Accumulator A, Address E0h, Bit-Addressable)

The Accumulator is one of the most used SFRs on the 8051 since it is involved in so many instructions. The Accumulator resides as an SFR at E0h, which means the instruction MOV A, #20h is really the same as MOV 0E0h, #20h. However, it is a good idea to use the first method since it only requires two bytes whereas the latter instruction requires three bytes.

Hex Byte Address	**Bit-addressable**								**Symbol**
E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
	ACC.7	ACC.6	ACC.5	ACC.4	ACC.3	ACC.2	ACC.1	ACC.0	Bit – ASM
	ACC^7	ACC^6	ACC^5	ACC^4	ACC^3	ACC^2	ACC^1	ACC^0	Bit – KEIL C

Table 1-19 ACC

1.8.21 B (B Register, Address F0h, Bit-Addressable)

The B register is used specifically in two instructions: the multiply (MUL AB) and divide (DIV AB) operations. The B register is also commonly used by programmers as an auxiliary register to temporarily store values.

Hex Byte Address	**Bit-addressable**								**Symbol**
F0	F7	F6	F5	F4	F3	F2	F1	F0	B
	B.7	B.6	B.5	B.4	B.3	B.2	B.1	B.0	Bit – ASM
	B^7	B^6	B^5	B^4	B^3	B^2	B^1	B^0	Bit – KEIL C

Table 1-20 B

1.8.22 Other SFRs

As we have already seen, Table 1-5 gives a summary of all the SFRs that exist in a standard 8051. All derivative micro-controllers of the 8051 must support these basic SFRs in order to maintain compatibility with the underlying MSCS51 standard.

A common practice when semiconductor firms wish to develop a new 8051 derivative is to add additional SFRs to support new functions that exist in the new chip. For example, the Dallas Semiconductor DS80C320 is upwards compatible with the 8051. This means that any program that runs on a standard 8051 should run without modification on the DS80C320. It also means that all the SFRs defined above apply to the Dallas device.

Download free eBooks at bookboon.com

However, since the DS80C320 provides many new features or devices which the standard 8051 does not support, there must be some way to control and configure these new features. This is accomplished by implementing additional SFRs to those listed here. For example, since the DS80C320 supports two serial ports (as opposed to just one on the 8051), the SFRs SBUF2 and SCON2 have been added. In addition to all the SFRs listed above, the DS80C320 also recognizes these two new SFRs as valid and uses their values to determine the mode of operation of the secondary serial port. Obviously, these new SFRs have been assigned to SFR addresses that were unused in the original 8051.

In this manner, new 8051 derivative chips may be developed which will still run existing 8051 programs. This is also one of the reasons stated earlier, why SFR addresses which are not utilised on one 8051 type should not be used, so that the program would still be compatible with other 8051 versions.

Download free eBooks at bookboon.com

2 Basic Registers

This chapter deals with the addressing modes, interrupts and internal peripherals (timers, serial and parallel input/output ports) of the basic 8051 and goes into more detail on the actual internal registers and how they are use in order to program and control the peripherals.

2.1 The Accumulator, Address E0H, Bit-addressable

Hex Byte Address	Bit-addressable								Symbol
E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
	ACC.7	ACC.6	ACC.5	ACC.4	ACC.3	ACC.2	ACC.1	ACC.0	Bit - ASM
	ACC^7	ACC^6	ACC^5	ACC^4	ACC^3	ACC^2	ACC^1	ACC^0	Bit - KEIL C

Table 2-1 ACC

The Accumulator, as its name suggests, is used as a general purpose register to accumulate the results of certain instructions. It can hold an 8-bit (1 byte) value and is the most versatile register the 8051 has, due to the large number of instructions that make use of this accumulator register. More than half of the 8051's 255 instructions manipulate or make use of the accumulator in some way or another.

Click on the ad to read more

Download free eBooks at bookboon.com

For example, if we want to add the numbers 10 and 20, the resulting answer 30 will be stored in the Accumulator. Once we have a value in the Accumulator we may continue processing the value or we may store it in another register or in memory.

2.2 The R registers

There are 4 banks of registers, with 8 registers, named R0, R1, R2, R3, R4, R5, R6 and R7 per bank. The default bank is Bank 0, with R0 having address 00H and R7 having address 07H.

These registers are used as auxiliary registers in many operations. To continue with the above example, suppose we are adding the numbers 10 and 20. The original number 10 may be stored in the Accumulator whereas the value 20 may be stored in, say, register R4. To process the addition we would use the command:

```
ADD A, R4
```

After executing this instruction the Accumulator will contain the value 30. We may think of the R registers as some very important auxiliary or helper registers. The Accumulator alone would not be very useful if it were not for these R registers.

These registers are also used to store values temporarily. For example, let us say we want to add the values in R1 and R2 together and then subtract the values of R3 and R4. One way to do this would be:

```
MOV A, R3     ; Move the value of R3 into the accumulator
ADD A, R4     ; Add the value of R4
MOV R5, A     ; Store the resulting value temporarily in R5
MOV A, R1     ; Move the value of R1 into the accumulator
ADD A, R2     ; Add the value of R2 to the accumulator
SUBB A, R5    ; Subtract the value of R5, which now contains R3 + R4
```

As you can see, we used R5 to temporarily hold the sum of R3 and R4. Of course, this is not the most efficient way to calculate

$$(R1+R2) - (R3 +R4)$$

but it does illustrate the use of the R registers as a way of storing values temporarily. Note that we are assuming that the resultant sum of (R1+R2) and (R3+R4) fits in an 8-bit register.

Download free eBooks at bookboon.com

2.3 The B Register, address F0H, Bit-addressable

The B register is very similar to the Accumulator in the sense that it may hold an 8-bit (1-byte) value.

Hex Byte Address	Bit-addressable								Symbol
F0	F7	F6	F5	F4	F3	F2	F1	F0	B
	B.7	B.6	B.5	B.4	B.3	B.2	B.1	B.0	Bit – ASM
	B^7	B^6	B^5	B^4	B^3	B^2	B^1	B^0	Bit – KEIL C

Table 2-2 B

The B register is only used directly by two 8051 instructions: MUL AB and DIV AB. Thus, if we want to quickly and easily multiply or divide A by another number, we may store the other number in B and make use of these two instructions.

Aside from the MUL and DIV instructions, the B register is often used as yet another temporary storage register much like a ninth R register.

2.4 The Data Pointer (DPTR)

The Data Pointer (DPTR) is the 8051's only user-accessable 16-bit (2-byte) register. The Accumulator, R registers, and B register are all 1-byte registers.

DPTR as the name suggests, is used to point to data. It is used by a number of commands which allow the 8051 to access external memory. When the 8051 accesses the external memory, it will access it at the address indicated by DPTR.

While DPTR is most often used to point to data in external memory, many programmers often take advantage of the fact that it is the only true 16-bit register available. It is often used to store 2-byte values which have nothing to do with memory locations. Moreover, it can be used as 2 separate and independent 8-bit registers, the high byte register DPH and the low byte register DPL.

2.5 The Program Counter (PC)

The Program Counter (PC) register is not part of the SFRs. It contains a 2-byte address which tells the 8051 where the next instruction to execute is found in memory. When the 8051 is initialized, the PC is set to 0000h and is incremented each time an instruction is executed. It is important to note that PC is not always incremented by one after each instruction. This is because of the fact that some instructions require 2 or 3 bytes and the PC will therefore be incremented by 2 or 3 in these cases.

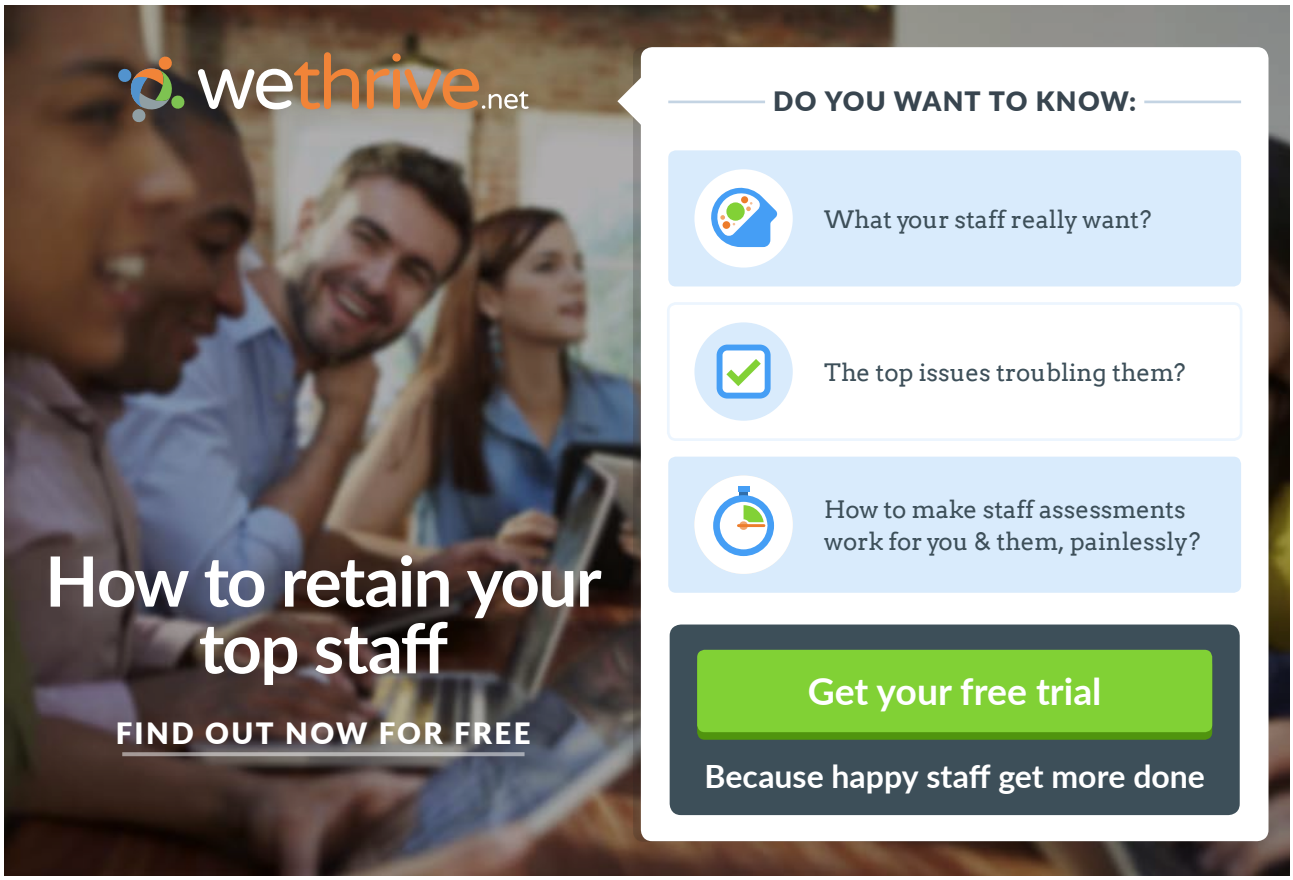
Download free eBooks at bookboon.com

The Program Counter is special in that there is no way to directly modify its value. That is to say, we cannot code something like PC = 2430h. On the other hand, if we execute LJMP 2430h (meaning junp to location 2430 hex), we would have effectively accomplished the same thing, since the micro-controller would need to load the program counter with the address of the location where it needs to jump to and continue the execution of the code from there.

It is also interesting to note that while we may change the value of the PC (by executing a jump instruction, etc.) there is no specific direct instruction to read the value of the PC. That is to say, there is no way to ask the 8051 "What is the address of the instruction you are about to execute?" As it turns out, this is not completely true; there is one trick that may be used to determine the current value of PC. When for example a CALL is executed, the address of the instruction after the CALL is pushed on stack (first the low byte followed by the high byte). Once it is on the stack, this address can be popped or modified at will! This trick is used extensively in the PaulOS Real-Time Operating System (RTOS) and other RTOSs in order to swap tasks.

2.6 The Stack Pointer (SP), address 81H

The Stack Pointer, like all registers except DPTR and PC, may hold an 8-bit (1-byte) value. The Stack Pointer is used to indicate where the next value to be removed from the stack should be taken from.



Click on the ad to read more

Download free eBooks at bookboon.com

When we push a value onto the stack, the 8051 first increments the value of the SP and then stores the value at the resulting indirectly addressable memory location.

When we pop a value off the stack, the 8051 returns the value from the indirectly addressable memory location indicated by the SP, and then decrements the value of the SP.

This order of operation is important. When the 8051 is initialized (reset), the SP will be set to 07h. If we immediately push a value onto the stack, the value will be stored in Internal RAM address 08h. This makes sense taking into account what was mentioned two paragraphs above: First the 8051 will increment the value of the SP (from 07h to 08h) and then it will store the pushed value at that memory address (08h).

The SP is modified directly by the 8051 by the following 6 instructions: PUSH, POP, ACALL, LCALL, RET, and RETI. It is also used intrinsically whenever an interrupt is triggered (more on interrupts in section 2.13). The SP always points to an indirectly addressable internal memory area and these instructions act as in the Indirect Addressing mode. (see section 2.7.3). They make use of or modify the contents of the indirectly addressable memory pointed to by the SP. Since the stack resides in the indirectly addressable internal memory, there is a limit to the size of stack which can be used, which is also affected by the number and type of the variables being stored in this same area.

2.7 Addressing Modes

An addressing mode refers to how we are addressing a given memory location. In summary, the addressing modes are as follows, with an example of each:

Immediate Addressing	MOV A,#20h
Direct Addressing	MOV A,30h
Indirect Addressing	MOV A,@R0
External Indirect	MOVX A,@DPTR
Code Indirect	MOVC A,@A+DPTR

Each of these addressing modes provides important flexibility. Moreover, the type of addressing mode also determines the memory area that is being accessed by the instruction. Reference to Table 13 would be helpful at this stage.

2.7.1 Immediate Addressing

Immediate addressing is so-named because the value to be stored in memory immediately follows the operation code in memory. That is to say, the instruction itself dictates what value will be stored in memory.

Download free eBooks at bookboon.com

For example, the instruction:

```
MOV A, #20h
```

uses Immediate Addressing because the Accumulator will be loaded with the value that immediately follows; in this case 20 (hexadecimal). Immediate addressing is very fast since the value to be loaded is included in the instruction. However, since the value to be loaded is fixed at compile-time it is not very flexible.

2.7.2 Direct Addressing – Data in Directly Addressable Internal RAM

Direct addressing is so-named because the value to be stored in memory is obtained by directly retrieving it from another memory location address which is given with the instruction. For example, the instruction:

```
MOV A, 30h
```

will read the data out of the Directly Addressable Internal RAM address 30 (hexadecimal) and store it in the Accumulator. Direct addressing is generally fast since, although the value to be loaded is not included in the instruction, it is quickly accessible since it is stored in the 8051's Internal Directly Addressable RAM. It is also much more flexible than Immediate Addressing since the value to be loaded is whatever is found at the given address; which may be variable.

Also, referring to the 8051 Internal memory map, in Table 1.2.3. it is important to note that when using direct addressing any instruction which refers to an address between 00h and 7Fh is referring to Internal Memory. Any instruction which refers to an address between 80h and FFh is referring to the SFR control registers that control the 8051 micro-controller itself.

Certain versions of the 8051 such as the 8032 have 256 bytes (0 to FF hex) of Internal ram. The obvious question that may arise is, "If indirect addressing, an address from 80h through FFh refers to SFRs, how can we access the upper 128 bytes of Internal RAM that are available on the 8032?" The answer is: We cannot access them using direct addressing. As stated earlier, if we directly refer to an address in the range of 80h through FFh, we will be referring to an SFR. However, we may access the 8032's upper 128 bytes of RAM by using the next addressing mode, which is indirect addressing.

Download free eBooks at bookboon.com

2.7.3 Indirect Addressing – Data in Indirectly Addressable Internal RAM

Indirect addressing is a very powerful addressing mode which in many cases provides an exceptional level of flexibility. Indirect addressing is also the only way to access the extra 128 bytes of Indirectly Addressable Internal RAM found on an 8032 or other improved 8051 versions. A typical instruction using Indirect addressing is the following:

```
MOV A, @R0
```

This instruction causes the 8051 to examine the value of the R0 register. The 8051 will then load the accumulator with the value from Internal RAM which is found at the address indicated by R0. R0 is simply acting as a pointer to an Indirectly Addressable Internal memory location.

For example, let us say R0 holds the value 40h and Internal RAM address 40h holds the value 67h. When the above instruction is executed the 8051 will check the value of R0. Since R0 holds 40h the 8051 will get the value out of Internal RAM address 40h (which holds 67h) and store it in the Accumulator. Thus, the Accumulator ends up holding 67h.

Indirect addressing always refers to the Indirectly Addressable Internal RAM only; it never refers to an SFR. In a prior example we mentioned that SFR 99h can be used to write a value to the serial port. Thus one may think that the following would be a valid solution to write the value 1 to the serial port:

Click on the ad to read more

Download free eBooks at bookboon.com

```
MOV R0, #99h        ;Load the address of the serial port into R0
MOV @R0, #01h       ;Send 01 to the serial port – Wrong!!
```

This is not the correct way. Since indirect addressing always refers to Indirect Internal RAM these two instructions would write the value 01h to Internal RAM address 99h on an 8032. On an 8051 these two instructions would produce an undefined result since the 8051 only has 128 bytes of Internal RAM. Indirect addressing cannot therefore be used to access the SFRs, which can only be accessed using direct addressing. The correct way would therefore be:

```
MOV 99h, #01h       ;Load location 99h (serial port SBUF register location) with 01
```

or since the assembler would know that SBUF resides at address 99h

```
MOV SBUF, #01h              ;Send 01 to the serial port SBUF register
```

2.7.4 External Indirect – 16-bit address

External Memory is accessed using a very limited number of commands. In the case of a 16-bit external data memory address, there are only two commands that can be used for External Indirect addressing mode:

```
MOVX A, @DPTR
MOVX @DPTR, A
```

The X in MOVX signifies that an External address is being used. As we can see, both commands utilize DPTR. In these instructions, DPTR must first be loaded with the address of external memory (or memory mapped device such as an 8255 input/output port chip) that we wish to read from or write to. Once DPTR holds the correct external memory address, the first command will move the contents of that external memory address into the Accumulator. The second command will do the opposite; it will allow us to write the value of the Accumulator to the external memory address pointed to by DPTR.

If the address to be accessed is the Program (or Code) area, then the following commands must be used:

```
MOVC A, @A + PC
```

or

```
MOVC A, @A + DPTR
```


Download free eBooks at bookboon.com

Here the address of the byte fetched is the sum of the original unsigned 8-bit Accumulator contents and the contents of either the 16-bit Program Counter (PC) or the 16-bit Data Pointer (DPTR). In some instances therefore, the accumulator has to be zeroed in order to use these commands.

In other cases, the value in the accumulator comes in handy when using translation or conversion tables. As a simple example assume that we have a table with the heights of a number of students (as an 8 bit integer 0–255 cms), and we want to get the height of a particular student.

The accumulator would be loaded with that student number (also in the range from 0 to 255) and DPTR would be loaded with the address of the start of the table.

Using MOVC A,@A + DPTR we can immediately get the height of that particular student loaded in the accumulator.

In conjunction with the MOVX and MOVC instructions, the micro-controller internal hardware would also set up the special control signals RD (Read), ALE (Address Latch Enable) and PSEN (Program Store Enable) which should be used by the external logic to enable the correct ROM or RAM for program and/or data access.

2.7.5 External Indirect – 8-bit

This form of addressing is usually only used in relatively small projects that have a very small (256 bytes max) amount of external data RAM. An example of this addressing mode is:

```
MOVX @R0, A
```

Once again, the value of R0 (containg the external RAM address) is first read and the value of the Accumulator is written to that address in External RAM. Since the value of @R0 can only be 00h through FFh the project would effectively be limited to 256 bytes of External RAM. There are relatively simple hardware/software tricks that can be implemented to access more than 256 bytes of memory using External Indirect 8-bit addressing; however, it is usually easier to use the DPTR version of addressing if the project in hand has more than 256 bytes of External RAM.

It should be noted here that if we are using C, the compiler when converting the C source program to machine code, is intelligent enough to choose the correct addressing mode to address the variables. When declaring the variable types in our C progam, the location of their storage space would also be given or implied. Thus the compiler would know in which part of memory they are being stored so that it would be able to refer to them in the correct addressing mode.

Download free eBooks at bookboon.com

2.8 Program Flow

When an 8051 is first initialised, it resets the PC to 0000h. The 8051 then begins to execute the instructions sequentially in memory unless a program instruction causes the PC to be otherwise altered. There are various instructions that can modify the value of the PC; specifically, conditional branching instructions, direct jumps, calls to subroutines, and returns from subroutines. Additionally, interrupts, when enabled, can cause the program flow to deviate from its otherwise sequential flow.

2.8.1 Conditional Branching

The 8051 contains a suite of instructions which, as a group, are referred to as conditional branching instructions. These instructions cause program execution to follow a non-sequential path if a certain condition is satisfied (true).

Let us take for example, the JB instruction. This instruction means Jump if Bit Set. An example of the JB instruction might be:

```
            JB 45h, HELLO
            MOV A, #10
            …….
            …….
HELLO:      ....
```

In this case, the 8051 will analyse the contents of bit 45h. If the bit is set (1) then the program execution will jump immediately to the label HELLO, skipping the MOV A, #10 instruction and those following it. If the bit is not set (0) the conditional branch fails and the program execution continues, as usual, with the MOV A, #10 instruction which follows.

Conditional branching is really the fundamental building block of program logic since all decisions are accomplished by using conditional branching. These 8051 assembly language conditional branching instructions can be thought of as the equivalent "IF…THEN" structure found in other higher level programming languages.

An important note worth mentioning about conditional branching is that the program may only branch to instructions located within 128 bytes prior to or 127 bytes following the address which follows the conditional branch instruction. This means that in the above example the label HELLO must be within +127 /-128 bytes of the memory address which contains the conditional branching instruction.

Download free eBooks at bookboon.com

If it so happens that in our program we cannot avoid having the label HELLO occurring very far from the conditional branch address, then we can use what is referred to as a Stepping Stone. This is easily understood by following this example, where the target jump for the conditional JB instruction is actually HELLO. We instead make use of the stepping stone label HELLO2:

```
                JB 45h, HELLO2          ; use the stepping stone, to a near location
                SJMP CONTINUE           ; skip over the stepping stone
HELLO2:
                LJMP HELLO              ; now jump to the far location HELLO
CONTINUE:
                MOV A, #10
                …….
                …….
HELLO:          ....        ; this label is very far away from the JB 45h, HELLO2 location
```

Click on the ad to read more
Download free eBooks at bookboon.com

2.8.2 Direct Jumps

While conditional branching is extremely important, it is often necessary to make a direct branch to a given memory location without basing it on a given logical decision. This is equivalent to the rarely used GOTO command in C. In this case we want the program flow to continue at a given memory address without considering any conditions.

This is accomplished in the 8051 using the Direct Jump and Call instructions. As illustrated in the last paragraph, this suite of instructions causes program flow to change unconditionally.

Consider the example:

```
        LJMP NEW_ADDRESS
        .
        .
        .
NEW_ADDRESS:    ....
```

The LJMP (Long Jump) instruction when executed, the PC is loaded with the address of NEW_ADDRESS location and program execution continues sequentially from there.

The obvious difference between the Direct Jump / Call instructions and the conditional branching is that with Direct Jumps and Calls program flow always changes. With conditional branching program flow only changes if a certain condition is true.

It is worth mentioning that, aside from LJMP, there are two other instructions which cause a direct jump to occur; the SJMP (Short Jump) and AJMP (Absolute Jump) commands. Functionally, these two commands perform exactly the same function as the LJMP command – that is to say, they always cause program flow to continue at the address indicated by the command. However, SJMP and AJMP differ in the following ways:

The SJMP command, like the conditional branching instructions, can only jump to an address within +127/-128 bytes of the SJMP command (hence the Short in the name).

The AJMP command can only jump to an absolute address that is in the same 2KB block of memory where the AJMP command is residing. That is to say, if the AJMP command is at code memory location 650h, it can only do a jump to addresses 0000h through 07FFh (0 through 2047, decimal).

Download free eBooks at bookboon.com

We may ask, "Why would we want to use the SJMP or AJMP command which have restrictions as to how far they can jump, if we can just use the LJMP command which can jump to any location in memory?" The answer is simply a matter of code usage.

The LJMP command requires three bytes of code memory whereas both the SJMP and AJMP commands require only two. Thus, if we are developing an application that has memory restrictions we can often save quite a bit of memory using the 2-byte AJMP/SJMP instructions instead of the 3-byte instruction. Speed is not affected since all the three instruction types require 2 machine cycles to execute.

Suppose we are writing a program that requires 2100 bytes of memory but we have a memory restriction of 2KB (2048 bytes). If we do a simple search/replace operation to change if possible the LJMPs to SJMPs or AJMPs, the program might shrink down to an allowable size. Thus, without changing any logic whatsoever in our program, we might save enough bytes to meet our 2048 byte code memory restriction.

Some quality assemblers will actually do the above conversion for us automatically. That is, they will automatically change our LJMPs to SJMPs whenever possible. This is a very powerful capability that we may want to look for in an assembler if we plan to develop many projects that have code memory restrictions.

If we are using C, most compilers, when converting the C source program to machine code, are intelligent enough to choose the correct JMP type in order to save code space.

2.8.3 Direct Calls

Another operation that will be familiar to seasoned programmers is the LCALL or ACALL instruction. This is similar to a function call in C.

When the 8051 executes an LCALL instruction, the PC is incremented twice to obtain the address of the following instruction. It then pushes the updated Program Counter onto the stack and then continues executing code at the 16-bit address indicated by the LCALL instruction.

When the 8051 executes an ACALL instruction, the PC is incremented twice to obtain the address of the following instruction. It then pushes the updated Program Counter onto the stack and then continues executing code at the 16-bit address formed by successively concatinating the 5 high-order bits of the updated PC with the 11-bit address supplied with the ACALL instruction. The subroutine called must therefore start within the same 2KB block, since its address must have the same higher 5-bits as the updated PC.

2.8.4 Returns from Routines

Another structure that can cause program flow to change is the "Return from Subroutine" instruction, known as RET in 8051 Assembly Language.

Download free eBooks at bookboon.com

The RET instruction, when executed, returns to the address following the instruction that called the given subroutine. More accurately, it returns to the address that is stored on the stack.

The RET command is unconditional in the sense that it always changes program flow without basing it on any condition. It is also variable in the sense that program flow can be different each time the RET instruction is executed, since this depends on the address of the CALL instruction (and the address popped on stack when the CALL was made).

2.8.5 Interrupts and RETI

An interrupt is a special feature which allows the 8051 to provide the illusion of multi-tasking, although in reality the 8051 is only doing one thing at a time. The word interrupt can often be substituted with the word event.

An interrupt is triggered whenever a corresponding event occurs. When the event occurs, the 8051 temporarily puts on hold the normal execution of the program (saving on stack the return address and updating the stack pointer register) and executes a special section of code referred to as an interrupt handler or an interrupt service routine (ISR) by changing the program counter contents. The interrupt handler performs whatever special functions are required to handle the event and then returns control to the 8051 (using the RETI instruction) at which point program execution continues as if it had never been interrupted (naturally some time would have been lost while executing the interrupt routine).

The topic of interrupts is somewhat tricky but very important. For that reason, an entire section (2.13) will be dedicated to the topic, but for now suffice it to say that Interrupts can cause program flow to change.

2.9 Low-Level Information

In order to understand and make better use of the 8051, it is necessary to understand some underlying information concerning timing.

The 8051 operations are based on an external crystal clock. This is an electrical device which, when supplied with energy, emits pulses at a fixed frequency. One can find crystals of virtually any frequency depending on the application requirements. When using an 8051, the most common crystal frequencies are 12 MHz or 11.059 MHz – with the latter being much more common. Why would anyone pick such an odd frequency? There is a good reason for it – it has to do with generating baud rates for the serial port and we will talk more about it in the Serial Communications section 2.12. For the remainder of this discussion, unless stated otherwise, we will assume that we are using an 11.0592 MHz crystal.

Download free eBooks at bookboon.com

Micro-controllers (and many other electrical systems) use crystals oscillators in order to synchronise operations and the 8051 is no exception. Effectively, the 8051 operates using what are called "machine cycles". A single machine cycle is the minimum amount of time (or clock cycles) in which a single 8051 instruction can be executed, although many instructions take multiple cycles.

A machine cycle in the basic 8051 is in reality 12 pulses of the crystal clock. That is to say, if an instruction takes one machine cycle to execute, it will take 12 pulses of the crystal to execute. Since we know the crystal is pulsing 11,059,200 times per second and that one machine cycle is 12 pulses, we can calculate how many instruction cycles the 8051 can execute in one second:

11,059,200 / 12 = 921,600

This means that the 8051 can execute 921,600 single-cycle instructions per second. Since a large number of 8051 instructions are single-cycle instructions it is often stated that the 8051 can execute roughly 1 million instructions per second, although in reality it is less – and depending on the instructions being used, an average estimate of about 600,000 instructions per second is more realistic.

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

Click on the ad to read more

Download free eBooks at bookboon.com

For example, if we are using exclusively 2-cycle instructions we would find that the 8051 would execute 460,800 instructions per second. The 8051 also has two really slow instructions (MUL AB and DIV AB) that require a full 4 cycles to execute – if we were to execute nothing but those instructions we would find the performance reduced to about 230,400 instructions per second.

Many 8051 derivative chips change the instruction timing. For example, many optimised versions of the 8051 execute instructions in 1 oscillator cycle instead of 12; such a chip would be effectively 12 times faster than the 8051 when used with the same 11.0592 MHz crystal. Moreover, these modern 8051 derivative micro-controllers use crystals of 22.1184 MHz or even higher, making them, overall at least 24 times faster than the standard 8051.

Since all the instructions require different amounts of time to execute a very obvious question comes to mind: How can we keep track of time in a time-critical application if we have no reference to time in the outside world?

Luckily, the 8051 includes timers which allow us to time events with high precision. This will be the topic of the next section.

2.10 Timers

The basic 8051 comes equipped with two timers, both of which may be controlled, set, read, and configured individually. The 8051 timers have three general functions:

- Keeping time and/or calculating the time elapsed between events.
- Counting the events themselves.
- Generating baud rates for the serial port.

The three timer uses are distinct so we will talk about each of them separately. The first two uses will be discussed in this chapter while the use of timers for baud rate generation will be discussed in section 2.11.2.

2.10.1 How does a timer count?

The answer to this question is very simple: A timer always counts up. It does not matter whether the timer is being used as a timer, as a counter, or as a baud rate generator: a timer is always **incremented** by the micro-controller. Moreover, when the timer register reaches the upper limit, a timer flag is set (TF0 or TF1) which can be checked by the program or it can even be made to generate an interrupt. The timer then resumes counting from zero unless instructed otherwise by having it setup in the auto-reload mode.

Download free eBooks at bookboon.com

2.10.2 Using Timers to measure time

Obviously, one of the primary uses of timers is to measure time. We will discuss this use of timers first in the following sections and then in section 2.10.15 we will subsequently discuss the use of timers to count events. When a timer is used to measure time it is also called an "interval timer" since it is measuring the time of the interval between two events.

2.10.3 How long does a timer take to count?

First, it is worth mentioning that when a timer is in interval timer mode (as opposed to event counter mode) and correctly configured, it will increment by 1 at every machine cycle. As you will recall from section 2.9, a single machine cycle consists of 12 crystal pulses. Thus a running timer will be incremented at the rate of

11,059,200 / 12 = 921,600 times per second

or

1/921600 seconds per count (1.0851 micro-seconds)

Unlike instructions which require 1, 2 or 4 machine cycles, the timers are consistent; they will always be incremented once per machine cycle. Even new variants of the 8051 which run very fast and use only one clock cycle per machine cycle, they all have the option to run the timers slower (dividing the clock frequency by twelve or more) so that the timings remain the same thus maintaining compatibility between different versions of the micro-controller. Thus if a timer has counted from 0 to 65535 (the maximum count of 65536 times) we may calculate the elapsed time to be:

65,536 / 921,600 = 0.0711 seconds or approximately 71 milliseconds

This would represent the maximum time we can use on a 16-bit timer. Normally we would need to execute a certain section of code say once every second, or we would need to have a delay of say 50 milliseconds. Since the timer registers can only hold integer values ranging from 0 to 65535 we should find suitable integer values which can give us some suitable delay, which we can then use to get our actual required delay (of 1 second) in our program.

So we come to another important calculation. Let us say we want to know how many times will the timer be incremented in 0.05 seconds (50 milliseconds). We can do simple multiplication:

0.05 * 921,600 = 46,080

which also happens to be an exact integer and thus we can use it in our timer to get accurate timings.

Download free eBooks at bookboon.com

This tells us that it will take 0.05 seconds to count from 0 to 46,080. Now this is a little more useful. If we know that it takes 1/20th of a second to count from 0 to 46,080 and we want to execute some event every second we simply wait for the timer to count from 0 to 46,080 twenty times (also an exact integer); then we execute our event. We would need to reset the timer every time it reaches 46080, unless we are using the auto-reload mode as will be explained later. We would need to wait for the timer to count up another 20 times. In this manner we will effectively be executing our event once every second, accurate to within thousandths of a second.

If we are using the timer as a 16-bit (0 to 65535) timer and since as we have already stated, the timer actually counts up and moreover noting that it will set the overflow flag or even generate an interrupt when it overflows, then we would actually load the timer registers with 19456, (which is 65536–46080) so that it would take another 46080 counts in order to overflow. Thus we can have an interrupt generated every 1/20 of a second and then counting 20 interrupts before executing the required code. Otherwise, we can start the timer, wait for the overflow flag to be set by the timer, resetting the overflow flag and reloading the timer registers with 19456 and repeating the process 20 times and then proceed once the 1 second has passed.

Thus, we now have a system with which to measure time. All we need to review is how to control the timers and initialise them to provide us with the interrupt delay that we need. Figure 2-1 shows the pins, bits and SFRs which control Timer 1, and similarly for Timer 0. These will be used to configure the timers as explained further down.

Click on the ad to read more
Download free eBooks at bookboon.com

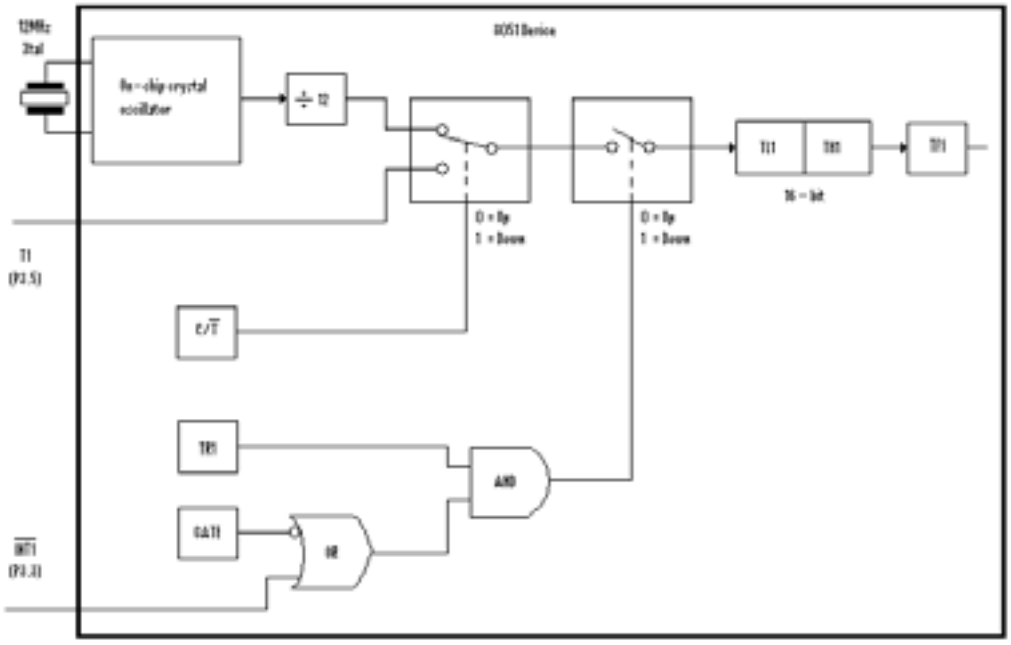


Figure 2-1 Timer/Counter 1 Mode 0 and Mode 1 operation

2.10.4 Timer SFRs

As mentioned before, the 8051 has two timers each of which functions essentially in the same way. One timer is TIMER0 and the other is TIMER1. The two timers share two SFRs (TMOD and TCON) which control the timer mode of operation, and each timer also has two SFRs dedicated solely to itself (TH0/TL0 and TH1/TL1).

The SFRs have been assigned names (which all assemblers and compilers know) in order to make it easier to refer to, but in reality an SFR has a numeric address. It is often useful to know the numeric address that corresponds to an SFR name. The SFRs relating to timers are shown in Table 2-3.

SFR Name	Description	SFR Hex Address
TH0	Timer 0 High Byte	8C
TL0	Timer 0 Low Byte	8A
TH1	Timer 1 High Byte	8D
TL1	Timer 1 Low Byte	8B
TCON	Timer Control Register	88
TMOD	Timer Mode Register	89

Table 2-3 Timer-related SFRs

Download free eBooks at bookboon.com

When we enter the name of an SFR into an assembler, it internally converts it to its correct address. For example, the command:

```
MOV TH0, #25h
```

moves the value 25h into the TH0 SFR. However, since TH0 is the same as SFR address 8Ch this command is equivalent to:

```
MOV 8Ch, #25h
```

Now, back to the timers. First, let us talk about Timer 0 which has two SFRs dedicated exclusively to TH0 and TL0. Without making things too complicated to start off with, we may just think of these as the high and low bytes of the timer counter. That is to say, when Timer 0 has a value of 0, both TH0 and TL0 will contain 0. When Timer 0 has the value 1000 decimal, TH0 will hold the high byte of the value (3 decimal) and TL0 will contain the low byte of the value (232 decimal). Reviewing low/high byte notation, recall that we must multiply the high byte by 256 and add the low byte to get the final 16-bit decimal value. That is to say:

$$(TH0 * 256) + TL0 = 1000$$

$$(3 * 256) + 232 = 1000$$

Or else, knowing the final decimal value (1000), we can calculate what values we need to load into TH0 and TL0 using the following simple C instructions:

```
TH0 = 1000/256;        // (integer division, just taking the integer part of the answer)
TL0 = 1000%256;        // (modular division, just taking the remainder after dividing)
```

Certain assembler/compilers can work out these simple equations for us or else we can write our own macros. We might also use the following alternative instructions, obviously giving the same result:

```
TH0 = 1000>>8;      //(shift the number 8 bits to the right, to get the high byte)
TL0 = 1000 & 255;   //(bitwise AND, in order to get the lower 8 bits)
```

Timer 1 works in exactly the same way, but its SFRs are designated as TH1 and TL1.

Since there are only two bytes devoted to the value of each timer it is obvious that the maximum value a timer may have is 65,535. If a timer contains the value 65,535 and is subsequently incremented, it will reset or overflow back to 0. It is this overflow action which triggers the interrupt if enabled.

Download free eBooks at bookboon.com

2.10.5 The TMOD SFR

Let us first talk about our first control SFR: TMOD (Timer Mode). The TMOD SFR is used to control the mode of operation of both timers. Each bit of this SFR gives the micro-controller specific information concerning how to run a timer. The higher four bits (bits 4 through 7) relate to Timer 1 whereas the lower four bits (bits 0 through 3) perform the exact same functions, but for Timer 0. TMOD is not bit-addressable.

The functions of the individual bits of TMOD are shown in Table 2-4.

Click on the ad to read more

Download free eBooks at bookboon.com

Not Bit-addressable			
Bit	**Name**	**Timer**	**Explanation of the Timer Functions**
7	GATE	1	When this bit is set (1), Timer 1 will only run when INT1 (P3.3, EXT1) pin is high, provided that TR1 is set to 1. When this bit is cleared (0), timer 1 will run as dictated by the state of TR1, regardless of the state of INT1 pin. In each case, TR1 (in TCON) must be set to 1 for the timer to run.
6	$C/\overline{T}$	1	When this bit is set (1), Timer 1 will count events (pulses) on T1 (P3.5) pin. When this bit is cleared (0), the timer will increment every machine cycle (XTAL/12)
5	M1	1	Timer 1 mode bit (see Table 25)
4	M0	1	Timer 1 mode bit (see Table 25)
3	GATE	0	When this bit is set (1), Timer 0 will only run when INT0 (P3.2, EXT0) pin is high, provided that TR0 is set to 1. When this bit is cleared (0), timer 0 will run as dictated by the state of TR0, regardless of the state of INT0 pin. In each case, TR0 (in TCON) must be set to 1 for the timer to run.
2	$C/\overline{T}$	0	When this bit is set (1), Timer 0 will count events (pulses) on T0 (P3.4) pin. When this bit is cleared (0), the timer will increment every machine cycle (XTAL/12)
1	M1	0	Timer 0 mode bit (see Table 25)
0	M0	0	Timer 0 mode bit (see Table 25)

Table 2-4 TMOD (89H) SFR

As we can see in the Table 2-5 below, four bits (two for each timer, TM0 and TM1) are used to specify a mode of operation for the particular timer.

M1	M0	Timer Mode	Description
0	0	0	13-bit timer
0	1	1	16-bit timer
1	0	2	8-bit auto-reload
1	1	3	Split timer mode

Table 2-5 Timer Mode Control bits

2.10.6 13-bit Timer Mode (mode 0)

Timer mode 0 is a 13-bit timer mode. This is a relic that was kept in the 8051 to maintain compatibility with its predecessor, the 8048. Generally the 13-bit timer mode is not used in new development projects.

Download free eBooks at bookboon.com

When the timer is in 13-bit mode, TLx (meaning TL0 or TL1) will count from 0 to 31. When TLx is incremented from 31, it will "reset" to 0 and increment THx. Thus, effectively, only 13 bits of the two timer bytes are being used: bits 0-4 of TLx and bits 0-7 of THx. This also means, in essence, the timer can only contain 8192 values. If you set a 13-bit timer to 0, it will overflow back to zero 8192 machine cycles later.

Again, there is hardly any reason to use this mode and it is only mentioned so we would not be surprised if we ever end up analysing archaic code which has been passed down through generations of programmers.

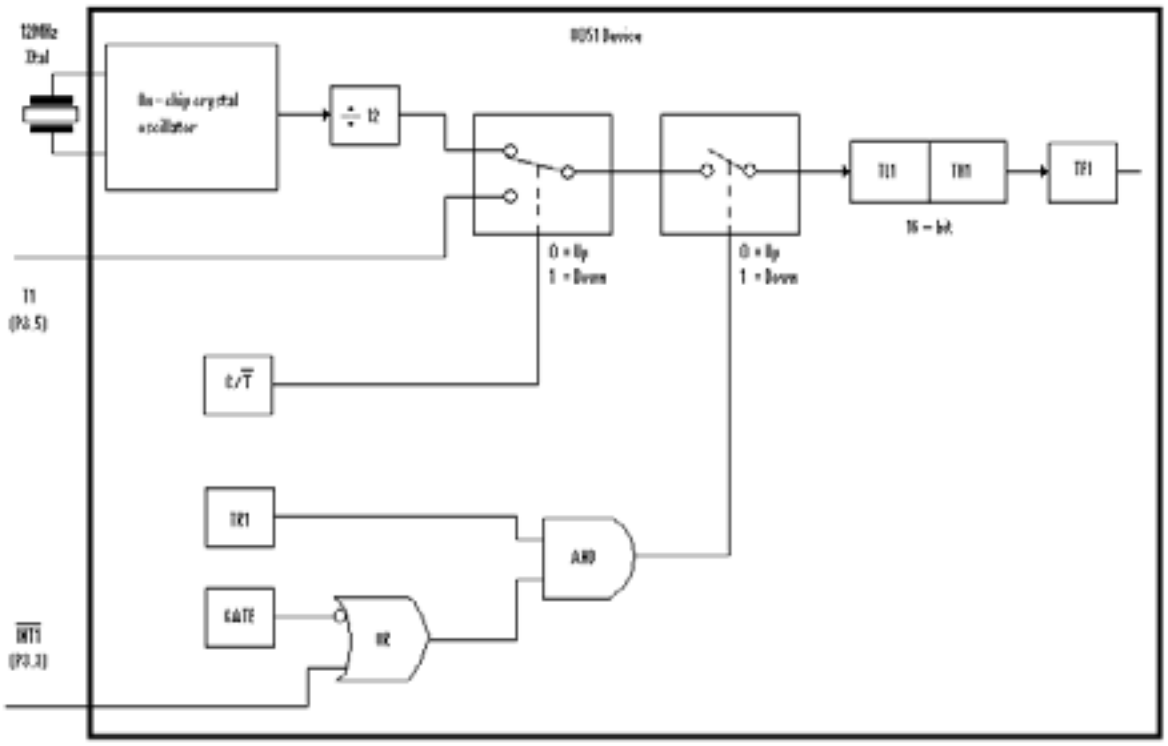


Figure 2-2 Timer 1 Mode 1

2.10.7 16-bit Timer Mode (mode 1)

Timer mode 1 is a 16-bit timer as shown in Figure 2-2 for the case of Timer 1. This is a very commonly used mode and it functions just like 13-bit mode except that all 16 bits are used.

TLx (TL0 or TL1) is incremented from 0 to 255. When TLx is incremented from 255, it resets to 0 and causes THx to be incremented by 1. Since this is a full 16-bit timer, the timer may contain up to 65536 distinct values. If you set a 16-bit timer to 0, it will overflow back to 0 after 65,536 machine cycles, resulting in the longest delay possible.

Download free eBooks at bookboon.com

2.10.8 8-bit Timer Mode (mode 2)

Timer mode 2 is an 8-bit auto-reload mode, as shown in Figure 2-3 for Timer 1. When a timer is in mode 2, THx holds the reload value and TLx is the 8-bit timer register itself. Thus, TLx starts counting up and when TLx reaches 255 and is subsequently incremented, instead of resetting to 0 (as in the case of modes 0 and 1), it will be reset to the reload value stored in THx.

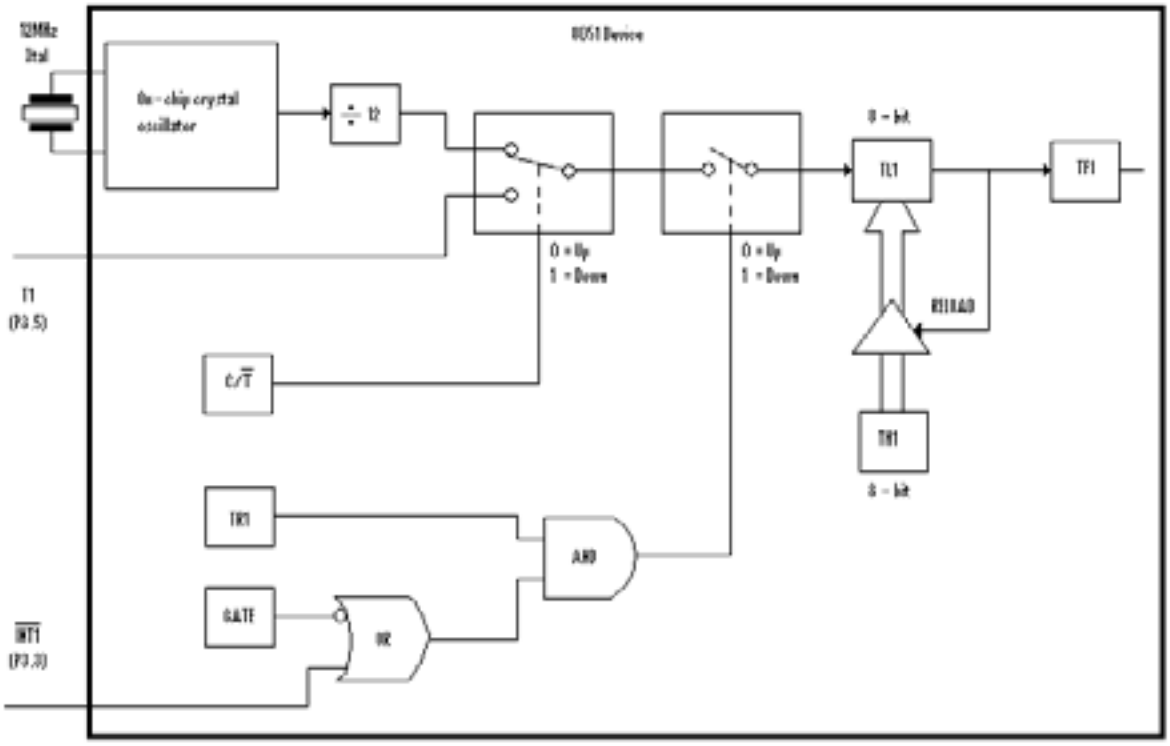


Figure 2-3 Timer 1 Mode 2

For example, let us say TH0 holds the value FDh and TL0 holds the value FDh. After 1 machine cycle, TL0 would be incremented to FEH and if we were to watch the values of TH0 and TL0 for a few machine cycles this is what we would see:

Machine Cycle	TH0 Hex Value	TL0 Hex Value
1	FD	FE
2	FD	FF
3	FD	FD
4	FD	FE
5	FD	FF
6	FD	FD
7	FD	FE

Table 2-6 Timer counters registers

Download free eBooks at bookboon.com

As we can see, the value of TH0 never changed. In fact, when we use mode 2 we almost always set THx to a known value and TLx is the SFR that is constantly incremented. Whenever TLx overflows, the overflow flag TFx will be set, and an interrupt will be generated if so desired.

What is the benefit of auto-reload mode? Perhaps we want the timer to always have a value from 200 to 255 (i.e. we always need the timer to overflow after 56 counts) . If we use mode 0 or 1, we would have to check in code to see if the timer had overflowed and, if so, reset the timer to 200. This wastes time in checking the value and/or to reload it. When we use mode 2 the micro-controller takes care of this for us. Once we have configured a timer in mode 2 we do not have to worry about checking to see if the timer has overflowed nor do we have to worry about resetting the value; the micro-controller hardware will do it all for us.

The auto-reload mode is very commonly used for establishing a baud rate which we will talk more about in the Serial Communications chapter. It is also frequently used whenever we need to have interrupt signals at regular intervals, thus avoiding the need to reset the timer counter registers in the Interrupt Service Routine. We will expand on this later on, in the Interrupts section.

It should be remembered that for Timers 0 and 1, auto-reload is only available in 8-bit mode. Enhanced versions of the 8051, such as the 8031 have other timers which have 16-bit auto-reload capabilities.

Click on the ad to read more
Download free eBooks at bookboon.com

2.10.9 Split Timer Mode (mode 3)

Timer mode 3 is a split-timer mode and can be used only with Timer 0. When Timer 0 is placed in mode 3, it essentially becomes two separate 8-bit timers. That is to say, Timer 0 runs using TL0 and a new Timer 00 running using TH0. Both timers count from 0 to 255 and overflow back to 0. The bits TR1 and TF1 that are related to the real Timer 1 will now be tied to Timer 00 and thus TH0 now controls the original Timer 1 interrupt.

While Timer 0 is in split mode, the real Timer 1 (i.e. TH1 and TL1) can be put into mode 0, 1 or 2 in the normal way but without any interrupt (since TF1 is being used by Timer 0 in mode 3), and may be started or stopped by switching it out of and into its own mode 3. An example of the timers operating in this mode is given in Appendix F.

The only real necessity of using split timer mode is if we need to have two separate timers and, additionally, a baud rate generator and we are using the original 8051 with only two timers available. In such a case we can use the real Timer 1 as a baud rate generator (usually in mode 2) and use TH0 and TL0 as two separate 8-bit timers, by setting Timer 0 in mode 3. Most modern upgrades of the 8051 family have 4 timers or more, making this mode not so really useful.

2.10.10 The TCON SFR

Finally, there is one more SFR that controls the two timers and provides valuable information about them. As we may notice, we have only defined the higher 4 (nibble) of the 8 bits. That is because the other lower 4 bits of the SFR do not have anything to do with timers – they have to do with external interrupts and they will be discussed in section 2.10.16.

The TCON SFR has the following structure, as shown in Table 2-7. A new piece of information in this table is the column bit address. This is because this SFR is bit-addressable (note that the address of the SFR is divisible by 8, hence it is bit-addressable as mentioned earlier on).

Download free eBooks at bookboon.com

Bit-addressable						
Bit	**Name**	**Alternate Name (ASM)**	**Alternate Name (Keil C)**	**Bit Hex Address**	**Explanation Of Function**	**Timer Number**
7	TF1	TCON.7	TCON^7	8F	Timer 1 overflow flag. This bit is set by the micro-controller when timer register overflows.	1
6	TR1	TCON.6	TCON^6	8E	Timer 1 start/stop. Timer runs if this bit is set to 1.	1
5	TF0	TCON.5	TCON^5	8D	Timer 0 overflow flag. This bit is set by the micro-controller when timer register overflows.	0
4	TR0	TCON.4	TCON^4	8C	Timer 0 start/stop. Timer runs if this bit is set to 1.	0
The lower 4 bits have nothing to do with the timers as such and are not being listed here. They are used to detect and initiate external interrupts and they are discussed in a later section, under external interrupts (section 2.10.16).						

Table 2-7 TCON (88H) SFR

This bit-addressing capability means that if we want to set the bit TF1, which is the highest bit of TCON, instead of executing:

```
MOV TCON, #80h      ;(sets bit 7, and clears the other bits)
```

or

```
ORL TCON, #80h      ;(sets bit 7 only, without modifying the other bits)
```

we could just execute the command:

```
SETB TF1            ;(sets bit 7 only)
```

which is much more easy and user friendly.

This has the benefit of setting the high bit of TCON without changing the value of any of the other bits of the SFR and also it is more easily understood by anybody seeing the code. Usually when we start or stop a timer we do not want to modify the other values in TCON, so we take advantage of the fact that this SFR is bit-addressable.

Download free eBooks at bookboon.com

2.10.11 Initialising a Timer

Now that we have discussed the timer-related SFRs we are ready to write a piece of code that will initialise the timer and start it running.

As we will recall, we must first decide what mode we want the timer to be in. In this case let us suppose that we want a 16-bit timer that runs continuously; that is to say it is not dependent on any external pin condition.

We must first initialise the TMOD SFR. Assume we are working with timer 0. We will therefore be using the lowest 4 bits of TMOD. The first two bits, GATE0 and C/$\overline{T}$ are both 0 since we want the timer to be independent of the external pins. 16-bit mode is timer mode 1 so we must clear T0M1 and set T0M0. Effectively, the only bit we want to turn on is bit 0 of TMOD. Thus to initialise the timer we execute the instruction:

```
MOV TMOD,#01h       ; sets bit 0 and clears the other bits, hence affecting Timer 1 too.
```

or

```
ANL TMOD, #0F0h     ; clears the lower T0 mode control bits, leaving T1 bits unchanged
                    ; momentarily placing Timer 0 in mode 0.
ORL TMOD, #01h      ; sets bit 0 only (mode 1), leaving the other bits unchanged.
```

Click on the ad to read more

Download free eBooks at bookboon.com

Timer 0 is now in 16-bit timer mode. However, the timer is not running. To start the timer running we must set the TR0 bit and we can do that by executing the instruction:

```
SETB TR0;
```

Upon executing these instructions Timer 0 will immediately begin counting, being incremented once every machine cycle (every 12 clock pulses).

2.10.12 Reading the Timer registers

There are two common ways of reading the value of a 16-bit timer; which one we use depends on the application. We may either read the actual value of the timer as a 16-bit number, or we may simply detect when the timer has overflowed.

2.10.13 Reading the value of a Timer register

If our timer is in an 8-bit mode, that is either 8-bit auto-reload mode or in split timer mode, then reading the value of the timer is simple. We simply read the 1-byte value of the timer register (TLx or THx depending on the mode we are in) and we are done.

However, if we are dealing with a 13-bit or 16-bit timer this gets a little more complicated. Let us suppose that the timer registers are presently loaded with the values 14 and 255 (high byte 14, low byte 255). Consider what would happen if we read the low byte first then go on to read the high byte of the timer. It could well happen that we read the low byte of the timer as 255, then read the high byte of the timer as 15. Why? We correctly read the low byte as 255, but when we executed the next instruction a small amount of time would have passed, small but long enough for the timer to increment again at which time the values of the register pairs THx,TLx would have rolled over from 14, 255 to 15, 0. But in the process we would have wrongly read the timer registers as being 15,255 and this is a problem which may well lead to complete failure of our program.

The solution is not too tricky, really. We read the high byte of the timer, then read the low byte, then read the high byte again. If the high byte which we read the second time is not the same as the high byte read the first time we repeat the cycle, because we would conclude there was a roll-over. In assembly code, this would appear as:

Download free eBooks at bookboon.com

```
REPEAT:   MOV A, TH0          ; read THO and store it in the Accumulator
          MOV R0, TL0         ; read TL0 and store it register R0
          CJNE A, TH0, REPEAT ; compare the new TH0 with the previous
                              ; value and jump to REPEAT if not the same
              ……
              ……
```

In this case, we load the accumulator with the high byte of Timer 0. We then load R0 with the low byte of Timer 0. Finally, we check to see if the high byte that we read out of TH0 the first time, which is now stored in the Accumulator is the same as the current TH0 high byte, now read by the CJNE A, TH0, REPEAT instruction. If it is not the same, it means that we have just rolled over and must read again the timer values – which we do by going back to REPEAT. When the loop exits we will correctly have the low byte of the timer register (TL0) in R0 and the high byte (TH0) in the Accumulator.

Another much simpler alternative is to simply turn off the timer run bit (i.e. CLR TR0), read the timer values and then turn on the timer run bit (i.e. SETB TR0). In this case, the timer is not running whilst we are taking the readings, so no special precautions are necessary. Of course, this implies that our timer will be stopped for a few machine cycles. Whether or not this is tolerable to us depends on the specific application.

2.10.14 Detecting Timer Overflow

Often it is necessary to determine when the timer has reset to 0. That is to say, we are not particularly interested in the value of the timer but rather we are interested in knowing when the timer has overflowed and starts back to 0.

Whenever a timer overflows from its highest value back to 0, the micro-controller automatically sets the TFx bit or flag in the TCON register. This is useful since rather than checking the exact value of the timer we can just check if the TFx bit is set. If TF0 is set it means that timer 0 has overflowed; if TF1 is set it means that timer 1 has overflowed.

We can use this approach to cause the program to execute a fixed delay. We calculated earlier that it takes the 8051 1/20th of a second to count from 0 to 46,080. However, the TFx flag is set when the timer overflows back to 0. Thus, if we want to use the TFx flag to indicate when 1/20th of a second has passed we must set the timer initially to 65536 less 46080, or 19456. If we therefore set the timer to 19456 then 1/20th of a second later the timer will overflow. Thus we come up with the following code to execute a pause of 1/20th of a second:

Download free eBooks at bookboon.com

```
; 50 millisecond delay
MOV TH0, #76          ; High byte of 19456 (19456/256 = 76 exactly)
                      ; You can use MOV TH0, #HIGH(19456)
MOV TL0, #00          ; Low byte of 19456 (19456%256 = 0 i.e. no remainder after dividing)
                      ; You can use MOV TLO, #LOW(19456)
ANL TMOD, #0F0h       ; clears the lower T0 mode bits, leaving T1 bits unchanged
ORL TMOD, #01         ; Put Timer 0 in 16-bit mode
CLR TF0               ; Clear the overflow flag
SETB TR0              ; Start Timer 0 in order to begin counting
JNB TF0,$             ; If TF0 is not set, jump back
                      ; to this same instruction, that is
                      ; wait here until timer overflows, and 0.05s have passed
CLR TR0         ; switch off timer 0
PROCEED: ..
```

Click on the ad to read more

Download free eBooks at bookboon.com

In the above code the first two lines initialise the Timer 0 registers starting value to 19456. The next four instructions configure Timer 0, clear the overflow flag and turn the timer on. Finally, the last instruction JNB TF0,$ translates to "Jump, if TF0 is not set, back to this same instruction." The $ operand means, in most assemblers, the address of the current instruction. Thus as long as the timer has not overflowed and the TF0 bit has not been set the program will keep executing this same instruction. After 1/20th of a second timer 0 will overflow, setting the TF0 bit, and program execution will then break out of this one-line loop and continues at label PROCEED.

The program can easily be modified as shown below, to get the one second delay mentioned earlier.

```
; ONE SECOND DELAY
MOV R0, #20             ; We need to count the 50ms delay twenty times
ANL TMOD, #0F0h         ; clears the lower T0 mode bits, leaving T1 bits unchanged
ORL TMOD, #01           ; Put Timer 0 in 16-bit mode

DELAY_50MS:
MOV TH0, #76            ; High byte of 19456 (19456/256 = 76 exactly)
                        ; You can use MOV TH0, #HIGH(19456)
MOV TL0, #00            ; Low byte of 19456 (19456%256 = 0 i.e. no remainder after dividing)
                        ; You can use MOV TLO, #LOW(19456)
CLR TF0                 ; Clear the overflow flag
SETB TR0                ; Start Timer 0 in order to begin counting
JNB TF0,$               ; If TF0 is not set, jump back
                        ; to this same instruction, that is
                        ; wait here until timer overflows, and 0.05s have passed
CLR TR0                 ; switch off timer 0
DJNZ R0, DELAY_50MS     ; repeat 0.05s delay 20 times to get the one second delay
PROCEED: ..
```

2.10.15 Timing the length of events

The 8051 provides another important option that can be used to time the length of events.

Download free eBooks at bookboon.com

For example, let us say that we are trying to save electricity in the office and we are interested in how long a light is turned on each day. When the light is turned on, we want to measure the time that it is on. When the light is turned off we do not want to time it. One option would be to connect the light switch (voltage level suitably converted to the 0-5V DC range) to one of the pins, constantly read the pin, and turn the timer on or off using TR0 based on the state of that pin. While this would work fine, the 8051 provides us with an easier method of accomplishing this.

Looking again at the TMOD SFR, there is a bit called GATE. So far we have always cleared this bit because we wanted the timer to run regardless of the state of the external pins. However, now it would be nice if an external pin could control whether the timer was running or not. It can (see Figure 2-4). All we need to do is connect the light switch (having the voltage level suitably scaled down and rectified, since obviously we cannot apply 230V AC directly to the 8051 pin) to pin INT0 (P3.2) on the 8051 and set the bits GATE and TR0 to 1. When both the GATE and TR0 are set, Timer 0 will only run if P3.2 is high. When P3.2 is low (i.e., the light switch is off) the timer will automatically be stopped.

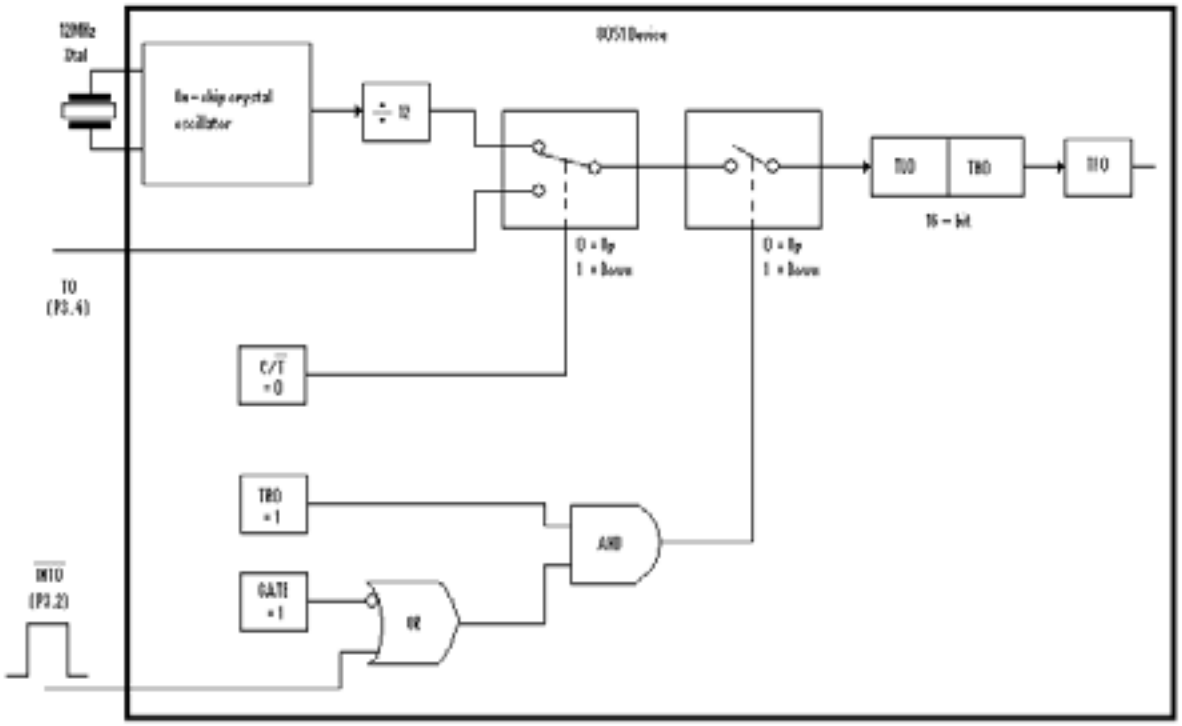


Figure 2-4 Timer 0 16-bit pulse-duration mode

Thus, with no control code whatsoever, the external pin P3.2 can control whether or not our timer is running. Naturally, our code would have to be adjusted so that we can then count also the number of overflows that have occurred so that at the end of the day we can add up the total time. This is explained in the next example.

Download free eBooks at bookboon.com

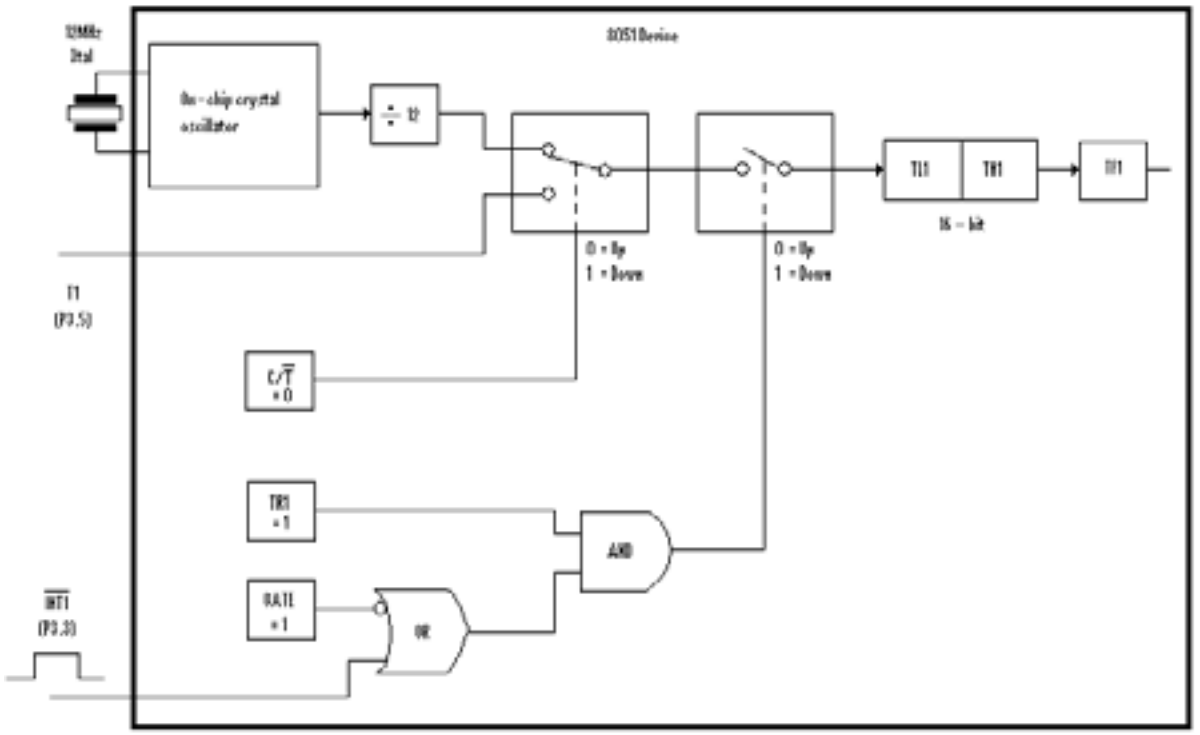


Figure 2-5 Timer 1 16-bit pulse duration mode

2.10.16 Using Timers to calculate a pulse or signal duration

The circuits shown in Figure 2-4 and in Figure 2-5 can also be used to calculate the duration of a positive pulse. Let us suppose we are using Timer 0 as a 16-bit timer (since Timer 1 would most probably be used for the serial port baud-rate generation). The positive pulse would be fed to pin P3.2 and the Gate and TR0 would be set to one. Thus due to the AND,OR logic gates the timer would effectively be operational only when there is a positive pulse on pin P3.2, and it would shut itself off as soon as the signal goes back down to zero. The counter registers TH0 and TL0 would have therefore counted the duration of the pulse. Assuming that we are using a crystal of 11.0592 MHz, and a timer counting rate of 1/12 this frequency, it would work out that every count is equivalent to 12/11.0592 microseconds (1.085 microseconds). To read the values of TH0,TL0 we would need to be monitoring the pulse so that we would know that it has finished. Thus a 16-bit timer starting from 0, would take 65536*1.085 microseconds or 71.11ms till it overflows.

It should be noted, that if the pulse lasts longer than approximately 71 ms, then the TH0,TL0 registers would overflow (setting TF0 =1) and the counter would continue counting from zero. Hence for longer pulses, the TF0 interrupt should be used so as to keep track of the number of overflows. Jumping to the interrupt service routine would automatically clear the TF0 flag for the next overflow interrupt.

Download free eBooks at bookboon.com

However, we can even use another facility which is provided by the 8051, so that we can automate this process even further, without the need to monitor or poll the signal to know when it has finished. Since the signal is being fed directly to pin P3.2, and since this 'happens' to be the external 0 (interrupt 0) pin, we can activate the EXT0 interrupt capability of the 8051, make it falling-edge triggered (setting IT0=1, see Table **2-8**), and thus the signal itself would generate an EXT0 interrupt when it falls back to zero. Thus we would have a timer counting whilst the signal is present, TF0 interrupt being used to trigger a routine so that we can count the number of overflows (if any occur during the duration of the pulse), and EXT0 interrupt to signal the end of pulse (using an appropriate Interrupt Service Routine (ISR) to read the registers and calculate the pulse duration).

Bit-addressable						
Bit	**Name**	**Alternate Name (ASM)**	**Alternate Name (Keil C)**	**Bit Hex Address**	**Explanation Of Function**	**Timer Number**
7	TF1	TCON.7	TCON^7	8F	Timer overflow. This bit is set by the micro-controller	1
6	TR1	TCON.6	TCON^6	8E	Start(1), Stop (0) timer	1
5	TF0	TCON.5	TCON^5	8D	Timer overflow. This bit is set by the micro-controller	0
4	TR0	TCON.4	TCON^4	8C	Start(1), Stop (0) timer	0
3	IE1	TCON.3	TCON^3	8B	Ext Interrupt flag. Set/ cleared by hardware	1
2	IT1	TCON.2	TCON^2	8A	Falling edege (1), low level (0) triggering selection	1
1	IE0	TCON.1	TCON^1	89	Ext Interrupt flag. Set/ cleared by hardware	0
0	IT0	TCON.0	TCON^0	88	Falling edege (1), low level (0) triggering selection	0
The lower 4 bits are used to detect and initiate external interrupts.						

Table 2-8 TCON (88H) SFR

Here is an example of the important routines written in assembly language. The KEIL IDE provides all the necessary explanations for the keywords used, such as SEGMENT, RSEG etc and the reader is urged to consult the KEIL IDE package for further details.

Download free eBooks at bookboon.com

```
; PulseT0.a51
; Test

MyBitData SEGMENT BIT
RSEG MyBitData
PULSE_OK: DBIT 1

MyData SEGMENT DATA
RSEG MyData

TIMER_OVF: DS 1

TIMER_OV: DS 1        ; values for use in other section of the program
TIMER_HI: DS 1        ; values for use in other section of the program
TIMER_LO: DS 1        ; values for use in other section of the program

MyCode SEGMENT CODE
RSEG MyCode
        LCALL Main

ORG 0003H     ; external 0 ivt
        LJMP EXT0_ISR

ORG 000BH     ; timer 0 ivt
        LJMP TIM0_ISR

ORG 30H
```


Download free eBooks at bookboon.com

```
Main:
; First clear the 8032 Internal RAM (from 0 to FFH)
        CLR A
        MOV R0, #0FFH
CLR_RAM:
        MOV @R0, A
        DJNZ R0, CLR_RAM

; next set up the stack pointer
        MOV SP, #2FH

; Program starts here
; set up timer 0 for pulse width counting
        ANL TMOD, #0F0H
        ORL TMOD, #09H ; set 16-bit timer mode, gate = 1
        MOV TH0, #0
        MOV TL0, #0 ; reset 16-bit counter
        CLR TF0 ; clear the timer overflow flag
        SETB IT0 ; falling-edge triggered ext0 interrupt
        SETB P3.2 ; set p3.2 for input
        SETB TR0 ; prepare timer to count whenever P3.2 is high
        SETB EX0 ; enable external zero interrupts
        SETB ET0 ; enable timer 0 interrupts
        SETB EA ; enable global interrupts

; display pulse width values
; the number of overflows, the value of TH0 and the value of TL0
; at the end of the pulse.
; the actual duration of the pulse would be
; (65536*timer_ov + 256*TH0 + TL0) * 1.085 microseconds
;
; assuming that we have a routine called disp_reg which
; displays on screen the 8-bit value of register r0, the
; program stays looping in this loop_again routine
; prints values only if bit pulse_ok is set by the end of pulse isr
```


Download free eBooks at bookboon.com

```
LOOP_AGAIN:
        JNB PULSE_OK, LOOP_AGAIN
        MOV R0,TIMER_OV
        LCALL DISP_REG
        MOV R0,TIMER_HI
        LCALL DISP_REG
        MOV R0,TIMER_LO
        LCALL DISP_REG
        CLR PULSE_OK
        SJMP LOOP_AGAIN

; this routine executes only every ext0 interrupt,
; that is when the pulse has just ended
; the main program would be halted and would continue only
; after this routine has finished
EXT0_ISR:
        MOV TIMER_OV, TIMER_OVF         ; store values
        MOV TIMER_HI, TH0
        MOV TIMER_LO, TL0

        MOV TIMER_OVF, #0               ; reset counters
        MOV TH0, #0
        MOV TL0, #0
        SETB PULSE_OK ; set flag indicating pulse time ready for printing
        RETI
; this interrupt service routine executes when timer0 overflows
TIM0_ISR:
        INC TIMER_OVF ; increment the overflow counter
        RETI

        end
```


Download free eBooks at bookboon.com

And here is the same example this time written in C:

```
/* Test program PULSE */

#include <reg52.h>
#include <absacc.h>
#include <stdio.h>
#include "SerP3Pkg.h" // used for setting the UART for serial input/output

// Variables
unsigned int Timer_OVF, Timer_Overflows;
unsigned long Timer_CNT;
unsigned char Timer_HI, Timer_LO;
bit PulseOK;

void init_timer0(void)
{
   TMOD &= 0xF0;                  // clear timer 0 bits, leaving timer 1 as it was set
   TMOD |= 0x09;                  // timer 0 as a 16-bit timer, GATE = 1
   P3 |= 0x04; // set P3.2 for input
   TF0 = 0;                       // clear the overflow flag
   TR0 = 1;                       // timer 0 ready to count, whenever pin P3.2 is a 1 (pulse present)
   IT0 = 1;                       // external 0 interrupt falling-edge triggered (pulse just off)
   EX0 = 1;                       // enable external 0 interrupts
   ET0 = 1;                       // enable timer 0 interrupts
   EA = 1;                        // enable global interrupts
}

// This ISR executes when the pulse has just ended
void ext0_isr (void) interrupt 0 using 1
{
   Timer_OVF = Timer_Overflows;          /* save all timer readings */
   Timer_HI = TH0;
   Timer_LO = TL0;
   Timer_CNT = Timer_OVF*65536 + Timer_HI*256 + TL0;
   TH0 = TL0 = Timer_Overflows = 0;      /* reset all timer readings */
   PulseOK = 1; /* indicates that a NEW pulse reading has been taken */
}
```


Download free eBooks at bookboon.com

```
// This ISR executes when timer 0 overflows (TF0 =1)
// TF0 is cleared (reset to 0) automatically when using interrupts
void tf0_isr (void) interrupt 1 using 2
{
   Timer_Overflows++;
}

void main(void)
{
        init_serial(57600);
        init_timer0();
        printf("\n\n Pulse Duration Timing\n\n\r");
        printf("\n\nToggle P3.2 to simulate pulse.\n\n\r");
        printf(" Timer Overflows TH0 TL0 Total Counts Microseconds\n\r");
   while (1)
        {
        if (PulseOK==1) {
                printf(" %05u %03bu %03bu %010lu %12.1f\r",
                        Timer_OVF,
                        Timer_HI,
                        Timer_LO,
                        Timer_CNT,
                        1.0851*Timer_CNT); /* 12/11.0592 = 1.0851 */
                PulseOK = 0;
        }
   }
}
```

And finally here is yet again the same example written in C, but this time making use of the UNION to make calculations even simpler.

Download free eBooks at bookboon.com

```
/* Test program PULSE2 */

#include <reg52.h>
#include <absacc.h>
#include <stdio.h>
#include "SerP3Pkg.h"

/* types */
typedef union UTYPELONG {
        unsigned long Long;
        unsigned int Int[2];
        unsigned char Char[4];
}UTYPELONG;

// Variables

unsigned int Timer_Overflows;
UTYPELONG Timer_CNT;
bit PulseOK;

void init_timer0(void)
{
   TMOD &= 0xF0;      // clear timer 0 bits, leaving timer 1 as it was set
   TMOD |= 0x09;      // timer 0 as a 16-bit timer, GATE = 1
   P3 |= 0x04;        // set P3.2 for input
   TF0 = 0;           // clear timer overflow flag
   TR0 = 1;           // timer 0 ready to count, whenever pin P3.2 is a 1 (pulse present)
   IT0 = 1;           // external 0 interrupt falling-edge triggered (pulse just off)
   EX0 = 1;           // enable external 0 interrupts
   ET0 = 1;           // enable timer 0 interrupts
   EA = 1;            // enable global interrupts
}
```


Download free eBooks at bookboon.com

```
void ext0_isr (void) interrupt 0 using 1
{
   Timer_CNT.Int[0] = Timer_Overflows;        /* save all timer readings */
   Timer_CNT.Char[2] = TH0;
   Timer_CNT.Char[3] = TL0;                   /* NOTE the Big Endian storage style */
   TH0 = TL0 = Timer_Overflows = 0;           /* reset all timer readings */
   PulseOK = 1;          /* indicates that a NEW pulse reading has been taken */
}

// This ISR executes when Timer 0 overflows (TF0=1)
// TF0 is cleared (reset to 0) automatically when interrupts are used
void tf0_isr (void) interrupt 1 using 2
{
   Timer_Overflows++;
}

void main(void)
{
        init_serial(57600);                   /* set up UART */
        init_timer0();                        /* set up timer 0 */

        printf("\n\n                Pulse Duration Timing\n\n\r");
        printf("\n\nToggle P3.2 to simulate pulse.\n\n\r");
        printf("Timer Overflows TH0 TL0 Total Counts Microseconds\n\r");

   while (1)
        {
        if (PulseOK == 1) {
                printf(" %05u %03bu %03bu %010lu %12.1f\r",
                        Timer_CNT.Int[0],
                        Timer_CNT.Char[2],
                        Timer_CNT.Char[3],
                        Timer_CNT.Long,
                        1.0851*Timer_CNT.Long); /* 12/(11.0592) = 1.0851 */
                PulseOK = 0;
        }
   }
}
```

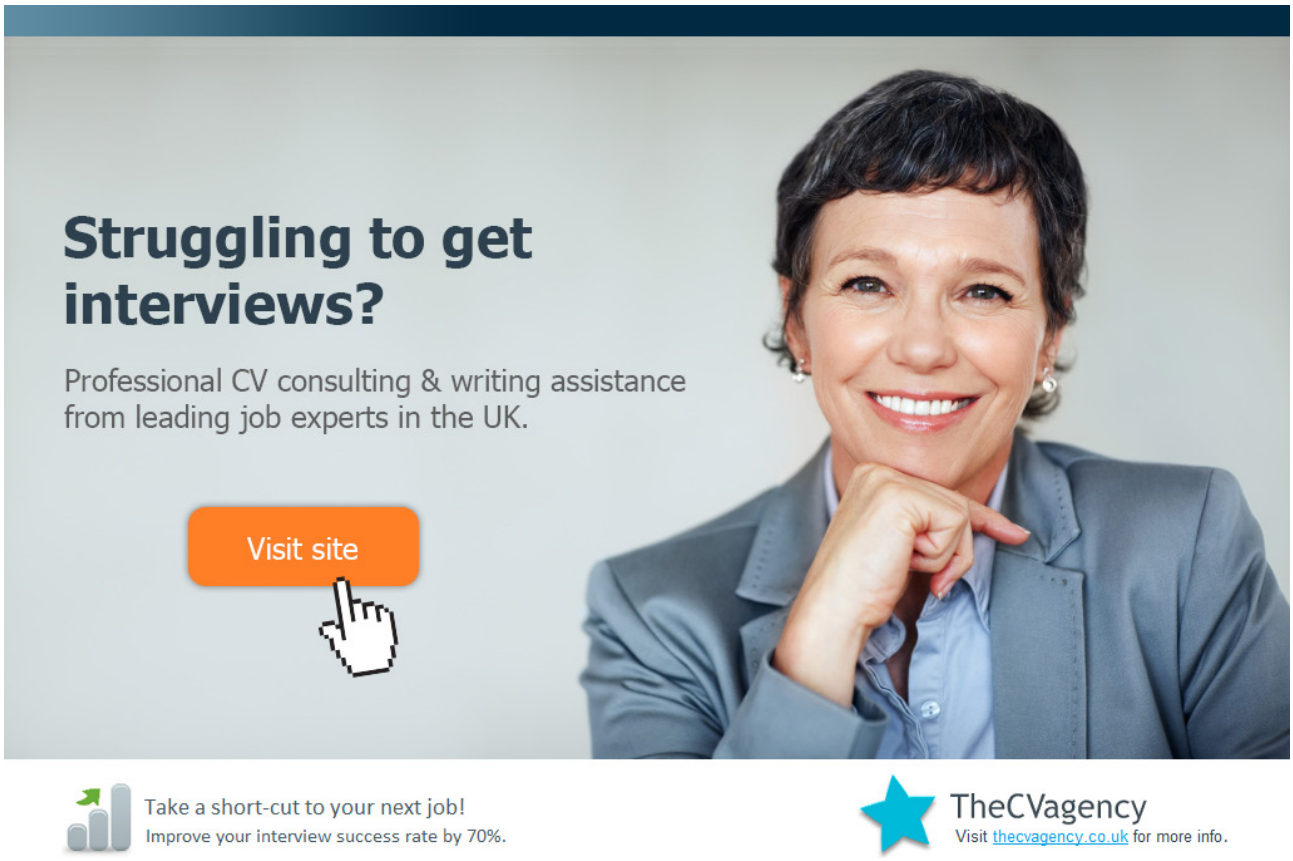

Download free eBooks at bookboon.com

2.10.17 Using Timers as event counters

We have discussed how a timer can be used for the obvious purpose of keeping track of time. However, the 8051 also allows us to use the timers to count events.

How can this be useful? Let us say we had a sensor placed across a road that would send a pulse every time a car wheel passed over it. This could be used to determine the volume of traffic on the road. We could attach this sensor to one of the 8051's I/O lines and constantly monitor it, detecting when it pulsed high and then incrementing our counter when it went back to a low state. This is not terribly difficult, but requires some code. Let us say we hooked the sensor to P1.0; the code to count cars passing would look something like this:

```
CAR:
JNB P1.0,$          ; If a car has not raised the signal, keep waiting
JB P1.0,$           ; The line is high which means the car is on the sensor right now
                    ; hence wait here until the car passes
INC COUNTER         ; The wheel has passed completely, so we count it
SJMP CAR            ; go back to wait for another car
```



Download free eBooks at bookboon.com

As we can see, it is only four lines of code. But what if we need to be doing other processing at the same time? We do not want to be stuck in the JNB P1.0,$ loop waiting for a car to pass if we need to be doing other things. Of course, there are ways to get around this limitation but the code quickly becomes big, complex, and ugly.

Luckily, since the 8051 provides us with a way to use the timers to count events we do not have to bother with it. It is actually easy since we only have to configure one additional bit.

Let us say we want to use Timer 0 to count the number of cars that pass. If we look back to the bit table for the TMOD SFR (see Table 2-4 and Figure 2-4) we will see that there is a bit called "C/$\overline{T0}$" – it is bit 2 (TMOD.2). Reviewing the explanation of the bit we see that if the bit is cleared then timer 0 will be incremented at every machine cycle, using the crystal oscillator. This is what we have already used in order to measure time. However, if we set C/$\overline{T0}$ to 1, then timer 0 will monitor the P3.4 line. Instead of being incremented every machine cycle, timer 0 will count events (pulses) on the P3.4 line. So in our case we simply connect our sensor to P3.4 and let the 8051 do the work. Then, when we want to know how many cars have passed, we just read the value of timer 0 registers TL0 and TH0. This value of timer 0 will be the number of wheels that have passed. If we expect more than 65535 pulses, then we would also need to take care of how many overflows have taken place, but this too is easy since the overflows are indicated by TF0 bit being set. This can also be programmed to cause an interrupt and hence the TF0 interrupt routine simply counts the number of overflows automatically. Each overflow would indicate that 65536 wheels have passed. For this setup, TR0 is set to 1 and the GATE is set to 0. Thus TMOD = xxxx0101, setting Timer 0 in 16-bit mode.

So what exactly is an event? What does timer 0 actually count? Speaking at the electrical level, the 8051 counts 1-0 (high to low) transitions on the P3.4 line. This means that when a wheel first runs over our sensor it will raise the input to a high ("1") condition. At that point the 8051 will not count anything since this is a 0-1 transition. However, when the car wheel has passed the sensor, the input will fall back to a low ("0") state. This is a 1-0 transition and at that instant the counter will be incremented by 1. If we are really counting cars, the final value would obviously have to be divided by two since each car has got front and rear wheels, with both pairs triggering the timer count.

It is important to note that the 8051 checks the P3.4 line each instruction cycle (12 clock cycles). This means that if P3.4 is low, goes high, and goes back low in say 6 clock cycles it will probably not be detected by the 8051. This also means the 8051 event counter is only capable of counting events that occur at a maximum of 1/24th the rate of the crystal frequency. That is to say, if the crystal frequency is 12.000 MHz it can detect a maximum of 500,000 events per second (12.000 MHz * 1/24 = 500,000), even though the timer itself works at twice that frequency. If the event being counted occurs more frequently than 500,000 times per second it will not be able to be accurately detected and counted by the 8051.

Download free eBooks at bookboon.com

2.11 Serial Port Operation

One of the 8051's many powerful features is its integrated UART, otherwise known as a serial port. The fact that the 8051 has an integrated serial port means that we may very easily read and write values to the serial port. If it were not for the integrated serial port, writing a byte to a serial line would be a rather tedious process requiring turning on and off one of the I/O lines in rapid succession to properly "clock out" each individual bit, including start bits, stop bits, and parity bits. The clocking out of the bits has to be done at the pre-defined speed or baud rate.

However, we do not have to do this. Instead, we simply need to configure the serial port's operating mode and baud rate. Once configured, all we have to do is write to an SFR (SBUF) to transmit a value from the serial port (through the TXD pin P3.1) or read the same SFR to get the received value from the serial port (using the RXD pin P3.0). The 8051 will automatically let us know when it has finished sending the character we wrote and will also let us know whenever it has received a byte so that we can process it. We do not have to worry about transmission at the bit level, which saves us quite a bit of coding and processing time.

2.11.1 Setting the Serial Port Mode

The first thing we must do when using the 8051's integrated serial port is, obviously, configure it. This instructs the 8051 about how many data bits we want, the baud rate we will be using, and how the baud rate will be determined.

Click on the ad to read more

Download free eBooks at bookboon.com

First, let us present the "Serial Control" (SCON) SFR and define what each bit of the SFR represents:

Bit-addressable					
Bit	**Name**	**Alternate Name (ASM)**	**Alternate Name (Keil C)**	**Bit Hex Address**	**Explanation of Function**
7	SM0	SCON.7	SCON^7	9F	Serial port mode bit 0
6	SM1	SCON.6	SCON^6	9E	Serial port mode bit 1
5	SM2	SCON.5	SCON^5	9D	Multiprocessor comms enable
4	REN	SCON.4	SCON^4	9C	Receiver enable. This bit must be set in order to receive characters.
3	TB8	SCON.3	SCON^3	9B	Transmit bit 8. The 9^{th} bit which is transmitted when operating in mode 2 or mode 3
2	RB8	SCON.2	SCON^2	9A	Receive bit 8. The 9^{th} bit which is received when operating in mode 2 or mode 3
1	TI	SCON.1	SCON^1	99	Transmit flag. Set when a byte has been completely transmitted. Will cause a serial interrupt if the interrupts are enabled. Must be cleared by software. Can also be set by software to signal that the transmitter is ready.
0	RI	SCON.0	SCON^0	98	Receive flag. Set when a byte has been completely received. Will cause a serial interrupt if the interrupts are enabled. Must be cleared by software.

Table 2-9 SCON (99H) SFR

It is necessary to define the function of SM0 and SM1 as in Table 2-10:

SM0	SM1	Serial Mode	Explanation	Baud Rate Clock
0	0	0	8-bit shift register	Oscillator/12
0	1	1	8-bit UART	Set by timer 1 (*)
1	0	2	9-bit UART	Oscillator/32 (*)
1	1	3	9-bit UART	Set by timer 1 (*)

(*) Note: The baud rate indicated in this table is doubled if PCON.7 (SMOD) is set.

Table 2-10 Serial Mode selection bits

The SCON SFR allows us to configure the Serial Port. Thus, we will go through each bit and review its function. The higher four bits (bits 4 through 7) are the configuration bits.

Download free eBooks at bookboon.com

Bits SM0 and SM1 set the serial mode to a value between 0 and 3 inclusive. The four modes are defined in Table 2-10. As we can see, selecting the Serial Mode selects the mode of operation (8-bit/9-bit, UART or Shift Register) and also determines how the baud rate will be calculated. In modes 0 and 2 the baud rate is fixed based on the oscillator's frequency. In modes 1 and 3 the baud rate is variable, based on how often Timer 1 overflows. We will talk more about the various Serial Modes in a moment.

The next bit, SM2, is a flag for "Multiprocessor communication." Generally, whenever a byte has been received the 8051 will set the "RI" (Receiver Interrupt) flag. This lets the program know that a byte has been received and that it needs to be processed. However when SM2 is set, the "RI" flag will only be triggered if the 9th bit received was a "1". That is to say, if SM2 is set and a byte is received whose 9th bit is cleared, the RI flag will never be set. This can be useful in certain advanced serial applications, where we need to communicate with one out of many microcontrollers. (see Master-Slave section 2.12.4 below). For now it is safe to say that we will almost always want to clear this bit so that the RI flag is set upon reception of any character.

The next bit REN is "Receiver Enable." This bit is very straightforward; if you want to receive data via the serial port, set this bit. We will almost always want to set this bit.

The lower four bits (bits 0 through 3) are operational bits. They are used when actually sending and receiving data, they are not used to configure the serial port.

The TB8 bit is used in modes 2 and 3. In these mode a total of nine data bits are transmitted. The first 8 bits are the 8 bits of the actual data to be transmitted (taken from SBUF), and the ninth bit is taken from TB8. If TB8 is set (1) and a value is written to the serial port, the data bits will be written to the serial line followed by a "set (1)" ninth bit. If TB8 is cleared the ninth bit will be "cleared (0)". (see Master-Slave section 2.12.4 below).

The RB8 also operates in modes 2 and 3 and functions essentially the same way as TB8, but on the reception side. When a byte is received in modes 2 or 3, a total of nine bits are received (from the RXD pin P3.0). In this case, the first eight bits received are the data of the serial byte received (stored in SBUF) and the value of the ninth bit received will be placed in RB8. (see 2.12.4).

TI means "Transmitter Interrupt." When a program writes a value to the serial port, a certain amount of time will pass before the individual bits of the byte are "clocked out" or transmitted out of the serial port (TXD pin P3.1). If the program were to write another byte to the serial port before the first byte was completely sent, the data being sent would be garbled. Thus, the 8051 lets the program know that it has "clocked out" the last bit by setting the TI bit. When the TI bit is set, the program may assume that the serial port is "free" and ready to send the next byte.

Download free eBooks at bookboon.com

Finally, RI means "Receiver Interrupt." It functions similarly to the "TI" bit, but it indicates that a byte has been received. That is to say, whenever the 8051 has received a complete byte it will trigger the RI bit to let the program know that it needs to read the value quickly, before another byte is read, which would overwrite the one just received.

2.11.2 Setting the Serial Port Baud Rate

Once the Serial Port Mode has been properly configured as explained above, the program must configure the serial port's baud rate. This only applies to Serial Port modes 1 and 3. The Baud Rate is determined based on the oscillator's frequency when in mode 0 and 2.

In mode 0, the baud rate is always the oscillator frequency divided by 12. This means if your crystal is 11.0592 MHz, mode 0 baud rate will always be 921,583 baud.

In mode 2 the baud rate is the oscillator frequency divided by 32 (if SMOD [PCON.7] = 1) or 64, (if SMOD [PCON.7] = 0), so an 11.0592 MHz crystal frequency will yield a baud rate of 345,600 or 172,800.

In modes 1 and 3, the baud rate is determined by how frequently timer 1 overflows. It is this timer 1 overflow frequency which is divided either by 16 or by 32 (again depending on SMOD) to give the required baud rate. The more frequently timer 1 overflows, the higher the baud rate. There are many ways one can cause timer 1 to overflow at a rate that determines a baud rate, but the most common method is to put timer 1 in the 8-bit auto-reload mode (timer 1 mode 2 as already seen in Figure 2-3) and set a reload value (TH1) that causes Timer 1 to overflow at a frequency appropriate to generate one of the standard baud rates. That is, the timer must overflow at 32 (SMOD=0) or 16 (SMOD=1) times the required baud rate, or in other words, the time it takes the timer to overflow must be equal to 1/32nd (or 1/16th) the time of one serial bit.

To determine the value that must be placed in TH1 to generate a given baud rate, we may use the following equation (assuming PCON.7 is cleared, that is we are dividing by 32).

Oscillator freq. = (Crystal freq. / 12)

Time for one timer count = 1/(Osc. Freq.)
= (12/Crystal freq.) seconds

Since PCON.7 is assumed to be zero, the pulses coming from the timer overflow are divided by 32 to give the correct baud rate. We must therefore ensure that we have (32*Baud Rate) overflows every second.

Time to overflow = 1/(32 * Baud Rate) seconds

Download free eBooks at bookboon.com

Thus, we need to determine how many counts are needed to get this overflow time. By simple proportion we can work it out very easily. If we have one count in (12/Crystal freq.) seconds, how many counts do we have in 1/(32 * Baud Rate) seconds. The answer is obviously

$$\frac{1}{32*Baud\ Rate} * \frac{Crystsal\ frequency}{12} \text{ counts}$$

Since the timers always count up, we must start off the timer with register TH1 set at this value below the top, or:

$$TH1 = 256 - ((Crystal\ freq.) / (32 * 12 * Baud\ Rate))$$

i.e.

$$TH1 = 256 - ((Crystal\ freq.) / (384 * Baud\ Rate)) \text{ ..... } \textbf{Equation 2-1}$$

If PCON.7 is set, then the divisor is set to 16 and not to 32 and the baud rate is effectively doubled, thus the equation becomes:

$$TH1 = 256 - ((Crystal\ freq.) / (192 * Baud\ Rate)) \text{ ..... } \textbf{Equation 2-2}$$

Click on the ad to read more

Download free eBooks at bookboon.com

For example, suppose we have an 11.0592 MHz crystal and we want to configure the serial port to 19,200 baud. Using Equation 2.1:

TH1 = 256 – ((11059200 / 384) / 19200)

TH1 = 256 – (28800 / 19200)

TH1 = 256 – 1.5 = 254.5

This is not an integer and therefore not possible to set correctly.

If we set TH1 to 254 we will get 14,400 baud and if we set it to 255 we will get 28,800 baud. It looks like we are stuck but there is a solution.

To achieve 19,200 baud we simply need to set PCON.7 (SMOD) to 1. When we do this, we double the baud rate and use equation 2-2. Thus we get:

TH1 = 256 – ((11059200 / 192) / 19200)

TH1 = 256 – ((57600) / 19200)

TH1 = 256 – 3 = 253

Here we get an exact integer TH1 value. Therefore, to obtain 19,200 baud with an 11.0592 MHz crystal we must:

- Configure Serial Port mode 1 (8-bit variable baudrate) or 3 (9-bit variable baudrate).
- Configure Timer 1 to timer mode 2 (8-bit auto-reload).
- Set TH1 and TL1 to 253
- Set PCON.7 (SMOD) to double the baud rate.

This is in fact the reason why the oddly numbered frequency of 11.0592 MHz is chosen. This will ensure that these calculations would always give an integer value for TH1 for the standard baud rates, as shown in Table 2-11 below. This table compares some values with another crystal frequency of 12 MHz (also used often since it results in timers incrementing every one microsecond).

Download free eBooks at bookboon.com

Target Baud Rate	Crystal Frequency MHz	PCON.7 PCON^7 SMOD	TH1 Reload Value	Actual Baud Rate	Error (%)
9600	12	1	249 (F9H) (-7)	8923	7
2400	12	0	243 (F3H) (-13)	2404	0.16
1200	12	0	230 (E6H) (-26)	1202	0.16
57600	11.0592	1	255 (FFH) (-1)	57600	0
19200	11.0592	1	253 (FDH) (-3)	19200	0
9600	11.0592	0	253 (FDH) (-3)	9600	0
2400	11.0592	0	244 (F4H) (-12)	2400	0
1200	11.0592	0	232 (E8H) (-24)	1200	0

Table 2-11 Baud Rate calculation

With the standard 11.0592 MHz crystal, the equations for calculating TH1, can be simplified to:

TH1 = 256 – (28800 / Baud Rate) if SMOD = 0 ..... **Equation 23**

TH1 = 256 – (57600 / Baud Rate) if SMOD = 1 ..... **Equation 24**

or

Baud Rate = 28800 / (256 – TH1) if SMOD = 0 ..... **Equation 25**

Baud Rate = 57600 / (256 – TH1) if SMOD = 1 ..... **Equation 26**

Once the Serial Port has been properly configured as explained above, it is ready to be used to send and receive data.

To write a byte to the serial port we must simply write the value to the SBUF (99h) SFR. For example, if we want to send the letter "A" (the 8-bit ASCII code of the letter A is 65 decimal) to the serial port, it could be accomplished simply by loading the serial buffer register SBUF:

```
MOV SBUF, #"A" or MOV SBUF, #65
```

Upon execution of the above instruction the 8051 will begin transmitting the character via the serial port, starting with the low Start bit, bit 0 to bit 7 of the actual data, followed by a high Stop bit). Obviously transmission is not instantaneous – it takes a measurable amount of time to transmit. Since the 8051 does not have a serial output buffer we need to be sure that a character is completely transmitted before we try to transmit the next character by loading a new value into SBUF.

Download free eBooks at bookboon.com

The 8051 lets us know when it is done transmitting a character by setting the TI bit in SCON. When this bit is set we know that the last character has been transmitted and that we may send the next character, if any. Consider the following code segment:

```
        CLR TI              ; Be sure the bit is initially cleared
        MOV SBUF, #"A"      ; Start sending the letter "A" via the serial port
HERE:   JNB TI, HERE        ; Wait here until the transmission is done, namely TI bit is set.
```

The above three instructions will successfully transmit a character and wait for the TI bit to be set before continuing. Thus the 8051 will pause on the JNB instruction until the TI bit is set by the 8051 upon successful transmission of the character. We can then transmit another character.

2.11.3 Reading the Serial Port

Reading data received by the serial port is equally easy. To read a byte from the serial port we just need to read the value stored in the SBUF (99h) SFR after the 8051 has automatically set the RI flag in SCON to indicate that a character has just been received.

For example, if our program wants to **wait** for a character to be received and subsequently read it into the Accumulator, the following code segment may be used:

Click on the ad to read more

Download free eBooks at bookboon.com

```
HERE: JNB RI, HERE      ; Wait here for the 8051 to set the RI flag
                        ; (wait for the reception of a character)
      MOV A, SBUF       ; Read the character from the serial port
      CLR RI            ; clear RI, ready for the next character to be received
```

The first line of the above code segment waits for the 8051 to set the RI flag; again, the 8051 sets the RI flag automatically when it receives a character via the serial port. So as long as the bit is not set the program repeats the "HERE: JNB RI, HERE" instruction continuously.

Once the RI bit is set upon character reception the above condition automatically fails and program flow falls through to the "MOV A, SBUF" instruction which reads the value and stores it in the accumulator. The RI flag is finally cleared so as to be ready for the next possible character to be received.

Section 3.2 describes a complete serial port example for transmitting and receiving text.

2.11.4 Master-Slave Operation

Mode 2 or mode 3, (9-bit mode), is generally used whenever inter micro-controller communication is desired. Generally speaking, mode 2 is used for high speed communications (up to 345600 baud with an 11.0592 MHz crystal, without using timers), and mode 3 is used when standard baud rates (using timer 1) are required.

For 9-bit mode 2, the baud rates are determined directly by the crystal frequency (assumed 11.0592 MHz) and the value of SMOD (PCON.7) as shown in the Table 2-12.

SMOD	Crystal divisor	Baud Rate (=Xtal/divisor)
0	64	172800
1	32	345600

Table 2-12 Crystal Divisor

When using the 9-bit mode, one micro-controller is generally configured to act as the master controller, with up to 256 other slave micro-controllers. They can be connected in a 3-wire setup, for two-way communication as shown in Figure 2-6, which shows the setup for a master board with two slave boards.

Download free eBooks at bookboon.com

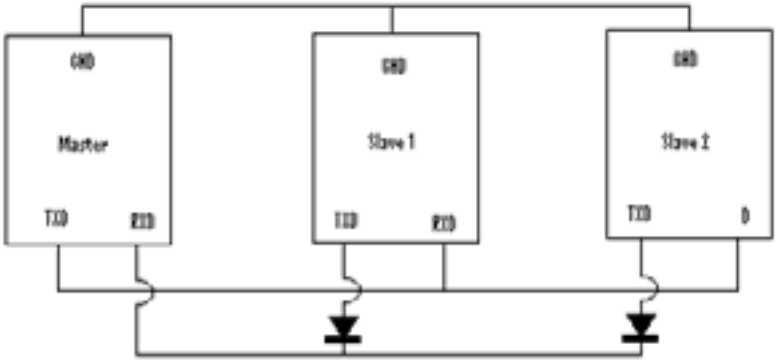


Figure 2-6 Master-slaves connection

The diodes are needed since all the slave TXD lines of the slaves are connected together and they are acting as output lines, feeding into the RXD line of the Master. Therefore one slave cannot send data ***into*** another TXD line of another slave and the diodes ensure that the TXD lines act only as output lines and do not sink any other signals. The switching time of these diodes will restrict the transmission speed of the slaves to the master.

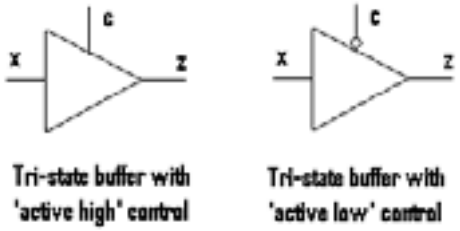


Figure 2-7 Tri-state buffers

Instead of these diodes, we can also use tri-state buffers (see Figure 2-7) which have the capability (when not in operation) to offer a high-impedance. Thus when a slave is not transmitting, it disables the tri-state buffer so that its TXD line is effectively isolated from the network. When a slave wants to transmit, it would enable the tri-state buffer so that the signal present on its TXD line is transferred on to the network. It would disable the tri-state buffer once it has finished with the transmission. In order to control the tri-state buffer, an additional pin from some port of the slave micro-controller has to be used in order to enable/disable the tri-state buffer. Depending on whether it is using active high or low control, a 1 or a 0 at the port pin would enable the buffer.

c	z
0	Z
1	x

Table 2-13 Tri-state truth table

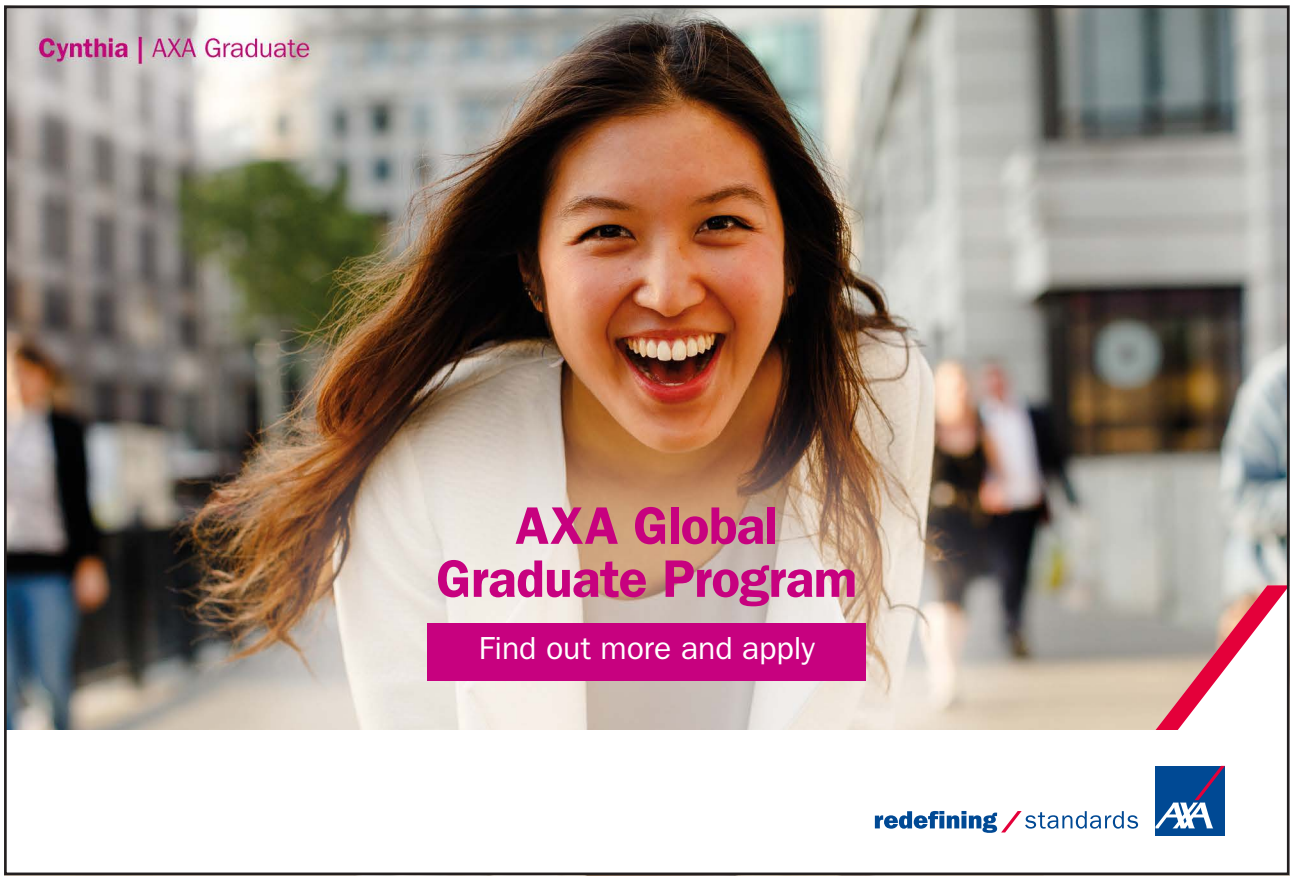
Download free eBooks at bookboon.com

As it can be seen in Table 2-13, when **c = 1** the tri-state active-high device is active and **z = x**, that is the output z connected to the network would reflect the TXD signal at x. When **c = 0** the tri-state device is not active, and **z = Z** (e.g., high impedance/no current).

If the slave units are not required to communicate back or acknowledge, then the diodes and their TXD lines can be left unconnected, and we just have a two-wire set up, with just the Ground and the TXD line from the master to the RXD pin of all the slave units.

Use is made of SM2, TB8 and RB8 as explained in the flow sequence below, which explains in detail how this master-slave communication can be achieved.

- All slave units have their UART serial device set to work under interrupt control, and with SM2 initially set 1. This means that their UART interrupt service routine (ISR) will only be called when the 9th received bit (bit number 8, since we normally count from bit 0) is a 1.
- All devices are set in 9-bit mode 2 or mode 3, depending on the required baud rates. Mode 3 is the most commonly used, especially if the diodes are being used. A slow baudrate would be required otherwise the diodes would not have enough time to pass through the data. TB8 is initially cleared, set to 0.
- All slave units are given (by means of a software #define statement) a particular unique 8-bit address (0–255).



Click on the ad to read more

Download free eBooks at bookboon.com

- The master starts a transmission (not necessarily under interrupt control) by sending an address corresponding to the slave to which it wants to send data. When sending this address, the master sets its own TB8 bit to 1, so that the master is effectively adding on a '1' bit to the address it is sending. SM2 for the master is left cleared (=0).
- All slave units receive this address with the extra '1' bit and since they would at this stage all have their own SM2 bit set to 1, each one of the slave units would get a serial RI interrupt. The ISR would be activated and each slave unit would receive, read and check the address to see whether the master is intending to communicate with it.
- Only the slave unit whose address corresponds to the received address would be taking further action. The other slave units would simple return from their own ISR without doing or changing anything. They would simply wait for another address (with a 9th '1' bit) to be received.
- The slave unit with the correct address would now set its own SM2 to 0, so that from now on and until SM2 is changed again to 1, its own serial ISR would come into action for every data byte sent by the master (even if with a 9th bit of '0').
- The master, after sending the slave address, depending on the software algorithm could:
  - Either wait for an acknowledgement from the addressed slave.
  - Or just wait a while to give time for the slave to change its setup (mainly setting SM2 to 0).
- The master, after this waiting period, would set its own TB8 to 0 so that when sending the data over, it will affix a '0' at the end of each data byte, so as not to be interpreted as an address and trigger the serial interrupt of the other slave units. Some pre-arranged 'end of data' character would be sent at the end of all the data, as an indication to the addressed slave unit that no more data is going to be sent.
- Only the addressed slave unit would be interrupted to receive this data, since it would be the only slave controller with its SM2 reset to 0. The other slaves would not even notice that there is data passing, since their RI bit would not be set with bytes having a 9th bit of '0', and hence their own ISR would not be triggered.
- When the addressed slave unit receives the 'end of data' marker, it would once again revert back to the original mode, by setting SM2 to 1, and the transmission would pause. All the slave units would now once again be waiting for an address to be sent by the master board.
- Naturally, whilst waiting for the serial ISR to be activated, the slave units could be executing some other code for their particular application, rather than staying idle.

In general, address 255 is reserved for a 'general call' to be used whenever the master needs to send data to ALL slave units (such as an emergency switch off). Every slave unit would programmed to recognise this address, and all slave units would then switch their SM2 to 0 and react to this general transmission. No acknowledgement is sent by the slave units, otherwise there would simply be rubbish on the TxD line since every slave unit would be transmitting the acknowledgement at the same time.

Download free eBooks at bookboon.com

We can also have special group addresses so that the master can send data simultaneously to a group of slave units, again without any acknowledgement coming from them.

This 3-wire multiprocessor communication is very effective over short distances and very easy to implement. A sample program is also given in the appendix.

2.12 Interrupts

As the name implies, an interrupt is some event which interrupts normal program execution.

As stated earlier, program flow is always sequential, being altered only by those instructions which expressly cause program flow to deviate in some way. However, interrupts give us a mechanism by means of which we can "put on hold" the normal program flow, execute a subroutine, and then resume normal program flow as if we had never left it. This subroutine, called an interrupt handler or an interrupt service routine (ISR), is only executed when a certain event (interrupt) occurs. The event may be any of the following:

- one of the timers overflowing,
- receiving a character via the serial port,
- transmitting a character via the serial port,
- one of two external events, normally pulses on dedicated pins.

The 8051 may be configured so that when any of these events occur the main program is temporarily suspended and control is passed to a special section of code or interrupt service routine (ISR) which presumably would execute some function related to the event that has just occurred. Once the ISR is completed, control would be returned to the original program. The main program so to speak, would never even know that it was interrupted.

The ability to interrupt normal program execution when certain events occur makes it much easier and much more efficient to handle certain events. If it were not for interrupts we would have to repeatedly check in our main program whether the timers had over-flowed, or whether we have received another character via the serial port, or if some external event has occurred. Besides making the main program ugly and hard to read, such a situation would make our program inefficient since we would be using precious instruction cycles, regularly and frequently checking for events even if they do not happen so frequently.

For example, let us say we have a program executing many subroutines performing many tasks. Let us also suppose that we want our program to automatically toggle the P3.0 port every time timer 0 overflows. The code to do this without using interrupts would look something like this:

Download free eBooks at bookboon.com

```
TOP:    . . .
. . .
                JNB TF0, SKIP_TOGGLE      ; check for timer overflow here (2 cycles)
                CPL P3.0                  ; toggle the bit (1 machine cycle)
                CLR TF0                   ; clear the overflow flag to be ready for
                                          ; the next overflow (1 cycle)

SKIP_TOGGLE:
                . . .                     ; assume that 98 cycles are involved here
                . . .
                JMP TOP                   ; loop endlessly
```

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

Click on the ad to read more

Download free eBooks at bookboon.com

Since the TF0 flag is set whenever timer 0 overflows, the above code will toggle P3.0 every time timer 0 overflows. This accomplishes what we want, but is inefficient. The JNB instruction consumes 2 machine cycles to determine that the flag is not set and jump over the unnecessary code. In the event that timer 0 overflows, the CPL and CLR instruction require an additional 2 machine cycles to execute. To make the arithmetic easy, let us say that the rest of the code (until JMP TOP) in the program requires 98 machine cycles. Thus in total, our code consumes 100 machine cycles (98 instruction cycles plus the 2 that are executed at every iteration to determine whether or not timer 0 has overflowed). If we are in 16-bit timer mode, timer 0 will overflow every 65,536 machine cycles. In the time between overflows we would have performed 65536/100 or 655 JNB tests, consuming 1310 machine cycles plus another 2 machine cycles to perform the code when there is the overflow. So to achieve our goal, we have spent 1312 out of 65536 or 2% of the time just checking when to toggle P3.0. Moreover, we would not be reacting immediately to the overflow since we would only notice it when we come to the check instruction. And our code is not efficient because we have to make that check during every iteration of our main program loop.

Luckily with interrupts this is not necessary and we can forget about checking for the overflow condition. The micro-controller itself will check for the condition automatically *after every instruction* (thus the reaction is much quicker) and when the condition is met it will jump to a subroutine, execute the code, then return to where it was before the interrupt. In this case, our subroutine would be nothing more than:

```
CSEG AT 000BH       ; this ensures routine is written in
                    ; the correct vector table location for Timer 0 interrupt
CPL P3.0
RETI
```

First, it should be noted that the ISR has to be located at a specified code location, depending on the interrupt being used. (see section 2.13.1).

Secondly, it can be noticed that the CLR TF0 command has disappeared. That is because when the 8051 executes our "timer 0 interrupt routine," it automatically clears the TF0 flag which had originally generated the interrupt. Also instead of a normal RET instruction we have a RETI instruction. The RETI instruction does the same thing as a RET instruction (that is it pops the high- and low-order bytes of the program counter successively from the stack), but it also tells the 8051 that an interrupt routine has finished so that it would restore the interrupt logic to accept additional interrupts at the same priority level as the one just processed. *We must always end our interrupt service routines with RETI instruction.*

Download free eBooks at bookboon.com

Thus, every 65536 instruction cycles (when timer 0 overflows), control is transferred to the ISR automatically at the end of the current instruction, and the CPL and the RETI instructions are executed only once. These two instructions together require 3 instruction cycles, and we have accomplished the same goal as the first example that required 1312 instruction cycles. As far as the toggling of P3.0 goes, our code is 437 times more efficient! Not to mention the fact that it is much easier to read and understand because we do not have to remember to always check for the timer 0 flag in our main program. We just set up the interrupt and forget about it, secure in the knowledge that the 8051 will execute our code whenever it is necessary.

The same idea applies to receiving data via the serial port. One way to do it is to continuously check the status of the RI flag in an endless loop. Or we could check the RI flag as part of a larger program loop. However, in the latter case we run the risk of missing characters. What happens if a character is received right after we do the check, the rest of our program executes, and before we even check RI again a second character has come in. We will lose the first character. With interrupts, the 8051 will put the main program "on hold" and call our special routine to handle the reception of a character. Thus, we neither have to put an ugly check in our main code nor do we lose characters.

2.12.1 What Events Can Trigger Interrupts?

We can configure the 8051 so that any of the following events will cause an interrupt:

- Timer 0 Overflow.
- Timer 1 Overflow.
- Reception/Transmission of Serial Character.
- External Event 0.
- External Event 1.

In other words, we can configure the 8051 so that when Timer 0 Overflows or when a character is sent/received, the appropriate interrupt routines are called.

Obviously we need to be able to distinguish between various interrupts and executing different code depending on what interrupt was triggered. This is accomplished by jumping to a fixed address when a given interrupt occurs.

By consulting Table 2-14 it can be seen that whenever Timer 0 overflows (i.e., the TF0 bit is set), the main program will be temporarily suspended and control will jump to 000BH. It is assumed that we have some code written at address 000BH that handles the situation of Timer 0 overflowing.

Download free eBooks at bookboon.com

Interrupt Name	Interrupt Number	Flag	Interrupt Hex Vector Address
External 0	0	IE0	0003
Timer 0	1	TF0	000B
External 1	2	IE1	0013
Timer 1	3	TF1	001B
Serial	4	RI or TI	0023

Table 2-14 8051 Interrupt Vector Table location

The Interrupt Vector Addresses shown in this table indicate the location where the ISR code for that particular interrupt should be written. Only 8 bytes are allocated for every interrupt (provided that one is using them all) and so if the ISR requires more than 8 bytes, then one would simply write

```
LJMP MY_ISR
```

at the Interrupt Vector Address and then at MY_ISR (located elsewhere in the main code area) we can write our routine which can be of any length. This is also shown in the A51 template in the A51 examples in Chapter 3.

Click on the ad to read more

Download free eBooks at bookboon.com

It should also be noted here, that both RI and TI interrupts cause the program to jump to the same address (0023H). Hence it is up to the ISR to check which event (RI or TI) caused the interrupt and subsequently clear the appropriate RI or TI flag. These are not cleared automatically by the controller.

2.12.2 Setting Up Interrupts

By default at power up, all interrupts are disabled. This means that even if, for example, the TF0 bit is set, the 8051 will not execute the interrupt service routine

Bit-addressable					
Bit	**Name**	**Alternate Name (ASM)**	**Alternate Name (Keil C)**	**Hex Bit Address**	**Explanation of Function**
7	EA	IE.7	IE^7	AF	Global Interrupt Enable/Disable
6	-	IE.6	IE^6	AE	Undefined on the 8051
5	-	IE.5	IE^5	AD	Undefined on the 8051
4	ES	IE.4	IE^4	AC	Enable/Disable Serial Interrupt
3	ET1	IE.3	IE^3	AB	Enable/Disable Timer 1 Interrupt
2	EX1	IE.2	IE^2	AA	Enable/Disable External 1 1nterrupt
1	ET0	IE.1	IE^1	A9	Enable/Disable Timer 0 Interrupt
0	EX0	IE.0	IE^0	A8	Enable/Disable External 0 Interrupt

Table 2-15 IE (A8H) SFR

Our program must specifically tell the 8051 that it wishes to enable interrupts and specifically which interrupts it wishes to enable. We can do this by modifying the IE SFR (A8h), setting the corresponding bits accordingly. As we can see in Table 2-15, each of the 8051's interrupts has its own bit in the IE SFR. We enable a given interrupt by setting the corresponding bit to 1. For example, if we wish to enable Timer 1 Interrupt only, we would execute either:

```
ORL IE, #08h
```

or

```
SETB ET1
```

Each of the above instructions set bit 3 of IE, thus enabling Timer 1 Interrupt. Once Timer 1 Interrupt is enabled, whenever the TF1 bit is set, the 8051 will automatically put "on hold" the main program and execute the Timer 1 Interrupt Handler at address 001Bh. In C, this would simply be one ET1 = 1; line.

Download free eBooks at bookboon.com

However, before Timer 1 Interrupt (or any other interrupt) is truly enabled, we must also set bit 7 of IE. (SETB EA). This bit, the Global Interrupt Enable/Disable bit, enables or disables all interrupts simultaneously. That is to say, if bit 7 is cleared then no interrupt servicing will occur, even if all the other bits of IE are set. Setting bit 7 will enable all the interrupts that have been selected by setting other bits in IE. This is useful in program execution if we have time-critical code that needs to execute. In this case, we may need the code to execute from start to finish without any interrupt getting in the way. To accomplish this we can simply clear bit 7 of IE (CLR EA) just before starting the critical code and then set it again to 1 after our time-critical code has been executed.

We should also clear the interrupt flag initially, just to be sure that we start at the right condition. So, to sum up what has been stated in this section, to enable the Timer 1 Interrupt the most common approach is to execute the following instructions (assuming we have already written and properly stored the corresponding ISR):

```
CLR TF1
SETB ET1
SETB EA
```

Thereafter, the Timer 1 Interrupt Handler at 01Bh will automatically be called whenever the TF1 bit is set (upon Timer 1 overflow). Moreover, the TF1 bit will automatically be cleared once the ISR is being executed.

2.12.3 Polling Sequence

The 8051 automatically evaluates whether an interrupt should occur after every instruction. When checking for interrupt conditions, it checks them in the following order, as shown in Table 2-16:

1. External 0 Interrupt
2. Timer 0 Interrupt
3. External 1 Interrupt
4. Timer 1 Interrupt
5. Serial Interrupt

Table 2-16 Polling Sequence Order

This means that if a Serial Interrupt occurs at the exact same instant that an External 0 Interrupt occurs, the External 0 ISR will be executed first and the Serial ISR will be executed only when the External 0 ISR has been completed. This order is only respected in the extreme case that interrupts happen exactly at the same time. It should be remembered, that interrupts having the same priority cannot interrupt each other, irrespective of the polling sequence order.

Download free eBooks at bookboon.com

2.12.4 Interrupt Priorities

The 8051 offers two levels of interrupt priority: high and low. By using interrupt priorities we may assign a higher priority to certain important or critical interrupt conditions.

For example, we may have enabled Timer 0 Interrupt which is automatically called every time Timer 0 overflows. Additionally, we may have enabled the Serial Interrupt which is called every time a character is received via the serial port. However, we may consider that receiving a character is much more important than the timer interrupt. In this case, if Timer 0 Interrupt is already executing we may wish that the serial interrupt itself interrupts the Timer 0 ISR if it happens to be executing. When the serial interrupt is complete, control passes back to Timer 0 ISR to continue from where it had been stopped and finally back to the main program. We may accomplish this by assigning a high priority to the Serial Interrupt and a low priority to the Timer 0 Interrupt.

Interrupt priorities are controlled by the IP SFR (B8h), where setting the bit to one raises the priority of that particular interrupt. The IP SFR has the following format:

Click on the ad to read more
Download free eBooks at bookboon.com

Bit-addressable					
Bit	**Name**	**Alternate Name (ASM)**	**Alternate Name (Keil C)**	**Bit Hex Address**	**Explanation of Function**
7	-	IP.7	IP^7	-	Undefined – future expansion
6	-	IP.6	IP^6	-	Undefined – future expansion
5	-	IP.5	IP^5	-	Undefined – future expansion
4	PS	IP.4	IP^4	BC	Serial interrupt priority
3	PT1	IP.3	IP^3	BB	Timer 1 interrupt priority
2	PX1	IP.2	IP^2	BA	External 1 interrupt priority
1	PT0	IP.1	IP^1	B9	Timer 0 interrupt priority
0	PX0	IP.0	IP^0	B8	External 0 interrupt priority

Table 2-17 IP (B8H) SFR

When considering interrupt priorities, the following rules apply:

- Nothing can interrupt a high priority interrupt; not even another high priority interrupt. Same priority interrupts cannot interrupt each other.
- A high priority interrupt may interrupt a low priority interrupt.
- A low priority interrupt may be dealt with only if no other interrupt is already executing.
- If two interrupts occur at the same time, the interrupt with higher priority will execute first. If both these interrupts happen to have the same priority, the interrupt which is serviced first is determined by polling sequence order of Table 2-16.
- An new interrupt cannot pause an already running ISR which was triggered by an interrupt having the same priority as the new one, irrespective of the polling sequence order mentioned in Table 2-16.

2.12.5 What Happens When an Interrupt Occurs?

When an interrupt is triggered, the following actions are taken automatically by the microcontroller:

1. The current Program Counter (address of the next code/instruction to be executed) is saved (pushed) on the stack, low byte first.
2. Interrupts of the same and lower priority are blocked.

Download free eBooks at bookboon.com

3. In the case of Timer and External interrupts, the corresponding interrupt flag is cleared automatically. Take special note of this third step: If the interrupt being handled is a Timer 0 or Timer 1 or External interrupt, the microcontroller automatically clears the interrupt flag before passing control to your interrupt handler routine. This is because there is no ambiguity as to what caused the interrupt, and it is not necessary to clear the bit in the ISR code. If the external interrupt was programmed as being level triggered, then the hardware has to ensure that the level is again restored so as no to cause repeated interrupts. *However we would have to clear the particular flag in the ISR software for the Serial port and Timer 2 (in the case of the 8032 controller).* This is due to the fact that for each of these devices, two different events can trigger the same interrupt number. For the serial port it can be a Received character interrupt (RI) or a Transmitted character interrupt (TI), or both, as explained in the section below. For the Timer 2, it could be the timer overflow flag (TF2) or EXT2 flag which caused the interrupt.
4. Program execution transfers to the corresponding interrupt handler vector address.
5. The Interrupt Service Routine executes.
6. At the end of the ISR, the Program Counter is popped back automatically from the stack once the RETI instruction is executed.

Additionally, apart from these automatic events, other precautions may need to be taken. It is good programming practice to save (push) the PSW register immediately at the beginning of the ISR so that we save the status of the various flags which might have been in use by the interrupted section of the code. The interrupt jump might have occurred just before our main program was about to execute a JC label (jump if carry bit is set). If the ISR routine modifies the carry bit, then when the ISR is finished and the main program resumes operation, it would not perform as expected. The PSW should then be popped back before executing the RETI instruction.

If we are absolutely certain that our ISR does not modify any flags, then there is no need to PUSH/POP the PSW register.

```
ISR_EXAMPLE_01:
                PUSH PSW            ;save flags
                …….
                …….
                POP PSW             ;restore flags
                RETI
```

If in our ISR we intend to overwrite and use some registers which are being used in our main program or in some other ISR, we would also need to be very careful.

Download free eBooks at bookboon.com

We can push all these registers at the start of your ISR and pop them back at the end, before executing the RETI instruction.

```
ISR_EXAMPLE_02:
                PUSH PSW        ;save flags
                PUSH ACC        ;save registers being used in this routine
                PUSH B          ;with their original values
                PUSH 0          ;save register r0 bank 0 (address 0h)
                PUSH 1          ;save register r1 bank 0 (address 1h)
                …….
                …….
                POP 1           ;restore registers to original values
                POP 0
                POP B
                POP ACC
                POP PSW         ; restore flags
                RETI
```

Note that the registers should be popped out of the stack in the reverse order from the way they were pushed. The first register that was pushed on the stack, should be the last register that is popped from the stack.

Instead of pushing and popping registers R0-R7 in the ISR, we might consider using a separate dedicated bank for the ISR routine. The PSW (and ACC, B, DPL and DPH if used) should still be pushed/popped. This could be done by setting the corresponding bits in the PSW register as shown:

```
ISR_EXAMPLE_03:
                PUSH PSW    ;save flags and register bank in use flags
                SETB RS0
                CLR RS1     ; use register bank 1
                …….
                …….
                POP PSW     ; restore flags and original register bank
                RETI
```


Download free eBooks at bookboon.com

The POP PSW instruction automatically restore the original register bank since RS0 and RS1 bits are actually part of the PSW SFR.

2.12.6 What Happens When an Interrupt Ends?

An interrupt ends when your program executes the RETI (Return from Interrupt) instruction. When the RETI instruction is executed the following actions are taken by the microcontroller:

- The high and low order bytes of the program counter are popped back from the stack. These contain the address of the instruction to be executed next.
- The interrupt logic is restored so as to accept additional interrupts at the same priority level as the one just processed.

Using RET instead of RETI at the end of the ISR would ultimately cause the programme not to run as expected since the controller would not handle other interrupts correctly.

2.12.7 Serial Interrupts

Serial Interrupts are slightly different from the rest of the interrupts. This is due to the fact that there are two interrupt flags: RI and TI. If either flag is set (or even both), a serial interrupt is triggered. As we will recall from the section on the serial port, the RI bit is set when a byte is received by the serial port and the TI bit is set when a byte has been sent.

This means that when our serial interrupt is executed, it may have been triggered because the RI flag was set or because the TI flag was set or perhaps because both flags were set. It should be remembered that both transmission and reception can work simultaneously in the 8051. Thus, our routine must check the status of these flags to determine what action is appropriate, and hence the 8051 cannot and does not automatically clear the RI and TI flags. We must therefore make sure to clear these bits in the ISR.

A brief code example of such a serial ISR is in order. Note that the serial interrupt might have occurred because of a received character and/or a character has just been transmitted:

Download free eBooks at bookboon.com

```
INT_SERIAL:
;First we check whether a character has just been transmitted
        JNB TI,CHECK_RX                 ; If the TI flag is not set, we
                                        ; jump to check RI

; transmitter section
        CLR TI                          ; Clear the TI flag
        JNB TX_BUFFER_FULL, CHK_RX      ; Check RI if nothing else to transmit
        MOV SBUF,TX_BUF                 ; Transmit character stored
                                        ; in location TX_BUF

        CLR TX_BUFFER_FULL              ; Buffer now empty, ready for
                                        ; the next character
; We still need to check the receiver since BOTH TI and RI might have occurred.
; Hence once we are finished with the TI case, we fall through to the RI case.
;
; receiver section
CHK_RX:
        JNB RI,EXIT_ISR                 ; Ignore if RI is not set
        CLR RI                          ; Clear the RI flag
        JNB RX_BUFFER_FULL,RTR          ; Check if ok to store received character
        SJMP EXIT_ISR                   ; If not, then exit service routine, without saving
                                        ; it, thus losing the character.

RTR:
        MOV RX_BUF,SBUF                 ; Store character in buffer RX_BUF
        SETB RX_BUFFER_FULL             ; Indicate new character in buffer
EXIT_ISR:
        RETI
```

The main program would regularly check variable RX_BUFFER_FULL and gets the character from RX_BUF when available. It would then clear RX_BUFFER_FULL.

As we can see, our code checks the status of both interrupts flags. If both flags were set, both sections of code will be executed. Also note that each section of code clears its corresponding interrupt flag. If we forget to clear the interrupt bits, the serial interrupt will be executed over and over again until we clear the bit. Thus it is very important that we always clear the interrupt flags in a serial interrupt.

The above code is an example of a simple serial routine. Other more complete routines can be found in the appendix.

Download free eBooks at bookboon.com

2.12.8 Important Interrupt Considerations:

We now list some important considerations to be made when using interrupts and writing interrupt service routines.

2.12.8.1 Register Protection

One very important rule applies to all interrupt handlers:

Interrupts must leave the processor in the same state that it was in when the interrupt was initiated.

Remember, the idea behind interrupts is that the main program is not aware that they are executing in the "background". However, consider the following code:

```
CLR C           ; Clear carry
MOV A, #25h     ; Load the accumulator with 25h
ADDC A, #10h    ; Add 10h, with carry
```

After the above three instructions are executed, the accumulator will contain a value of 35h.

Click on the ad to read more

Download free eBooks at bookboon.com

But what would happen if right after the MOV instruction an interrupt has occurred. Let us assume that during this interrupt service routine, the carry bit was set and the value of the accumulator was changed to 40h. When the interrupt finished and control is passed back to the main program, the ADDC would add 10h to 40h, and add an additional 01h because the carry bit is set. In this case, the accumulator will contain the value 51h at the end of execution.

The program has calculated the wrong answer. A programmer who is unfamiliar with interrupts would be convinced that the microcontroller was damaged in some way, provoking problems with mathematical calculations.

What has happened, in reality is that the interrupt did not protect the registers it was using.

What does this mean? It means that if our interrupt uses the accumulator, it must ensure that the value of the accumulator is the same at the end of the interrupt as it was at the beginning. This is generally accomplished with a PUSH and POP sequence. For example:

```
PUSH ACC
PUSH PSW
MOV A, #0FFh
ADD A, #02h
…….
…….
POP PSW
POP ACC
```

The main code of the ISR is the MOV instruction and the ADD instruction. However, these two instructions modify the Accumulator (the MOV instruction) and also modify the value of the carry bit (the ADD instruction can cause the carry bit to be set). Since an interrupt routine must guarantee that the registers remain unchanged by the routine, the routine pushes the original values onto the stack using the PUSH instruction. It is then free to use the registers that it has protected or pushed on stack. Once the interrupt has finished its task, it pops the original values back into the registers. When the interrupt exits, the main program will never know the difference because the registers are exactly the same as they were before the interrupt was executed.

Download free eBooks at bookboon.com

In general, our interrupt routine must protect the following registers if somehow they are being modified in the ISR:

- PSW
- DPTR (DPH/DPL)
- ACC
- B
- Registers R0-R7

Remember that PSW consists of many individual bits that are set by various 8051 instructions. Unless we are absolutely sure of what we are doing and have a complete understanding of which instructions set which bits, it is generally a good idea to always protect PSW by pushing and popping it off the stack at the beginning and end of our interrupts. The PSW register would preserve the status of the register bank we were using prior to the interrupt, the carry flag, the zero flag etc.

Note also that most assemblers (in fact, ALL assemblers that I know of) will not allow us to execute the instruction:

```
PUSH R0
```

This is due to the fact that depending on which register bank is selected, R0 may refer to either internal RAM address 00h, 08h, 10h, or 18h. Hence R0 is not a valid memory address that the PUSH and POP instructions can use.

Thus, if we are using any "R" register in our interrupt routine, we will have to push that register's absolute address onto the stack instead of just saying PUSH R0. For example, when using bank 0 instead of PUSH R0 we would execute:

```
PUSH 00h
```

Of course, this only works if we have selected the default register bank 0. If we are using an alternate register set, we must PUSH the address which corresponds to the register we are using in that alternate bank. For example, if we are using register bank 1, then the register R0 for that bank would have an address of 08h, hence we would use:

```
PUSH 08h
```

Certain assemblers allow special keywords (such as PUSH AR2) to be used in order to calculate automatically the correct address for the register being pushed of popped. Such as:

Download free eBooks at bookboon.com

```
; if we intend using register 2 of bank 1 in our ISR
PUSH    PSW     ; save PSW
SETB    RS0     ; select bank 1 (RS0=1, RS1=0)
CLR     RS1
USING   1       ; advise pre-processor to use register bank 1 (no code)
PUSH    AR2     ; push R2 in bank 1 (address 0Ah)

; if we intend using register 7 of bank 3 in our ISR
PUSH    PSW     ; save PSW
SETB    RS0     ; select bank 3 (RS0 = RS1 = 1)
SETB    RS1
USING   3       ; advise pre-processor to use register bank 3 (no code)
PUSH    AR7     ; push R7 in bank 3 (address 1Fh)
```

Note that the keyword USING does not generate any code. It is used by the pre-processor to calculate the correct address for the register being pushed or popped.

Alternatively, we might want to make use of a separate register bank for our ISR, a register bank which is used only in the ISR. In this case, provided that we do not use any other registers which are used elsewhere, there would not be the need to push any of the registers R0-R7. We push only the PSW and ACC (and perhaps B, DPH and DPL if we use them in the ISR, since they would be common with other sections of the code) and then set RS0, RS1 (two bits themselves residing in the PSW register) to select our reserved bank. Then, before leaving, we simply pop back the pushed registers in reverse order. There would not be the need to reset again RS0 and RS1 separately, since their original value would be re-instated anyway when we pop back the PSW register, where the original RS0 and RS1 bit settings were stored.

2.12.9 Common Problems with Interrupts

Interrupts are a very powerful tool available to the 8051 developer, but if used incorrectly they can be the source of a number of bugs. Errors in interrupt routines are often very difficult to diagnose and correct.

If we are using interrupts and our program is crashing or does not seem to be performing as we would expect, we should always review the interrupt-related issues. See section 11.7 for some hints on using interrupts.

Download free eBooks at bookboon.com

3 A51 Examples

In this chapter we present a few assembly language programs which use most of the topics discussed in the previous chapters. Before starting to write the first program, we provide a template which explains the general organisation of an 8051 assembly language program. The remarks by the side of the instructions in the template and in the other example programs provide most of the explanations required.

Further examples in C are provided in the Appendix.

3.1 Template.a51

```
; Template.a51
$NOMOD51
#include <reg52.inc>                    ; assuming that we are using an 8032 instead of an 8051
                                        ; reg52.inc would include all the SFRs present on the 8032

start   equ 0000H                       ; these equates can be changed if using a development
                                        ; board, depending on where the code is to reside.
Ext0_IVA        equ 0003H               ; Interrupt Vector address for External 0 interrupt number 0
TF0_IVA         equ 000BH; Interrupt Vector address for Timer 0 interrupt number 1
Ext1_IVA        equ 0013H               ; Interrupt Vector address for External 1 interrupt number 2
TF1_IVA         equ 001BH; Interrupt Vector address for Timer 1 interrupt number 3
Ser_IVA         equ 0023H               ; Interrupt Vector address for Serial interrupt number 4
TF2_IVA         equ 002BH; Interrupt Vector address for Timer 2 interrupt number 5

Past_IVT equ 0030H

; The following equates are used for RAM zero initialisation routines
IDATASTART      EQU 0H                  ; the absolute start-address of IDATA memory is always 0
IDATALEN        EQU 100H                ; the length of IDATA memory in bytes for the 8032 (256 bytes).

XDATASTART      EQU 0H                  ; the absolute start-address of XDATA memory (say 8100H)
XDATALEN        EQU 0H                  ; the length of XDATA memory in bytes.

CSEG AT start
        LJMP Main                       ; this is the first instruction executed on reset

CSEG AT Ext0_IVA
        RETI                            ; good practice to include this if not using interrupt, just in case.
                                        ; comment above code if this interrupt is being used
; or if the ISR code is within 8 bytes long, it can be written directly here.
; if not then use
;       LJMP Ext0_ISR                   ; to jump to the correct ISR

CSEG AT TF0_IVA
;       RETI                            ; good practice to include this if not using interrupt, just in case.
                                        ; comment above code if this interrupt is being used
; or if the ISR code is within 8 bytes long, it can be written directly here.
; if not then use
        LJMP TF00_ISR                   ; to jump to the correct ISR
; and so on for the other interrupts
```


Download free eBooks at bookboon.com

```
CSEG AT Ext1_IVA
        RETI                    ; good practice to include this if not using interrupt, just in case.
                                ; comment above code if this interrupt is being used
; or if the ISR code is within 8 bytes long, it can be written directly here.
; if not then use
;       LJMP Ext1_ISR           ; to jump to the correct ISR

CSEG AT TF1_IVA
        RETI                    ; good practice to include this if not using interrupt, just in case.
                                ; comment above code if this interrupt is being used
; or if the ISR code is within 8 bytes long, it can be written directly here.
; if not then use
;       LJMP TF1_ISR            ; to jump to the correct ISR

CSEG AT Ser_IVA
        CLR RI                  ; good practice to include this if not using interrupt, just in case.
        CLR TI                  ; good practice to include this if not using interrupt, just in case.
        RETI                    ; good practice to include this if not using interrupt, just in case.
                                ; comment above 3 code lines if this interrupt is being used
; or if the ISR code is within 8 bytes long, it can be written directly here.
; if not then use
;       LJMP Ser_ISR            ; to jump to the correct ISR

CSEG AT TF2_IVA
        CLR TF2                 ; good practice to include this if not using interrupt, just in case.
        CLR EXF2            ; good practice to include this if not using interrupt, just in case.
        RETI                    ; good practice to include this if not using interrupt, just in case.
                                ; comment above 3 code lines if this interrupt is being used
; or if the ISR code is within 8 bytes long, it can be written directly here.
; if not then use
;       LJMP Ext0_ISR           ; to jump to the correct ISR

; skip over Interrupt Vector Table in the code area
org Past_IVT
Main:
; First clear the 8032 Internal RAM (from 0 to FFH)
        CLR A
        MOV R0,#(IDATALEN - 1)
CLR_RAM:
        MOV @R0,A
        DJNZ R0,CLR_RAM

; then clear external RAM if required, using conditional assembly, depending on XDATALEN
IF XDATALEN <> 0
        MOV     DPTR,#XDATASTART
        MOV     R7,#LOW (XDATALEN)
IF (LOW (XDATALEN)) <> 0 ; check needed so that the DJNZ checks below will work
                                ; correctly, since if R7 is zero before the DJNZ, it will loop
                                ; for 256 times and not zero times.
                                ; ( check with XDATALEN of 255 bytes and then 256 bytes !! )
        MOV     R6,#(HIGH (XDATALEN) +1)
ELSE
        MOV     R6,#(HIGH (XDATALEN))
ENDIF
```


Download free eBooks at bookboon.com

```
        CLR     A
CLR_XDATA:
        MOVX    @DPTR, A
        INC     DPTR
        DJNZ    R7,CLR_XDATA
        DJNZ    R6,CLR_XDATA
ENDIF

; set up stack pointer, above Bit-addressable area (not necessarily always set to this point)
        MOV SP,#2FH

; Program starts here
…….
……..
…….

; Long Interrupt Service Routines can be written here, after the main program
TF0_ISR:
        PUSH PSW
    ……..
    ……
        POP PSW
        RETI

; Constants can be stored here, at the end of the code area.
 OneHundred: DB 100
 SixHundred: DW 600
 Message: DB “Hello !!”,10,13

; Variables can be stored either in the internal 256 bytes data area or in the external volatile
; memory. Bit variables are stored in the bit data area

MyBits SEGMENT BIT
RSEG MyBits
        Flag1:  DBIT 1          ; 1 bit in Bit-addressable area, allocated to Flag1
        Flag2:  DBIT 1          ; 1 bit in Bit-addressable area, allocated to Flag2

Var1 SEGMENT DATA
RSEG Var1
        Answer: DS      1       ; 1 byte in data area, allocated to Answer
        Year:   DS      2       ; 1 bytes in data area, allocated to Year
        Month:  DS`     1       ; 1 byte in data area, allocated to Month

Var2 SEGMENT XDATA
RSEG Var2
        Numbers: DS     500     ; 500 bytes allocated to Numbers, in external RAM

end
```

The first real program, SerP3.a51 is a serial port example program (section 3.2) which basically initializes the UART and then provides routines for reading and writing characters via the UART.

Download free eBooks at bookboon.com

The second program (3.3) is a simple Traffic light controller which also uses the SerP3.a51 routines. It makes use of Timer 2 interrupt which is used to accurately time the duration in seconds for each traffic pattern. Note the way the Interrupt Vector Table (IVT) is jumped over when the program starts executing from location 0000H. For those Interrupts which are not in use, it is generally a good practice to insert a simple RETI instruction at the corresponding IVT location just in case an inadvertent event causes an undesired interrupt to occur.

3.2 Serial Port Example Program

```
; SERP3.A51
; march 2003 - paul p. debono
; works fine using p3 serial socket
; no interrupts
;
$NOMOD51
#include <reg52.inc>

; These routines are declared PUBLIC so that they can be used in other modules
PUBLIC INIT_SERIAL, TX_CHAR, RX_CHAR
PUBLIC TX_IMSG,TX_CMSG, TX_XMSG
;
; SERIAL PORT RELATED ROUTINES
;
; INIT_SERIAL(BAUDRATE)          Initialise Serial port, 9600, 19200 or 57600 baud.
; RX_CHAR()                      Receive character from port, (WAIT FOR CHARACTER)
; TX_CHAR(ALPHA)                 Send character to Port
; TX_MSG(*MESSSAGE)              Transmit null terminated string (Internal RAM)
; TX_CMSG(*MESSSAGE)             Transmit null terminated string (PROGRAM CODE AREA)
; TX_XMSG(*MESSSAGE)             Transmit null terminated string (External DATA AREA)
;
SERIAL_RTN SEGMENT CODE
RSEG SERIAL_RTN

; SUBROUTINES USED IN APPLICATION
;
; ***********************************************
;
; serial port support
;
; initialise the serial port for required baud rate,
; not under interrupt control.

; baud rate passed in r7 bank 0
;       parameter 96 => 9600 baud
;       parameter 192 => 19200 baud
;       parameter 57 => 57600 baud
```


Download free eBooks at bookboon.com

```
INIT_SERIAL:
        ANL TMOD, #00001111B        ; clear all timer 1 bits in tmod
        ORL TMOD, #20H              ; timer 1 8-bit auto reload mode 2
        CLR RCLK                    ; use timer 1 for receive baud rate
        CLR TCLK                    ; use timer 1 for transmit baud rate
        MOV TH1, #0FDH              ; 9600/19200 baud counter value
        MOV TL1, #0FDH
        MOV PCON, #0H               ; choose 9600 baud
        CJNE R7, #192, CHK_IF_57
        MOV PCON, #80H              ; choose 19200 baud, smod=1
        SJMP BAUD_OK
CHK_IF_57:
        CJNE R7, #57, BAUD_OK
        MOV TH1, #0FFH              ; 57600 baud counter value
        MOV TL1, #0FFH
        MOV PCON, #80H              ; smod=1
BAUD_OK:
        CLR ET1                     ; disable timer 1 interrupts, just in case
        SETB TR1            ; start timer 1 (tr1 = 1) or mov tcon,#40h
        MOV SCON, #52H              ; 1 start, 8 data, 1 stop bit, RI=0, and setTI=1
                                    ; so as to be ready to start the first time
                                    ; enable receiver (ren=1)

    RET

; ***********************************************

; ***********************************************
; character received through serial port p3, is passed on to r7 bank 0 (address 07)
RX_CHAR:
        JNB RI, $                   ; wait here for character
        CLR RI
        MOV 07, SBUF
        RET

; ***********************************************

; ***********************************************
; character in r7 is transmitted through serial port p3
TX_CHAR:
        JNB TI, $           ; if tx is ready, then you are clear to send, else wait
        CLR TI
        MOV SBUF, 07                ; transmission starts, t1 set to 1 when ready

; the following delay might be needed depending on receiving equipment requirements
        PUSH B
        MOV B, #0A0H                ; small delay between transmissions
        DJNZ B, $           ; since we are not using any handshaking
        POP B

    RET

; ***********************************************

; ***********************************************
```


Download free eBooks at bookboon.com

```
; transmit message residing in internal memory
; pointer to message passed in r1
; message must terminate with a null (0) character.
; on exit, r1 is corrupted

TX_IMSG:
        MOV A, @R1
        CJNE A, #0, SEND_IT
        RET
SEND_IT:
        MOV 07, A
        ACALL TX_CHAR
        INC R1
        SJMP TX_IMSG

;***********************************************

;***********************************************
; transmit message residing in program (code) memory
; pointer to message passed in dph (hi) and dpl (lo)
; message must terminate with a null (0) character.
; on exit, dptr is corrupted.
TX_CMSG:
        CLR A
        MOVC A, @A + DPTR
        CJNE A, #0, SEND_IT2C
        RET
SEND_IT2C:
        MOV 07, A
        ACALL TX_CHAR
        INC DPTR
        SJMP TX_CMSG

;***********************************************

;***********************************************
; transmit message residing in external memory
; pointer to message passed in dph (hi) and dpl (lo)
; message must terminate with a null (0) character.
; on exit dptr is corrupted.
TX_XMSG:
        MOVX A, @DPTR
        CJNE A, #0, SEND_IT2
        RET
SEND_IT2:
        MOV 07, A
        ACALL TX_CHAR
        INC DPTR
        SJMP TX_XMSG

;***********************************************
 END
;***********************************************
```


Download free eBooks at bookboon.com

The second program (section 3.3) is Traffic lights program, and it is targeted to be run from an EPROM. This means that the code area starts from location 0000H. It is also targeted for the FLT-32 development board, which has an 8255 input/output chip added on, providing three additional 8-bit ports, labelled as PORTA, PORTB and PORTC in this program.

Click on the ad to read more

Download free eBooks at bookboon.com

3.3 Traffic Lights A51 Program

```
; lesson07EP.a51 targeted for eprom

$NOMOD51
#include <reg52.inc>
; use model for 8032/8052
; this will ensure that the assembler will recognise
; all the labels referring to the various
; Special Function Registers (SFRs).
;
;
; Traffic lights program with TIMER2 delay
; in 16-bit AUTO-RELOAD mode
;
; Timers count up at 12/11.0592 microseconds per count
; i.e. at 1.085 microseconds per count.
; Thus for a 50 millisecond delay, they need to count
; up 50000/1.085 = 46082 times. Hence the counters
; have to be loaded with 65536-46082 = 19454 decimal
; or 4BFEH
;
;
; The following routines are declared as EXTRN (within brackets)
; since they are actually defined in a different module.
;
EXTRN CODE (INIT_SERIAL, RX_CHAR, TX_CHAR)
EXTRN CODE (TX_IMSG, TX_CMSG, TX_XMSG)
;
; serial port related routines found in serp3.a51
;
; INIT_SERIAL(BAUDRATE)          Initialise Serial port, 9600, 19200 or 57600 baud.
; RX_CHAR()                      Receive character from port, (WAIT FOR CHARACTER)
; TX_CHAR(ALPHA)                 Send character to Port
; TX_MSG(*MESSSAGE)              Transmit null terminated string (Internal RAM)
; TX_CMSG(*MESSSAGE)             Transmit null terminated string (PROGRAM CODE AREA)
; TX_XMSG(*MESSSAGE)             Transmit null terminated string (External DATA AREA)
;
CR              EQU 13
LF              EQU 10

CTRL_WD         EQU 91H         ; control word for the 8255
PORTA           EQU 0FF40H      ; 8255 ports addresses in FLT-32 board
PORTB           EQU 0FF41H
PORTC           EQU 0FF42H
CONTROL         EQU 0FF43H

; Interrupts vector table location when targeting EPROM.
RESET           EQU 0000H
EXT0_ISR_VEC            EQU 0003H
T0_ISR_VEC              EQU 000BH
EXT1_ISR_VEC            EQU 0013H
T1_ISR_VEC              EQU 001BH
SERIAL_ISR_VEC          EQU 0023H
T2_ISR_VEC              EQU 002BH
```


Download free eBooks at bookboon.com

```
PROG_AREA               EQU 0030H       ; Main program area starting point
;
;
        ORG RESET
        LJMP MAIN                       ; Continue with MAIN, jumping over the interrupt vector table.

        ORG EXT0_ISR_VEC
;       LJMP EXT0_ISR                   ; point to my interrupt service routine
        RETI                            ; remark this line if using ISR

        ORG T0_ISR_VEC
        ; LJMP T0_ISR                   ; point to my interrupt service routine
        RETI                            ; remark this line if using ISR

        ORG EXT1_ISR_VEC
;        LJMP EXT1_ISR                  ; point to my interrupt service routine
        RETI                            ; remark this line if using ISR

        ORG T1_ISR_VEC
;       LJMP T1_ISR                     ; point to my interrupt service routine
         RETI                           ; remark this line if using ISR

        ORG SERIAL_ISR_VEC
;       LJMP SERIAL_ISR                 ; point to my interrupt service routine
        RETI                            ; remark this line if using ISR

        ORG T2_ISR_VEC
        LJMP T2_ISR                     ; point to my timer 2 service routine
;       RETI                            ; remark this line if using ISR

ORG PROG_AREA

MAIN:
; initialise stack pointer past bit-addressable area
        MOV SP, #30H

; First clear the 8032 Internal RAM (from 0 to FFH)
        CLR A
        MOV R0, #0FFH
CLR_RAM:
        MOV @R0, A
        DJNZ R0, CLR_RAM

; initialise serial port
        MOV R7, #57
        LCALL INIT_SERIAL

; print message
        MOV DPTR, #MESSAGE1
        LCALL TX_CMSG

; initialise 8255 i/o chip
        MOV DPTR, #CONTROL
        MOV A, #CTRL_WD
        MOVX @DPTR, A                   ; initialise 8255 ports mode
```


Download free eBooks at bookboon.com

```
; initialise timer 2
        MOV RCAP2H, #4BH        ; re-load timer counters
        MOV RCAP2L, #0FEH       ; 50 msec timer delay
        MOV TH2, RCAP2H         ; used for the first interrupt
        MOV TL2, RCAP2L

        MOV T2CON, #0           ; set timer 2 16-bit auto-reload

        SETB TR2                ; start timer 2
        SETB ET2                ; enable interrupts from timers
        SETB EA                 ; allow interrupts
        SETB PT2                ; Timer 2 with high priority

; start traffic lights
        MOV DPTR, #TABLE        ; point DPTR to table
        MOV A, DPL              ; DEC DPTR
        JNZ DECSKIP
        DEC DPH
DECSKIP:
        DEC DPL                 ; DPTR now points ahead of TABLE
        MOV R7, #1
        MOV R6, #1              ; R6,R7 set to 1 to start immediately from Top of Table
        SETB TF2                ; simulate timer 2 interrupt

LOOP: SJMP LOOP                 ; main program simply loops here
                                ; forever. It could be doing something
                                ; else whilst the timer takes care of
                                ; scheduling the display pattern.

; traffic control isr.
; permanent data stored in code (eprom) area
T2_ISR:
        CLR TF2                 ; clear interrupt flag
        PUSH ACC
        DJNZ R6, EXIT_NOW       ; exit immediately if 1 second
                                ; has not yet passed
        MOV R6, #20             ; reset R6 otherwise
        DJNZ R7, EXIT_NOW       ; has required time passed ?
        INC DPTR                ; yes
        CLR A                   ; we need to clear it first
        MOVC A, @A + DPTR       ; get next pattern
        JNZ SKIP                ; 0 pattern indicates end of table, hence start again
        MOV DPTR, #TABLE        ; acc=0 hence no need to clear it
        MOVC A, @A + DPTR       ; load 1st pattern in Acc,
SKIP: PUSH DPH
        PUSH DPL
        MOV DPTR, #PORTB
        MOVX @DPTR, A           ; light up leds with pattern
        POP DPL
        POP DPH
        INC DPTR
        CLR A
        MOVC A, @A + DPTR       ; get duration byte and
        MOV R7, A               ; store the seconds in R7
```


Download free eBooks at bookboon.com

```
; initialise timer 2
        MOV RCAP2H, #4BH        ; re-load timer counters
        MOV RCAP2L, #0FEH       ; 50 msec timer delay
        MOV TH2, RCAP2H         ; used for the first interrupt
        MOV TL2, RCAP2L

        MOV T2CON, #0           ; set timer 2 16-bit auto-reload

        SETB TR2                ; start timer 2
        SETB ET2                ; enable interrupts from timers
        SETB EA                 ; allow interrupts
        SETB PT2                ; Timer 2 with high priority

; start traffic lights
        MOV DPTR, #TABLE        ; point DPTR to table
        MOV A, DPL              ; DEC DPTR
        JNZ DECSKIP
        DEC DPH
DECSKIP:
        DEC DPL                 ; DPTR now points ahead of TABLE
        MOV R7, #1
        MOV R6, #1              ; R6,R7 set to 1 to start immediately from Top of Table
        SETB TF2                ; simulate timer 2 interrupt

LOOP: SJMP LOOP                 ; main program simply loops here
                                ; forever. It could be doing something
                                ; else whilst the timer takes care of
                                ; scheduling the display pattern.

; traffic control isr.
; permanent data stored in code (eprom) area
T2_ISR:
        CLR TF2                 ; clear interrupt flag
        PUSH ACC
        DJNZ R6, EXIT_NOW       ; exit immediately if 1 second
                                ; has not yet passed
        MOV R6, #20             ; reset R6 otherwise
        DJNZ R7, EXIT_NOW       ; has required time passed ?
        INC DPTR                ; yes
        CLR A                   ; we need to clear it first
        MOVC A, @A + DPTR       ; get next pattern
        JNZ SKIP                ; 0 pattern indicates end of table, hence start again
        MOV DPTR, #TABLE        ; acc=0 hence no need to clear it
        MOVC A, @A + DPTR       ; load 1st pattern in Acc,
SKIP: PUSH DPH
        PUSH DPL
        MOV DPTR, #PORTB
        MOVX @DPTR, A           ; light up leds with pattern
        POP DPL
        POP DPH
        INC DPTR
        CLR A
        MOVC A, @A + DPTR       ; get duration byte and
        MOV R7, A               ; store the seconds in R7
```

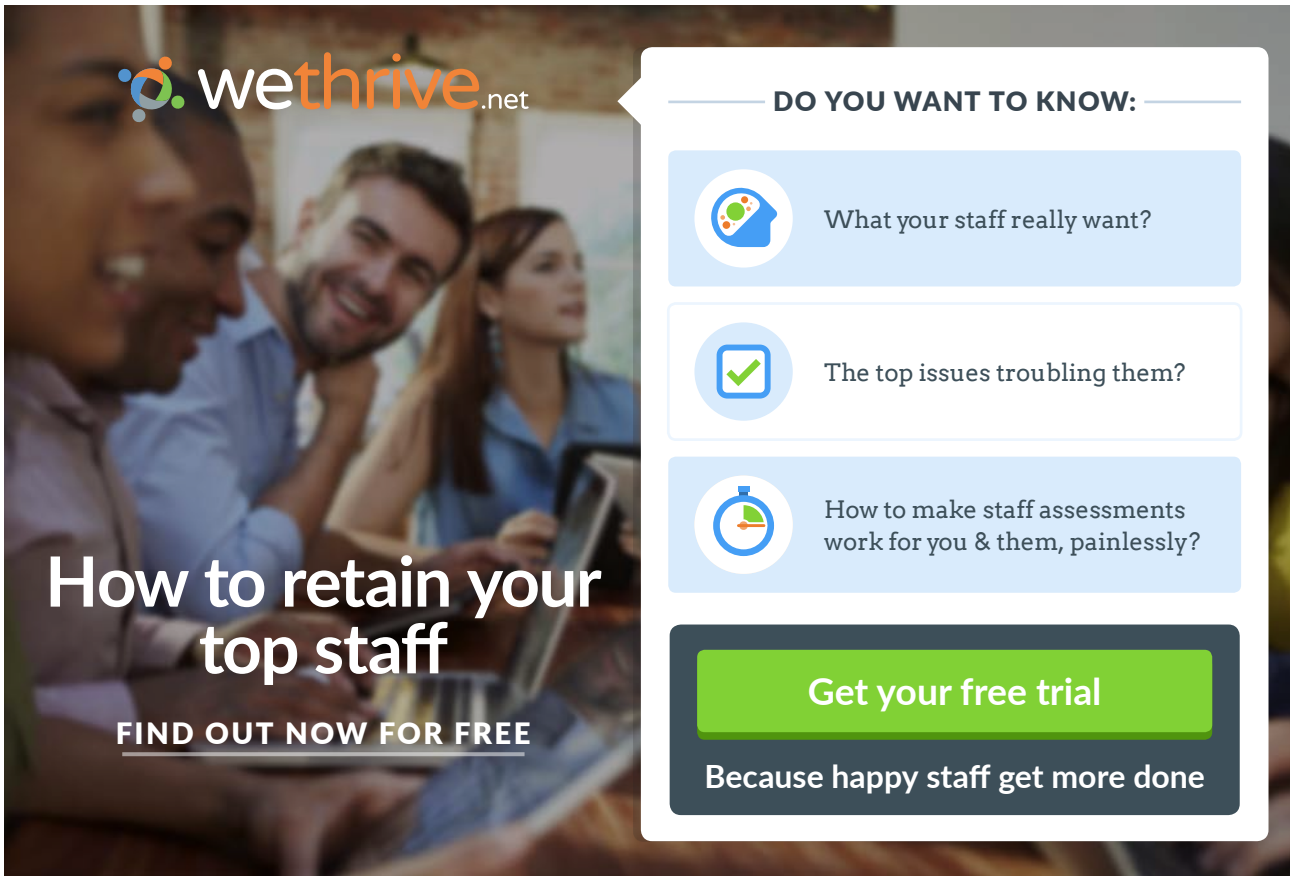

Download free eBooks at bookboon.com

```
EXIT_NOW:
        POP ACC
        RETI

; permanent data, can be stored in code area (not rewriteable)
; Table storing Pattern and Duration in seconds
;
TABLE:
        DB 82H,10           ; R - G
        DB 84H,2            ; R - Y
        DB 88H,1            ; R - R
        DB 0C8H,2           ; RY - R
        DB 28H,8            ; G - R
        DB 48H,2            ; Y - R
        DB 88H,1            ; R - R
        DB 8CH,2 ; R - RY
        DB 0                ; end of array marker

; Message must terminate with a zero for correct performance of the print routine
MESSAGE1:   db      13,10,'This is a serial port test',CR,LF,LF
            db      'Read the program carefully and try to',CR,LF
            db      'understand it well.',CR,LF,0

        END
```



Click on the ad to read more

Download free eBooks at bookboon.com

4 8032 Differences

The 8051 is the very basic micro-controller. In this chapter we present one of the first improved versions or variants, namely the 8032/8052 micro-controller, with an enhanced internal memory and an additional timer. An explanation of the new special function registers associated with the new internal peripheral is also given.

4.1 8032 Extras

The 8032 microcontroller is the 8051's "big brother." It is a slightly more powerful microcontroller, sporting a number of additional features which the developer may make use of.

Hex Byte Address	Notes	Notes	Hex Byte Address
FF Upper 128 bytes 80	(8032 ONLY) Accessible by Indirect Addressing only	SFR area Acces-sible by Direct Address-ing only	FF 80
7F Lower 128 bytes 00	Accessible by Direct and Indi-rect Ad-dressing.		

Table 4-1 8032 Total Internal RAM organisation

- 256 bytes of Internal RAM (compared to 128 in the standard 8051). The lower 128 bytes are accessible using either Direct or Indirect addressing modes. The additional upper 128 bytes can only be accessed using the Indirect addressing mode.
- A third 16-bit timer (Timer 2), capable of a number of new operating modes, interrupts and 16-bit reloads.
- The serial port can now make use of either Timer 1 or Timer 2 to generate the baud rates.
- Additional SFRs to support the functionality offered by the third timer. These SFRs still reside in the 80h-FFh area accessible only by Direct Addressing to differentiate them from the Indirectly addressable internal RAM used for program stack and/or variables.

Table 4-1 shows the internal memory differences that there are between the 8051 and 8032. The remainder of this chapter will explain these additional features offered by the 8032, and how they are used within user programs.

Download free eBooks at bookboon.com

4.2 256 Bytes of Internal RAM

The standard 8051 microcontroller contains 128 bytes of Internal RAM that are available to the developer as working memory for variables and/or for the operating stack. Instructions that refer to internal ram addresses in the range of 00h through 7Fh refer to the basic 8051.

Addresses which are accessible using direct addressing, in the range of 80h through FFh refer to Special Function Registers (SFRs).

Although the 8032 has 256 bytes of Internal RAM, the above mentioned method of referencing them remains true. Using Direct Addressing, any address between 00h and 7Fh refers to Internal RAM whereas any address in the range of 80h through FFh refers to SFRs.

The 8032's additional Internal RAM may only be accessed using Indirect Addressing. Indirect addressing always refers to Internal RAM and never to an SFR.

Thus, to read the value contained in Internal RAM address 90h, we would need to code something along the following lines:

```
MOV R0, #90h        ;Set the indirect address to 90h
MOV A, @R0          ;Read the contents of Internal RAM pointed to by R0
```

The above code first assigns the value 90h to the register R0. It subsequently reads, indirectly, the contents of the address contained in (pointed by) R0 (90h). Thus, after these two instructions have executed, the Accumulator will contain the value of Internal RAM address 90h.

Download free eBooks at bookboon.com

Hex Byte Address	Hex Bit Address								Notes
FF 80	Additional Indirectly Addressable General Purpose RAM								Can be used as a STACK Area by loading SP with 7FH or higher
7F 30	Directly and Indirectly Addressable General Purpose RAM								
2F	7F	7E	7D	7C	7B	7A	79	78	Bit Addressable Section
2E	77	76	75	74	73	72	71	70	
2D	6F	6E	6D	6C	6B	6A	69	68	
2C	67	66	65	64	63	62	61	60	
2B	5F	5E	5D	5C	5B	5A	59	58	
2A	57	56	55	54	53	52	51	50	
29	4F	4E	4D	4C	4B	4A	49	48	
28	47	46	45	44	43	42	41	40	
27	3F	3E	3D	3C	3B	3A	39	38	
26	37	36	35	34	33	32	31	30	
25	2F	2E	2D	2C	2B	2A	29	28	
24	27	26	25	24	23	22	21	20	
23	1F	1E	1D	1C	1B	1A	19	18	
22	17	16	15	14	13	12	11	10	
21	0F	0E	0D	0C	0B	0A	09	08	
20	07	06	05	04	03	02	01	00	
1F 18	Register Bank 3 (R0 – R7)								Bank is selected using RS0 and RS1 in the PSW register. See SFRs.
17 10	Register Bank 2 (R0 – R7)								
0F 08	Register Bank 1 (R0 – R7)								
07 00	Register Bank 0 (R0 – R7)								

Table 4-2 8032 Internal RAM organisation

It is very important to understand that the above code just mentioned, is not the same as the following:

```
MOV A, 90h ;Reads the contents of SFR 90h (P1)
```


Download free eBooks at bookboon.com

This instruction uses direct addressing; recall that direct addressing reads Internal RAM when the address is in the range of 00h through 7Fh, and reads an SFR when the address is in the range of 80h through FFh. Thus in the case of this second example, the move instruction reads the value of SFR 90h, which happens to be P1 (I/O Port 1).

The importance of using the correct addressing mode cannot be over-emphasised. It should however be noted here, that when using a compiler to compile a C source code program into machine code, the compiler automatically would use the correct addressing form. The compiler would know where the variable or SFR is located and could therefore decide which type of addressing mode is required to access that variable or SFR.

4.3 Additional Timer 2

An important addition for the 8032 is the availability of a third timer, referred to as Timer 2. We shall now deal with the SFRs connected with this timer, as well as the various modes of operation for this extra timer.

Click on the ad to read more

Download free eBooks at bookboon.com

4.3.1 New SFRs for 8032's third timer (T2)

In addition to the 8051's standard 21 SFRs, the 8032 adds an additional 5 SFRs related to the 8032's third timer as shown shaded in Table 4-3a. All of the original 8051 SFRs shown in Tables 4-3a and 4-3b function exactly as they do in the 8051 – the 8032 simply adds new SFRs, it doesn't change the definition of the standard SFRs. The five new SFRs are in the range of C8h to CDh (SFR C9h is not defined), plus some additional bits shown shaded.

Note that the TL2/TH2 register pair and the RCAP2L/RCAP2H register pair occupy consecutive memory addresses, low byte first, contrary to the registers available for Timer 0 and Timer 1. This means that in KEIL C, we can load the whole 16-bit Timer 2 counter registers TL2 and TH2 by using the SFR16 data type.

Hex Byte Address	Hex Bit Address								Symbol
FF – F9	Not implemented on chip								–
* F8 *	Not implemented on chip								–
F7 – F1	Not implemented on chip								–
* F0 *	F7	F6	F5	F4	F3	F2	F1	F0	B
EF – E9	Not implemented on chip								–
* E8 *	Not implemented on chip								–
E7 – E1	Not implemented on chip								–
* E0 *	E7	E6	E5	E4	E3	E2	E1	E0	ACC
DF – D9	Not implemented on chip								–
* D8 *	Not implemented on chip								–
D7 – D1	Not implemented on chip								–
* D0 *	D7	D6	D5	D4	D3	D2	D1	D0	PSW
CF – CE	Not implemented on chip								–
CD									TH2
CC									TL2
CB									RCAP2H
CA									RCAP2L
C9	Not implemented on chip								–
C8	CF	CE	CD	CC	CB	CA	C9	C8	T2CON
C7 – C1	Not implemented on chip								–
* C0 *	Not implemented on chip								–
BF – B9	Not implemented on chip								–
* B8 *	–	–	BD	BC	BB	BA	B9	B8	IP
B7 – B1	Not implemented on chip								–
* B0 *	B7	B6	B5	B4	B3	B2	B1	B0	P3
AF – A9	Not implemented on chip								–
* A8 *	AF	–	AD	AC	AB	AA	A9	A8	IE
A7 – A1	Not implemented on chip								–
* A0 *	A7	A6	A5	A4	A3	A2	A1	A0	P2

Table 4-3a 8032 Special Function Registers (SFRs)-DIRECT addressing ONLY

Download free eBooks at bookboon.com

Hex Byte Address	Hex Bit Address								Symbol
9F – 9A	Not implemented on chip								-
99									SBUF
* 98 *	9F	9E	9D	9C	9B	9A	99	98	SCON
97 – 91	Not implemented on chip								-
* 90 *	97	96	95	94	93	92	91	90	P1
8F –8E	Not implemented on chip								-
8D									TH1
8C									TH0
8B									TL1
8A									TL0
89									TMOD
* 88 *	8F	8E	8D	8C	8B	8A	89	88	TCON
87									PCON
86 – 84	Not implemented on chip								-
83									DPH
82									DPL
81									SP
* 80 *	87	86	85	84	83	82	81	80	P0

Table 4-3b 8032 Special Function Registers (SFRs)-DIRECT addressing ONLY

The procedure would be as follows:

- We first declare a variable of type SFR16, say using

  SFR16 T2REG = 0xCC;

- We then simply write

  T2REG = 0x1234; //This would load 34H in TL2 and 12H in TH2.

Download free eBooks at bookboon.com

4.3.2 T2CON SFR (C8H)

Bit-addressable				
Bit	**Name**	**Alternate Names**	**Bit Hex Address**	**Explanation of Function**
7	TF2	T2CON.7	CF	Timer 2 overflow. This bit is set when T2 overflows. When T2 interrupt is enabled, this bit will cause the interrupt to be triggered. This bit will not be set if either TCLK or RCLK bits are set
6	EXF2	T2CON.6	CE	Timer 2 External flag. Set by a reload or capture caused b a 1-0 transition on T2EX (P1.1), but only when EXEN2 is set. When T2 interrupt is enabled, this bit will also trigger the interrupt.
5	RCLK	T2CON.5	CD	Timer 2 Receiver Clock. When this bit is set, Timer 2 will be used to determine the serial port receive baud rate. When cleared, Timer 1 will be used as the baud rate generator.
4	TCLK	T2CON.4	CC	Timer 2 Transmitter Clock. When this bit is set, Timer 2 will be used to determine the serial port transmitter baud rate. When cleared, Timer 1 will be used as the baud rate generator.
3	EXEN2	T2CON.3	CB	Timer 2 External enable. When set, a 1-0 transition on T2EX (P1.1) will cause a capture or a reload to occur.
2	TR2	T2CON.2	CA	Timer 2 run. When set, timer 2 will start. Timer 2 will stop when this bit is cleared.
1	C/T2	T2CON.1	C9	Timer 2 Counter/Interval timer. If cleared, Timer 2 is an interval counter. If set, Timer 2 is incremented by 1-0 transitions on T2 (P1.0).
0	CP/RL2C	T2CON.0	C8	Timer 2 Capture/Reload. If cleared, auto reload occurs on timer 2 overflow, or T2EX 1-0 transition if EXEN2 is set. If set, a capture will occur on a 1-0 transition of T2EX, if EXEN2 is set.

Table 4-4 T2CON (C8H) SFR

The operation of Timer 2 (T2) is controlled almost entirely by the T2CON SFR shown in Table 4-4, at address C8h. Note that since this SFR has an address which is divisible by 8, then it is Bit-addressable.

Download free eBooks at bookboon.com

4.3.3 Timer 2 as a baud-rate generator

Timer 2 may be used as a baud rate generator. This is accomplished by setting either RCLK (T2CON.5) or TCLK (T2CON.4). With the standard 8051, Timer 1 is the only timer which may be used to determine the baud rate of the serial port. Additionally for the standard 8051 the receive and transmit baud rate must be the same.

With the 8032, however, we may configure the serial port to receive at one baud rate and transmit at another baud rate. For example, if RCLK is set and TCLK is cleared, serial data will be received at the baud rate determined by Timer 2 whereas the baud rate of transmitted data will be determined by Timer 1.

Determining the auto-reload values of Timer 1 for a specific baud rate was discussed in section 2.12.2. Timer 2 can similarly be programmed, the only difference is that in the case of Timer 2, the auto-reload value is placed in RCAP2H and RCAP2L, and the value is a 16-bit value rather than an 8-bit value.

The baud rates (in serial modes 1 and 3) are determined by Timer 2's overflow rate as follows:

$$\text{Baud Rate} = (\text{Timer 2 Overflow Rate})/ 16$$

The Timer can be configured for either timer or counter operation. The timer operation is a little different for Timer 2 when it is being used as a baud rate generator. Normally, as a timer it would increment every machine cycle (oscillator frequency/12). However, when being used as a baud rate generator, it increments at every state time (oscillator frequency/2) and the equations for determining the variable baud rate (serial modes 1 or 3), using Timer 2 become:

$$\text{Baud Rate} = (\text{Osc. Frequency})/(32 [65536 – (\text{RCAP2H,RCAP2L})]) \quad \textbf{… Equation 4-1}$$

or

$$(\text{RCAP2H,RCAP2L}) = [2097152 – (\text{Osc. Freq.})/(\text{Baud Rate})] / 32 \quad \textbf{… Equation 4-2}$$

Where (RCAP2H,RCAP2L) is the content of RCAP2H and RCAP2L taken as a 16-bit unsigned integer.

Thus to get a baudrate of 345600 baud with an 11.0592 MHz clock, using equation 4-2 we would need to load (RCAP2H,RCAP2L) with 65535. Thus the intial Timer 2 registers TH2 and TL2 as well as the reload registers RCAP2H and RCAP2L would all be loaded with 255 or FFH. An example using Timer 2 as the baud rate generator is given in Appendix F.

Download free eBooks at bookboon.com

Note that when Timer 2 is used as a baud rate generator (having either TCLK or RCLK set), the Timer 2 Overflow Flag (TF2) will not be set, therefore the Timer 2 interrupt does not have to be disabled. Thus when Timer 2 is being used as a baud rate generator, T2EX can still be used as an extra external interrupt if required.

4.3.4 Timer 2 in auto-reload mode

The first mode in which Timer 2 may be used is Auto-Reload. The auto-reload mode functions just like Timer 0 and Timer 1 in auto-reload mode, except that the Timer 2 auto-reload mode performs a full 16-bit reload (recall that Timer 0 and Timer 1 only have 8-bit reload capability). When a reload occurs, the value of TH2 will be reloaded with the value contained in RCAP2H and the value of TL2 will be reloaded with the value contained in RCAP2L.

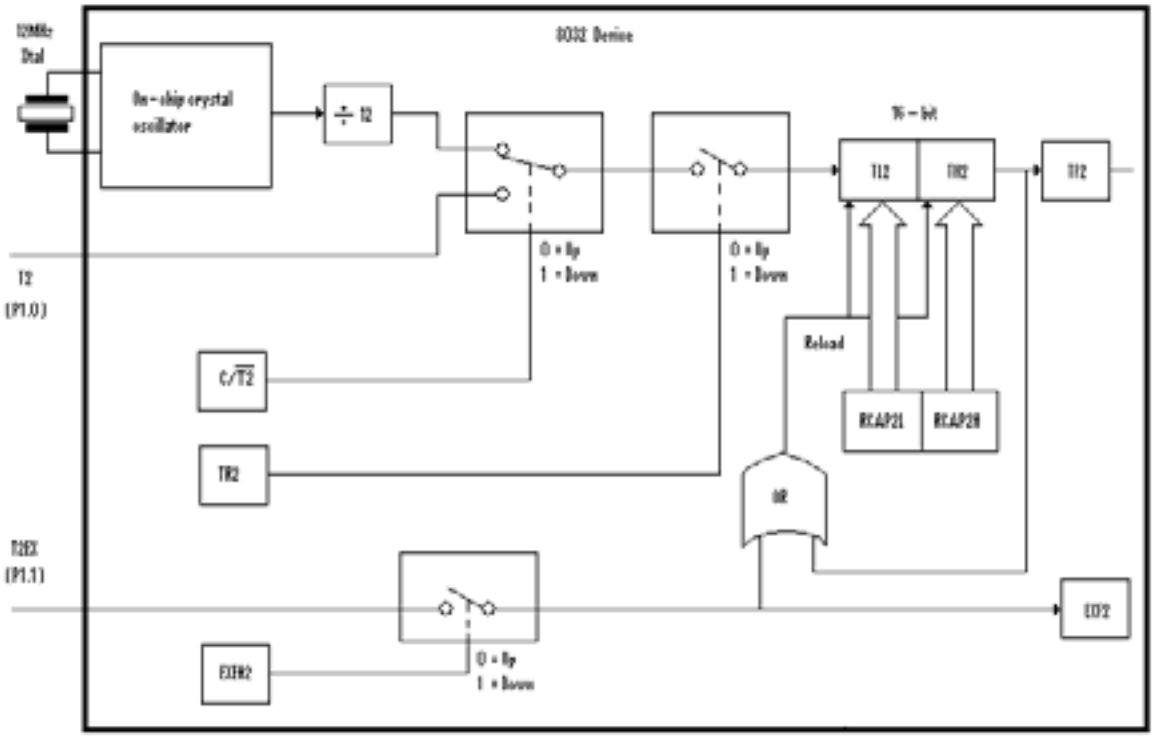


Figure 4-1 Timer 2 Auto-reload Mode

To operate Timer 2 in auto-reload mode, the CP/RL2 bit (T2CON.0) must be cleared. In this mode, Timer 2 (TH2/TL2) will be reloaded with the reload value (RCAP2H/RCAP2L) whenever it overflows from FFFFh back to 0000h. An overflow of Timer 2 will cause the TF2 bit to be set, which will cause an interrupt to be triggered, if Timer 2 interrupt is enabled. Note that TF2 will not be set on an overflow condition if either RCLK or TCLK (T2CON.5 or T2CON.4) are set, which is the case if Timer 2 is being used as a baud rate generator.

Download free eBooks at bookboon.com

Additionally, by also setting EXEN2 (T2CON.3), a reload will also occur whenever a 1-0 transition is detected on T2EX (P1.1). A reload which occurs as a result of such a transition will cause the EXF2 (T2CON.6) flag to be set, triggering a Timer 2 interrupt if the said interrupt has been enabled.

4.3.5 Timer 2 in Capture mode

A new mode specific to Timer 2 is called "Capture Mode." As the name implies, this mode captures the value of Timer 2 (TH2 and TL2) into the capture SFRs (RCAP2H and RCAP2L). To put Timer 2 in capture mode, CP/RL2 (T2CON.0) must be set, as must be EXEN2 (T2CON.3).

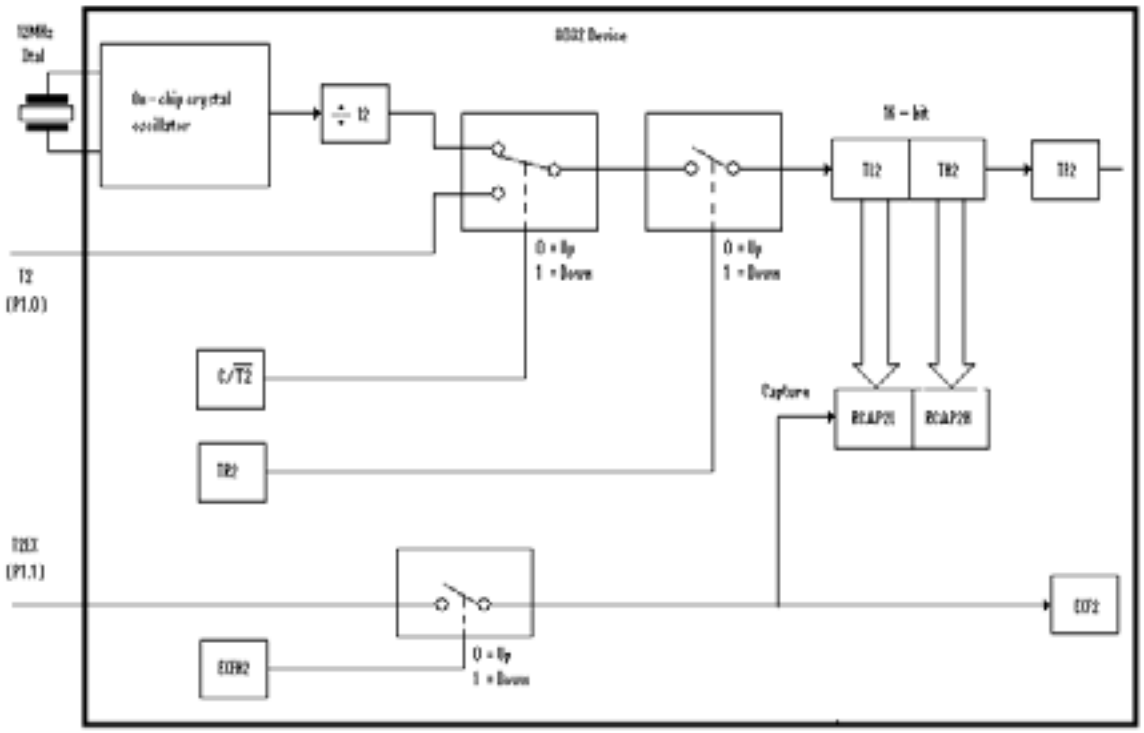


Figure 4-2 Timer 2 in 16-bit capture mode

When configured as mentioned above, a capture will occur whenever a 1-0 transition is detected on T2EX (P1.1). At the moment the transition is detected, the current values of TH2 and TL2 will be copied into RCAP2H and RCAP2L, respectively. At the same time, the EXF2 (T2CON.6) bit will be set, which will trigger an interrupt if Timer 2 interrupt is enabled.

NOTE 1: Even in capture mode, an overflow of Timer 2 will result in TF2 being set and an interrupt being triggered.

Download free eBooks at bookboon.com

NOTE 2: Capture mode is an efficient way to measure the time between events. At the moment that an event occurs, the current value of Timer 2 will be copied into RCAP2H/L. However, Timer 2 will not stop and an interrupt will be triggered. Thus our interrupt routine may copy the value of RCAP2H/L to a temporary holding variable without having to stop Timer 2. When another capture occurs, our interrupt can take the difference between the two values to determine the elapsed time. Again, the main advantage is that we do not have to stop Timer 2 to read its value, as is the case with Timer 0 and Timer 1, where there is the possibility of reading the wrong value if the timer count happens to be close to a roll-over .

4.3.6 Timer 2 Interrupt

As is the case with the other two timers, Timer 2 can be configured to trigger an interrupt. In fact, as can be seen in Table 4-5 a number of situations can trigger a Timer 2 interrupt.

To enable Timer 2 interrupt, set ET2 (IE.5) and it should be noted that this bit of IE is only valid on an 8032 or other devices of the 8051 family which have a Tmer 2 on board. Similarly, the priority of Timer 2 interrupt can be configured using PT2 (IP.5). As always, we have to make sure to also set the EA (IE.7) bit when enabling any interrupt. This will ensure that the controller would recognize the interrupt.

Download free eBooks at bookboon.com

Interrupt Name	Interrupt Number	Flag	Interrupt Hex Vector Address
External 0	0	IE0	0003
Timer 0	1	TF0	000B
External 1	2	IE1	0013
Timer 1	3	TF1	001B
Serial	4	RI or TI	0023
Timer 2	5	TF2 or EXF2	002B

Table 4-5 8032 Interrupt Vector Table location

Once Timer 2 interrupt has been enabled, a Timer 2 interrupt will be triggered whenever TF2 (T2CON.7) or EXF2 (T2CON.6) are set. The Timer 2 Interrupt routine must be placed at 002Bh in code memory.

NOTE: Like the Serial Interrupt, Timer 2 interrupt does not automatically clear the interrupt flag that triggered the interrupt. Since there are two conditions that can trigger a Timer 2 interrupt, either TF2 or EXF2 being set, the microcontroller does not reset these flags automatically when jumping to the ISR. Therefore we have to add some code in the interrupt routine which determines the source of the interrupt and act accordingly. It is possible (and even probable!) that we will want to do one thing when the timer overflows and something completely different when a capture or reload is triggered by an external event. Thus it is imperative to always clear TF2 and/or EXF2 in the Timer 2 ISR. Failing to do so will cause the interrupt to be triggered repeatedly until the bits are cleared.

Download free eBooks at bookboon.com

5 Evaluation Boards

Here we discuss just a few of the many development boards which are widely available for the 8051 family of micro-controllers. These evaluation boards can be used to develop and test the program on the actual hardware and arte especially useful for students whilst gaining experience on the micro-controller. Actual add-on hardware (such as LCD displays, servo motors, LEDs, keyboards) can also be connected to these boards in order to implement the required project. We discuss and explain the main features of the Flite-32 board from Flite Electronics International Limited (http://www.flite.co.uk) using an 8032 micro-controller, the NMIY-0032 8051 board from New Micros, Inc. (http://www.newmicros.com) using an 8051 and the C8051F020TB from Silicon Labs (http://www.silabs.com) using the very high performance C8051F020 micro-controller. These are all available at the University of Malta Communications and Computer Engineering department laboratories for student use.

Naturally, if you are using another kind of board, you might wish to skip this chapter completely or just skim through it perhaps you might pick some new idea.

5.1 FLITE-32 Development Board

The FLITE-32 development board is available from Flite Electronics International Limited (see http://www.flite.co.uk/flite-flt-32-803251-training-system.htm), is used extensively in our course program to train the students on the 8051-family of microcontrollers. It uses the 8032 device, thus having available three timers and 256 bytes of internal RAM. Co effectively mplete schematics and manuals are available from this site.

The board also has some additional peripherals, which further enhance its capabilities. Namely it has:

- 32KB monitor EPROM program
- 8KB (which we have expanded to 32KB) external memory
- 8255 Peripheral Interface Adaptor IC providing an additional 24 I/O lines (3 I/O ports having 8-bits each)
- 26C91 Universal Asynchronous Receiver/Transmitter (UART) providing an additional serial port, running at a maximum of 38400 baud, using socket P2

Download free eBooks at bookboon.com

Figure 5-1 Flite-32 Board

Discover the truth at www.deloitte.ca/careers

Deloitte.

© Deloitte & Touche LLP and affiliated entities.

Click on the ad to read more

Download free eBooks at bookboon.com

5.1.1 FLITE-32 General Setup

Microprocessor: Intel 8032

Internal RAM: 256 bytes

External RAM: either 8KB using the 6264 SRAM IC.
(address 8000H to 9FFFH, with J2/J3 having links 2-3)
or 32KB using the 62256 SRAM IC.
(address 8000H to FDFFH, with J2/J3 links 1-2)

User area for code and data starts from 8100H

Reserved Areas: The monitor program residing in the 32KB EPROM, having Address range 0000 to 7FFFH, also uses some memory in the internal RAM and in the external RAM. In particular if some printing routines from the monitor are being used, internal RAM locations 20H and 21H should be avoided together with external RAM area 8000H to 80FFH.

5.1.2 Peripherals:

The lists below describe the address mapping and register names given to the peripherals found on the board.

8255 Input-Output IC:

Port A – FF40H

Port B – FF41H

Port C – FF42H

Control – FF43H

2691 External UART (P2):

```
UART_2691_BASE   EQU   0FFF8H                ; UART BASE ADDRESS, on Flite-32
UART_MR1         EQU   UART_2691_BASE        ; MR1 – Mode Register 1
UART_MR2         EQU   UART_2691_BASE        ; MR2 – Mode Register 2
UART_SR          EQU   UART_2691_BASE + 1; READ SR – Channel Status Register
UART_CSR         EQU   UART_2691_BASE + 1; WRITE CSR – Clock Select Register
UART_CR          EQU   UART_2691_BASE + 2; WRITE CR – Command Register
UART_RHR         EQU   UART_2691_BASE + 3; READ RHR – Rx Holding Reg
```


Download free eBooks at bookboon.com

```
UART_THR        EQU   UART_2691_BASE + 3; WRITE THR – Tx Holding Reg
UART_ACR        EQU   UART_2691_BASE + 4; WRITE ACR – Auxiliary Control
UART_ISR        EQU   UART_2691_BASE + 5; READ ISR – Interrupt Status Register
UART_IMR        EQU   UART_2691_BASE + 5; WRITE IMR – Interrupt Mask Reg
UART_CTU        EQU   UART_2691_BASE + 6
; READ/WRITE CTU – Counter Timer Upper Register
UART_CTL        EQU   UART_2691_BASE + 7
; READ/WRITE CTL – Counter Timer Lower Register
RX              EQU   0FFE8H              ; READ RX DATA input (socket P2).
                                          ; This is used if required to auto-determine the baud rate
```

The 2691 can also be used under interrupt control, by connecting the link J8 to External 0 Interrupt (link 5-6), or External 1 Interrupt (link 5-4).

The main memory map of the board is shown in Table 5-1.

<table>
<tr><th>Internal RAM</th><th>External EPROM or RAM</th><th>Remarks</th></tr>
<tr><td rowspan="6">Not
Available</td><td>FFFFH</td><td>UART</td></tr>
<tr><td>FE40H</td><td>8255</td></tr>
<tr><td>FE00H</td><td>Peripherals</td></tr>
<tr><td>FDFFH
If 32KB External
RAM
(Code and Data)
A000H</td><td>User
Area</td></tr>
<tr><td>9FFFH
8KB External
RAM
(Code and Data)
8100H</td><td>User
Area</td></tr>
<tr><td>80FFH
to
8000H</td><td>Reserved
For monitor
Use</td></tr>
<tr><td>FFH
Internal
On-chip
Memory
00H</td><td>7FFFH
External
EPROM
(Monitor Code Area)
0000H</td><td>Monitor
Program</td></tr>
</table>

Table 5-1 FLT-32 Memory map

Download free eBooks at bookboon.com

Additional RAM can be added by replacing the default RAM chip. Even the EPROM can be replaced with a smaller or larger capacity EPROM. In every case, some jumper links would have to be re-arranged to get the correct address coverage.

The Interrupt Vector Table, which normally resides in the low ROM area, (0000H to 0030H), is re-mapped on start-up and points to the external RAM area starting at 8000H. For example, in the monitor EPROM, at address location 0003H, which is the normal EXT0 interrupt vector address, there is written the instruction LJMP 8000H (jump to address 8000H).

Click on the ad to read more

Download free eBooks at bookboon.com

Interrupt Number	0	1	2	3	4	5
Interrupt Name	External 0	Timer 0	External 1	Timer 1	Serial	Timer 2
Standard Vector Address	0003H	000BH	0013H	001BH	0023H	002BH
FLT-32 Vector Address (Monitor Version V0)	8000H	N/A	8010H	8018H	8020H	8028H
FLT-32 Vector Address (Monitor Version 3)	8000H	* 8008H	8010H	8018H	8020H	8028H
Flags Causing Interrupt	IE0	IT0	IE1	TF1	RI & TI	TF2 & EXF2
Interrupt Enable bit (+ EA)	EX0	ET0	EX1	ET1	ES	ET2
Interrupt Priority bit	PX0	PT0	PX1	PT1	PS	PT2
Falling Edge Triggering	IT0		IT1			

Table 5-2 FLT-32 Interrupt Vector Table

* This 8008H address is only available if using version 3 monitor EPROM which we modified. The modification in the EPROM involved over-writing locations starting at 0008H with a JMP 8008H instruction, thus replacing the original jump to the Single Step command which existed in the standard monitor program. Otherwise, it would not be available since this interrupt is normally used by the default EPROM monitor for the SINGLE STEP command.

Note:

Link J8 can be used to divert signals to EXT0 or EXT1 interrupt pins of the CPU as described here:

External Interrupt pin 5 on the optional P4 connector can be connected to EXT0 (link 2-3) or to EXT1 (link 1-2)

Also interrupts from the external 2691 UART can also be diverted to EXT0 (link 5-6) or to EXT1 (link 5-4)

Download free eBooks at bookboon.com

Timers:

Timer 0 is used by the monitor Version 0 program (under interrupt control) whenever Single Step or Trace is being performed. Hence it is available to the user (not using interrupts since the vector is not available in RAM) providing no tracing is being done. Moreover, if the program is intended for stand-alone EPROM use (that is we eventually will replace the monitor EPROM with another EPROM which we will write ourselves containing just our application program), then even the interrupt can be used (using Timer 0 vector address 000BH). Otherwise, the interrupts connected with Timer 0 can only be used with the modified version 3 of the monitor program, which removes the Single Step function.

Timer 1 is used (not under interrupt control) as the baud rate generator whenever socket P3 (the 8032 internal UART) is being used by the monitor program, usually to output characters to a printer or to a terminal. Hence if the monitor routines for printing using socket P3 are not being used, then Timer 1 can be used in any mode as we deem fit.

Serial Printer:

Socket P3 (serial printer) is selected by pressing

W (set baud rate),

WO (enable printer) and

WX (disable printer),

when in the monitor prompt.

5.1.3 Some Important FLITE-32 Monitor Routines:

Use LCALL <address> to use these routines which are already coded in the monitor program. CALLed from the user program, they will RETurn on completion.

- 0090H Convert character in the ACC to upper case.
- 0093H Send the character in the ACC to external UART (socket P2).
- 0096H Get character from external UART (socket P2) to the ACC.
- 0099H Send Carriage Return and Line Feed to external UART (socket P2).
- 009CH Send to UART message pointed to by DPTR, terminated by 00H.
- 009FH Restart board without initialising.
- 00A2H As above but with sign-on message.
- 00A5H Send to UART the Hex word in found in registers AB as ASCII characters.
- 00A8H Convert the ASCII value in ACC to Hex.

Download free eBooks at bookboon.com

It should be noted here that the monitor program makes use of the internal RAM memory from 00H to 5FH. In fact, the monitor program intialises the stack pointer SP to 5FH, thus having the actual stack starting at 60H. Use of the internal RAM area 00 to 7FH should be avoided if one intends to make use of the functions/commands of the monitor program. For the PaulOS RTOS, all monitor commands and functions are ignored, and all the 256 bytes of the internal RAM area, from 00H to FFH are used extensively by the RTOS and any application program which is running.

5.2 Typical Settings for KEIL uV2

USE the following settings in Options for Target 1

Memory Model: LARGE: VARIABLES IN XDATA
Code Model: LARGE: 64K Program

	START	SIZE	(If using 32KB RAM)
CODE:	0X8100	0X5D00	
RAM:	0XDE00	0X2000	

Interrupt Vector address at 0x7FFD (click on C51 tab)

Click on the ad to read more

Download free eBooks at bookboon.com

	START	SIZE	(If using 8KB RAM)
CODE:	0X8100	0X1B00	
RAM:	0X9C00	0X0400	

Interrupt Vector address at 0x7FFD (click on C51 tab)

	START	SIZE	(If using 32KB EPROM)
CODE:	0X0000	0X8000	
RAM:	0X8000	0X7E00	

Interrupt Vector address at 0x0000 (click on C51 tab)

5.3 The NMIY-0031 Board

This is another low-cost 8051 evaluation board, available from New Micros Inc. (see http://www.newmicros.com/cgi-bin/store/order.cgi?form=prod_detail&part=NMIY-0031). Complete schematics and user manual are available from this site.

5.3.1 NMIY General Data

NMIY uses an 8051 micro-controller having two timers (T0 and T1), one UART (equivalent to the P3 socket found on the FLITE-32 board) and only 128 bytes internal RAM.

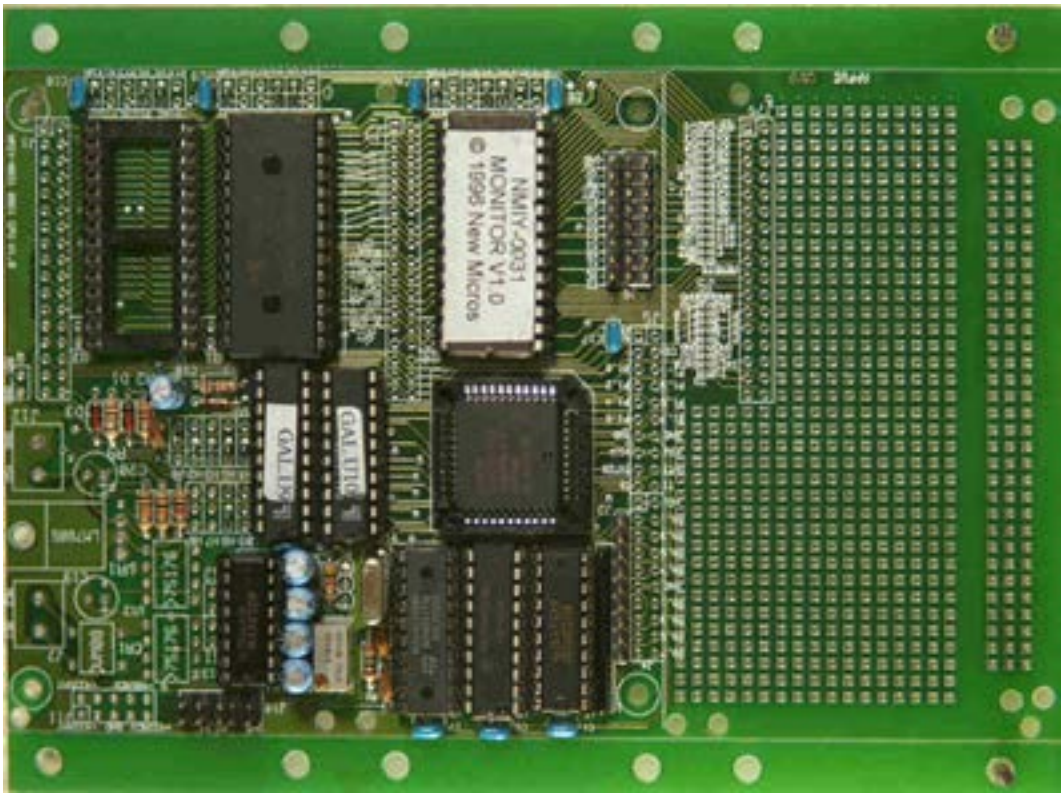


Figure 5-2 NMIY-0031 Board

Download free eBooks at bookboon.com

T1 can be used for the serial port. The serial interrupt is not available when using the monitor EPROM.

Initially, whilst developing and loading the hex file onto the board, use only a serial baud-rate of 9600 baud (8-N-1-N), 2 ms/character and 5 ms/line delays in the terminal software (such as TERATERM) settings. The baud rate can then be changed to any standard value in the user's source file program, depending on the application requirements.

Use CAPITAL letters to talk to the monitor program:

H – Help
L – Load
X – Execute

As standard, the board has only 8KB of monitor code and 8KB of RAM to use for code and data. Additional RAM (for decimal/hex sizes see Table 5-3 where the hex size is shown using the 0x prefix notation instead of the H suffix notation) can be plugged in, up to 64Kbytes (say two 32KB RAM ICs in sockets U3 and U4). See link settings Table 5-4.

We may start program code from 8100H (same as in FLITE-32), so we may adjust the ORG position in the STARTUP.A51 file accordingly.

Click on the ad to read more
Download free eBooks at bookboon.com

Also, in the STARTUP.A51 file, the IDATALEN has to be modified to 80H (128 bytes).

The Interrupt Vector Base address should be adjusted to 8000H (not 7FFDH as for the FLITE-32). This is because the interrupt vectors use slightly different mapping addresses than on the FLITE-32 board.

RAM SIZE KBYTES	RAM SIZE BYTES (DECIMAL)	RAM SIZE BYTES (HEX)
0.5	512	0x0200
1	1024	0x0400
2	2048	0x0800
3	3072	0x0C00
4	4096	0x1000
5	5120	0x1400
6	6144	0x1800
7	7166	0x1C00
8	8192	0x2000
9	9216	0x2400
10	10240	0x2800
11	11264	0x2C00
12	12288	0x3000
13	13312	0x3400
14	14336	0x3800
15	15360	0x3C00
16	16384	0x4000
17	17408	0x4400
18	18432	0x4800
19	19456	0x4C00
20	20480	0x5000
30	30720	0x7800
32	32768	0x8000

Table 5-3 RAM Size Dec-Hex Conversion

5.3.2 MEMORY MAPPING:

Addresses above 0XFC00 (or FC00H) are reserved.
In fact address 0xFFFC refers to the additional external IC input/output latched port.
This is available from the J4 socket.
You may use for example, in your C program:

Download free eBooks at bookboon.com

```
#define MyPort          XBYTE [0XFFFC]
// define MyPort address (FFFCH) as the input/output port
// PB0 to PB7 will be the output bits
// PA0 to PA7 will be the input bits
unsigned char dataout, datain;

MyPort = dataout;       /* send data stored in variable dataout, to the output port (PB0-PB7) */
datain = MyPort;        /* read data from MyPort (PA0-PA7) to the datain variable */
```

Socket	U2 (EPROM)			U3 (RAM)		U4 (RAM)
	H3 (1-2) Code only	H3 (2-3) Code + Data	H4 32K or 64K	H2 (1-2) Code + Data	H2 (2-3) Data only	Data only
2764 8KB	N = 0011001 0x0000 – 0x1FFF		(1-2)	O = 011001 0x8000 – 0x9FFF		P = 011001 0x0000 - 0x1FFF
27128 16KB	N = 0010101 0x0000 – 0x3FFF		(1-2)	O = 010101 0x8000 – 0xBFFF		P = 010101 0x0000 – 0x3FFF
27256 32KB	N = 0010110 0x0000 – 0x7FFF		(1-2)	O = 100101 0x8000 – 0xFC00		P = 100101 0x0000 – 0x7FFF
27512 64K	N = 1000110 0x0000 – 0xFFFF		(2-3)	Not available		Not available

Table 5-4 External Memory (Link Settings)

Normally, whilst developing the program:

- U2 is set to code only (containing the monitor program)
- U3 is set to code + data
- U4 (if available) is set to data only (cannot set it in any other mode)

We must make sure to set the memory map in the Target Options to reflect our particular memory setup.

5.3.3 Input-Output connections

J4 is the latched input – output port (address 0xFFFC)
PA pins are the INPUT pins, and PB pins are the OUTPUT pins.
Pin 1 is in the direction of U1 and closest to U2.
Make sure that we have the correct pin orientation!

Download free eBooks at bookboon.com

Pin No:	Signal		Pin No:	Signal
1	+5 V		2	+5 V
3	PA^0		4	PA^1
5	PA^2		6	PA^3
7	PA^4		8	PA^5
9	PA^6		10	PA^7
11	GND		12	GND
13	PB^0		14	PB^1
15	PB^2		16	PB^3
17	PB^4		18	PB^5
19	PB^6		20	PB^7

Table 5-5 NMIY J4 Pinouts

J5 is the 8051 port 1, external timer inputs and external interrupts.
Pin 1 is in the direction of U1 and closest to U2.
Make sure that we have the correct pin orientation!

Click on the ad to read more

Download free eBooks at bookboon.com

Pin No:	Signal		Pin No:	Signal
1	GND		2	GND
3	RST		4	CSX
5	P1^0		6	P1^1
7	P1^2		8	P1^3
9	P1^4		10	P1^5
11	P1^6		12	P1^7
13	GND		14	GND
15	+5 V		16	+5 V
17	INT 0		18	INT 1
19	T0		20	T1

Table 5-6 NMIY J5 Pinouts

J6 is the LCD connector. P1 is located in the row nearest to U2 and closest to the board edge. Address 0xFFF8 is used to send COMMANDS and address 0xFFF9 is used for sending DATA to the LCD.

Pin No:	Signal		Pin No:	Signal		Pin No:	Signal
1	GND		2	+5 V		3	GND
4	CONTRAST		5	A0		6	CONTRAST
7	WR1		8	E1		9	WR1
10	D0		11	D1		12	D0
13	D2		14	D3		15	D2
16	D4		17	D5		18	D4
19	D6		20	D7		21	D6
22	N.C.		23	E3		24	N.C.

Table 5-7 NMIY J6 Pinouts

5.4 C8051F020TB

This is at the time of writing, one of the latest super-charged versions of the 8051 family. It is the product of Silicon Labs.

Download free eBooks at bookboon.com

Figure 5-3 C8051F020 Board

Further details, manuals and example programs can be found at the Silicon Labs site, whose contact details are being listed here under:

Silicon Laboratories Inc.
4635 Boston Lane
Austin,
Texas TX 78735
USA
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032
Email: MCUinfo@silabs.com
Internet: www.silabs.com

Download free eBooks at bookboon.com

6 Programming in C with KEIL µV2 IDE

C Compilers are used to translate programs written in C to the native language of the intended target processor. In our case the intention is to write programs in C which ultimately are intended to be executed on an 8051 micro-controller or any of its derivatives. There are a large number of such compilers available, and even some device makers themselves have their own Integrated Development Environment (IDE) software.

This chapter explains the use of the KEIL IDE which is practically the industry-standard in this field. It supports every level of software developer from the professional applications engineer to the student just learning about embedded software development. Further information about this IDE can be obtained from the web site http://www.keil.com/uvision/uv4.asp. Detailed examples are given so that after reading this chapter, we would be able to set it up so as to reflect the actual hardware which we intend to use for our particular task or project. Example programs written in C are given to help the reader grasp the basic principles involved when programming micro-controllers. Most of the tables and diagrams are taken directly from the Keil uV2 user manuals and screen shots.

Click on the ad to read more

Download free eBooks at bookboon.com

Programming in C for the 8051 micro-controller with the KEIL development system and using the RTOS is not problematic. The Integrated Development Environment (IDE) provided with KEIL µVx (the latest version at time of printing is µV4) is very user friendly and similar to other IDEs used on the standard PCs for C# or C++ etc. Familiarity with any such tools will greatly help in grasping the basics of the KEIL environment. Before explaining how to set up a project to write programs with KEIL µVx, let us first mention a very important point about the way variables are stored on the 8051 and how they are handled by the compiler. The KEIL user manual provides more detailed information on the subject, but the most important details are being reproduced here.

6.1 Byte Ordering – BIG ENDIAN and LITTLE ENDIAN

Most microprocessors have a memory architecture that is composed of 8-bit address locations known as bytes. Many data items (addresses, numbers, and strings) are too long to be stored using a single byte and must be stored in a series of consecutive bytes.

When using data that are stored in multiple bytes, byte ordering becomes an issue. Unfortunately, there is not just one standard for the order in which bytes in multi-byte data are stored. There are two popular methods of byte ordering currently in widespread use.

The first method is called "little endian" and is often referred to as the Intel order. In little endian, the least significant, or low-order byte is stored first. For example, a 16-bit integer value of 0x1234 (4660 decimal) would be stored using the little endian method in two consecutive bytes as follows:

Address	+0	+1
Contents	0x34	0x12

A 32-bit integer value of 0x57415244 (1463898692 decimal) would be stored using the little endian method as follows:

Address	+0	+1	+2	+3
Contents	0x44	0x52	0x41	0x57

A second method of accessing multi-byte data is called "big endian" and is often referred to as the Motorola order. In big endian, the most significant, or high-order byte is stored first, and the least significant, or low-order byte is stored last. For example, a 16-bit integer value of 0x1234 would be stored using the big endian method in two consecutive bytes as follows:

Address	+0	+1
Contents	0x12	0x34

Download free eBooks at bookboon.com

A 32-bit integer value of 0x004A4F4E would be stored using the big endian method as follows:

Address	+0	+1	+2	+3
Contents	0x00	0x4A	0x4F	0x4E

The 8051 is an 8-bit machine and has no instructions for directly manipulating data objects that are larger than 8 bits. Multi-byte data are stored according to the following rules.

- The 8051 LCALL instruction stores the address of the next instruction on the stack. The address is pushed onto the stack low-order byte first. The address is, therefore, stored in memory in little endian format.
- All other 16-bit and 32-bit values are stored, contrary to other Intel processors, in big endian format, with the high-order byte stored first. For example, the LJMP and LCALL instructions expect 16-bit addresses that are in big endian format.
- Floating-point numbers are stored according to the IEEE-754 format and are stored in big endian format with the high-order byte stored first. It should be noted here that the 8051 does not have any floating point handling instructions and normally this is handled by the compiler.

If the 8051 embedded application performs data communications with other microprocessors or devices, it may be necessary to know the byte ordering method used by the other CPU or peripheral.

Here is an example which may be used to test and understand the storage format of the micro-controller and of the KEIL compiler itself.

Download free eBooks at bookboon.com

```
/*-------------------------------------------------------
   Endian.c -- Big Endian and Little Endian explanation
-------------------------------------------------------*/
#include <stdio.h>
#include <string.h>

#include "reg52.h"
#include "UART0b.h"

typedef unsigned char         UCHAR;
typedef unsigned int   UINT;
typedef unsigned long  ULONG;

typedef union UTYPELONG {
   ULONG Long;
   UINT Int[2];
   UCHAR Char[4];
}UTYPELONG;

typedef union UTYPEINT {
   UINT Int;
   UCHAR Char[2];
}UTYPEINT;

UTYPELONG bdata Y;

sbit y0 = Y.Long^0;
sbit y1 = Y.Long^1;
sbit y2 = Y.Long^2;
sbit y3 = Y.Long^3;
sbit y4 = Y.Long^4;
sbit y5 = Y.Long^5;
sbit y6 = Y.Long^6;
sbit y7 = Y.Long^7;
sbit y24 = Y.Long^24;
sbit y25 = Y.Long^25;
sbit y26 = Y.Long^26;
sbit y27 = Y.Long^27;
sbit y28 = Y.Long^28;
sbit y29 = Y.Long^29;
sbit y30 = Y.Long^30;
sbit y31 = Y.Long^31;
```


Download free eBooks at bookboon.com

```
char Bit2Char ( bit x )
    {
        return (char) ( x == 1 ? '1' : '0' );
    }

/*-------------------------------------------------------
MAIN C function
-------------------------------------------------------*/
void main (void)
{
UART0_Init (57600);

printf("\n\n*** Notes on BIG ENDIAN and LITTLE ENDIAN ***\n\n");
printf("Compilers, processors and devices all have their own choice of storing numbers.\n\n");
printf("The KEIL compiler is BIG ENDIAN, that is HIGH BYTE FIRST.\n");
printf("However, since the 16-bit SFR16 registers are stored in the 8051 as\n");
printf("LITTLE ENDIAN, the KEIL compiler deals with SFR16 types as LITTLE ENDIAN.\n");
printf("For example, RCAP2L has an address CAH, and RCAP2H has an address CBH.\n");
printf("Thus, RCAP2 can be declared to be of type SFR16 at address CAH, (SFR16 RCAP2=0xCA;)\n");
printf("and if you want to load RCAP2L with 01H and RCAP2H with 23H,\n");
printf("you may use RCAP2=0x2301; and it will be loaded correctly, LOW BYTE FIRST.\n\n");

RCAP2 = 0x2301;
printf("This is shown below, showing the contents of RCAP2, RCAP2H and RCAP2L\n");
printf("RCAP2, is declared as an sfr16 at address 0xCA is loaded with %04XH\n", RCAP2);
printf("RCAP2L, at address 0xCA therefore contains %02BXH\n", RCAP2L);
printf("RCAP2H, at address 0xCB therefore contains %02BXH\n\n", RCAP2H);

printf("The architecture of the 8051 is also BIG ENDIAN (except LCALL stack pushes\n");
printf("and some SFRs!)\n");
printf("However certain other processors and devices (peripherals) can be\n");
printf("either BIG or LITTLE ENDIAN.\n\n");
printf("Care should therefore be taken when reading or writing data from/to such\n");
printf("devices, to ensure that the correct order is maintained.\n\n");
       Y.Long = 0x01234567;
printf("Y = %08LXH (type long)\n\r",Y.Long);
printf("Int(0) = %04XH and Int(1) = %04XH\n\r",Y.Int[0],Y.Int[1]);
printf("Byte(0) = %02BXH, Byte(1) = %02BXH, Byte(2) = %02BXH and Byte(3)= %02BXH\n\n",
       Y.Char[0],Y.Char[1],Y.Char[2],Y.Char[3]);
printf("This check explains how the bytes are stored by the COMPILER in arrays.\n");
printf("Namely, HIGH BYTE FIRST.\n\n");

printf("One has also to be careful with the way one addresses the bits.\n");
printf("Bit Y.0 would refer to the first bit of the variable Y as it is stored\n");
printf("by KEIL, which would be bit 0 of the MSB as shown below.\n\n");
```

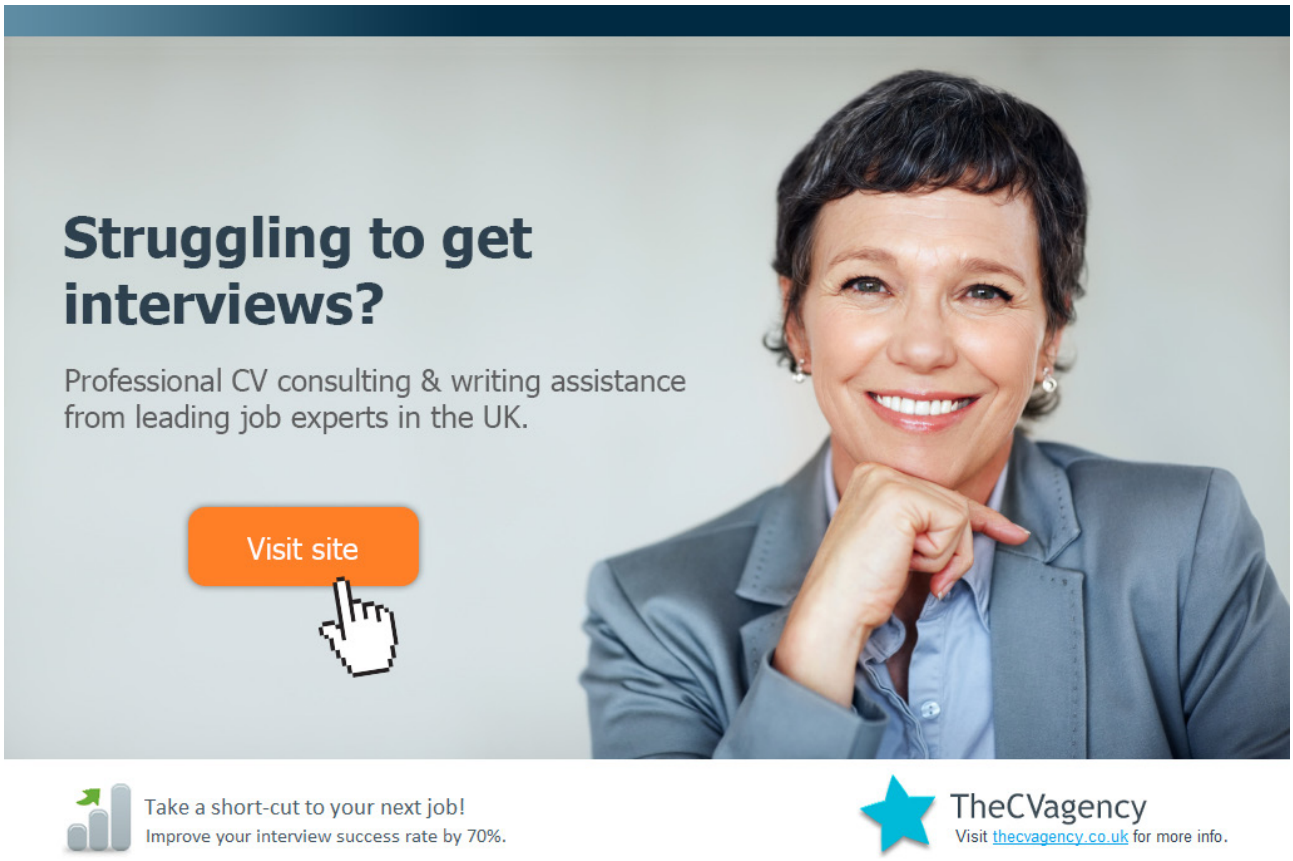

Download free eBooks at bookboon.com

```
printf("bit Y.0 = %c bit zero of the most significant byte\n", Bit2Char(y0));
printf("bit Y.1 = %c bit one of the most significant byte\n", Bit2Char(y1));
printf("bit Y.2 = %c bit two of the most significant byte\n", Bit2Char(y2));
printf("bit Y.3 = %c bit three of the most significant byte\n", Bit2Char(y3));
printf("bit Y.4 = %c bit four of the most significant byte\n", Bit2Char(y4));
printf("bit Y.5 = %c bit five of the most significant byte\n", Bit2Char(y5));
printf("bit Y.6 = %c bit six of the most significant byte\n", Bit2Char(y6));
printf("bit Y.7 = %c bit seven of the most significant byte\n\n\n", Bit2Char(y7));

printf("bit Y.24 = %c bit zero of the least significant byte\n", Bit2Char(y24));
printf("bit Y.25 = %c bit one of the least significant byte\n", Bit2Char(y25));
printf("bit Y.26 = %c bit two of the least significant byte\n", Bit2Char(y26));
printf("bit Y.27 = %c bit three of the least significant byte\n", Bit2Char(y27));
printf("bit Y.28 = %c bit four of the least significant byte\n", Bit2Char(y28));
printf("bit Y.29 = %c bit five of the least significant byte\n", Bit2Char(y29));
printf("bit Y.30 = %c bit six of the least significant byte\n", Bit2Char(y30));
printf("bit Y.31 = %c bit seven of the least significant byte\n", Bit2Char(y31));
while(1);
}

/*-------------------------------------------------------------
--------------------------------------------------------------*/
```

We now move to the actual KEIL µV2 IDE and describe briefly the basic setup required so as to be able to write the program and ultimately store the program on the device.



Download free eBooks at bookboon.com

The first screen shot (Figure 6-1) shows the layout under the KEIL µV2 environment.

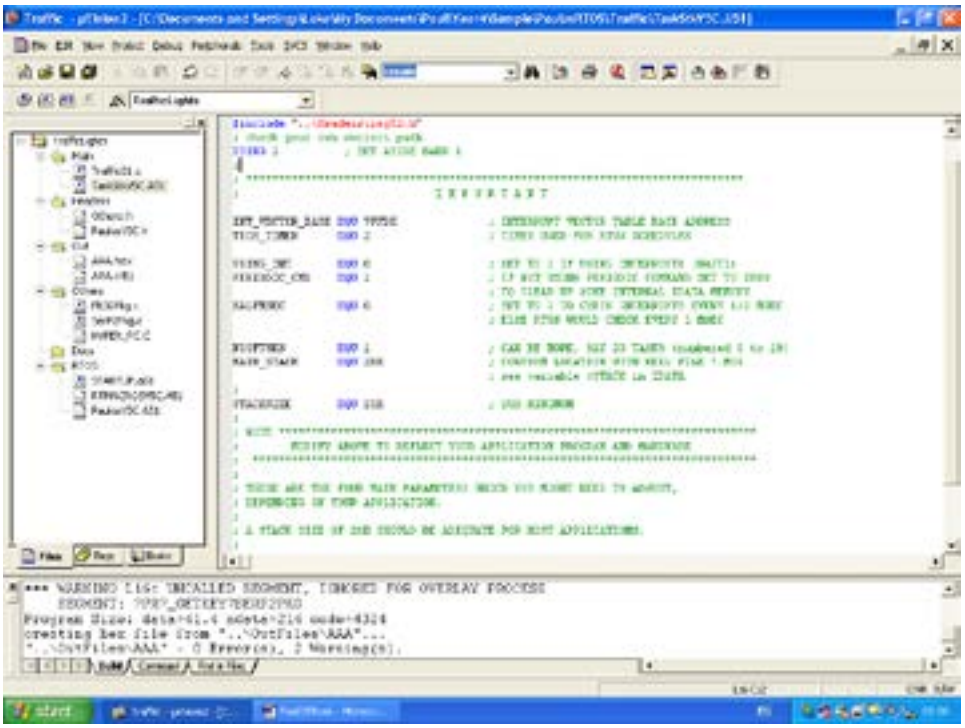


Figure 6-1 The KEIL µV2 environment

We would need first to create a project by selecting the New Project from the Project Tab (fourth item from the top left hand corner) and give it a name. It will then also ask us for the type of micro-controller (say Generic, 8051 or 8032) and whether we want to copy the Standard 8051 Startup code (automatically named Startup.a51) to the Project folder and Add it to the project. We normally answer yes to this prompt. Later on, by right clicking on the topmost folder in the Files list area (TrafficLights in our Figure 6-2), we could select a different micro-controller (device) for the Target. A 'Generic' type would do as a start as shown in the second screen shot (see Figure 6-2).

Download free eBooks at bookboon.com

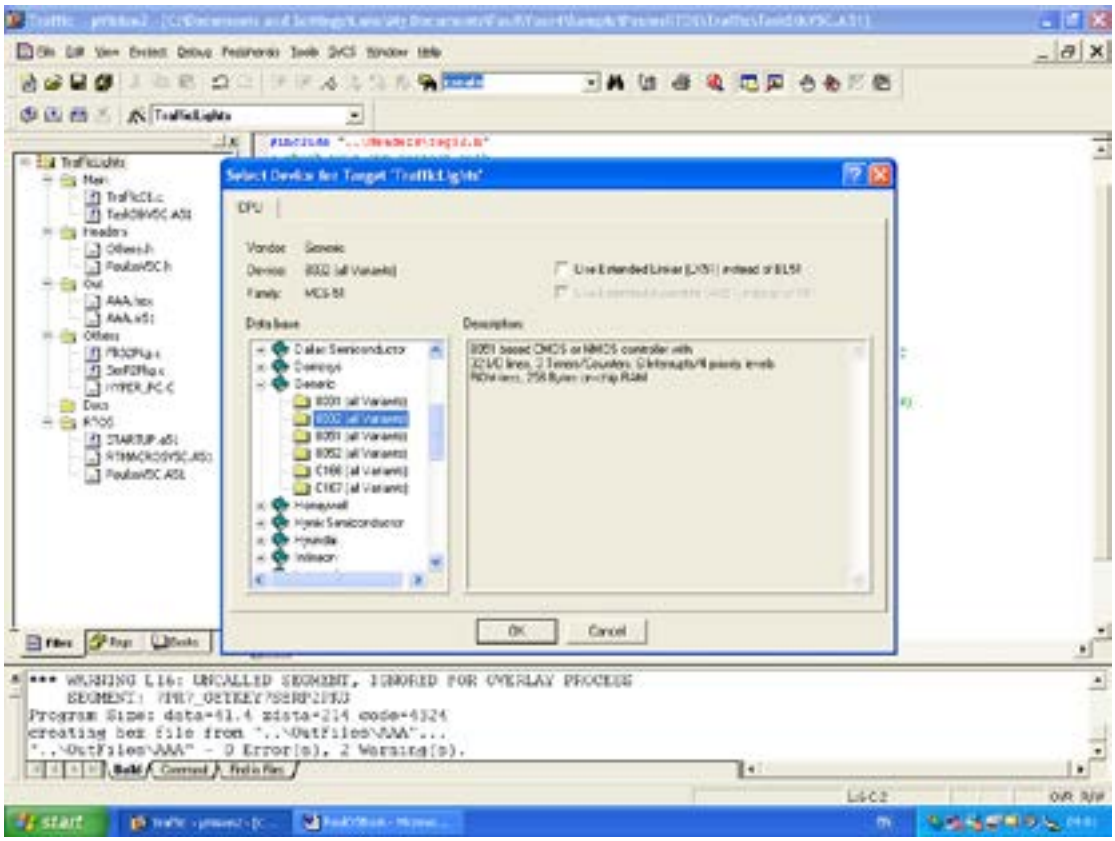


Figure 6-2 KEIL µV2 CPU type selection

We would then need to define the options for the particular target board, in particular the code and data area which are available. Screenshot 3 (Figure 6-3) shows the setup for the FLITE-32 board with 32KB EPROM and 32KB RAM. The clock speed is set to the crystal frequency on the board, 11.0592 MHz in this case. This crystal setting is not that critical but it has to be set to the correct value if we want to calculate the delays or duration of certain code using the debugging functions available in KEIL.

The settings for the off-chip Code area shown give a start address of 8100 hex and a size of 5000 hex, that is the code will reside from 8100H to D0FFH. This reflects the fact that the intended board is the FLITE-32 and thus the code (AAA.hex file) has to be dumped on to the external RAM of the board so that it can be tested while the FLITE-32 monitor program still resides on the EPROM from 0000H to 7FFFH. The first 100 hex bytes in RAM (8000H to 80FFH) are reserved for monitor use.

The settings for the off-chip XDATA area shown give a start address of D100 hex and a size of 2D00 hex, and the data area will therefore reside from D100H to FDFFH. The area from FE00H to FFFFH is reserved for other peripherals (8255 parallel i/o chip and SC2691 UART) and therefore must be left free.

Download free eBooks at bookboon.com

These off-chip areas must be set to reflect the actual hardware. If we intend to have the program run on a 32KB EPROM (replacing the monitor EPROM), then code area would start at 0000H with a size of 8000H. The XDATA area could then be set to start at 8000H with a size of FE00H.

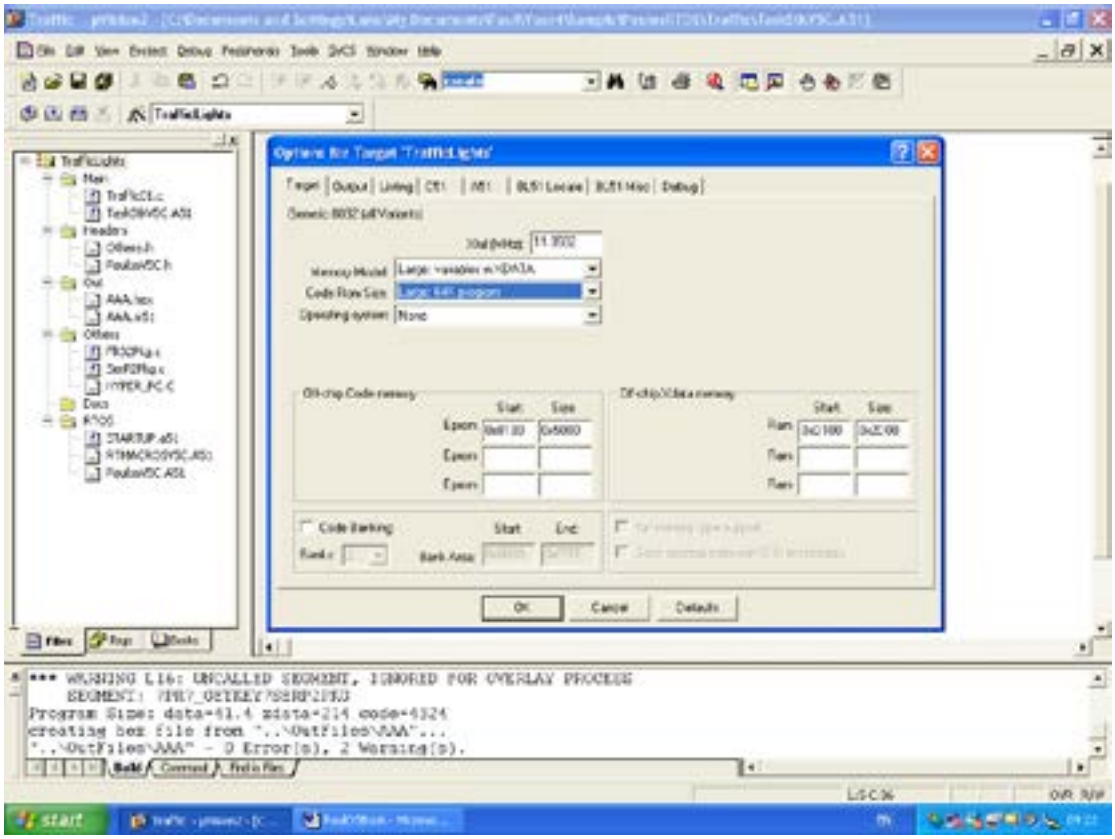


Figure 6-3 KEIL µV2 Target setup

The output and listing tabs should be selected to define the directory and file name for the output files generated by the compiler/linker, as shown in Figure 6-4 and in Figure 6-5.

Download free eBooks at bookboon.com

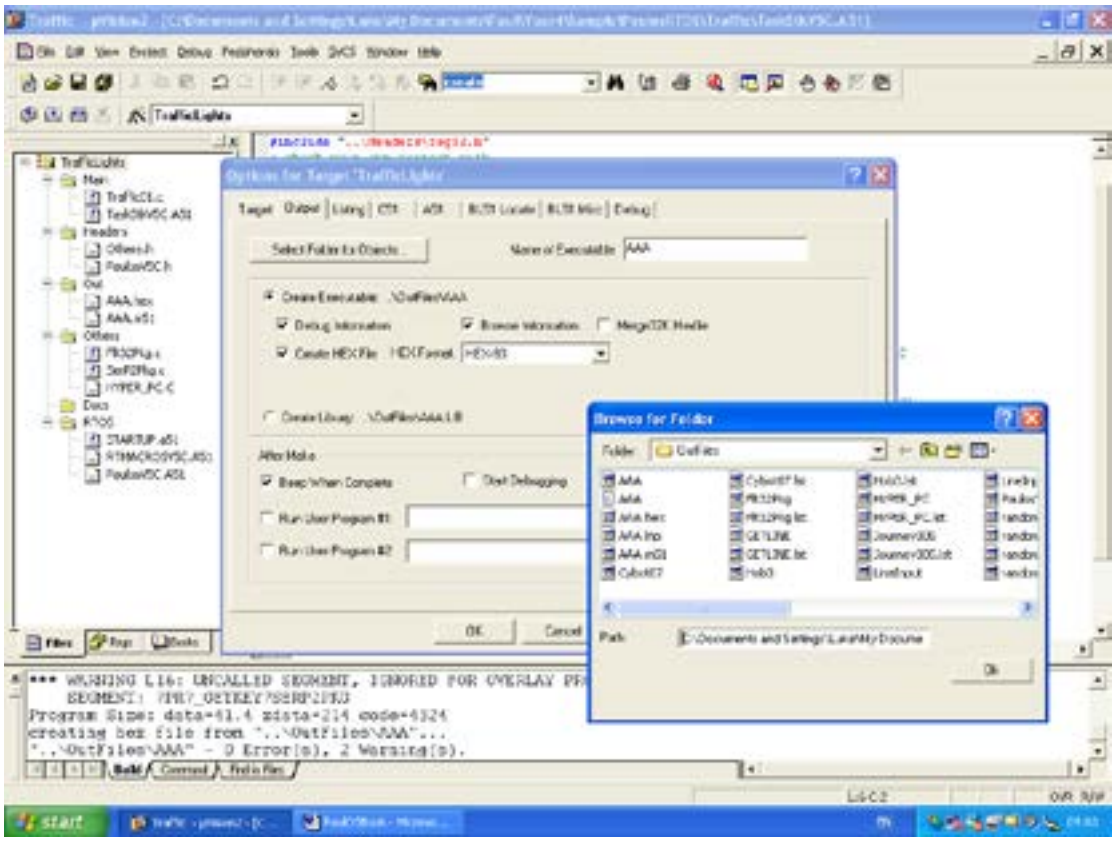


Figure 6-4 KEIL µV2 Target Output options

Click on the ad to read more

Download free eBooks at bookboon.com

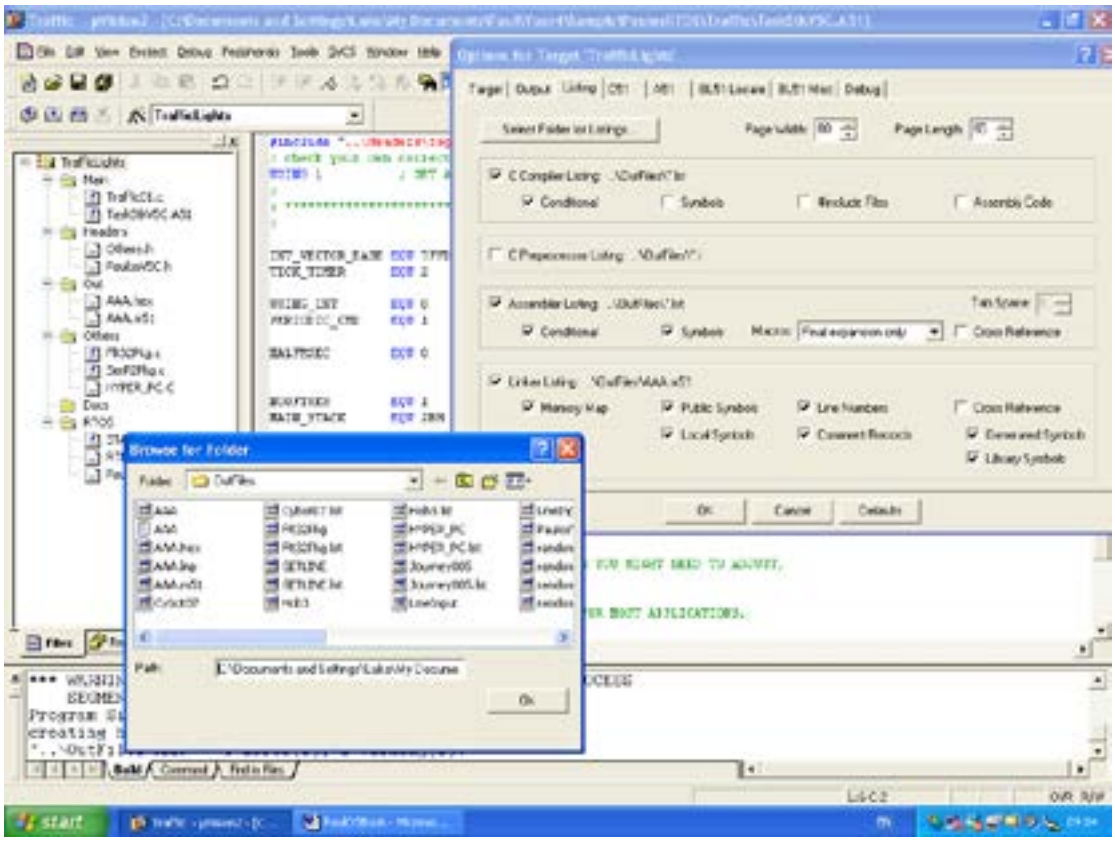


Figure 6-5 KEIL µV2 Target listing options

When using interrupts, and with the RTOS we would be using at least one timer interrupt, and it is therefore imperative to set correctly the Interrupt Vector Base address. The interrupt addresses, are shown once again in Table 6-1 below for the normal 8032.

Interrupt Name	Interrupt Number	Flag	Interrupt Hex Vector Address
Reset			0000
External 0	0	IE0	0003
Timer 0	1	TF0	000B
External 1	2	IE1	0013
Timer 1	3	TF1	001B
Serial	4	RI or TI	0023
Timer 2	5	TF2 or EXF2	002B

Table 6-1 8032 Interrupt Vector Table location

Download free eBooks at bookboon.com

If the code is in an EPROM, starting at address 0000H is the Reset vector and this is the location where the controller fetches its first instruction on startup. Usually the three bytes stored in this location contain an instruction to jump to the start of the program which is the main() function when writing in C. In KEIL, the start location can also be set manually by setting the ?START variable in the STARTUP.A51 file. At the other interrupt vector addresses, all spaced 8 bytes apart, we could have either ALL the code of the interrupt service routine (ISR), if it is short enough to fit in the space without overwriting areas reserved for other interrupts which are going to be used, or else it would have a jump instruction to the main interrupt service routine. Note that if we use say External 0 and Timer 2 interrupts only, then we could use the space form 0003H to 002AH for our external 0 ISR. There is a CODE keyword command which can be used to specify exactly where we want the code to be stored, but this is dealt with later on.

Setting the Interrupt vector address base to 0000H in the C51 tab would leave the addresses for the vectors set as shown in the table above, that is they would reside in EPROM code area. It should be noted once again that the first interrupt (External 0) is offset by 3 bytes from the base address of 0000H, and all the other interrupts are offset an additional 8 bytes from each other.

On a development board, since we would be loading the program in RAM and since it would not be possible to write on an EPROM (without an EPROM programmer), these interrupt vector addresses would in general be re-mapped by the monitor program on to the RAM area. That is, the monitor program residing in ROM, would have pre-programmed jumps at these 0003H to 002BH locations to re-route the program on to the RAM area. For example, our modified FLITE-32 monitor program version 3 has the following re-mappings:

Interrupt Name	Int. No:	Original Int. Hex Vector Address (ROM)	Instruction On monitor ROM	Modified Int. Hex Vector Address (RAM)
Base address ➔		0000		7FFD
External 0	0	0003	LJMP 8000H	8000
Timer 0	1	000B	LJMP 8008H	8008
External 1	2	0013	LJMP 8010H	8010
Timer 1	3	001B	LJMP 8018H	8018
Serial	4	0023	LJMP 8020H	8020
Timer 2	5	002B	LJMP 8028H	8028

Table 6-2 FLITE-32 Interrupt Vector Table location

The interrupts are still spaced 8 bytes apart, but the first interrupt is now 3 bytes away from the base of 7FFDH and not 3 bytes away from 0000H. Thus in order for the compiler to know exactly where the interrupt routines should be placed, all we need to do is to set the base address, (using the C51 tab in the target options) to 7FFDH, as shown in Figure 6-6.

Download free eBooks at bookboon.com

Other development boards may use a different base address, but the setup would still be done in a similar manner.

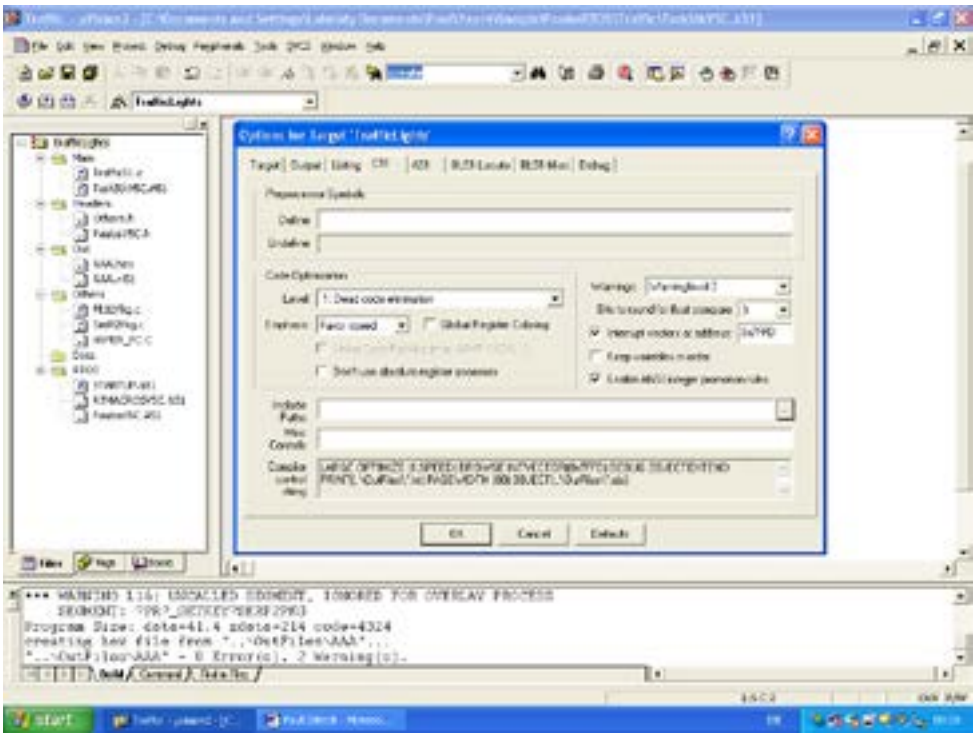


Figure 6-6 KEIL µV2 C51 options

Click on the ad to read more

Download free eBooks at bookboon.com

These are the main important setup parameters to be adjusted when creating a new project. With this setup completed, we can proceed to describe the various C related functions and methods which might have to be adapted for the 8051–KEIL environment.

6.2 Explicitly Declared Memory Types

We may specify where the variables are to be stored by including a memory type specifier in the variable declaration. The following table summarises the available memory specifiers:

Memory Type	Description
code	Program memory (max size 64KBytes code)
data	Directly addressable internal data memory, faster access to variables. (max size128 bytes)
idata	Indirectly addressable internal data memory; accessed across the full internal address space (max size 256 bytes) using MOVC @A+DPTR
bdata	Bit-addressable internal data memory; supports mixed bit and byte access. (max size 16 bytes)
xdata	External data memory (max size 64KBytes), accessed by MOVX @DPTR
far	Extended RAM and ROM memory spaces (up to 16Mbytes)
pdata	Paged (256 bytes) external data memory, accessed by MOVX @Rn

Table 6-3 Locations of Variables

When variables are declared at a specific location, the C compiler automatically uses the correct addressing mode (direct or indirect or external) when referring to these variables. If during the declaration no specific memory location is specified, the compiler will place the variable at the default location, as specified in the target option setting. Some examples on the use of these declarations are given below:

```
char data variable1;                    /* variable1 byte placed in internal data area */
unsigned long xdata array[100];         /* array placed in external data area */
unsigned char bdata sensor;             /* sensor byte placed in Bit-addressable area */
```

6.3 Data types

The following are the data types with the corresponding value ranges, available when using the C51 compiler:

Download free eBooks at bookboon.com

Data Types	Bits	Bytes	Value Ranges (decimal)
bit (*)	1		0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	8 or 16	1 or 2	-128 to +127 or –32768 to +32767
signed short	16	2	–32768 to +32767
unsigned short	16	2	0 to 65535
signed int	16	2	–32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2147483648 to +2147483647
unsigned long	32	4	0 to 4294967295
float	32	4	+- 1.175494E-38 to +-3.402823E+38
sbit (*)	1		0 to 1
sfr (*)	8	1	0 to 255
sfr16 (*)	16	2	0 to 65535

(*) the bit, sbit, sfr and sfr16 data types are not provided in ANSI C standard and are unique to the C51 compiler.

Table 6-4 C51 compiler data types

The bit data type in particular is well suited to make full use of the bit-addressable area capability of the 8051. A header file, reg51.h or reg52.h is already available in the INC directory of the Keil program, where all the 8032/8051 SFR registers are defined as well as the bit-addressable flags. Similar files (with the .INC suffix exist for A51 use, when using assembly language).

Note that an array of type bit is not allowed.

For example to start timer 2 we could simply use:

```
TR2 = 1;
```

in the C program, provided that the reg52.h header file is included. This is equivalent to the assembly language command:

```
SETB TR2
```

Another possibility of using the bit and sbit data type is for example when we are reading a data byte from some port and each bit of this data byte corresponds to some particular sensor or flag. Let us suppose that we have 8 sensors connected to some input port, say P1. To define each bit of this port (P1 is a bit-addressable SFR), we would use the sbit data type as shown:

Download free eBooks at bookboon.com

```
sbit fire       = P1 ^ 0;       /* bit 0 of port P1 */
sbit window     = P1 ^ 1;       /* bit 1 of port P1 */
sbit door       = P1 ^ 2;       /* bit 2 of port P1 */
…
…
```

The ^ (caret) is used instead of the . (dot) which is normally used in assembly. Even though the caret sign is normally used in C to denote an XOR operation, when used with the sbit keyword, it denotes the bit number of the variable, **which must be in a bit-addressable area**. Incidentally, the dot is used in C structure types.

The above declarations, one could check whether there is a fire simply by using:

```
if (fire)
   {
         ….
         ….
   }
```

Click on the ad to read more

Download free eBooks at bookboon.com

We might also need to have a variable that needs to be addressed either as one-byte or as individual bits. This can be achieved by declaring the variable to reside in the bit-addressable data area bdata.

```
char bdata display;                    /* byte variable display */
                                       /* residing in Bit-addressable area */
sbit red        = display ^ 0;         /* bit 0 of display byte */
sbit amber      = display ^ 1;         /* bit 1 of display byte */
sbit green      = display ^ 2;         /* bit 2 of display byte */
```

To clear all the byte (say switch off all leds or lights), simply use:

```
display = 0;
```

To set a particular bit on, we can use:

```
red = 1;
```

We may also have functions which return a bit value, for example:

```
bit validvalue (char idata value) {
…
…
}
```

6.4 Interrupt routines

Using interrupts in C is very straight forward. All we have to do is to declare a function as shown below:

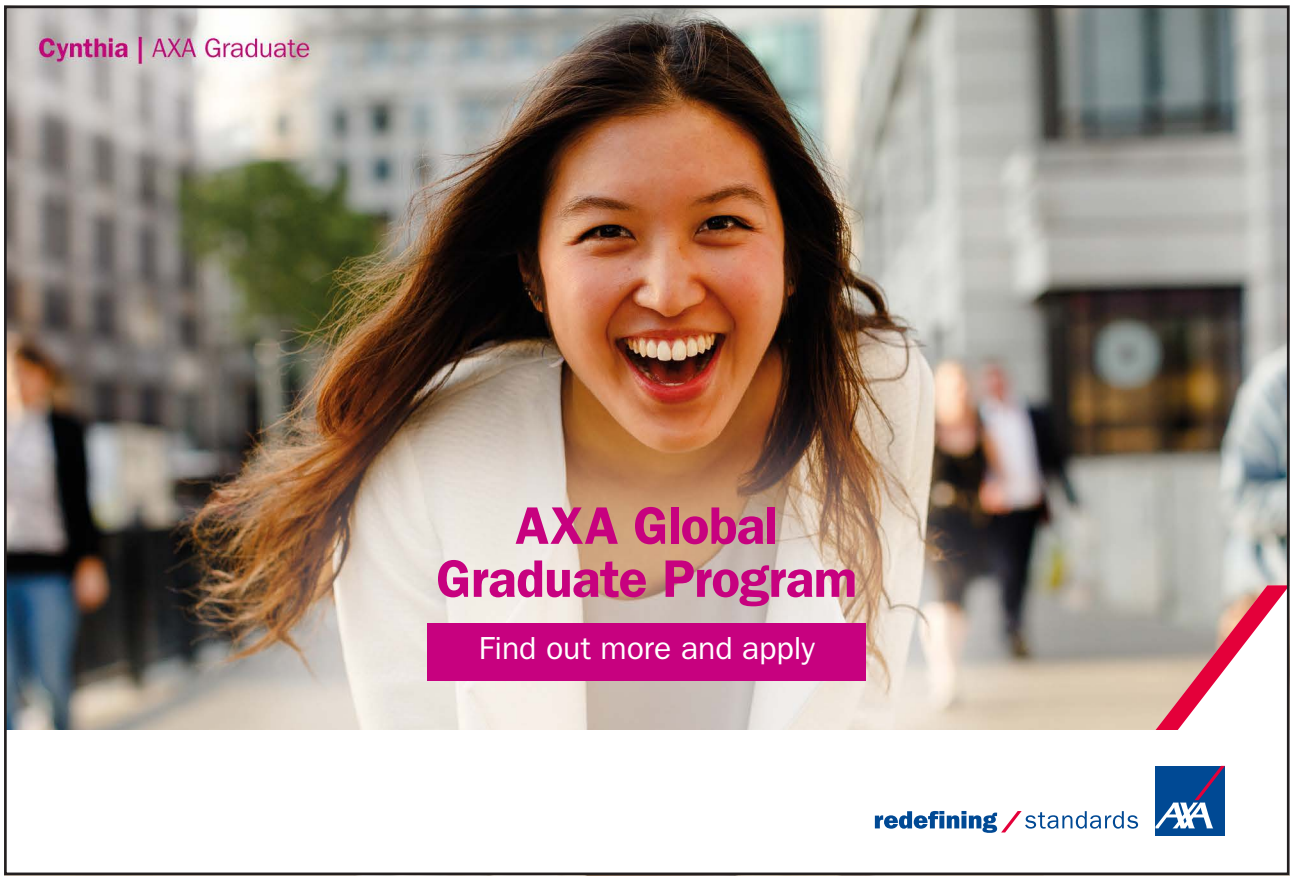
```
void serial (void) interrupt 4 using 2 { /* using register bank 2 for the ISR */

…
…
}
```


Download free eBooks at bookboon.com

Here the function named 'serial' is to run under interrupt number 4 (see Table 6-1) , which is the serial interrupt. The compiler automatically inserts at the correct vector table address (using the Interrupt Base address given in the target options setup, say 7FFDH as reference) a jump to this function. Hence, whenever a serial interrupt occurs, the program first jumps to the interrupt vector address location and there it executes another jump to this serial function. The pushes, pops and RETI instruction are handled automatically by the compiler.

The 'using 2' at the end of the declaration instructs the compiler to switch over to register bank 2 whenever this serial interrupt occurs. This saves the compiler from using additional pushes and pops to save the registers of the bank which was being used prior to the interrupt. Ideally and when possible, we should use a separate register bank for every interrupt function.



Click on the ad to read more

Download free eBooks at bookboon.com

7 Real-Time Operating System

We now come to the Real-Time Operating System (RTOS) and we start by giving the general principles behind the RTOS concept. An explanation of the three main variations of RTOSs which we will deal with is given, namely the round-robin, co-operative and pre-emptive versions of the RTOS. These categories are explained in section 7.2 I have developed operating systems of these three main versions of RTOSs and will be explained in detail in the following chapters. There are the ParrOS (assembly language version, Appendix A) and SanctOS (C language version, see Chapter 8 and Appendix C) which are both of the round-robin type. Then there is the PaulOS RTOS (see Chapter 9 and Appendix B and Appendix D) which is of the co-operative type and finally I have the MagnOS (see Chapter 10 and Appendix E) which falls in the pre-emptive category.

7.1 What is a Real-Time Operating System

This chapter introduces the concept of the RTOS. Such a system is not something out of this world, and once the concept is understood, one would be able to modify and expand the programs listed in this book, to suit his/her own requirements. It should be pointed out at the outset, that all the programs found in this book, are experimental. They all work, but I do not accept any responsibility if they are used in a system.

The RTOS is an operating system that guarantees a certain capability within a specified time constraint. For example, an operating system might be designed to ensure that a certain part or item is available at a certain point on an assembly line. In what is usually called a “hard” real-time operating system, if the program code for such an operation cannot be performed in time for making the part available at the designated time and place, the operating system would terminate with a failure. In a “soft” real-time operating system, the assembly line would continue to function but the production output might be lower as objects fail to appear at their designated time, causing the operator (or robot) to be temporarily unproductive. Some real-time operating systems are created for a special specific application and others are more of a general purpose type which can be adapted to various situations. It immediately becomes clear that real-time does not have an absolute value of time but the reaction time can vary depending on the application.

One might already be familiar with other operating systems used on personal computers, such as DOS, Windows, Unix or Linux. The RTOS to be discussed in this chapter is a much smaller version, written specifically for small micro-controllers.

The general idea is to write an operating system which would take overall control of the whole situation, particularly scheduling tasks (routines) at appropriate times according to the program logic or algorithm.

Download free eBooks at bookboon.com

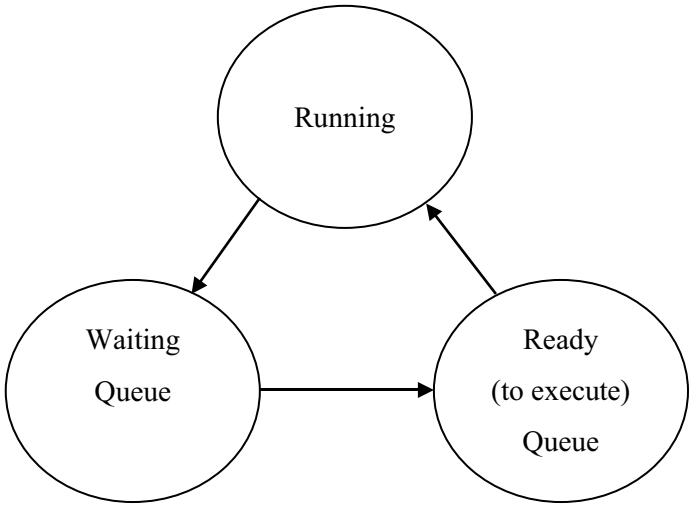


Figure 7-1 RTOS Task states diagram

The application program would be split up into "short" ENDLESS programs (routines, functions or procedures) known in the RTOS environment as TASKs. The RTOS can then be thought of as an organiser or scheduler of these individual tasks, controlling which task should be running and which task should run next. There would of course be only one task running at a particular time, but tasks would be switching in and out so fast that they would give the impression of running simultaneously or multi-tasking. The RTOS is simply time-slotting each task, in a time-multiplexing technique.

As show in Figure 7-1, a task can be in one of the following three states:

- RUNNING: Only one task would be in this state, since the micro-controller can only execute one program at any one time. As the name implies, this would be the task currently executing.
- WAITING: Tasks end up here after running for a preset time or because the task itself requested to wait. Tasks can be made to wait for:
  - Timeout: wait for a specified time
  - Signal: wait for a signal which would come from another task
  - Interrupt: wait for some external interrupt
  - Some other event: usually, tasks which have finished waiting are placed in the Ready queue.
- READY: This is a queue where all the tasks which are ready to execute are held whilst waiting their turn. Once the task currently running either has its time-slot expired or itself opts to wait, then the task next in line in this ready queue will take over and become the running task.

Download free eBooks at bookboon.com

It should be emphasised here that each task should be written as an endless loop or sub-program. Since tasks would be switching on and off, and each particular task can make use of a certain number of registers and it might also push on the stack some registers (or addresses in case of some call instructions), the biggest problem of the RTOS is how to handle these situations so that the tasks do not overwrite the registers used by other tasks, and they do not disturb the stack area.

Since the 8051 family of microprocessors have 4 register banks, allocating a different register bank to each task would solve the first problem. This would restrict the number of tasks to 4 and therefore is not that good unless you have only a very limited number of tasks.

The second problem could be solved by having a different stack area for each task, and loading the stack pointer (SP) accordingly before switching tasks. Now the following question arises: How can we switch tasks? If we remember what happens during an interrupt (or call instruction) we would be on the right track to answer the question. Whenever a CALL is executed, the address of the next instruction in the program is pushed on the stack. This address is retrieved by the RET (or RETI) instruction, using the locations pointed to by SP, so that the program continues where it left from. The RTOS operates on an interrupt basis, usually using a Timer interrupt at regular (for example 1ms) intervals. All that is required to be done in order to change tasks is therefore to have on the stack the address of the next task instead of the present one. This can be done by changing the actual address on the stack or else (and this is the method used in our RTOSs), point the SP to a different stack area, where the address of the new task is stored. With this method, the task swapping would be done and moreover, each task would have its own stack area for pushing registers, call routines etc.

7.2 Types of RTOSs

We could split the RTOSs into 3-types. There is the round-robin , co-operative and pre-emptive RTOS. These are described in more detail in later chapters but as a brief explanation the main differences and properties can be mentioned here.

7.2.1 Round-robin RTOS

The round-robin RTOS is a very simple operating system which allocates each task a specific time to operate. After this time elapses, the RTOS would stop this task, save its environment, replace the environment with that of the next task and then start the next task again for a specific time. In general, the time for each task would be the same, but not necessarily so.

It should be made clear at this point that once this allocated time slot is over, the processor shelves this task at whatever instruction it happens to be executing and starts (or continues) the next task in the queue. Each task could be a very small routine, written as an endless loop, which would be repeating on and on, and executing in bursts, a few instructions at a time (depending on the slot time) when its allocated time is used by the micro-controller.

Download free eBooks at bookboon.com

If a task needs to be stopped permanently at some time, there could be a

```
Here:   SJMP Here       (or while(1); if written in C)
```

so that the task would actually be running when its slot time comes around, but not doing anything useful. This is not so efficient since it would still be wasting valuable processor time. A more efficient way would be to kill the task completely by removing it from the queue.

More on this type of RTOS can be found in the assembly language version program ParrOS (see Appendix A) and in the SanctOS, written in C which is fully described in Chapter 8.

7.2.2 Co-operative RTOS

The co-operative RTOS (such as PaulOS in Appendix B and D and Chapter 9), is a further improvement on the round-robin RTOS. In this case, a task, which is again written as an endless loop would run until the task itself would issue an RTOS command which would cause a change of task. These commands would depend on the operating system itself and for the case of PaulOS, there are commands which would cause the task to pause and go into a waiting queue, thus giving up its processor time to another task. The task may for example wait for a specified time delay, or it may wait for an interrupt or for some signal from another task.

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

Click on the ad to read more

Download free eBooks at bookboon.com

Once this command is executed, the RTOS would initiate a change of task process similar to the round-robin case, and the next task in the queue which is ready to execute would take over. There could be instances where all the tasks would be waiting for a time delay or for an event to occur, in which case there would not be any tasks ready to execute. In this case, the main() code would run, which would normally be executing a useless *while(1);* loop (also setting the controller in an idle mode to save energy).

If all the tasks are independent and all have the RTOS instruction to wait for say 5ms, then the co-operative RTOS would be working exactly the same as the round-robin type, with each task coming into action in sequence.

More on this type of RTOS can be found when describing the PaulOS RTOS in Chapter 9.

7.2.3 Pre-Emptive RTOS

The pre-emptive RTOS such as MagnOS is still a further improvement on the previously mentioned RTOSs. Here each task is given a priority number and the task with the highest priority is given the go-ahead to execute by the RTOS. Unless the task itself executes a 'wait' instruction to give up its time, then it will continue to run since a task of a lower priority would not be permitted to run and thus interrupt the higher priority one. On the other hand, if a low-priority task is currently running and a higher priority task moves to the ready (to execute) queue (for example because it was waiting for some time delay which has now passed), then the RTOS would stop the lower priority task and place it in the ready queue and the higher priority task would then take over and start/continue to execute. More information on the MagnOS RTOS is given in Chapter 10.

In this environment it is very important to allocate the right priorities to the tasks so as to be sure that all tasks are given a chance to run. Various theories or ideas exist about priority allocation techniques, the Rate Monotonic Scheduling or Algorithm ([19] C.L. Liu and J.W. Layland) being one of the most popular.

Download free eBooks at bookboon.com

8 SanctOS – a Round-Robin RTOS

This chapter explains the very simple round-robin RTOS called SanctOS, where each task (or function) works for a specified amount of time before passing on the processor time to the next task.

This RTOS is a direct adaptation of the home-brew round-robin ParrOS (Paul Round-Robin Operating System) RTOS assembly language program described in great detail in the appendix. This is the improved version written in the C language so as to make it more versatile and more easily portable to other micro-controllers. The name SanctOS is an acronym for Small ANd CompacT Operating System, and before proceeding further, it would be very advantageous if the ParrOS program is understood by reading the appropriate appendix.

Most of the commands are exactly the same as those for the ParrOS RTOS, (with the additional OS_ prefix), and the settings regarding the number of tasks, tick time and stack size can be set in the PARAMETERS.H file shown below.

There are some very immediate advantages in using C to write the RTOS. Parameters can be easily changed from char to integer or long types and the routines would automatically reflect the changes when they are compiled. An example here would be the OS_CREATE_TASK(parameter list) command where in the A51 version, the slot time parameter was of type integer (0-65535). If we had to change the parameter to long in order to be able to accommodate longer wait periods, we would have had to re-write the routines so as to increment or decrement double words (32-bit) rather than words (16-bit). In the C version this could be done fairly easily simply by changing the type declarations.

Naturally there are some memory space and speed penalties to pay for this versatility. However the improvements more than outweigh the penalties, especially as far as student understanding of the RTOS is concerned. Here we now list the RTOS commands, this time for the C version. The full SanctOS RTOS source program listing can be found in Appendix C.

8.1 SanctOS System Commands

The following are the only RTOS system calls available :

- OS_INIT_RTOS (uchar iemask); // Initialises all RTOS variables
- OS_RTOS_GO (void); // Starts the RTOS
- OS_CREATE_TASK (uchar task_num, uint task_add, uint slot_time);
  // Creates a task, allocating a slot_time during which it can execute

These commands are more fully explained in section 8.2 and its sub-sections.

Download free eBooks at bookboon.com

8.2 Variations from the A51 version

The C version of the RTOS provides some variations and additional commands from ParrOS, which can be implemented easily, after using the program for a while.

8.2.1 OS_INIT_RTOS (uchar iemask)

This system command is used only once in the main program and its function is to initialise all the RTOS variables. It sets up the RTOS timer (the so called tick timer, which generates the regular the critical interrupt which calls the Interrupt Service Routine that handles the slot time counter and task swapping) and enables the required interrupts according to the iemask parameter given within the command.

An example of the syntax used for this command is:

```
OS_INIT_RTOS(0x01);
```

This would initiatilise the RTOS, enabling the external interrupt 0 since the iemask contents correspond to the interrupts shown in Table 8-1. The timer to be used for the RTOS tick timer (say Timer 0) and its corresponding interrupt would be enabled automatically (irrespective of the iemask setting), depending on the TICK_TIMER value declared in the SanctOS_Param.h header file. It should be noted here that this tick timer interrupt is therefore used by the RTOS and cannot be used by the user program.

Click on the ad to read more

Download free eBooks at bookboon.com

Interrupt		IE MASK		Notes
No:	**Name**	**Binary**	**Hex**	
0	External Int 0	00000001	01	
1	Timer Counter 0	00000010	02	Default RTOS timer for 8051
2	External Int 1	00000100	04	
3	Timer Counter 1	00001000	08	
4	Serial Port	00010000	10	
5	Timer 2 (8032 only)	00100000	20	Default RTOS timer for 8032

Table 8-1 IEMASK Parameter (SanctOS)

8.2.2 OS_RTOS_GO(void)

This system command is also used only once in the main program, when the RTOS would be required to start supervising and scheduling the individual processes or tasks. It does not take any parameters.

An example of the syntax used for this command is:

```
OS_RTOS_GO();
```

This would start the RTOS ticking, at a reference time signal of TICKTIME milliseconds as set in the parameters header file SanctOS_Param.h. This TICKTIME in milliseconds would then become the basic reference unit for other system commands which use any timeout parameter, such as the OS_CREATE() function.

8.2.3 OS_CREATE_TASK (uchar task_num, uint task_add, uint slot_time)

This system command creates the tasks by setting the appropriate variables corresponding to the task number (usually starting from 0), the function name (which actually corresponds to the address location where the task function actually starts in the program code area and the slot-time required for this task.

An example of the syntax used for this command gives some further explanation of its function and purpose.

```
OS_CREATE_TASK(0,Task_Zero_Routine,25);
```


Download free eBooks at bookboon.com

This would create a task which refers to the function or sub-routine Task_Zero_Routine, having a task number 0 to be handled by the RTOS. The slot-time given for this task is 25 ticktimes. The value of TICKTIME milliseconds, declared in the SanctOS_Param.h would be used as the basic reference unit for this slot-time. Thus if TICKTIME was declared as 1 (meaning one millisecond), then the above task would run for 25ms each time it is given the go-ahead to run, that is this task would run in bursts of 25ms duration, stopping after 25ms when the next task would run for its own specified slot time and so on until all the tasks would have run and the turn for Task_Zero_Routine comes up again.

8.2.4 Other add-on MACROS

These macros (#define statements) add some more basic commands and flexibility.

```
OS_PAUSE_RTOS()          // Disable the RTOS
OS_RESUME_RTOS()         // Re-enable the RTOS

OS_CPU_IDLE()            // Sets the microprocessor in idle mode
                         // This is usually used in the main program endless loop after
                         // initialising and starting the RTOS.
OS_CPU_DOWN()            // Sets the microprocessor in power-down mode
```

These #define statements are simply substitutions for some instructions which might seem meaningless if they are just written in the normal way. For example, using these 2 definitions

```
#define OS_CPU_IDLE()          PCON |= 0x01 // Sets the microprocessor in idle mode

#define OS_CPU_DOWN()          PCON |= 0x02 // Set microprocessor in power-down mode
```

It would make the program much easier to understand if we use

```
OS_CPU_IDLE();
```

rather than just writing

```
PCON |= 0x01;
```

This SanctOS operating system is very simple to use and is ideal for situations where we have totally independent tasks. That is we have various jobs to do which do not rely on any input or event from some other task. If our particular requirement stipulates that we need to do all jobs together rather than sequentially, than this RTOS can be our solution. The tasks would all appear to be running simultaneously although in fact they would be alternating and using the processor time for a few milliseconds each.

Download free eBooks at bookboon.com

8.3 SanctOS example program

A simple example would help to explain how the OS works. The programs or modules required, apart from the main example program are:

SanctOS_Startup.a51, SanctOS_A01.A51, SanctOS_V01.c using the header files SanctOS_V01.h and SanctOS_param.h

This example program creates 255 tasks, which happen to be practically all the 'same' task just to make it simpler to program. Each task simply outputs the task number to port P1. Thus the first task would output a zero and the last task would output 254 to this port. The number 255 refers to the main() function. The variable *Running* is actually used in the SanctOS RTOS to refer to the task number of the currently executing task and is declared as an external variable so that it can be used in the example or application program too.

Since the tasks, when created, are allocated a slot time of 50 and the parameter TICKTIME is given a value of 1, then when the RTOS program is executing, it would first start task 0, thus P1 would be 0 for 50 ms.

Then task 1 would be invoked, and P1 would be 1 for another 50ms, then task 2 and so. Thus effectively, P1 would be counting and showing 0 to 254 in 50ms steps!! So after Task 0 executes, it would have to wait for the other tasks and the main() program(255 other routines) to each execute in turn for 50ms before it can continue executing again. Because of the large number of tasks in this particular example, this delay which works out to 12.75s might not be acceptable for the particular application. If on the other hand we have fewer tasks (say 10) and you allocate a slot time of say 2ms per tasks, then the waiting period for each task between successive executions would only be 20ms which might be more acceptable. This can easily be checked in the example program, simply by changing the NOOFTASKS parameter in the SanctOS_Param.h file and the slot-time in the OS_CREATE_TASK() command in the main program.

One can compare this round-robin RTOS with the Chinese juggler spinning plates on those long sticks in some circus. Each stick (task) is 'touched' in turn, making sure that each plate (task) is visited before it is too late. Depending on our application, we can determine the slot-time which we require for each task, remembering that only one task would actually be executing at any particular moment.

It should also be mentioned at this point, that most applications can be written without using any RTOS, making use instead of the various interrupt service routines. However, the use of an RTOS can most of the time help us to write a more user-friendly programme which is neater and simpler to maintain. Some time is needed to get familiar with the RTOS commands and the way it is initialised, but once this is mastered, it should be relatively simple to implement our project using the RTOS.

Download free eBooks at bookboon.com

Example01.c

```
/*************************************************************************/
/*                                                                       */
/*      Example01.c: Demo                                                */
/*              SanctOS demo                                             */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

#include <reg52.h>     /* special function registers 8052                */
#include <absacc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "SanctOS_V01.h"      /* SanctOS RTOS system calls definitions    */

/*************************************************************************/
/*      Task X:                                                          */
/*************************************************************************/
        // This variable 'Running is declared in SanctOS_V01.h as
        // 'extern data unsigned char Running; '
        // and contains the number of the currently running task
        // which can therefore be used in the main program if required
void OutPort(void){    /* Output Task Number on to Port P1               */
   while(1)
        {
                P1 = Running;
        }
   }

/*************************************************************************/

/*************************************************************************/
/* Main: Initialise and CREATE tasks */
/*************************************************************************/

void main (void)       {       /* program execution starts here         */
unsigned char i;
        OS_INIT_RTOS(0x20);    /* initialise RTOS variables and stack    */
                               /* using Timer 2 interrupts               */
                               /* OS_INIT_RTOS(0x00); would also be correct */
                               /* since the tick-timer interrupt         */
                               /* will be set automatically              */

        for(i=0;i<NOOFTASKS;i++) OS_CREATE_TASK(i, OutPort, 50);

        P1 = 0;
        OS_RTOS_GO();          /* start SanctOS RTOS                     */

        while (1) OS_CPU_IDLE();
}

/*************************************************************************/
```


Download free eBooks at bookboon.com

```
SanctOS_Param.H
/*
*********************************************************************************
*                      SanctOS_Param.H --- RTOS KERNEL HEADER FILE
*
* For use with SanctOS_V01.C - Round-robin RTOS written in C by Ing. Paul P. Debono
*                      for use with the 8051 family of microcontrollers
*
* File          : Parameters_V01.H
* Revision      : 8
* Date          : February 2006
* By            : Paul P. Debono
*
*                       B. Eng. (Hons.) Elec. Course
*                       University Of Malta
*
*********************************************************************************
*/
#ifndef __SANCTOS_PARAM_H__
#define __SANCTOS_PARAM_H__

/*
*********************************************************************************
* RTOS USER DEFINITIONS
*********************************************************************************
*/
#define STACKSIZE       0x10    // size of stack for each task - no need to change
#define CPU             8032    // set to 8051 or 8032
#define TICK_TIMER      2       // Set to 0, 1 or 2 to select which timer to use as the RTOS tick
timer
#define TICKTIME        1       // Length of RTOS basic tick in ms
#define NOOFTASKS       255     // Number of tasks used in the application

/*
*********************************************************************************
*/

#endif
```

Example 2

The second example shows a 3-task application, each task having a 5ms time-slot. Each task toggles a different pin on port P1 every 1s, 1.5s and 3s respectively and these timings are worked out by a Timer 0 interrupt service routine, running independently from the RTOS interrupt. Timer 0 is set up to overflow every 50ms and a simple counter can be used to determine the number of overflows so as to get the correct pin-toggling timings. The RTOS this time uses Timer 2 as the tick-timer source, which is used to give each task a 5ms time-slot in which to run.

Download free eBooks at bookboon.com

When executing, this program really gives the impression that all the three tasks are running simultaneously. The timing of the LEDs is a bit approximate here since the task might not be actually running when the LED toggle time expires. In the worst case scenario, the task might have to wait up to 10 milliseconds for the other two tasks to use their slot time before it would notice (when its time-slot comes up) that the toggling time has passed.

Click on the ad to read more

Download free eBooks at bookboon.com

Example02.c

```
/*********************************************************************************/
/*                                                                               */
/* Example02.c: Demo                                                             */
/* SanctOS demo                                                                  */
/*                                                                               */
/*                                                                               */
/*********************************************************************************/

#include <reg52.h> /* special function registers 8052                           */
#include <absacc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "SanctOS_V01.h" /* SanctOS RTOS system calls definitions                */

sbit Led0 = P1^0;
sbit Led1 = P1^1;
sbit Led2 = P1^2;

bit Sec, OneSecFive, ThreeSecFive;

/* set up timer 0 as a 16-bit timer                                              */
/* Overflows every 50 milliseconds with 11.0592 MHz clock                        */
/* Timer needs to count 46080 before it overflows                                */
void SetUp_Timer0 (void)
{
    TMOD &= 0xF0;              // clear timer 0 control bits only
    TMOD |= 0x01;              // 16-bit
    TH0 = (65536-46080)/256;
    TL0 = (65536-46080)%256;   // Overflows every 50 milli seconds
    TR0 = 1;                   // Timer ON
    ET0 = 1;                   // Enable TF0 interrupt
}

void TF0_ISR (void) interrupt 1 using 2
{
    static unsigned int data overflow_count;
    TH0 = (65536-46080)/256;
    TL0 = (65536-46080)%256;   // Overflows every 50 milli seconds
    overflow_count = (overflow_count + 1) % 420;              // 420 is the LCDM of 20,30 and 70
    if (overflow_count%20UL == 0) Sec = 1;                    // 20 overflows = 1s
    if (overflow_count%30UL == 0) OneSecFive = 1;             // 30 overflows = 1.5s
    if (overflow_count%70UL == 0) ThreeSecFive = 1;           // 70 overflows = 3.5s
}

/*********************************************************************************/
/* Task 0: */
/*********************************************************************************/
```


Download free eBooks at bookboon.com

```
void ToggleLed0(void){        /* Toggle LED 0                                  */
               while(1)
                       {
                       Led0 = ~Led0;
                       while(Sec == 0);
                       Sec = 0;
               }
 }

/*****************************************************************************/

/*****************************************************************************/
/*      Task 1:                                                              */
/*****************************************************************************/

void ToggleLed1(void){        /* Toggle LED 1                                  */
               while(1)
                       {
                       Led1 = ~Led1;
                       while(OneSecFive == 0);
                       OneSecFive = 0;
               }
 }

/*****************************************************************************/

/*****************************************************************************/
/* Task 2: */
/*****************************************************************************/

void ToggleLed2(void){        /* Toggle LED 2                                  */
               while(1)
                       {
                       Led2 = ~Led2;
                       while(ThreeSecFive == 0);
                       ThreeSecFive = 0;
               }
 }

/*****************************************************************************/

/*****************************************************************************/
/* Main: Initialise and CREATE tasks */
/*****************************************************************************/

void main (void)       {      /* program execution starts here                 */
       OS INIT_RTOS(0x22);    /* initialise RTOS variables and stack           */
                              /* using Timer 0 & Timer 2 interrupts */
       OS CREATE_TASK(0, ToggleLed0, 5);
       OS CREATE_TASK(1, ToggleLed1, 5);
       OS CREATE_TASK(2, ToggleLed2, 5);
       P1 = 0;
       SetUp_Timer0(); /* Timer 0 interrupts are once again enabled here !!   */
       OS_RTOS_GO();          /* start SanctOS RTOS                            */
       while (1) OS_CPU_IDLE();
}

/*****************************************************************************/
```


Download free eBooks at bookboon.com

9 PaulOS – a Co-operative RTOS

The PaulOS (PAUL's Operating System) co-operative RTOS is described here. This is the 'flagship' RTOS which we regularly use during the year with our students. It is heavily used also for their final year theses and it has been regularly refined to reflect the changes and upgrading requested by the students as they became more and more familiar with the performance and limitations of this co-operative RTOS. In this RTOS, each task is free to run for as long as it wishes. The task itself can control when to give up the processor time to allow other tasks to run.

The original idea for this RTOS came from the book "C and the 8051 – Building Efficient Applications – Volume II" by Thomas W. Schultz.[1] This RTOS is a direct adaptation of my PaulOS assembly language program, re-written in C so as to make it more versatile and more easily portable to other micro-controllers. In fact it was even successfully ported to the Intel 8086 microprocessor and an 8086 version with an example is also given in the Appendix. The main task of translating it from assembly to C was undertaken years ago as a final year engineering degree thesis [20] (Blaut 2004), then a student under my supervision. It was further developed and improved throughout the years by myself, thanks also to input and suggestions from other students taking my study-units during their degree program, into the version shown here. I consider this RTOS as providing a good basis to the study of a real-time operating system for the 8051.

This e-book
is made with
SetaPDF
SETASIGN
PDF components for PHP developers
www.setasign.com
Click on the ad to read more

Download free eBooks at bookboon.com

Most of the commands are exactly the same, (with the additional OS_ prefix) as explained in PaulOS.a51 RTOS assembly language version, also found in the appendix. The settings regarding the number of tasks and stack size and location can be set in the parameters file, which is also listed at the end of this chapter.

There are some very immediate advantages in using C to write the RTOS. Parameters can be easily changed from char to integer or long types and the routines would automatically reflect the changes when they are compiled. An example here would be the 'wait for timeout' OS_WAITT(parameter) command where in the A51 version, the parameter was of type integer (0-65535). In the A51 version, if we had to change the parameter to long in order to be able to accommodate longer wait periods, we would have to re-write the routines so as to increment or decrement double words (32-bit) rather than words (16-bit). In the C version this could be done fairly easily simply by changing the type declarations. The compiler would do the rest.

Naturally there are some memory space and speed penalties to pay for this versatility. However the improvements are more than worth the penalties, especially as far as student understanding of the RTOS is concerned. In the next paragraph we now list once again the RTOS commands, including the improvements, mainly achieved with the use of MACROS which are listed in section 9.3.14. The full source program can be found in appendix D.

9.1 Description of the RTOS Operation

The PaulOS RTOS is a co-operative RTOS and hence, as explained earlier in the RTOS chapter, each task has to take the initiative to give up its own time so as to allow other tasks to run. It has to be kept in mind that this OS is running on an 8051-based micro-controller which can only run one program at a time and hence this task swapping RTOS only gives the impression of having tasks running simultaneously. In actual fact we can only have one task actually running, and at the time that the RTOS is doing its own checks, no tasks at all would be running. This time ideally should be kept as short as possible.

The operation of the RTOS is as follows:

Each task, when created, would have its own memory area in external memory where there would be stored all the registers (R's, A, B, DPTR, PSW), stack area (including the return address of the task or function). Once a change of task is required, the RTOS would take care to swap the relevant registers and stack areas so that the micro-controller would have the correct data for the new task in its own internal RAM.

Download free eBooks at bookboon.com

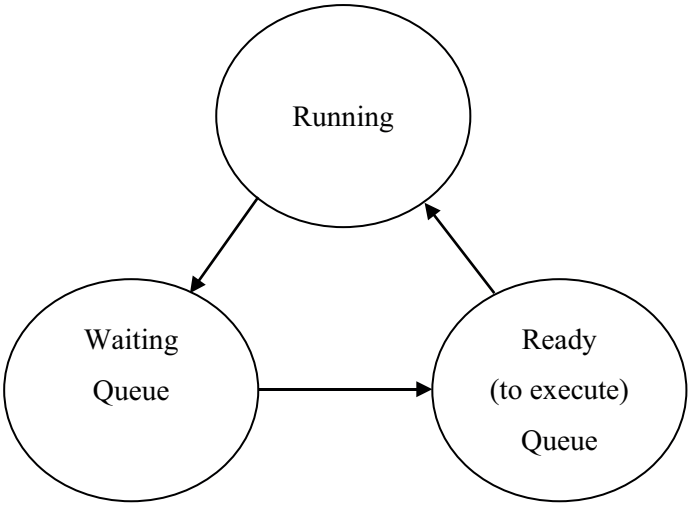


Figure 9-1 RTOS Task states diagram

The RTOS tick-timer can be chosen by the user who can select from the different timers available on the controller. Once set, at every timer overflow, an interrupt call is made to the main RTOS tick timer interrupt service routine. This is the most important routine in the program since at every interrupt the RTOS has to check the status of all the tasks so as to be able to decide whether a task can be moved from the Waiting queue to the Ready queue (see Figure 9-1) or a task swap if the main() was running is required. The RTOS achieves this by counting down the parameter variables holding the individual waiting time required for those tasks in the waiting queue. When anyone of these timeout parameters reaches 0, it means that the time to move on has arrived. Once again, being a co-operative RTOS, the scheduler cannot swap tasks on its own accord. Only the main() code can be forced to give up its time, so that if at any time whilst the main() code is running, there is a task which moves into the Ready queue, then that task takes over.

On the other hand, when one of the OS commands which forces a task change is encountered in a task, then it is only at that instance that a task swap is implemented. The currently running task is marked as being in the Waiting queue and the first task in the Ready queue takes over, with the stack and registers being conveniently swapped.

The idea behind the PaulOS RTOS is that any task (a function or a routine in a program) can be in any ONE of three states, Running, Waiting (for some event or time delay) or Ready (to execute) state.

RUNNING

A task can be RUNNING, (obviously in the single 8051 environment, there can only be one task which is actually in the running state). If there are no tasks which are ready to execute, then the RTOS will set the main() as the running task. This will be interrupted at any time, as soon as a task becomes ready to run.

Download free eBooks at bookboon.com

WAITING

A task can be in the WAITING (sometimes also referred to as SLEEPING) queue. Here a task could be waiting for any one of the following time delays or events to occur:

- a specified amount of time delay, selected by the user with OS_WAITT command. OS_DEFER command is actually an OS_WAITT(2) – wait for 2 ticks.
- a specified amount of time delay, selected by the user with OS_PERIODIC command. The actual task is placed in the waiting queue when the OS_WAITP (wait for periodic interval) is encountered.
- a specified interrupt to occur within a specified time, selected by the user with the OS_WAITI command.
- a signal from some other task within a specified timeout, selected by the user with the OS_WAITS(timeout) command.
- a signal from some other task indefinitely, selected by the user with the OS_WAITS(0) command.
- a never-ending waiting period. A task could be waiting here indefinitely, effectively behaving as if the task did not exist. This is specified by the OS_KILL_IT command.

Click on the ad to read more

Download free eBooks at bookboon.com

READY

It can also be in the READY QUEUE, waiting for its turn to execute. This can be visualised in Figure 9-1 which shows how the tasks can move from one state to another. The RTOS, when permitted to do so, will select the top task from this queue to execute instead of the currently running task, which would then be placed in the waiting queue.

The RTOS itself always resides in the background, and comes into play:

- At every RTOS TIMER interrupt (usually when Timer 2 or Timer 0 overflows, say every one millisecond) so as to update the waiting time left for any tasks.
- At any other interrupt from other timers or external inputs so as to check whether it needs to move to the ready queue any tasks which were waiting for such events or interrupts.
- Whenever an RTOS system command is issued by the main program or tasks, to perform that system command.

The RTOS which is effectively supervising and scheduling all the other tasks, then has to make a decision whether it has to pause the current task and resume a new one or whether it can let the current task run on. There could be various reasons for changing tasks, as explained further on, but in order to do this task swap smoothly, the RTOS has to save all the environment of the presently running task and substitute it with the environment of the next task which is about to run. This is accomplished by saving all the BANK 0 registers, the ACC, B, PSW, and DPTR registers. The STACK too has to be saved since the task might have pushed some data on the stack (apart from the address at the point that the task was interrupted, where it has to return to after the interrupt). This is the crux of the PaulOS RTOS.

9.2 PaulOS.C System Commands

We now list and explain all the PaulOS RTOS system commands. These are first listed or grouped according to whether or not they take any parameters. The list is then repeated, this tme sorted according to whether the command causes a task swap or not.

The following RTOS system calls do not receive any parameters :

- OS_DEFER (void); // Stops current task and passes control to next task in queue
- OS_KILL_IT (void); // Kills a task - sets it waiting forever
- OS_RUNNING_TASK_ID(void); // Returns the task number of the currently executing task
- OS_SCHECK (void); // Checks if running task's signal bit is set, returns a bit value
  // of 1 if signal is already present.
- OS_WAITP (void); // Waits for end of task's periodic interval, set by
  // the OS_PERIODIC command.

Download free eBooks at bookboon.com

The following RTOS system calls do receive parameters:

- OS_CREATE_TASK (uchar tasknum, uint taskadd); // Creates a task
- OS_INIT_RTOS (uchar iemask); // Initialises all RTOS variables
- OS_PERIODIC (uint ticks); // Tasks run periodically every number of ticks
- OS_RESUME_TASK (uchar tasknum); // Resumes a task which was previously KILLed
- OS_RTOS_GO (uchar prior); // Starts the RTOS with priorities if required
- OS_SIGNAL_TASK (uchar tasknum); // Signals a task
- OS_WAITI (uchar intnum); // Waits for an event (interrupt) to occur
- OS_WAITS (uint ticks); // Waits for a signal within a number of ticks
- OS_WAITT (uint ticks); // Waits for a timeout defined by number of ticks

The list of commands can also be grouped as those which cause a change of task, might cause a change of task and those which do not cause a task swap.

The following RTOS system calls force a task change after executing this command:

- OS_DEFER (void); // Stops current task and passes control to next task in queue
- OS_KILL_IT (void); // Kills a task – sets it waiting forever
- OS_WAITI (uchar intnum); // Waits for an event (interrupt) to occur
- OS_WAITT (uint ticks); // Waits for a timeout defined by number of ticks
- OS_WAITP (void); // Waits for the end of the task's periodic interval

The following RTOS system calls might force a task change after executing this command:

- OS_WAITS (uint ticks); // Waits for a signal within a number of ticks

If the signal is already present when the command is issued, then no task swap is made, otherwise a task change is performed.

The following RTOS system calls do not force a task change, and the task using any of these commands would continue to run after executing the command:

- OS_CREATE_TASK (uchar tasknum, uint taskadd); // Creates a task
- OS_INIT_RTOS (uchar iemask); // Initialises all RTOS variables
- OS_PERIODIC (uint ticks); // Tasks run periodically every number of ticks
- OS_RESUME_TASK (uchar tasknum); // Resumes a task which was previously KILLed

Download free eBooks at bookboon.com

- OS_RTOS_GO (uchar prior); // Starts the RTOS with priorities if required
- OS_RUNNING_TASK_ID(void); // Returns the task number of the currently running task
- OS_SCHECK (void); // Checks if running task's signal bit is set
- OS_SIGNAL_TASK (uchar tasknum); // Signals a task

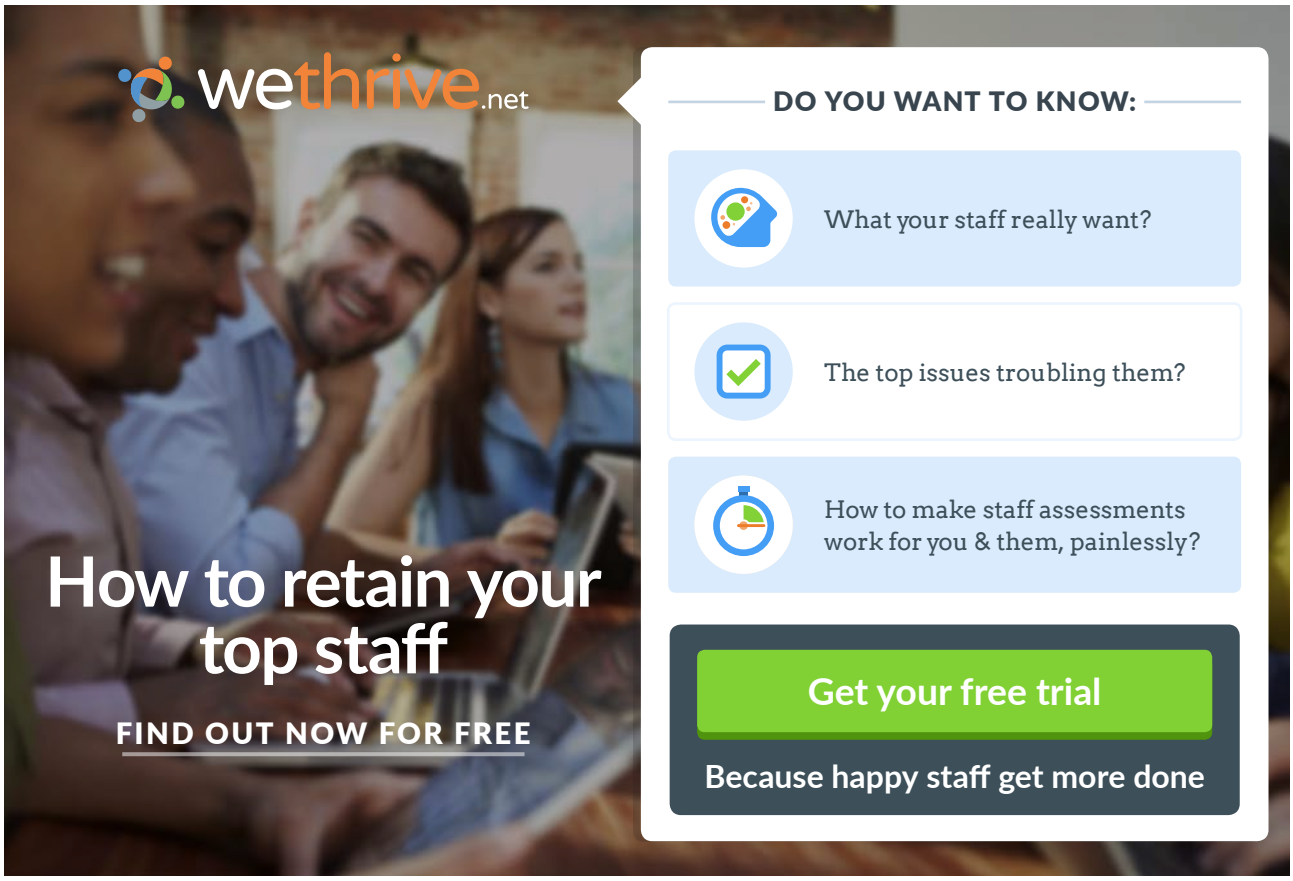
9.3 Descriptions of the commands

The C version of the RTOS provides some variations and additional commands which were implemented after having used the A51 program for a while. Some of the additions were only implemented in the C version although they can be easily added in the assembly version as well. The detailed description of the commands now follows, which would completely describe the RTOS. The complete PaulOS RTOS source program can be found in the Appendix D and examples are given at the end of this chapter which should make it easier to understand.

9.3.1 OS_INIT_RTOS(IEMASK)

This system command must be the **first** command to be issued in the main program in order to initialise the RTOS variables. It is called from the main program and takes the interrupt enable mask (IEMASK) as a parameter. An example of the syntax used for this command is:

```
OS_INIT_RTOS(0x30);
```



Click on the ad to read more

Download free eBooks at bookboon.com

which would imply that the application program intends to use the Timer 2 interrupt (IEMASK=20H) for the RTOS as well as the Serial Interrupt (IEMASK=10H). Hence the 0x30 parameter in the command.

Interrupt		IE MASK		Notes
No:	**Name**	**Binary**	**Hex**	
0	External Int 0	00000001	01	
1	Timer Counter 0	00000010	02	Default RTOS timer for 8051
2	External Int 1	00000100	04	
3	Timer Counter 1	00001000	08	
4	Serial Port	00010000	10	
5	Timer 2 (8032 only)	00100000	20	Default RTOS for 8032

Table 9-1 IEMASK Parameter (PaulOS)

The correct mask for the RTOS timer (defined in the file parameters.h) is always added (or ORed) by the RTOS automatically to any other mask, even if one forgets to enable it in the IEMASK parameter. The interrupts which are valid are shown in Table 9-1. This implies that in order to change the tick timer (that is the interrupt number for the RTOS) we have to change the TICK_TIMER parameter in the parameters.h file.

This system command performs the following:

- Clears the external memory area which is going to be used to store the stack of each task.
- Sets up the IE register (location A8H in the SFR area).
- Selects edge triggering on the external interrupts. This can be amended if a different triggering is required by changing directly the default initialisation in the RTOS source code listing found in Appendix D or by re-setting the correct triggering mode after having initialised the RTOS so as to override the default value. This is done by setting the correct bit value for IT0 and IT1 residing in the TCON SFR as already stated in section 2.10.16.
- Loads the Ready Queue with the main idle task number, so that initially only the main task will execute.
- Initialises all tasks as being not waiting for a timeout.
- Sets up the Stack Pointer (SP) variable of each task to point to the correct location in the stack area of the particular task. The stack pointer, initially, is made to point to an offset of 14 bytes above the base of the stack [(MAIN_STACK - 1) + NOOFPUSHES + 2] since NOOFPUSHES in this case is 13. The first 13 locations would initially all contain a zero. This is done so as to ensure that when the first RET instruction is executed after transferring the stack from external RAM on to the 8032 RAM, the SP would be pointing correctly to the address of the task to be started. This is seen in the QSHFT routine, where before the last RET instruction, there is the Pop_Bank0_Reg macro which effectively pops 13 registers. The RET instruction would then read the correct address to jump to from the next 2 locations.

Download free eBooks at bookboon.com

9.3.2 OS_CREATE_TASK(Task No:, Task Name)

This system command is used in the main program for each task to be created. It takes two parameters, namely the task number (the first task is normally numbered as task 0), and the task address, which in the C environment, would simply be the name of the procedure or function. An example of the syntax used for this command is:

```
OS_CREATE_TASK(0, MotorOn);
```

This would create a task, numbered 0 which would refer to the MorotOn() procedure or function.

This system command performs the following:

- Places the task number in the next available location in Ready Queue, meaning that this task is ready to execute. The location pointer in Ready Queue is referred to as READYQTOP in the program, and is incremented every time this command is issued.
- Loads the address of the start of the task at the bottom of the stack area in external ram allocated to this task. The SP for this task would have been already saved, by the INIT_ RTOS command, pointing to an offset 13 bytes above this, to compensate for the pops.

9.3.3 OS_RTOS_GO(Priority)

This system command is used only ONCE in the main program, when the RTOS would be required to start supervising the processes. It takes one Priority bit parameter.

The Priority bit parameter (0 or 1) if set to 1, implies that those tasks placed in the Ready Queue (meaning ready to execute), would be sorted in descending order before the RTOS selects the next task to run. A task number of 0 is taken to mean by the RTOS as the **highest** priority task, and would obviously be given preference during the sorting. The main() task or function is automatically given the highest task number (thus meaning the lowest priority) by the RTOS, so as all the other tasks would be able to interrupt it.

An example of the syntax used for this command is:

```
OS_RTOS_GO(1);
```

This would start the RTOS ticking with priority enabled. The tick time interval is determined by the parameter TICKTIME set in the parameters header file (say 1ms, 5ms or 10ms). This value would then become the basic reference unit for other system commands which use any timeout parameter.

The RTOS would also be required to execute “ready-tasks” sorting prior to any task change, since the priority parameter was set to 1.

Download free eBooks at bookboon.com

Assuming Timer 2 is being used to generate the ticktime this system command performs the following:

- Loads the variable DELAY (LO and HI bytes), with the number of BASIC_TICKS required to obtain the required ticktime delay.
- Sets the PRIORITY bit according to the priority parameter supplied.
- Loads RCAP2H and RCAP2L, the Timer 2 registers, with the required count in order to obtain the required delay between Timer 2 overflow interrupts. The value used depends on the crystal frequency used on the board. The clock registers count up at one twelfth the clock frequency, and using a clock frequency of 11.0592 MHz, each count would involve a time delay of 12/11.0592 μs or 1.08507 μs. Therefore to get a delay of 1ms (1000 μs), 1000/1.08507 or 921.6 counts would be needed. We would use integer 921 to get this delay, hence the reload registers (RCAP2H,RCAP2L) would be loaded with 65536 – 921 since the timers count up till they overflow.
- Stores the reference time signal parameter in GOPARAM and TICKCOUNT.
- Starts Timer 2 in 16-bit auto-reload mode.
- Enables interrupts.
- Sets TF2, which is the Timer 2 overflow interrupt flag, thus causing the 1st interrupt immediately.

9.3.4 OS_RUNNING_TASK_ID( )

This system command is used by a task to get the number of the task itself. It returns an unsigned character (1 byte) value and the same task continues to run after executing this system command.

An example of the syntax used for this command is:

```
X = OS_RUNNING_TASK_ID();          /* where X would be an unsigned character */
```

9.3.5 OS_SCHEK( )

This system command is used by a task to test whether there was any signal sent to it by some other task.

- It returns a bit value of:
  - 0 if Signal is not present
  - 1 if Signal is present
- If the signal was present, the signal flag (bit) is also cleared before returning to the calling task. The same task continues to run, irrespective of the returned value.

An example of the syntax used for this command is:

```
X = OS_SCHEK();        /* where X would be a bit-type variable */
```


Download free eBooks at bookboon.com

or one may use it as in the following example to test the presence of the signal bit:

```
if (OS_SCHEK() == 1)
 {
 /* do these instructions if a signal was present */
 }
```

9.3.6 OS_SIGNAL_TASK(Task No:)

This system command is used by a task to send a signal to another task. If the other task was already waiting for a signal, then the other task is placed in the Ready Queue and its waiting for signal flag is cleared. The task issuing the OS_SIGNAL_TASK command continues to run, irrespective of whether the called task was waiting or not waiting for the signal. If we need to halt the task after the OS_SIGNAL_TASK command to give way to other tasks, we must use the OS_DEFER() system command after the OS_SIGNAL_TASK command.

Click on the ad to read more

Download free eBooks at bookboon.com

This system command performs the following:

- It first checks whether the called task was already waiting for a signal.
- If the called (signaled) task was not waiting, it sets its waiting for signal (SIGW) flag and exits to continue the same task.
- If the signaled task was already waiting, it places the called task in the Ready Queue and it clears both the waiting for signal (SIGW) and the signal present (SIGS) flags.
- It also sets a flag (TINQFLAG) to indicate that a new task has been placed in the Ready Queue. This flag is used by the RTOS_TIMER_INT routine (every half a millisecond) in order to be able to decide whether there has to be a task change. It then exits the routine to continue the same task.

An example of the syntax used for this command is:

```
OS_SIGNAL_TASK(1);        // send a signal to task number 1
OS_DEFER();               // give cpu time to other tasks, if necessary
```

9.3.7 OS_PERIODIC (uint ticks)

This command initalises the task to repeat periodically, every certain number of ticks given as a parameter in the command. It is used at the beginning of a task, OUTSIDE of the endless loop, as shown in the next sub-section 9.3.8. An example of its usage is also given in that same sub-section.

We now deal with the commands that do perform a voluntary (co-operative) change of task:

9.3.8 OS_WAITP (void)

This command sets the task waiting for the preset periodic interval (set previously by the OS_PERIODIC(ticks) command. The task goes into a waiting state and the next ready task takes over.

If the interval has already passed when this command is executed, then the task would continue to execute. This is not normally the case, and only happens when there is a programming logic or algorithm mistake, since it would generally mean that the task is actually taking longer to execute than the requested periodic interval between executions.

It performs the following:

- Saves task environment in preparation for the expected task swap.
- If the periodic interval has not yet passed, as is generally the case, it sets the periodic interval flag to indicate that it is waiting for the periodic interval and issues a voluntary task change.

Download free eBooks at bookboon.com

- If however the periodic interval has already elapsed (this is usually due to bad programming, in cases where the code of the task itself takes a longer time to execute than the required periodic interval), then it clears the periodic interval flag and exits.

Such a command is used in a task, in conjunction with the OS_PERIODIC() command and an example of its usage is shown below:

```
OS_PERIODIC(50);                    // declare task as wishing to execute every 50 ticks
while(1)                            // repeat forever
{
….                                  // code to be executed every 50 ticks
….                                  // which should not take longer than
….                                  // 50 ticks to execute.
OS_WAITP();                         // wait for the periodic interval to pass
}
```

9.3.9 OS_WAITI(Interrupt No:)

This system command is called by a task to sleep and wait for an interrupt to occur. Another task, next in line in the Ready Queue would then take over. If the interrupt never occurs, then the task will effectively sleep for ever. This is one way of writing Interrupt Service Routines under PaulOS RTOS control. ISRs can also be written in such a way as to run independently, as describe in section 9.3.15.

If required, this command can be modified to allow another timeout parameter to be passed, so that if the interrupt does not arrive within the specified timeout, the task would resume. A timeout of 0 would still leave the task waiting for the interrupt forever. The modification required to the RTOS source listing would be similar to the OS_WAITS command, and the operation would then be as explained further down in sub-section 9.3.10.

This system command performs the following:

- It sets the bit which corresponds to the interrupt number passed on as a parameter.
- It then calls the QSHFT routine in order to start the task next in line.

An example of the syntax used for this command is:

```
OS_WAITI(0);                        // wait for an interrupt from external int 0
```

The task would then go into the sleep or waiting mode and a new task would take over.

Download free eBooks at bookboon.com

9.3.10 OS_WAITS(Timeout)

This system command is called by a task to sleep and wait for a signal to arrive from some other task. If the signal is already present (previously set or signaled by some other task), then the signal is simply cleared and the task continues on. If the signal does not arrive within the specified timeout period, the task resumes just the same. However, a timeout number of 0 would imply that the task has to keep on waiting for a signal indefinitely. If the signal does not arrive, then the task never resumes to run and effectively the task is killed.

This system command performs the following:

- It first checks whether the signal is already present.
- If the signal is present, then it clears the signal flag, exits and continues running.
- If the signal is not present, then:
  - It sets its own waiting for signal (SIGW) flag.
  - It also sets the waiting for timeout variable according to the supplied parameter.
  - It then jumps to the QSHFT routine in order to start the task next in line.

Click on the ad to read more
Download free eBooks at bookboon.com

An example of the syntax used for this command is:

```
OS_WAITS(50);
// wait for a signal within 50 units or ticks, the value of the unit depends on
// the TICKTIME parameter used.
```

If for example, the TICKTIME was set to 10 milliseconds in the header file, an OS_WAITS(50) would then imply waiting for a signal to arrive within 500 milliseconds.

or you can use:

```
OS_WAITS(0); // this would wait for a signal for ever
```

In both examples, if the signal is not already present, the task would then go into the sleep or waiting mode and a new task would take over.

9.3.11 OS_WAITT(Timeout)

This system command is called by a task to sleep and wait for a specified timeout period. The timeout period is in units whose value depends on the TICKTIME parameter used. Valid values for the timeout period are in the range 1 to 65535. A value of 0 is reserved for the OS_KILL_IT command, meaning permanent sleep, and therefore is not allowed for this command. The OS_WAITT system command therefore performs the required check on the parameter before accepting the value. If by mistake a value of 0 is given as a timeout parameter, then it is automatically changed to a 1. Once the timeout period passes, the task which had issued this command, would be moved from the waiting to the ready queue.

This system command performs the following:

- If the parameter is 0, then set it to 1, to avoid permanent sleep.
- Save the correct parameter in its correct place in the TTS table.
- Jump to the QSHFT routine in order to start the task next in line.

An example of the syntax used for this command is:

```
OS_WAITT(60);
// wait for a signal for 60 units, the value of the unit depends on
// the TICKTIME parameter used.
```


Download free eBooks at bookboon.com

If for example, the command TICKTIME was set to 10, the reference unit would be 10 milliseconds, and OS_WAITT(60) would then imply waiting or sleeping for 600 milliseconds. The task would then go into the sleep or waiting mode for 600ms and a new task would take over. After 600ms it would move to the ready queue.

9.3.12 OS_KILL_IT( )

This system command is used by a task in order to stop or terminate the task. As explained earlier in OS_WAITT, this is simply the command OS_WAITT with an allowed timeout of 0. The task is then placed permanently waiting and never resumes execution.

This system command performs the following:

- First it clears any waiting for signal or waiting for interrupt flags, so that that task would definitely never restart.
- Then it sets its timeout period in the TTS table to 0, which is the magic number the RTOS uses to define any non-timing task.
- Then it sets the INTVLRLD and INTVLCNT to 0, again implying that it is not a periodic task.
- Finally it jumps to the QSHFT routine in order to start the task next in line.

An example of the syntax used for this command is:

```
OS_KILL_IT();
/* the task simply stops to execute and a new task would take over.*/
```

9.3.13 OS_DEFER( )

This system command is used by a task in order to hand over processor time to another task. The task is simply placed at the end of the Ready Queue, while a new task resumes execution.

This system command performs the following:

- It sets its timeout period in the TTS table to 0, which is the magic number the RTOS uses to describe any non-timing task.
- It places the task in the Ready Queue, by simply placing the task number in the next available location in Ready Queue area.
- It then flows on to the QSHFT routine in order to start the task next in line.

Download free eBooks at bookboon.com

An example of the syntax used for this command is:

```
OS_DEFER();
/* the task simply stops execution and is placed in Ready Queue.*/
/* A new task would then take over. */
```

9.3.14 Enhanced event-waiting and other add-on MACROS

These macros (#define statements) perform the same functions of the OS_WAITT, OS_WAITS and OS_PERIODIC calls but rather than ticks they accept absolute time values as parameters in terms of minutes, seconds and millisecs. This difference is denoted by the _A suffix (the A standing for Absolute) – eg. OS_WAITT_A(0,0,300) would cause a task to wait for 300ms and is the absolute-time version of OS_WAITT(x), where x would have to be calculated to give the required number of ticks equivalent to a 300ms delay.

Range of values (65535 TICKTIMES) accepted is listed below:

Using a minimum TICKTIME of 1ms :

Range from 1ms to 1m, 5s, 535ms in steps of 1ms.

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

Click on the ad to read more

Download free eBooks at bookboon.com

Using a recommended TICKTIME of 10ms:

Range from 10ms to 10m, 55s, 350ms in steps of 10ms.

Using a maximum TICKTIME of 50 ms:

Range from 50ms–54m, 36s, 750ms in steps of 50ms

If the conversion from absolute time to ticks results in 0 (all parameters being 0 or overflow) this result is only accepted by OS_WAITS() by virtue of how the OS_WAITT(), OS_WAITS() and OS_PERIODIC() calls were written. In the case of the OS_WAITT() and OS_PERIODIC() calls the tick count would automatically be changed to 1 meaning an interval of 1 ticktime.

```
OS_WAITT_A(M,S,ms)      // Absolute OS_WAITT for minutes, seconds and milliseconds
OS_WAITS_A(M,S,ms)      // Absolute OS_WAITS for minutes, seconds and milliseconds
OS_PERIODIC_A(M,S,ms)   // Absolute OS_PERIODIC for minutes, seconds and milliseconds

OS_PAUSE_RTOS()         // Disable the RTOS, used in a stand-alone ISR
OS_RESUME_RTOS()        // Re-enable the RTOS, used in a stand-alone ISR

OS_CPU_IDLE()           // Sets the µC in idle mode in PCON SFR (section 1.8.10).
                        // This is usually used in the main program endless loop after
                        // initialising and starting the RTOS.
OS_CPU_DOWN()           // Sets the µC in power-down mode in PCON SFR (section 1.8.10).
```

9.3.15 Stand-alone Interrupt Service Routines

In the C version of the RTOS, a simple method of having one or more stand-alone interrupt service routine (ISR) which would run whenever some interrupt is generated has been included.

All we have to do is to set to '1' the corresponding interrupt in the PaulOS.H file. For example if we intend to have an ISR running under the EXT 0 interrupt (and not under RTOS control), then we have to make sure to set to one the corresponding #define statement in PaulOS.H file. In some examples or listings shown in this book, these STAND_ALONE_ISR parameters were moved to the parameters.h header file so as to have all parameters which can be changed by the user in one file, but the effect is the same.

```
#define STAND_ALONE_ISR_00 1 // EXT0 – set to 1 if using this interrupt as a stand alone ISR
```


Download free eBooks at bookboon.com

Then in the ISR itself we should also include the commands OS_PAUSE_RTOS() when starting the ISR and then OS_RESUME_RTOS() in order to resume the RTOS before exiting the ISR. This would ensure that the RTOS does not interfere with the stand-alone ISR.

It is best to use register banks 2 or 3 for these ISRs.

Example of a stand-alone ISR, interrupting the RTOS and executing immediately when the interrupt occurs.

```
void ISR_EXT0 (void) interrupt 0 using 2
  {

      OS_PAUSE_RTOS()          // Disable the RTOS, used in a stand-alone ISR
      /* Our service routine code goes in here */
      /* Our service routine code goes in here */
      /* Our service routine code goes in here */
      OS_RESUME_RTOS()         // Re-enable the RTOS, before exiting the stand-alone ISR
  }
```

9.4 PaulOS parameters header file

This is the RTOS parameters header file. We eould mainly be needing to set the TICK_TIMER, TICKTIME and NOOFTAKS parameters to reflect out particular application program.

```
/*
*************************************************************************
* PARAMETERS.H -- RTOS USER DEFINITIONS
*************************************************************************
*/
#define STACKSIZE       0x0F    // Number of bytes to allocate for the stack
#define CPU             8032    // set to 8051 or 8032
#define TICK_TIMER 2            // Set to 0, 1 or 2 to select which timer to
                                // use as the RTOS tick timer
#define TICKTIME         2      // Length of RTOS basic tick in msec - refer
                                // to the RTOS timing definitions
#define NOOFTASKS        8      // Number of tasks used in application
/*
*************************************************************************
* PARAMETERS.H -- RTOS USER DEFINITIONS
*************************************************************************
*/
```


Download free eBooks at bookboon.com

9.5 Example using PaulOS RTOS

This is an example using the PaulOS RTOS. The same function is used to represent 62 different tasks.

Each task would generate random x,y co-ordinates to represent the column (0–79) and row (5–20) where to display a character to represent the task number (A = task 0, B = task 1 and so on). LEDs are connected to Port B (assuming we have the FLT-32 development board) which display the running task number as a binary number. Three other tasks are created to clear the screen, display the stack size used by each task and to generate the random seed. As the program executes, the screen is populated with different characters to represent the 62 tasks.

Click on the ad to read more

Download free eBooks at bookboon.com

```
/*
*********************************************************************
*                                  PAULOS
*                            The Real-Time Kernel
*
*
*
*                            EXAMPLE random06.c
*********************************************************************************
*/
#include <reg52.h>            /* special function registers 8052 */
#include "PaulOS.h"           /* PaulOS C version system calls definitions */
#include <absacc.h>
#include <stdio.h>
#include <stdlib.h>
#include "..\Others\SerP2Pkg.h"
#include "..\Others\Flt32Pkg.h"
#include "..\Others\HYPER_PC.H"

extern uchar MaxSPTs[NOOFTASKS];

#define TaskWaitmSec 900

/*
*********************************************************************
*                                  TASKS
*********************************************************************
*/

void CommonTask (void)
{

        uchar x,y,z,s[5];

        z = 1 + OS_RUNNING_TASK_ID();
        OS_PERIODIC_A(0,z,TaskWaitmSec);      /* Run every (1 + Task ID) seconds */
        while(1)
        {
            x = rand()%80;       /* Get X position (0-79) where task number will appear */
            y = 5 + rand()%16;   /* Get Y position (5-20) where task number will appear */
            z = OS_RUNNING_TASK_ID();
            PC_DispChar(y,x,z+'A');   /* Display the task number on the screen */
            WritePort('B',z);         /* LEDs connected to Port B show running task No.*/
            sprintf(s,"%02bu",z);
            PC_DispStr(22,40,s);
            OS_WAITP();
        }
}
/*
*********************************************************************
*/
```


Download free eBooks at bookboon.com

```
/*
*********************************************************************
*/
void StackSize (void) {
        #if STACK_CHECK
                uchar i,j;
                uchar s[20];
                ulong k;

        #endif

        while(1) {
                #if STACK_CHECK

                        OS_WAITT_A(0,20,0);

                        j = OS_RUNNING_TASK_ID();
                        sprintf(s,"%02bu",j);
                        PC_DispStr(22,40,s); /* Display the task number on screen */
                        WritePort('B', j);

                        for (i=2;i<=20;i++) PC_DispClr2EndOfRow(i,0);
                        PC_DispStrCntr (2,"Maximum Stack size (per task) used so far");
                        PC_DispStr(3,5,"Task Stack Size Task Stack Size Task Stack Size
Task Stack Size");
                        PC_DispStr(4,5," No Bytes No Bytes No Bytes No Bytes");
                        j=0;
                        for (i=0;i<=NOOFTASKS;)       {
                        sprintf(s," %02bu      %03bu",i,MaxSPTs[i++] - MAINSTACK);
                        if(i<=NOOFTASKS+1) PC_DispStr(5+j,5,s);
                        sprintf(s," %02bu      %03bu",i,MaxSPTs[i++] - MAINSTACK);
                        if(i<=NOOFTASKS+1) PC_DispStr(5+j,22,s);
                        sprintf(s," %02bu      %03bu",i,MaxSPTs[i++] - MAINSTACK);
                        if(i<=NOOFTASKS+1) PC_DispStr(5+j,41,s);
                        sprintf(s," %02bu      %03bu",i,MaxSPTs[i++] - MAINSTACK);
                        if(i<=NOOFTASKS+1) PC_DispStr(5+j++,58,s);
 }
                for (k=0;k<80000;k++){};
                for (i=2;i<=21;i++) PC_DispClr2EndOfRow(i,0);
        PC_DispStrCntr (2,"by Paul P. Debono - EXAMPLE Random 06");
        PC_DispStrCntr (3,"C Version by John Blaut");

                #else

                        OS_KILL_IT();

                #endif

 }
}
/*
/*
*********************************************************************
*/
```


Download free eBooks at bookboon.com

```
void ClearArea (void)
{
        uchar i,s[3];
        OS_PERIODIC_A(0,25,0);        /* Repeat every 25 seconds */
  while(1)
 {
        i = OS_RUNNING_TASK_ID();
        sprintf(s,"%02bu",i);
        PC_DispStr(22,40,s);          /* Display the task number on the screen */
        WritePort('B', i);
        for (i=5;i<=20;i++) PC_DispClr2EndOfRow(i,0);
        PC_DispStr(22,40,s);          /* Display the task number on the screen */
        OS_WAITP();
 }
}

/*
*********************************************************************
*/
void RandomSeed (void)
{
        uint x;
        uchar z,s[3];
        OS_PERIODIC_A(0,3,500);              /* Run every 3.5 seconds */
    while(1)
    {
                z = OS_RUNNING_TASK_ID();
                sprintf(s,"%02bu",z);
                PC_DispStr(22,40,s);          /* Display the task number on the screen */
                WritePort('B',z);
                x = (x+1)%0xFFFF;
                srand(x);
                OS_WAITP();
    }
}
/*********************************************************************/
/*********************************************************************/
/*$PAGE*/
/*********************************************************************/
/********************************************************************* MAIN
*********************************************************************/
/* Using ANSI.SYS Escape control sequence           */
/* Clear Screen              Esc[2J                 */
/* Position Cursor           Esc[row;colH           */
/* Clear to end of line Esc[K                       */
void main (void)
{
        uchar i;

        OS_INIT_RTOS(0x20); /* initialise RTOS variables and stack */
        Init_8255(0x91); /* Initialise the 8255 */
        Set_P2_BaudRate (38400);
```


Download free eBooks at bookboon.com

```
        PC_DispClrScr(); /* Clear the screen */
        PC_DispStrCntr (1,"PaulOS, The Real-Time 8051 Co-Operative Kernel");
        PC_DispStrCntr (2,"by Paul P. Debono - EXAMPLE Random 06 with 65 tasks");
        PC_DispStr(22,31,"Task No:");

        for(i=0;i<=61;i++)
               {
                  OS_CREATE_TASK(i,CommonTask); /* CREATE common tasks */
               }
        OS_CREATE_TASK (62,StackSize);         /* CREATE task */
        OS_CREATE_TASK (63,ClearArea); /* CREATE task */
        OS_CREATE_TASK (64,RandomSeed); /* CREATE task */
        OS_RTOS_GO(0); /* Start multitasking */
        while (1)
          {
                OS_CPU_IDLE();
/* Go to idle mode if doing nothing, to conserve energy */
          }
}

/*
*********************************************************************
*/
```

Click on the ad to read more
Download free eBooks at bookboon.com

10 MagnOS – a Pre-Emptive RTOS

The final RTOS which we discuss is the MagnOS (MAGNus Operating System, Magnus meaning Great in Latin), which gives a demonstration of a pre-emptive RTOS. In this system, each task is given a priority, and the basic control logic of this RTOS is that the highest priority task runs for as long as necessary, until a higher priority task becomes ready to execute. The trick here is to learn to decide what priority to give to the individual tasks so as to avoid having a single task take over completely the processor time, without giving a chance for other tasks to run.

This RTOS, which is a pre-emptive RTOS is a further modification of the PaulOS co-operative RTOS program, written in C to make it more versatile and easier to port to other micro-controller variations and types. It can be further developed into a more complex RTOS if one can dedicate more time to it. The original idea behind this RTOS once again came from the book by Prof. Thomas W. Schultz "C and the 8051 – Volume II"[2] which described the basic ideas and workings of such a simple but effective pre-emptive RTOS.

Obviously, the main improvement of this RTOS over the PaulOS RTOS, is in the pre-emptive swapping of tasks capability. If the RTOS sees any task which is ready to execute and which has a higher priority than the current one running, it will interrupt that running task and it will start (or resume) executing the higher priority task instead. The task will then continue to run until it either gives up the processor/controller time on its own accord by some command similar to the PaulOS method (such as OS_WAITT(x)) or else another task having an even higher priority becomes ready to execute and therefore the RTOS would give it the priority to run.

Great care has to be taken in deciding what priority to allocate to each of the individual tasks and also in the use of variables and/or resources by more than one task.

Naturally there are some memory space and speed penalties to pay for this versatility. Because of this, if one can perform the required project with a co-operative RTOS, then one has no need for the pre-emptive RTOS. However the improvements more than outweigh the penalties, and since it is written in C, the student can better understand the workings of the RTOS. The full source code listing of this RTOS can be found in appendix E. Here is the list of the MagnOS RTOS commands and some description of each:

10.1 MagnOS System Commands

Some of the commands are exactly the same as those used in the PaulOS RTOS, but are also being listed here for the sake of completeness.

Download free eBooks at bookboon.com

The following RTOS system calls do not receive any parameters:

- OS_RTOS_GO (void); // Starts the RTOS with priorities if required
- OS_WAITP (void); // Waits for end of task's periodic interval
- OS_RUNNING_TASK_ID(void); // Returns the number (unsigned char) of current task

The following RTOS system calls do receive parameters :

- OS_INIT_RTOS (uchar iemask); // Initialises all RTOS variables
- OS_WAITI (uchar intnum); // Waits for an event (interrupt) to occur
- OS_WAITT (uint ticks); // Waits for a timeout defined by number of ticks

- OS_CHECK_TASK_PRIORITY (uchar task_num) // gets the requested task priority setting
- OS_CHANGE_TASK_PRIORITY (uchar task_num, uchar new_prio) // sets the task priority

- OS_RELEASE_RES (uchar Res_Num) // releases the resource, for use by other tasks
- OS_WAIT4RES (uchar Res_Num) // wait for the resource

- OS_SEND_MSG (struct letter xdata *msg) // send a message to a task
- OS_CLEAR_MSG (struct letter xdata *msg) // clears the message
- OS_CHECK_MSG (struct letter xdata *msg) // checks if message is present
- OS_GET_MSG (struct letter xdata *msg) // gets the message
- OS_WAIT_MESSAGE (struct letter xdata *msg) // waits for a message

- OS_CHECK_TASK_SEMA4 (uchar task_num) // checks the semaphore
- OS_SEMA4MINUS (uchar task_num, uchar units) // deducts units from the semaphore
- OS_SEMA4_PLUS (uchar task_num, ucahr units) // adds units to the semaphore
- OS_WAIT4SEM (uint ticks) // waits for the semaphore to get to zero within ticks time

- OS_KILL_TASK (uchar tasknum); // Kills a task – sets it to wait forever
- OS_CREATE_TASK (uchar tasknum, uint taskadd, uchar priority); // Creates a task

Download free eBooks at bookboon.com

10.2 Detailed description of commands

This pre-emptive RTOS (source listing given in the appendix E) provides some variations and additional commands which were implemented after having used the first test versions of the program for some time. We now describe what these commands actually do and how they were implemented. Although we might be repeating ourselves the commands which are very similar to PaulOS are once again described here since they might have some slight changes due to the priority and pre-emptive components of the MagnOS RTOS. Moreover it eliminates the need to continuously flick over the pages for references. Later, after reading this chapter, one can refer back to the PaulOS Chapter 9, and to the source code and the remarks in appendices D and E for further explanations and comparisons.

Each task has its own set of parameters, as declared in MagnOS.h file (see Appendix E) and shown here, since some of these parameters are used when explaining the commands in the various 10.2.x sub-sections.

Click on the ad to read more
Download free eBooks at bookboon.com

```
struct task_param {                 /* 13 bytes + 13 registers + stack per task */
        uchar catalog;              /* task id */
        uchar status1;              /* status flags, see below for details */
        uchar status2;              /* status flags, see below for details */
        uchar priority;             /* priority number, low = high priority */
        uchar semaphore;            /* counting semaphore for each task */
        uchar resource;             /* resource number required */
        uchar stackptr;             /* stack pointer SP storage location */
        uchar intnum;               /* task waiting for this interrupt number */
        uint timeout;               /* task waiting for this timeout in ticks, */
                                    /* 0 = not waiting */
        uint interval_count;        /* time left to wait for this periodic */
                                    /* interval task in ticks */
        uint interval_reload;       /* periodic tick interval reload value */
        uchar rega;                 /* registers storage area, ready for context */
                                    /* switching use */
        uchar regb;
        uchar rdph;
        uchar rdpl;
        uchar rpsw;
        uchar reg0;
        uchar reg1;
        uchar reg2;
        uchar reg3;
        uchar reg4;
        uchar reg5;
        uchar reg6;
        uchar reg7;
        char stack[STACKSIZE];          /* stack storage area */
                };
```

10.2.1 OS_RTOS_GO (void)

This is the command which starts the RTOS going.

It performs the following:

- It loads the correct timer (selected by the parameter TICK_TIMER) with correct reload value so as to generate overflow interrupts according to the BASIC_TICK selected.
- Starts the timer and sets the timer overflow interrupt flag so that it would cause an interrupt immediately.
- Signals to the RTOS that there are tasks in the READY queue, by setting flag TinQFlag to 1.
- Enables global interrupts, so that the timer interrupt is recognised immediately.

Download free eBooks at bookboon.com

Such a command is used only once, normally in the main() function so as to start the RTOS going. An example of its usage is shown below:

```
OS_RTOS_GO();
```

10.2.2 OS_PERIODIC (uint ticks)

This command initalises the task to repeat periodically, every certain number of ticks given as a parameter in the command. It is used at the beginning of a task, OUTSIDE of the endless loop, as shown in the next sub-section 10.2.3. An example of its usage is also given in that same sub-section.

10.2.3 OS_WAITP (void)

This command sets the task waiting for the preset periodic interval (set previously by the OS_PERIODIC(ticks) command. The task goes into a waiting state and the next ready task with the highest priority takes over.

If the interval has already passed when this command is executed, then the task would continue to execute. This is not normally the case, and only happens when there is a programming logic or algorithm mistake, since it would generally mean that the task is actually taking longer to execute than the requested periodic interval between executions.

It performs the following:

- Saves task environment in preparation for the expected task swap.
- If the periodic interval has not yet passed, as is generally the case, it sets the periodic interval flag to indicate that it is waiting for the periodic interval and issues a voluntary task change.
- If however the periodic interval has already elapsed (this is usually due to bad programming, in cases where the code of the task itself takes a longer time to execute than the required periodic interval), then it clears the periodic interval flag and exits.

Such a command is used in a task, in conjunction with the OS_PERIODIC() command and an example of its usage is shown below:

Download free eBooks at bookboon.com

```
OS_PERIODIC(50);                    // declare task as wishing to execute every 50 ticks
while(1)                            // repeat forever
{
….                                  // code to be executed every 50 ticks
….                                  // which should not take longer than
….                                  // 50 ticks to execute.
OS_WAITP();                         // wait for the periodic interval to pass
}
```

10.2.4 OS_RUNNING_TASK_ID(void)

This command simply returns the number (id) of the task which is currently running. This command does not cause a voluntary task change. Its usage is very straight forward, assuming that CurrentTask was previously declared as an *unsigned char* variable:

```
CurrentTask = OS_RUNNING_TASK_ID();
```

10.2.5 OS_INIT_RTOS (uchar iemask)

This command initalises all the RTOS and tasks variables, stacks, interrupt masks to their default values (mostly zeroes). The iemask is used to enable the interrupts which one intends to use for the tasks and the RTOS tick timer itself. The iemask bits reflect the interrupt numbers used in the 8051as shown in Table 10-1.

Bit	7	6	5	4	3	2	1	0
Interrupt	NA	NA	Timer 2	Serial	Timer 1	Ext 1	Timer 0	Ext 0

Table 10-1 IEMASK values

It is normally the first RTOS command used at the very beginning of the main() function, before creating the tasks and its use is shown below:

```
OS_INIT_RTOS(0x25); // mask=00100101, use interrupts EXT0, EXT1 and TF2
                    // Timer 2 would presumably be the RTOS tick timer
```


Download free eBooks at bookboon.com

10.2.6 OS_WAITI (uchar interrupt)

This command causes the task to wait for the required interrupt. The task goes into a waiting state and the next ready task with the highest priority takes over. The interrupt cannot be the same as that being used for the tick timer and obviously there cannot be a stand-alone ISR routine which is being activated this interrupt.

It performs the following:

- Sets the corresponding 'waiting for interrupt' flag for the task.
- Stores environment in preparation for the voluntary task swap.
- Performs the task swap.

It can be used by a task as follows:

```
OS_WAITI(4); // wait for the serial interrupt number 4
```

10.2.7 OS_WAITT (uint ticks)

This command causes the task to wait for the required number of ticks. The task goes into a waiting state and the next ready task with the highest priority takes over.

Click on the ad to read more

Download free eBooks at bookboon.com

It performs the following:

- Sets the corresponding 'timeout waiting parameter' for the task.
- Stores environment in preparation for the voluntary task swap.
- Performs the task swap

It can be used by a task as follows:

```
OS_WAITT(100); // wait for 100 ticks to pass
```

10.2.8 OS_CHECK_TASK_PRIORITY (uchar task_num)

This command returns the value (type unsigned char) of the required task priority. This command does not cause a voluntary task change and is used as follows:

```
Task_5_Priority = OS_CHECK_TASK_PRIORITY(5);
```

where Task_5_Priority would be a previously declared variable of type *unsigned char*.

It is mainly used in a task so as to be able to store the task priority before changing it. A higher value indicates a higher priority, which is the opposite of what was the practice in PaulOS.

10.2.9 OS_CHANGE_TASK_PRIORITY (uchar task_num, uchar new_prio)

This command changes the value of the required task priority. This command does not cause a voluntary task change and is normally used so as to increase or reset the priority of a particular task. This is normally required when for example something which cannot be interrupted is about to be executed, in which case the priority is temporary set to the highest value until the time critical code is executed. The task priority can then be restored to the original value. Such a task would be coded as follows:

```
P = OS_CHECK_TASK_PRIORITY(thistask);
OS_CHANGE_TASK_PRIORITY(thistask, highest_priority);
……..    // time critical code here
……..
OS_CHANGE_TASK_PRIORITY(thistask, P);
……..
```


Download free eBooks at bookboon.com

10.2.10 OS_RELEASE_RES (uchar Res_Num)

This command frees the given resource, thus making it available to other tasks. This command may cause a voluntary task change if there was another higher priority task waiting for this resource. The resource can be used to represent a function, a device or a variable.

The command performs the following:

- Stores environment in preparation for the voluntary task swap.
- Checks all tasks in order to see which is the highest priority task that was waiting for this resource.
- If no other higher priority task was waiting for this resource, then it simply marks this resource as being free and exits without performing any task swap.
- If a higher priority task is found, then:
  - the resource is marked as being used by the new task.
  - The current task is placed in the waiting queue (code default is for 3 ticks, the number was chosen for no particular reason).
  - Performs the task swap.

It can be used by a task as follows:

```
OS_RELEASE_RES(10);        // release resource number 10, which could represent a printer
                           // for example.
```

10.2.11 OS_WAIT4RES (uchar Res_Num, uint ticks)

This command causes the task to wait for the required resource to become available within a given timeout. A ticks value of zero implies keeping on waiting for the resource forever. The task goes into a waiting state if the resource is not available and the next ready task with the highest priority takes over. If the resource is already free and available, the task simply continues to execute. This command therefore may or may not perform a task swap.

This command performs the following:

- Stores environment in preparation for the voluntary task swap, if needed.
- If the resource is already available, it simply marks the resource as being in use by this task and exits without performing any task swap.
- If the resource is being used by some other task:
  - It sets the flags indicating that the task is waiting for a resource.
  - Marks which resource it is waiting for.
  - Sets the timeout tick time.
  - Performs the voluntary task change.

Download free eBooks at bookboon.com

The usage of this command is as follows:

```
        OS_WAIT4RES(10,0);   // wait forever for resource number 10
or
        OS_WAIT4RES(10,80); // wait for a maximum of 80 ticks for resource number 10,
                            // it will be placed in the ready queue once resource is available,
                            // if resource is still not available after 80 ticks, the task will
                            // still go ahead, which might be catastrophic!
```

10.2.12 OS_SEND_MSG (struct letter xdata *msg)

This command sends a message to another task. There are two message arrays stored in external RAM area associated with messages:

- A message (msg) array where messages are written to when a task wants to send a message. This same array is also used when a task reads a message.
- A mailbox (mbox) array where messages are stored whilst waiting to be read by the destination task.

Download free eBooks at bookboon.com

If the other task was already waiting for this message, then a voluntary task change is invoked. The message format (structure) carries information regarding the sender, recipient and data of the message itself. The structure of the message is composed of bytes representing the destination, source, length of message and up to 16 bytes for the message itself as shown in the declarations below:.

```
union dataformat {struct{ulong HI1,LO1,HI0,LO0;}dblwords;
                  struct{uint Hi3,Lo3,Hi2,Lo2,Hi1,Lo1,Hi0,Lo0;}words;
                  struct{uchar hi7,lo7,hi6,lo6,hi5,lo5,hi4,lo4,hi3,lo3,hi2,lo2,hi1,lo1,hi0,lo0;}bytes;
                  struct{char s[DATASIZE];}string;};

struct letter{uchar dest,src,len; union dataformat dat;};
```

The 'union' is used so that the data can be accessed easily in any form.

The command does the following:

- Stores the environment in preparation for the voluntary task swap, if needed.
- Goes through the mbox array and if it finds another task already waiting for this message:
  - Copies the message for that task on to the msg array.
  - Moves the task waiting for message into the ready queue.
  - Places the task sending the message into the waiting (for 1 tick) queue. It would then be placed in the normal ready queue automatically by the RTOS at the next tick.
  - Performs the voluntary task change.

- If there is no other task currently waiting for this message, it simply leaves the message in the mbox array and exits without performing any task swap.

The usage of this command is as follows:

```
OS_SEND_MSG(letter1);
```

10.2.13 OS_CLEAR_MSG (uchar task_num)

This command simply clears a message from the mbox array, destined for the particular task number. This command does not cause a voluntary task change.

The usage of this command is as follows:

```
OS_CLEAR_MSG(3);    // clear message destined for task number 3
```

Please not that this does not cater for multiple messages to the same task.

Download free eBooks at bookboon.com

10.2.14 OS_CHECK_MSG (uchar task_num)

This command checks if there is a message destined for a particular task. It returns a bit value of 1 if the message is present and a bit value of zero if there is no message. This command does not cause a voluntary task change.

The use of this command is as follows:

```
Bit_Message_Present = OS_CHECK_MSG(3);    // checks if there is message for task 3
```

where Bit_Message_Present would be declared of type *bit*.

10.2.15 OS_GET_MSG (struct letter xdata *msg)

This command reads the message destined for the current task. This command does not cause a voluntary task change.

This command does the following:

- It goes through the mailbox mbox array and copies the message destined to it on to the message msg array of the current task.
- It then clears the mailbox.

The use of this command is as follows:

```
OS_GET_MSG(letter3);          // get the message for task number listed within the
                              // letter3 variable (type struct letter)
```

10.2.16 OS_WAIT_MESSAGE (struct letter xdata *msg, uint ticks)

This command waits for a message within ticks time. If the ticks parameter is set to zero, then the task would wait until the message is received, whenever that may occur. If the message is not already present, then a voluntary task change is performed and the next ready task with the highest priority takes over. If on the other hand the message is already present, no task swap is performed and the message is transferred from the mbox to the msg array of the task.

This command performs the following:

- Saves task environment in preparation for the expected voluntary task swap.
- It checks the mbox array and if the message is already present:
  - it reads the message to the msg array.
  - clears the mbox and exits.

Download free eBooks at bookboon.com

- If the message is not present, then it has to wait for it and therefore
  - It searches the mbox for a free location and reserves that area in mbox for the message to be received and performs the task swap, waiting for the message within the specified ticks.
  - If no free location is found in mbox it exits, marking the NO_MBOX_FREE_F flag and exits without performing the task swap.

This command can be used as follows:

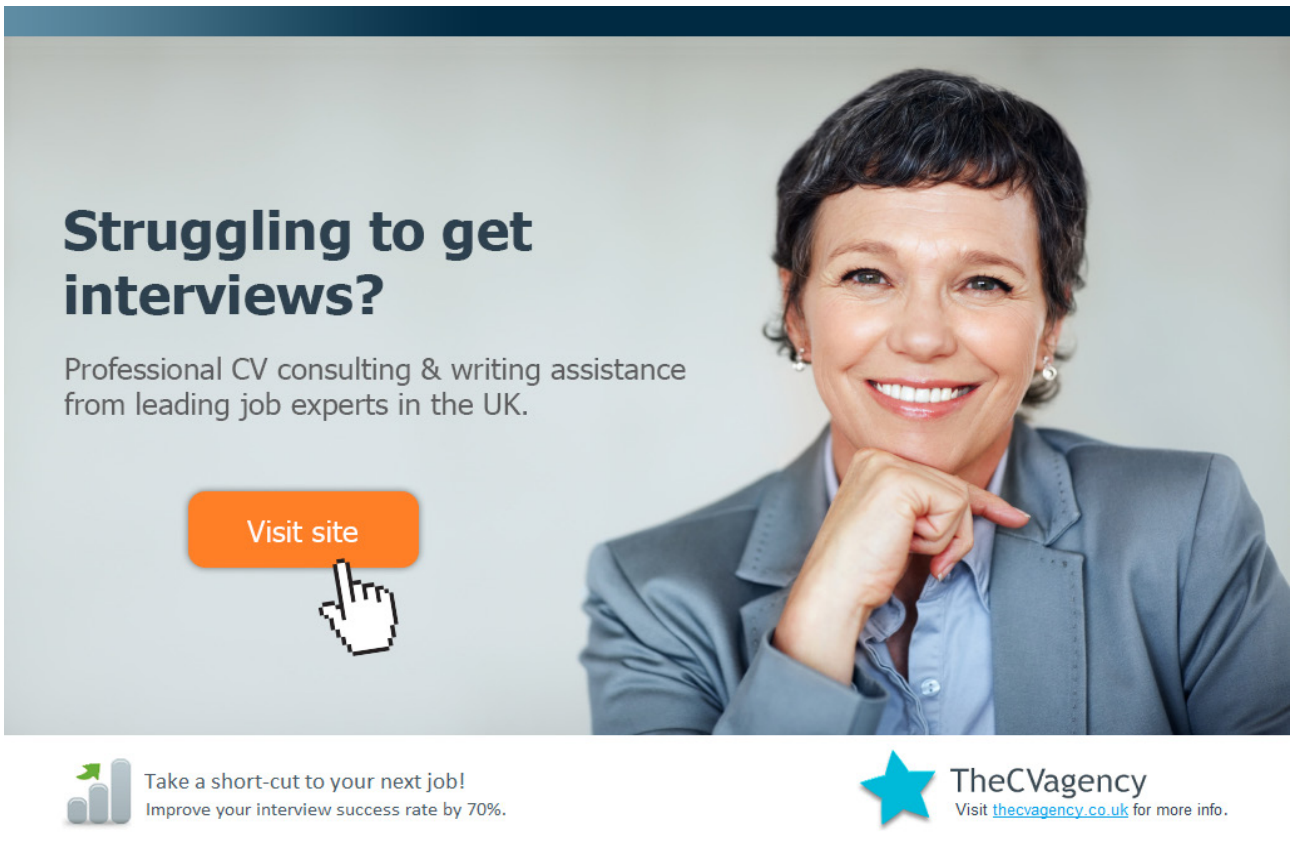
```
OS_WAIT_MESSAGE(letter,0);
if(task[Running].status2 & NO_MBOX_FREE_F)== NO_MBOX_FREE_F) {…..}
        // implies no free storage space found in mailbox
```

For a full explanation of status2, please refer to Appendix E header file MagnOS.h.

10.2.17 OS_CHECK_TASK_SEMA4 (uchar task_num)

This command checks for a semaphore of the particular task and simply returns an *unsigned char* value of the semaphores left. Each task can have its own semaphore, stored in its parameters array (sec section 10.2). This command does not cause a voluntary task change.

This command can be used as follows:



Download free eBooks at bookboon.com
Click on the ad to read more

```
SemaphoresLeft = OS_CHECK_TASK_SEMA4(5); // checks the semaphores left for task 5
```

where SemaphoresLeft is of type *unsigned char*.

10.2.18 OS_SEMA4MINUS (uchar task_num, uchar units)

This command deducts the given number of units (normally 1) from a semaphore of the particular task number. This command causes a voluntary task change only if the semaphores for the required task drop down to zero after the subtraction takes place. It therefore performs the following:

- Saves task environment in preparation for the eventual voluntary task swap.
- Deducts the required number of semaphore units (final resultant semaphores will be zero or greater).
- If the semaphore is now zero and there was a task waiting for the semaphore (to reduce to zero) then:
  - The task which was waiting for the semaphore is now placed in the waiting queue (the usual 1 tick time waiting delay, then placed automatically in the ready queue by the RTOS at the next tick) after clearing its semaphore waiting flag.
  - The present task is placed in the waiting queue (for 5 ticks, so as to give some time to other tasks. This can be changed in the source code).
  - A task swap is performed.

This command is used as follows:

```
OS_SEMA4MINUS(4,1);          // deduct 1 unit from the semaphore of task number 4
```

10.2.19 OS_SEMA4_PLUS (uchar task_num, ucahr units)

This command simply adds the given number of units (normally 1) to a semaphore of the particular task number. This command does not cause a voluntary task change.

This command is used as follows:

```
OS_SEMA4PLUS(4,1);           // add 1 unit to the semaphore of task number 4
```

10.2.20 OS_WAIT4SEM (uint ticks)

This command causes the task to wait for its semaphore to reach a value of zero within a given timeout period. A timeout ticks of zero implies having to wait forever until the semaphore reaches zero. The task goes into a waiting state if the semaphore is not zero and the next ready task with the highest priority takes over.

Download free eBooks at bookboon.com

The command does the following:

- Saves task environment in preparation for the eventual voluntary task swap.
- If the semaphore is already zero, it clears the wait for semaphore flag and resumes execution.
- If however the semaphore is not yet zero it sets the wait for semaphore flag and the wait for timeout flag and then it performs a voluntary task change.

The command can be used as follows:

```
OS_WAIT4SEM(0);                // wait forever until the semaphore becomes zero
```

10.2.21 OS_KILL_TASK (uchar tasknum)

This command kills the specified task and it will not execute again. This command will cause a voluntary task change. It performs the following:

- If the task was already killed by some other task, it simply exits.
- Otherwise it
  - Marks it as killed by setting the TASK_KILLED_F flag.
  - Clears and frees any mbox messages intended for this task.
  - If the task happens to be the one currently running (the task wants to commit suicide!), it clears all its flags and sets the timeout to zero so that it will appear to be waiting for ever. A task change is not called, without the need to save the environment.
  - If the task happens to be in the Ready queue, then the task number is changed to that of the idle task. Any multiple idle tasks entries in the queue are eliminated, so that at the end we would have only one idle task in the Ready queue.
  - All the task flags are reset.

This command can be used as follows:

```
OS_KILL_TASK(4);               // kill task number 4
```

10.2.22 OS_CREATE_TASK (uchar task_num, uint task_add, uchar task_priority)

This last command in the list creates a task and is used in the main task after intialising the RTOS but before starting the RTOS. This command does not cause a voluntary task change. The command does the following:

- It increments the Ready queue pointer and stores the task number in the Ready queue.
- It places the task start address in the stack area of that task, which would ultimately end in the SP once a task change is performed.

Download free eBooks at bookboon.com

- Stores the priority and other flags in the status area of that task and exits. A zero value would represent a low priority, while a value of 255 would represent the highest (top) priority.
  The command is normally used in the main() function, once for every task that needs to be created:

```
OS_CREATE_TASK(2,task_two,5);
```

// creates task of function named task_two, giving it a task number of 2 with a priority of 5.

MagnOS Parameters.h header file

This is the header file which we would need to modify depending on the application program. Typically we would need to set the TICKTIME and NOOFTASKS variables so as to reflect the actual tick time (say 1, 10 or 50 milliseconds) and number of tasks which we have in our main program.

Click on the ad to read more

Download free eBooks at bookboon.com

```
/*
************************************************************************
*                 PARAMETERS.H --- RTOS KERNEL HEADER FILE
*
* For use with MagnOS_V01.C
* Co-Operative RTOS written in C by Ing. Paul P. Debono
*                 for use with the 8051 family of microcontrollers
*
* File      : Parameters_V01.H
* Revision  : 8
* Date      : February 2006
* By        : Paul P. Debono
*
*                 B. Eng. (Hons.) Elec. Course
*                 University Of Malta
*
************************************************************************
*/

/*
************************************************************************
*                                       RTOS USER DEFINITIONS
************************************************************************
*/

#define STACKSIZE   0x10
            // Max size of stack for each task - no need to change
#define CPU         8032 // set to 8051 or 8032
#define TICK_TIMER  2    // Set to 0, 1 or 2 to select which timer to
                         // use as the RTOS tick timer
#define TICKTIME    50   // Length of RTOS basic tick in msec
                         // - refer to the RTOS
                         // timing definitions
#define NOOFTASKS   6    // Number of tasks used in the application program

/*
************************************************************************
************************************************************************
************************************************************************
*/
```


Download free eBooks at bookboon.com

We now give an example using the MagnOS RTOS just for demonstration purposes.

- Task 0 runs every minute and displays the alphabet in upper case letters, priority 1.
- Task 1 runs every four and a half secomds and displays the alphabet in lower case letters, priority 2.
- Task 2 runs every 700 milliseconds and displays the alphabet in Capital letters, priority 5.

It is interesting to change the periodicity (in the task functions) and priorities (in the main program) of the tasks and see the effect on the overall performance of the program. We need to remember that as far as the priority is concerned, a 0 value represents the lowest priority, and 255 would represent the highest (top) priority available.

Certain values can cause the RTOS to fail to start/finish the tasks within the required intervals. Scheduling probl;ems arise and the reader is urged to read material on task scheduling and assigning priorities for pre-emptive RTOSs.

Click on the ad to read more

Download free eBooks at bookboon.com

```
/*********************************************************************/
/*                                                                   */
/*    MagnosTest_00.c: Demo using pre-emptive Tasks                  */
/*                                                                   */
/*                                                                   */
/*********************************************************************/

#include <reg52.h>            /* special function registers 8052      */
#include <MagnOS_V01.h>       /* RTOS system calls definitions        */
#include <absacc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "UART_REENTRANT.h"

/*********************************************************************/
/*    Task 0:                                                        */
/*********************************************************************/

void CAPS (void){             /* Prints CAPITAL Letters               */
char code msg[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ-ZYXWVUTSRQPONMLKJIHGFEDCBA";

            static unsigned long i;
            static unsigned int j;

            OS_PERIODIC_A(1,0,0);    /* runs every 1 minute          */
            while(1)
            {
            TX_STRING("\nRunning Task: ");
            TX_CHAR('0'+ OS_RUNNING_TASK_ID());
            TX_STRING(" Priority: ");
            TX_CHAR ('0' + OS_CHECK_TASK_PRIORITY(OS_RUNNING_TASK_
ID()));
            TX_STRING("\n\r");
            j=0;
                  while (msg[j]!='\0')
                    {
                        TX_CHAR(msg[j++]);
                        for (i=1;i<10000;i++);
                    /* just a delay to simulate long process */
                    }
            TX_STRING("\n\r");
            OS_WAITP();
            }
 }

/*********************************************************************/
```


Download free eBooks at bookboon.com

```
/*************************************************************************/
/* Task 1: */
/*************************************************************************/

void Small (void){ /* Prints small Letters */
char code msg[]="abcdefghijklmnopqrstuvwxyz-zyxwvutsrqponmlkjihgfedc-
ba";

             static unsigned long i;
             static unsigned int j;

             OS_PERIODIC_A(0,4,500);       /* runs every 4.5 seconds */
             while(1)
            {
             TX_STRING("\nRunning Task: ");
             TX_CHAR('0'+ OS_RUNNING_TASK_ID());
             TX_STRING(" Priority: ");
             TX_CHAR ('0' + OS_CHECK_TASK_PRIORITY(OS_RUNNING_TASK_
ID()));
             TX_STRING("\n\r");
             j=0;
                   while (msg[j]!='\0')
                    {
                     TX_CHAR(msg[j++]);
                     for (i=1;i<300;i++);
                     /* just a delay to simulate long process */
                    }
             TX_STRING("\n\r");
             OS_WAITP();
      /* wait for Periodic timeout */
             }
 }

/*************************************************************************/
/*    Task 2:                                                            */
/*************************************************************************/

void Numbers (void){ /* Prints Numbers */
char code msg[]="0 1 2 3 4 5 6 7 8 9 - 9 8 7 6 5 4 3 2 1 0";

            static unsigned long i;
            static unsigned int j;
```


Download free eBooks at bookboon.com

```
            OS_PERIODIC_A(0,0,700);  /* runs every 700 milliseconds */
            while(1)
            {
            TX_STRING("\nRunning Task: ");
            TX_CHAR('0'+ OS_RUNNING_TASK_ID());
            TX_STRING(" Priority: ");
            TX_CHAR ('0' + OS_CHECK_TASK_PRIORITY(OS_RUNNING_TASK_
ID()));
            TX_STRING("\n\r");
            j=0;
                   while (msg[j]!='\0')
                    {
                     TX_CHAR(msg[j++]);
                     for (i=1;i<100;i++);
                    /* just a delay to simulate long process */
                    }
                   TX_STRING("\n\r");
                   OS_WAITP();      /* wait for Periodic timeout */
            }
 }

/*********************************************************************/
/*    Main: Initialise and CREATE tasks */
/*********************************************************************/

void main (void)        {     /* program execution starts here        */
      INIT_SERIAL_T1(57600);

      TX_STRING("Initialising MagnOS Pre-Emptive RTOS\n\r");
       OS_INIT_RTOS(0x20);/* initialise MagnOS RTOS variables and stack */
                                 /* using Timer 2 interrupts       */
/* A HIGH PRIORITY NUMBER, MEANS A HIGH PRIORITY TASK */
      OS_CREATE_TASK(0, CAPS,       1);   // priority 1
      OS_CREATE_TASK(1, Small,      2);   // priority 2
      OS_CREATE_TASK(2, Numbers,    5);   // priority 5

      TX_STRING("Tasks Created, running MagnOS RTOS\n\r");
      OS_RTOS_GO();                       /* start RTOS */

      while (1)
      {
       OS_CPU_IDLE();
      }
}
/*********************************************************************/
```


Download free eBooks at bookboon.com

11 Interfacing

This chapter deals with interfacing various devices to the 8051 family of micro-controllers. The list here is endless but the basic add-ons such as simple LEDs, switches, keypads, LCDs, DC motors (including servos and stepper motors) are all well covered with example programs.

11.1 Interfacing add-ons to the 8051

The 8051 on its own can be of little use unless we somehow manage to connect it to the real world. Minimally we would need some form of output device, such as an LED or a buzzer and an input interface which might even be a simple ON-OFF switch. Before going further, let us mention two important notes:

- A common fault when interfacing devices (even if simple) or other boards to the 8051 is to forget to connect the ground of the external device to the ground of the 8051 board. This would result in floating signals which would give indeterminate results.
- We should also remember when using the 8051 ports that port 0 needs external pull-up resistors whilst ports 1, 2 and 3 do not need any since they have them already internally wired. These pull-up resistors are not always shown in the following diagrams since it depends to which port we are connecting the interface circuit.

Click on the ad to read more

Download free eBooks at bookboon.com

With these notes in mind, we can list and describe a number of interface components which we can connect to the 8051.

11.2 LEDs

The simplest output indicator which we can connect would be a Light Emitting Diode (LED). We can connect the LED to either light up when the port pin is High (see Figure 11-1) or to light up when the port pin is Low (see Figure **11-2**).

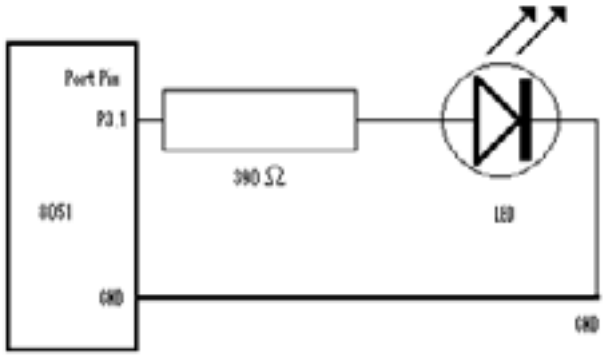


Figure 11-1 Port Driving LED (pin High = LED on)

The option shown in Figure **11-2** is better since the port is being used to sink the current rather than providing the source voltage.

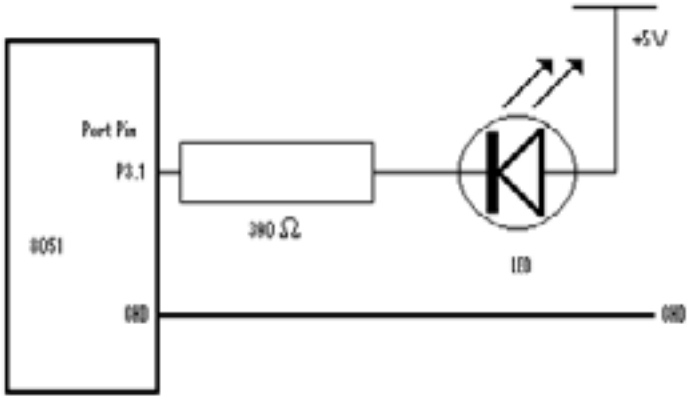


Figure 11-2 Port Sinking LED (pin Low = LED on)

We now list a section of code in C using Keil µVision4 for the circuit shown in **Figure 11-2**.

It will flicker the LED, switching it off for 1 second and then on for another second and so on until the microcontroller circuit is switched off.

Download free eBooks at bookboon.com

```
#include <REG52.H>
void msdelay( unsigned int );

sbit LED1 = P3^1;          // refer to bit P3.1 (port 3 bit 1) as LED1

#define led_on 0

#define led_off 1


void main(){

LED1=0;                    // set pin 1 of PORT3 as output

while(1){                  //infinite loop

LED1 = 1;                  //pins high, LED is off, or use LED1 = led_off;
msdelay(250);              // some delay
LED1 = 0;                  // pin low, LEDs are on, or use LED1 = led_on;
msdelay(250);              // some delay

}

}

//delay function
void msdelay(unsigned int value){

        unsigned int x,y;
        for(x=0;x<value;x++)
        for(y=0;y<1275;y++);
}
```

In C programs we cannot be sure of software delays, because they depend a lot on how the compiler optimizes the loops. As soon as we make some changes in the compiling options, the delay time changes.

A better option would be to use the in-built micro-controller timers if we want to have exact delays. Shown below is a function equivalent to a 1 second delay using timer 0, assuming we have an 11.0592 MHz crystal clock driving the micro-controller. The idea is to make a 50ms timer delay and repeat it for 20 times (20×50ms = 1000ms = 1s) so as to obtain the required one second delay. The timer would be counting at the rate of 12/11.0592 micro-seconds per count. Thus we need 46080 counts to get the required 50ms delay, and therefore, as we recall, we need to load the timer with the value of 19456 (which is 65536 – 46080) or 4C00 hex since our timer would be counting up until it overflows.

Download free eBooks at bookboon.com

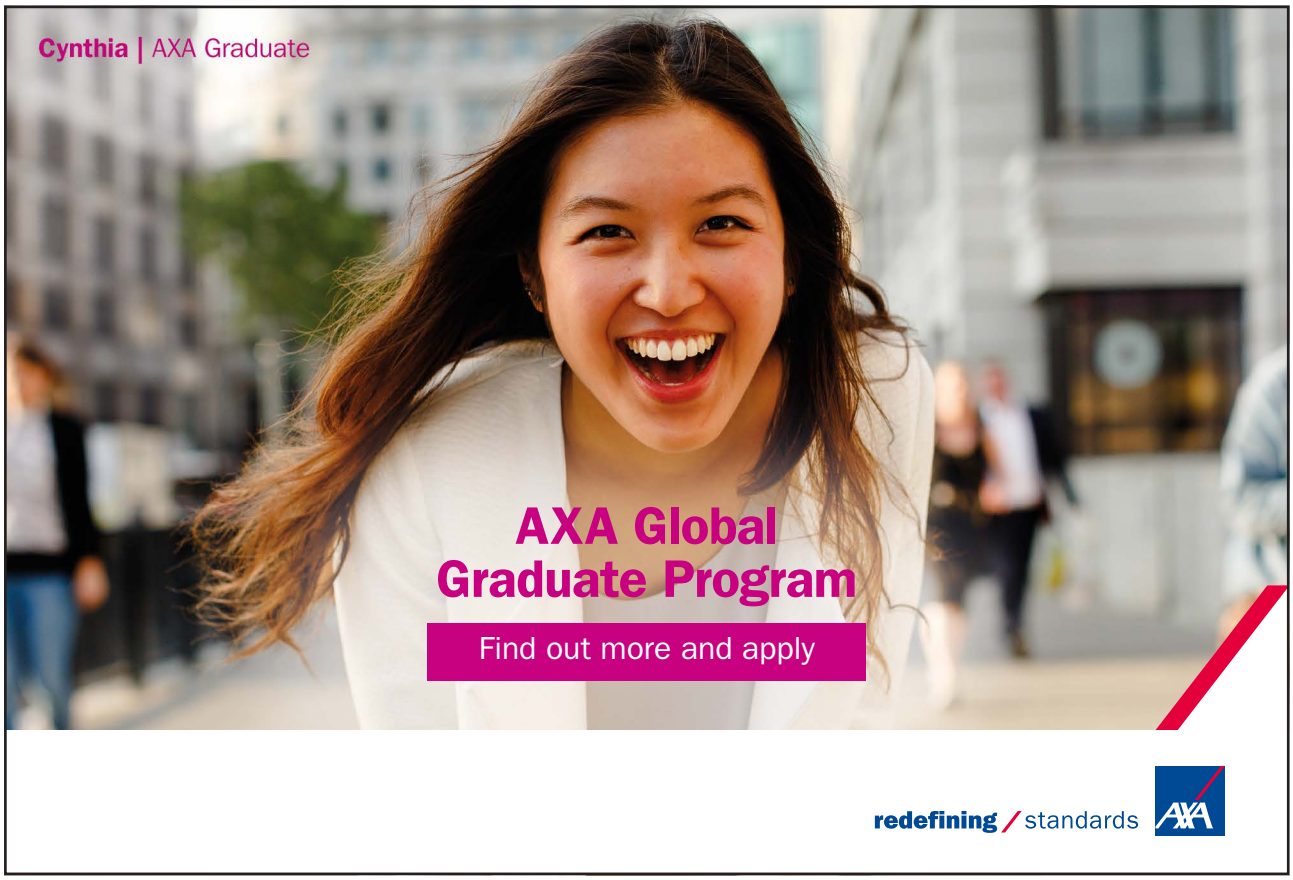
```
delay_1s()                      // using Timer 0 to get a 1 sec delay
{
int d;

TMOD &= 0xF0;                   // clear Timer 0 mode settings, temporarily to mode 0
TMOD |= 0×01;                   // set Timer 0 in mode 1, 16-bit
TF0 = 0;                        // clear Timer 0 overflow flag

for (d=0; d<=20; d++)           // repeat 20 times
 {
    TL0 = 0x00;                 // load it for 50ms overflow delay
    TH0 = 0x4C;                 // 4C00 hex = 19456
    TR0 = 1;                    // start Timer 0.
    while (TF0 == 0);           // run until TF0 = 1, indicating overflow, waiting 50ms

    TR0 = 0;                    // stop Timer 0
    TF0 = 0;                    // reset the Timer 0 overflow flag
 }
}
```

This type of problem is very simple to write using the PaulOS RTOS. Just one task would be needed to implement this LED flickering action:



Download free eBooks at bookboon.com
Click on the ad to read more

```
void Task_LED (void) {

    while(1){                  //infinite loop

          LED1 = 1;                          //pins high, LED is off, or use LED1 = led_off;
          OS_WAITT_A(0,0, 250);              // 250 millisecond delay
          LED1 = 0;                          // pin low, LEDs are on, or use LED1 = led_on;
          OS_WAITT_A(0,0, 250);              // 250 millisecond delay

          }

}
```

11.2.1 Seven-Segment LED Displays

Another simple output indicator which we can use is the familiar 7-segment LED display. There are basically two types of such displays, either the so-called Common Cathode (all the cathodes or negative connections are connected together to one common ground [GND] terminal) or the Common Anode type where all the anodes (or positive connections) are connected to one common supply [Vcc] terminal as shown in Figure 11-3. Apart from the 7 segments (a-g) forming the digit, some displays have an optional 8th segment which we could use to represent a decimal point (dp).

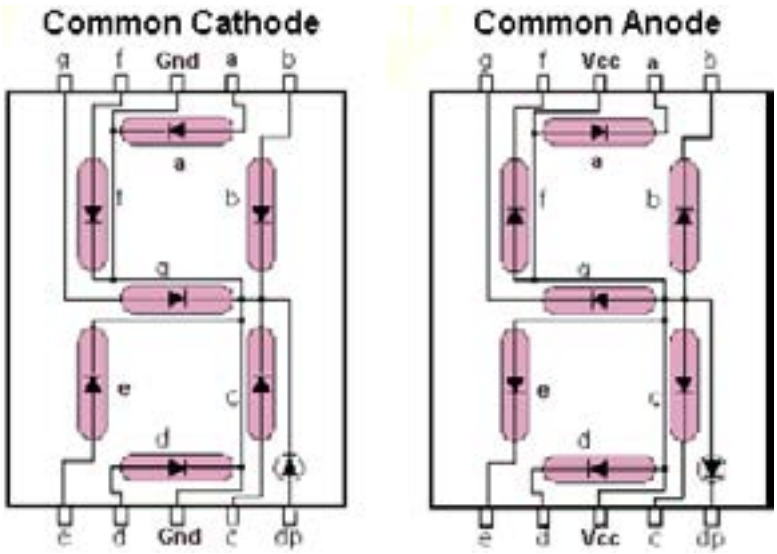


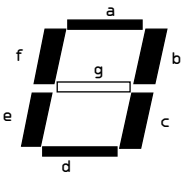
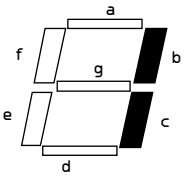
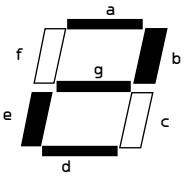
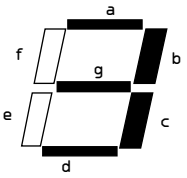
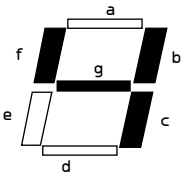
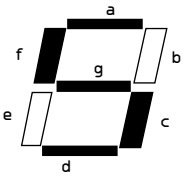
Figure 11-3 7-segmnet LED displays

In order to switch on the required decimal digit, we can connect the 7 or 8 segment diodes to the 8-bit port of the 8051 as we have already seen in the example with just one LED in **Figure 11-2**.

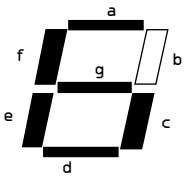
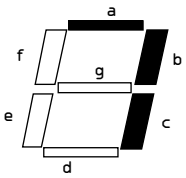
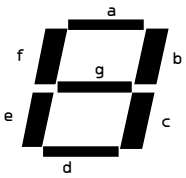
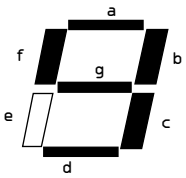
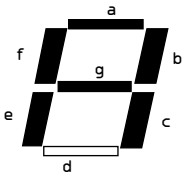
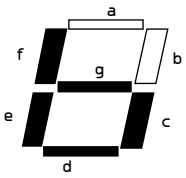
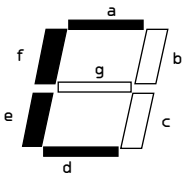
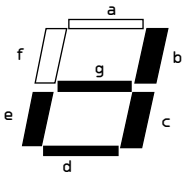
Download free eBooks at bookboon.com

The software would be written in such a way so as to switch on the required LEDs to display our decimal number. Thus to display the number 3, we would need to light up segments a, b, c, d and g and switch off the other segments. We should remember that with this direct drive method, the port must keep on presenting the same data to the 7-segment display, otherwise the display would change.

The following Table 11-1 shows how we can display the various digits. The 2nd and 3rd column in this Table shows the output byte for the port, depending on the way the segments are connected to the port..

Digit	gfedcba 6543210	abcdefg 6543210	a	b	c	d	e	f	g	
0	0×3F	0×7E	**on**	**on**	**on**	**on**	**on**	**on**	off	
1	0×06	0×30	off	**on**	**on**	off	off	off	off	
2	0×5B	0×6D	**on**	**on**	off	**on**	**on**	off	**on**	
3	0×4F	0×79	**on**	**on**	**on**	**on**	off	off	**on**	
4	0×66	0×33	off	**on**	**on**	off	off	**on**	**on**	
5	0×6D	0×5B	**on**	off	**on**	**on**	off	**on**	**on**	

Download free eBooks at bookboon.com

Digit	gfedcba 6543210	abcdefg 6543210	a	b	c	d	e	f	g	
6	0×7D	0×5F	**on**	off	**on**	**on**	**on**	**on**	**on**	
7	0×07	0×70	**on**	**on**	**on**	off	off	off	off	
8	0×7F	0×7F	**on**	**on**	**on**	**on**	**on**	**on**	**on**	
9	0×6F	0×7B	**on**	**on**	**on**	**on**	off	**on**	**on**	
A	0×77	0×77	**on**	**on**	**on**	off	**on**	**on**	**on**	
b	0×7C	0×1F	off	off	**on**	**on**	**on**	**on**	**on**	
C	0×39	0×4E	**on**	off	off	**on**	**on**	**on**	off	
d	0×5E	0×3D	off	**on**	**on**	**on**	**on**	off	**on**	

Download free eBooks at bookboon.com

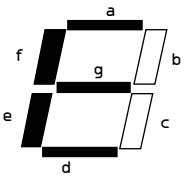
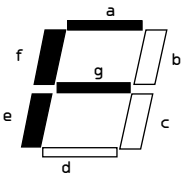
Digit	gfedcba 6543210	abcdefg 6543210	a	b	c	d	e	f	g	
E	0×79	0×4F	**on**	off	off	**on**	**on**	**on**	**on**	
F	0×71	0×47	**on**	off	off	off	**on**	**on**	**on**	

Table 11-1 LED 7 segment connections

We can also multiplex more than one 7-segment display by using a circuit as shown in Figure 11-4. One port supplies the data to all the displays, whilst the transistors T1–T4 switch on one display at a time as programmed by port 2. The first digit display would be left on for a few milliseconds and then switched off. The data is then changed to reflect the second digit display which is then switched on also for a few milliseconds. All the digits would be similarly switched on and off and this strobing action is repeated indefinitely so as to the viewer all the displays would appear to be lighted up continuously. A sample code program is listed to describe the program flow. We could also write the program using an RTOS where a OS_WAITT_A() command would be used to replace the delay function, thus the processor can be doing something else while waiting and driving the display.

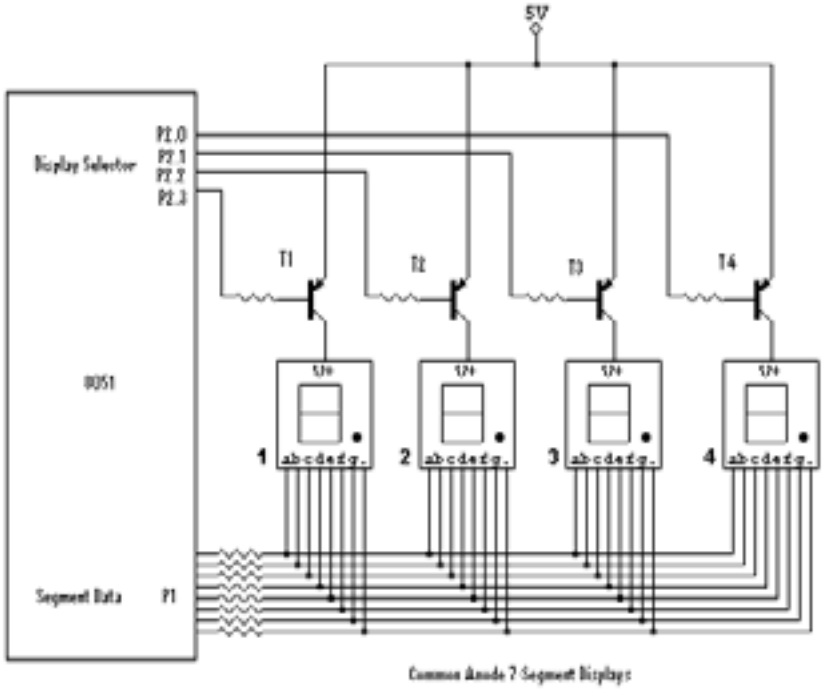


Figure 11-4 Multiplexing displays

Download free eBooks at bookboon.com

```
sbit digit0 = P2^0;
sbit digit1 = P2^1;
sbit digit2 = P2^2;
sbit digit3 = P2^3;

// Assuming segment a is connected to bit P1.6, segment b to bit P1.5 etc, then from Table 11-1
// we can select the segments to light up for each decimal digit 0–9 by sending the correct
// segment data from the array segment[].
// The digit can be selected by outputting a 1 on ONE pin from P2.0 to P2.3
unsigned char segments[10] = {0x7E, 0x30, 0x6D, 0x79, 0x33, 0x5B, 0x5F,
                                        0x70, 0x7F, 0x7B};

P2 &= 0xF0;             // switch off all digits
while(1)                // keep on looping
{
P1 = segments[0];       // send data to reflect the segments which need to be lighted up
 // in this case the number shown would be 0
digit0 = 1;             // switch on digit 0
delay();                // wait for some time, calling the delay function
digit0 = 0;             // switch off digit 0
// Now repeat for the second 7 segment LED digit
P1 = segments[1];       // send data to reflect the segments which need to be lighted up
 // in this case the number shown would be 1
Digit1 = 1;             // switch on digit 1
delay();                // wait for some time, calling the delay function
digit1 = 0;             // switch off digit 1
// and so on for the other digits.
……
……
}
```


Download free eBooks at bookboon.com

To make programming easier and at the same time provide a data latching (memory) capability, avoiding the need to keep on strobing the data, various 7-segment driver ICs were developed, the 4511 being one of them. These generally have 4 data input pins (D_1 to D_4) to represent the digit number which we want to display, D_1 being the least significant bit. Some are decimal drivers, accepting a 4-bit BCD (binary coded decimal number 0–9). Numbers greater than 9 (10–15) would show as blank. There are also Hex drivers which can display the normal 0–9 decimal digits and also a, b, c, d, e and f with the limitations of the 7-segment display. Thus A, C (not all drivers), E and F are shown as capital letters, whereas b, d (and sometimes c) are shown as small letters. The latching (latch enable or LE pin) mechanism ensures that once the data is latched on the IC (by putting LE low for a few micro-seconds, done in software by setting the port pin which is connected to this LE terminal from high to low and then back to high), then there is no need to keep the data at the 4511 input pins; the display would remain showing the latched digit data until some new data is latched to that same LED driver.

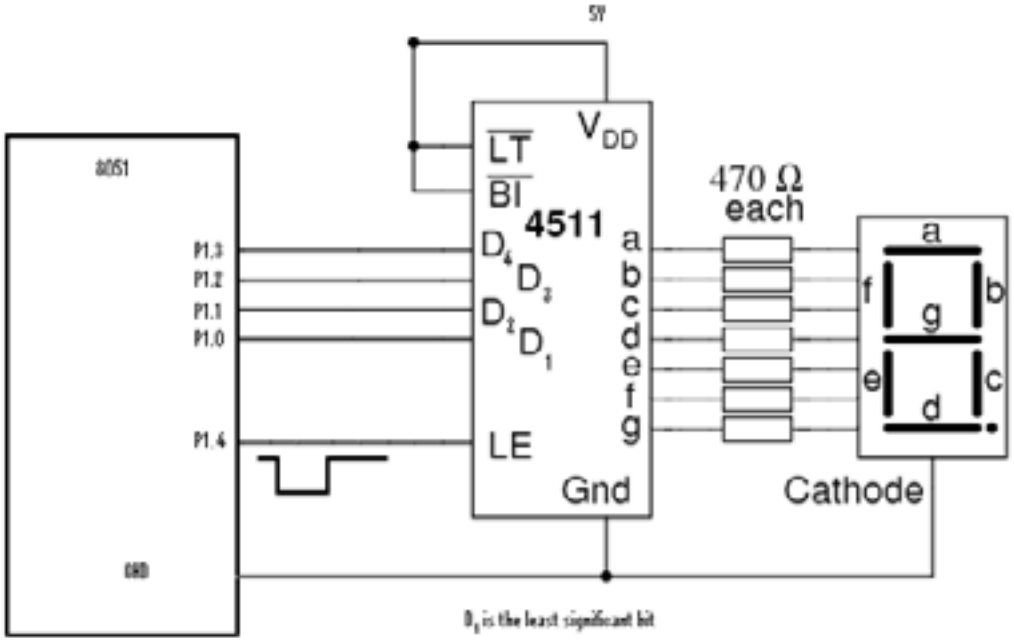


Figure 11-5 LED BCD driver

If we have the port P1 connections as shown in Figure 11-5 then we can display the number 7 with the following simple C code:

```
sbit LE = P1^4;
LE = 1;                 // ensure latch is High
P1 &= 0xF0;             // clear lower 4 data bits
P1 |= 0x07;             // set the correct data bits (in this case 7)
LE = 0;                 // toggle the Latch Enable bit
LE = 1;
```


Download free eBooks at bookboon.com

Two other control pins are usually available. The LT (lamp test) pin is usually used just to check that all the segments are working, and when set to low, the number 8 is displayed, irrespective of the D_1–D_4 input conditions. The BL (blanking input) pin is used to blank the display and is usually used to blank the leading zeroes in a multi-digit display. If not required, these two control signals are usually connected directly to the positive supply as shown in Figure 11-5.

We can also use the latch enable pin to multiplex more than one digit display to the same port. By latching sequentially different 7-segment digits, we can easily have a 6-digit display to use as a clock to display HH:MM:SS (the colon [:] can be obtained by using 4 separate LEDs, permanently on). Figure 11-6 shows how we can connect two 7 segment displays using the 4511 BCD-to-7-segment driver, which we can easily extend to more digits as required. The BCD data coming out of pins P1.0 to P1.3 is common to all the digits but the display is selected by pulsing the correct LE pin, using P1.4 or P1.5

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

Click on the ad to read more
Click on the ad to read more

Download free eBooks at bookboon.com

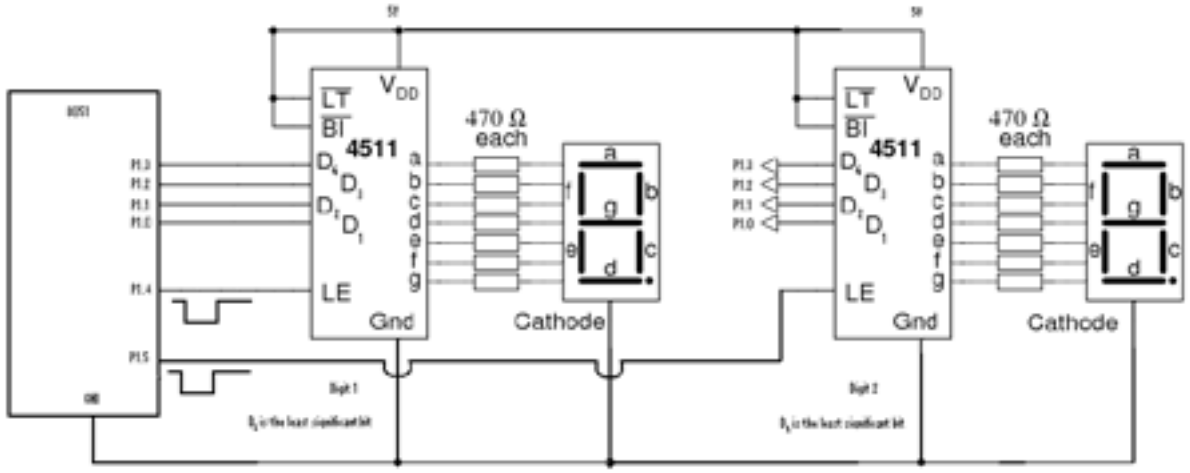


Figure 11-6 Multiplexing 4511s

For the circuit shown above in Figure 11-6 we can easily write a function which will handle everything:

```
        sbit LE1 = P1^4;
        sbit LE2 = P1^4;
Void Display(unsigned char Digit, unsigned char BCD_Data) {
        P1 &= 0xF0;             // clear lower 4 bits
        P1 |= BCD_Data;         // place data on output lines
        if (Digit == 1) {
                        LE1 = 1;        // latch data to digit 1
                        LE1 = 0;
`                       LE1 = 1;
                        }
        if (Digit == 2) {
                        LE2 = 1;        // latch data to digit 2
                        LE2 = 0;
`                       LE2 = 1;
                        }
}
```


Download free eBooks at bookboon.com

11.3 Input Switches

The simplest input is the switch, as shown in Figure 11-7. Here we can easily see that whenever the switch is open, the microcontroller port pin would be effectively connected to the 5V supply through the 10k ohm resistor. The microcontroller would read a high logic level or a 1. Closing the switch would ground the pin and the microcontroller would read a zero logic level. The port pin would be programmed for the input mode by initially writing a 1 to that pin.

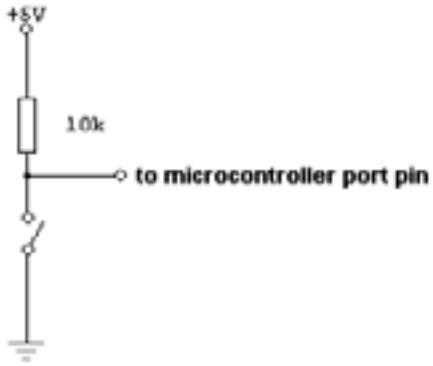


Figure 11-7 Switch (normally open, high on port pin)

On the other hand, in Figure 11-8 we can easily see that whenever the switch is open (normal position), the microcontroller port pin would be effectively connected to the ground through the 10k ohm resistor. The microcontroller would read a low logic level or a 0. Closing the switch would connect the pin to the 5V rail and the microcontroller would read a high logic level or a 1.

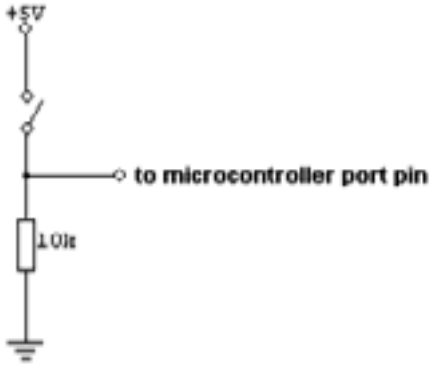


Figure 11-8 Switch (normally open, low on port pin)

11.3.1 Switch Bounce

When a physical switch is closed the contacts bounce opened and closed rapidly for about 20 to 30 ms, as illustrated below in Figure 11-9. The opening of a switch is normally clean and without bounce.

Download free eBooks at bookboon.com

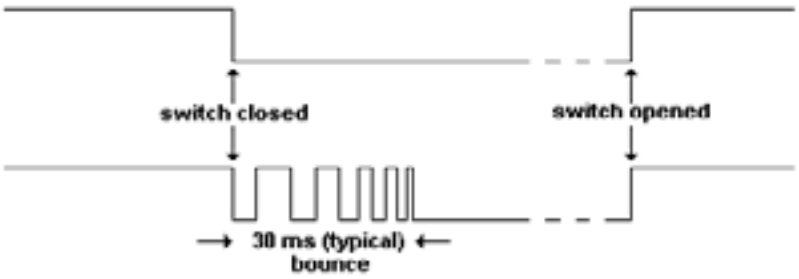


Figure 11-9 Switch bounce

While switch-close bounce is a very short time in human terms it is a very long time for a micro-controller (the basic 8051 running on a system clock of 12 MHz executes a 1-byte instruction in 1 μs). Without switch de-bouncing, the microcontroller would 'think' the switch was opened and closed repeatedly. Imagine if a push-button switch was being used to increment the output to a digital to analogue convertor. The software routine to poll the push button switch (expecting an off-on-off action on the push switch, returning a one when pressed, otherwise wait) would normally be:

- Wait while the switch is off
- Wait while the switch is on
- Switch can now be taken as pressed (off-on-off) and return a '1'

Click on the ad to read more

Download free eBooks at bookboon.com

The main program would then normally stay in the endless:

{

Call switch polling routine (outlined above)

Increment voltage routine

}

If the switch was connected to the microcontroller without any switch de-bouncing mechanism, then a user pressing the switch once would actually result in the DAC output voltage being increased many times because the microcontroller would respond as if the switch had been pressed many times.

De-bouncing mechanisms can be implemented:

1. By means of a software delay of around 30ms between two successive readings of the switch (to let the bouncing die down), whilst it is being polled. If the switch readings agree, then the switch is really on.
   - Wait while the switch is off
   - Wait 30ms
   - Exit if switch is off (return a '0'), else
   - Wait while the switch is on
   - Switch can now be taken as pressed (off-on-off) and return a '1'

2. Another software technique is to connect the switch to an interrupt pin instead of polling it routinely. It would be easier if a normally-high output from the switch is used and connected to the external interrupt, negative-edge triggered mode. As soon as the switch is pressed, we would have a high-to-low transition which would trigger an external interrupt. The ISR is called where we would immediately disable the external interrupt (otherwise we would have lots of them due to bouncing), wait for 30ms and then read the switch again. If we still read an ON condition, then we have detected a valid switch-on event and proceed accordingly. We can then enable the interrupts again and exit the ISR once finished with the response required. If the second reading shows an OFF condition, then we can take it as a glitch (or still bouncing) and that no switch has been pressed and once again we enable the interrupts again and exit the ISR without taking further action. If the bouncing is still going on, we would detect another interrupt and automatically repeat the ISR again.
3. By hardware, usually using a one shot device, which means that as soon as the switches flickers to the on position, the output of the one-shot will remain steadily on and bouncing is thus eliminated.

Download free eBooks at bookboon.com

11.4 Keypad

Multiple switches (or keypads and keyboards) are normaly connected in the form of a matrix where the vertical lines (columns) and horizontal lines (rows) are connected to the controller ports (either directly or via pull-up resistors) as shown in Figure 11-10. The port connections can be programmed to act as either input or output lines as required in order to be able to decide which key, if any, has been pressed.

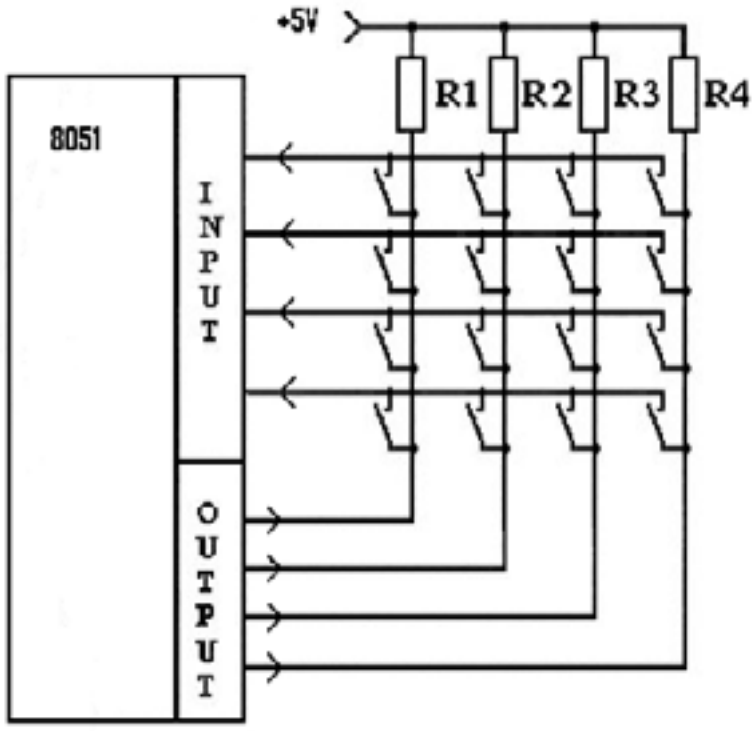


Figure 11-10 Keypad switch matrix

The method to detect a key press is as follows:

1. We set all output columns bits to 0.
2. The input row pins are then read.
3. If any row pin is a zero, then we know that a key in that row is being pressed, although we cannot tell yet which one of the four it is. If the input is not zero, we just have to wait and keep on reading the input port, waiting for a key press (going back to step 2).
4. If in the input row reading we do indeed detect a zero, then usually a bouncing delay is initiated so as to eliminate any bouncing or erroneous key contact (unless the bouncing is being taken care of by other hardware devices).
5. We read once again the input after this delay, and if the same row is giving a zero then we can start the process to determine exactly which column switch in that row is being pressed (the correct row is now known). If we do not detect a zero in arrow, then we take it that it was a glitch and go back to step 2, waiting for a key press.

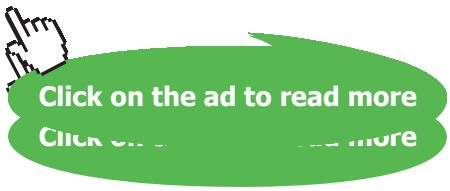
Download free eBooks at bookboon.com

6. We can determine which key is being pressed by setting the input to zero for one column at a time and reading the row state until we read a zero. When the correct column is determined, then we have effectively decoded the key press, since we had already determined the row in step 5.

11.4.1 Keypad: interrupts vs polling

Instead of using this algorithm, where we are effectively waiting (whilst reading the port input) for any key press, we can modify the circuit to that shown in Figure 11-11. Note that in this figure, the rows are the output bits (P1.0 to P1.3) of the port, while the higher nibble of the port (P1.4 to P1.7) act as the input to read the column values.

All the rows are first set to zero and the external INT0 interrupt is enabled. The column input signals are ANDed together to provide an external INT0 interrupt low logic signal whenever any column goes low (negative edge triggered, activated when the signal goes from high to low). The INT0 interrupt service routine (ISR) would then be activated so that we can determine which key is being pressed.



Download free eBooks at bookboon.com

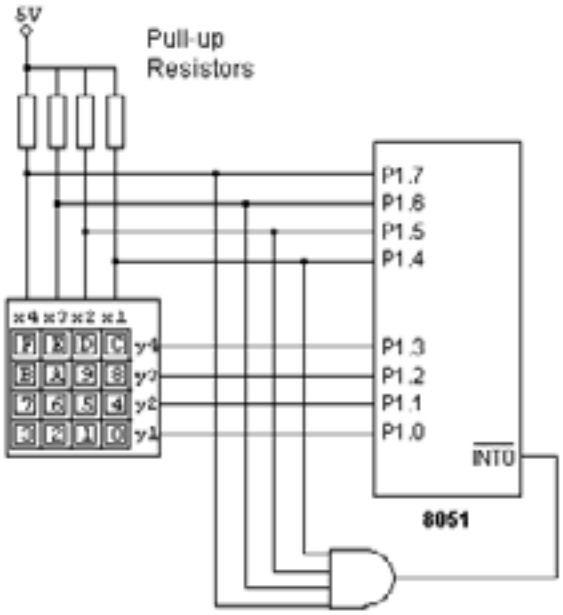


Figure 11-11 Interrupt keypad interface

We set all output row bits to 0, and enable the external negative-edge INT0 interrupt in the main program. We cannot obviously keep on looping and waiting within the ISR itself so the algorithm is modified as described below.

The ISR, activated whenever there is a key press would then perform the following:

1. The external interrupt is disabled. This is especially important in this case, since the bouncing effect of a switch would otherwise cause repeated interrupts.
2. A de-bounce delay (typically 30 ms) is initiated so as to wait for any bouncing or erroneous key contact to die down.
3. The input column pins are then read.
4. If any column pin is a zero, then we know that a key in that column is being pressed, although we cannot tell yet which one of the four it is. If we do not detect a zero on any input line, then the interrupt was probably caused by some glitch or intermittent key contact and we jump immediately to step 7 to exit the ISR.
5. If in the input column reading we do indeed detect a zero, then we can start the process to determine exactly which row switch in that column is being pressed (the correct column is now known from the input data pattern).
6. We can determine which actual key is being pressed by setting the input to zero for one row at a time and reading the column state until we read a zero. When the correct row is determined, then we have effectively decoded the key press, since we had already determined the column in step 5.
7. Enable once again the external INT0 interrupt, and exit the ISR.

Download free eBooks at bookboon.com

10.5 LCD Display

A Liquid Crystal Display (LCD) provides a versatile output screen where normal text and graphics can be displayed, thus providing more versatility than the simple LED devices mentioned above. LCD displays come in many different versions, but here we shall deal with the cheap and simple 2 or 4 line display, providing 16 or 20 characters per line capability. It can be programmed to run either in the 8-bit data or in the 4-bit data mode if we do not have the luxury of using an 8-bit port dedicated to supply just the data bits to the LCD.

Figure 11-12 shows how we can connect a standard LCD (such as the Hitachi HD44780) to an 8051 microcontroller. Apart from the ground, supply, back lighting and contrast pins, we would need 8 data bits (D0–D7) in 8-bit mode or just 4 data bits (D4–D7) in the 4-bit mode so that we can communicate with the LCD. There are also 3 additional control signals RS, R/W and E (or EN) which we need to connect to the 8051 to provide the required hand-shaking control signals.

- RS is the register select signal, so that the LCD would know whether we are sending data to be displayed or sending a command intended to give some instructions to the LCD.
- R/W, as the name implies is the Read or Write signal which determines the direction of the data flow (reading from the LCD or writing to the LCD).
- E (or EN) is the enable pin, which has to be toggled so that any data is latched on to the device.

This e-book
is made with
SetaPDF
SETASIGN
PDF components for PHP developers
www.setasign.com
Click on the ad to read more
Download free eBooks at bookboon.com

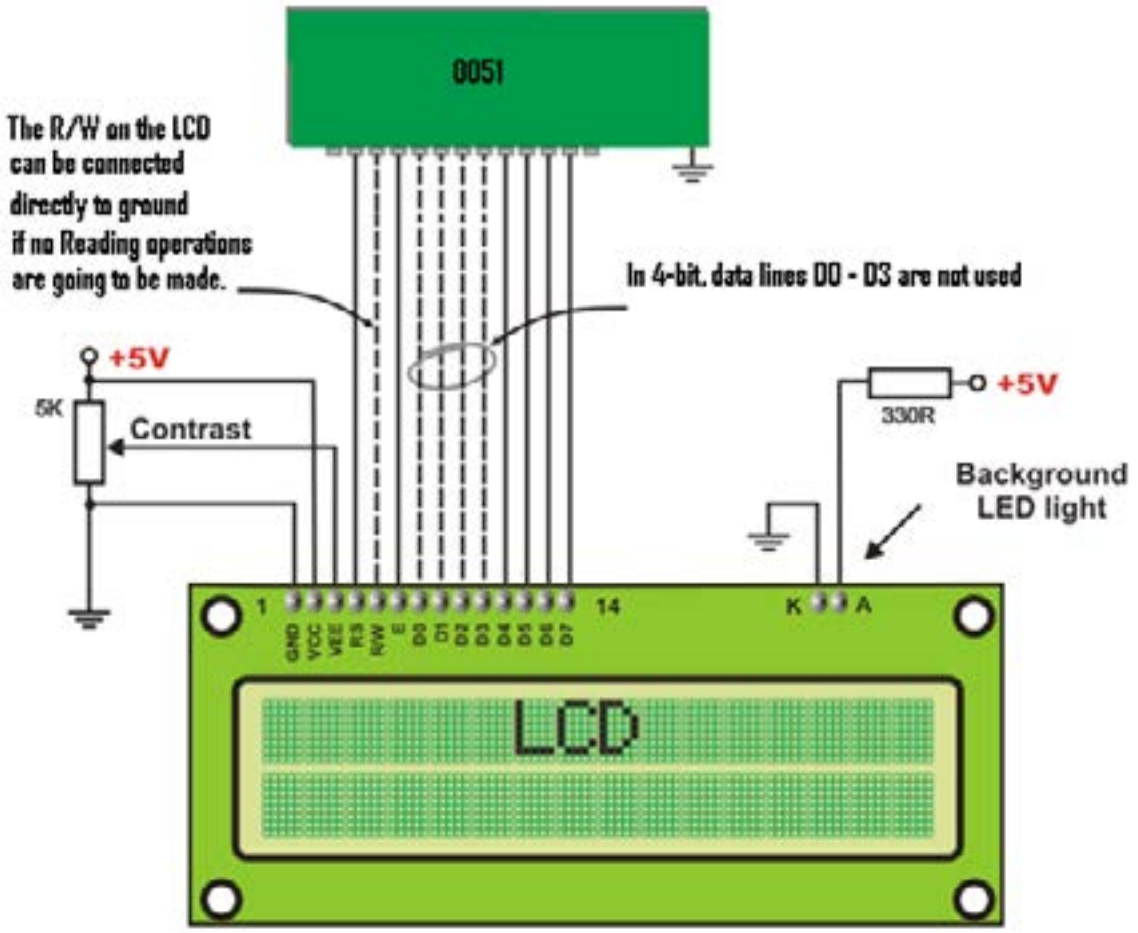


Figure 11-12 Standard LCD connections

The Read capability is mainly used to read the status of the LCD so that we can make sure that the LCD is ready to receive the next data or instruction. This is because the LCD takes some time to perform the required instructions, and not all instructions take the same amount of time to be executed. Hence the need to read the status of the LCD and wait for the LCD ready signal before proceeding. In many applications, we may only be required to write to the LCD, without the need to read anything. In this case we may simply initiate a fixed delay between issuing commands or data transfers, so as to be sure that the LCD has finished from the previous command, without the need to check the LCD status. Since the Write pin is active low, we can connect this pin permanently to ground in such cases. So, if we are only writing to the LCD and if we are using the 4-bit mode, we would then need only 6 bits (4-bits data, the EN and RS control signals) to communicate with the LCD. The LCD ground line naturally has to be common with the 8051 ground.

Download free eBooks at bookboon.com

11.5.1 Programming the HD44780

In order to write a command or data, the following sequence of commands needs to be made, depending on the mode of operation of the LCD:

8-Bit Write Sequence
Make Sure "EN" is 0 or low
Set "R/S" to 0 for a command, or 1 for data/characters
Put the data/command on D7-0
Set "EN" (EN= 1 or High)
Wait At Least 450 ns!!!
Clear "EN" (EN= 0 or Low)
Wait 5ms for command writes, and 200us for data writes.

Table 11-2 LCD 8-bit write sequence

4-Bit Write Sequence
Make Sure "EN" is 0 or low
Set "R/S" to 0 for a command, or 1 for data/characters
Put the HIGH BYTE of the data/command on D7-4
Set "EN" (EN= 1 or High)
Wait At Least 450 ns!!!
Clear "EN" (EN= 0 or Low)
Wait 5ms for command writes, and 200us for data writes.
Put the LOW BYTE of the data/command on D7-4
Wait At Least 450 ns!!!
Clear "EN" (EN= 0 or Low)
Wait 5ms for command writes, and 200us for data writes.

Table 11-3 LCD 4-bit write sequence

11.6 LCD Command Set

There are certain instructions or commands which we need to get familiar with in order to be able to program or setup the LCD display. The R/S and R/W control lines are also used depending on the type of the command required. Dedicated functions can be written which can take care of the initialisation and LCD mode setup as explain in the following sub-sections.

Download free eBooks at bookboon.com

R/S	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Instruction/Description
0	0	0	0	0	0	0	0	0	1	Clear Display and Home the Cursor
0	0	0	0	0	0	0	0	1	*	Return Cursor and LCD to Home Position
0	0	0	0	0	0	0	1	ID	S	Set Cursor Move Direction
0	0	0	0	0	0	1	D	C	B	Enable Display/Cursor
0	0	0	0	0	1	SC	RL	*	*	Move Cursor/Shift Display
0	0	0	0	1	DL	N	F	*	*	Set Interface Length
0	0	0	1	A	A	A	A	A	A	Move Cursor into CGRAM
0	0	1	A	A	A	A	A	A	A	Move Cursor to Display
0	1	BF	*	*	*	*	*	*	*	Poll the "Busy Flag"
1	0	D	D	D	D	D	D	D	D	Write a Character to the Display at the Current Cursor Position
1	1	D	D	D	D	D	D	D	D	Read the Character on the Display at the Current Cursor Position

Table 11-4 LCD Command set

Download free eBooks at bookboon.com
Click on the ad to read more

The bit abbreviations used in Table 11-4 for the different commands are explained in the following list:

"*" - Not Used/Ignored. This bit can be either "1" or "0"

Set Cursor Move Direction:
ID - Increment the Cursor After Each Byte Written to Display if set
S - Shift Display when Byte Written to Display if set

Enable Display/Cursor
D - Turn Display On(1)/Off(0)
C - Turn Cursor On(1)/Off(0)
B - Cursor Blink On(1)/Off(0)

Move Cursor/Shift Display
SC - Display Shift On(1)/Off(0)
RL - Direction of Shift Right(1)/Left(0)

Set Interface Length
DL - Set Data Interface Length 8(1)/4(0)
N - Number of Display Lines 1(0)/2(1)
F - Character Font 5x10(1)/5x7(0)

Poll the "Busy Flag"
BF - This bit is set while the LCD is processing

Move Cursor to CGRAM/Display
A - Address

Read/Write ASCII to the Display
D - Data

We now provide some basic initialisation code for the 8-bit and for the 4-bit connection so that we can interface and communicate with this LCD.

	General Initialisation	Example Initialisation
1	Wait 20ms for LCD to power up	
2	Write D7-0 = 30 hex, with RS = 0	
3	Wait 5ms	
4	Write D7-0 = 30 hex, with RS = 0, again	
5	Wait 200us	
6	Write D7-0 = 30 hex, with RS = 0, one more time	
7	Wait 200us	
8	Write Command "Set Interface"	Write 38 hex (8-Bits, 2-lines)
9	Write Command "Enable Display/Cursor"	Write 08 hex (don't shift display, hide cursor)
10	Write Command "Clear and Home"	Write 01 hex (clear and home display)
11	Write Command "Set Cursor Move Direction"	Write 06 hex (move cursor right)
12	--	Write 0C hex (turn on display)
Display is ready to accept data.		

Table 11-5 LCD 8-bit mode initialisation

Download free eBooks at bookboon.com

11.6.1 The 8-bit mode LCD initialisation sample program

From the above tables, we can write some basic initialisation program for the LCD, staring with the 8-bit mode of operation. In this program we are making certain assumptions regarding the port pin connections to the LCD lines as can be seen from the initial remarks found in the code listing.

```
/* Assume that LCD-RS is connected to bit 0 of Port 2 (or LCD_CTRL_PORT)*/
/*                 0 = Command, 1 = Data                */
/* Assume that LCD-RW is connected to bit 1 of Port 2 (or LCD_CTRL_PORT)          */
/*                 0 = Write, 1 = Read                  */
/* Assume that LCD-EN is connected to bit 2 of Port 2 (or LCD_CTRL_PORT)          */
/*      A high (1) to low (0) transition is needed to latch data/command */
#define LCD_CTRL_PORT P2
sbit RSbit = LCD_CTRL_PORT^0;
sbit RWbit = LCD_CTRL_PORT^1;
sbit ENbit = LCD_CTRL_PORT^2;

/* If we only use the Control Port just for this purpose, we can send any one of the        */
/* following defined items to set all three control lines simultaneously                    */
                                        /* bit      2       1       0       */
#define ClearLines              0x00            /* EN = 0, RW = 0, RS = 0 */

#define LatchCommand1           0x04            /* EN = 1, RW = 0, RS = 0 */
#define LatchCommand2           0x00            /* EN = 0, RW = 0, RS = 0 */

#define LatchData1              0x05            /* EN = 1, RW = 0, RS = 1 */
#define LatchData2              0x01            /* EN = 0, RW = 0, RS = 1 */

#define ReadDataLines1          0x06            /* EN = 1, RW = 1, RS = 0 */
#define ReadDataLines2          0x02            /* EN = 0, RW = 1, RS = 0 */

/* Assume that the 8-bits data are connected to Port 1 (or LCD_DATA_PORT) */
#define LCD_DATA_PORT P1

/*************************************************************************/
void LCD_SOFT_WAIT (int x)
 {
 unsigned int i,j;
 for(j=1; j<=x; j++){
   for(i=0; i<=120; i++){}; /* JUST A DELAY */
   }
 }
```


Download free eBooks at bookboon.com

```
/*****************************************************************************/
void LCD_SHORT_WAIT (void)
 {
 unsigned char i;
 i++;
 i++;
 }

/*****************************************************************************/

/* This Wait If Busy routine can be used ONLY after the initialisation */
void LCD_WAIT_IF_BUSY()
 {
  unsigned char Status;
  LCD_DATA_PORT = 0xFF;                 /* set DATA port to input mode */
  do
   {
      RWbit = 1; RSbit = 0; ENbit = 1;      */ set reading mode */
                            /* or LCD_CTRL_PORT    = ReadDataLine2; */
      LCD_SHORT_WAIT();
      ENbit = 0;            /* or LCD_CTRL_PORT    = ReadDataLine1; */
      Status = LCD_DATA_PORT;
   } while ((Status & 0x80) == 0x80);

 ENbit = 1;
 LCD_DATA_PORT = 0x00;                  /* set DATA port to output mode */
}

/*****************************************************************************/
/*****************************************************************************/

void LCD_SEND_INIT(char ch) /* send display init to lcd */
 {
 LCD_DATA_PORT = ch;
 ENbit = 1; RWbit = 0; RSbit = 0;              // command sending mode
 LCD_SHORT_WAIT();
 ENbit = 0;
 LCD_SOFT_WAIT(20);                    /* wait for at least 5 milliseconds */
 ENbit = 1;
                    /* cannot check busy line yet, not until the initialisation has finished */
  }
```


Download free eBooks at bookboon.com

```
/*****************************************************************************/

void LCD_Send_Command(char ch)        /* write display command to lcd */
 {
 LCD_WAIT_IF_BUSY();
 LCD_DATA_PORT = ch;
 ENbit = 1; RWbit = 0; RSbit = 0;             // command sending mode
 LCD_SHORT_WAIT();
 ENbit = 0;
 }                  /* end lcd write */

/*****************************************************************************/

void LCD_Send_Data(char ch)                /* write display data to lcd */
 {
 LCD_WAIT_IF_BUSY();
 LCD_DATA_PORT = ch;
 ENbit = 1; RWbit = 0; RSbit = 1;             // data sending mode
 LCD_SHORT_WAIT();
 ENbit = 0;
 }                  /* end lcd write */

/*****************************************************************************/

/* 8-bit mode */
void LCD_INIT(void)                /* reset lcd display */
 {
 LCD_CTRL_PORT = LCD_DATA_PORT = 0;      /* set both 8251 ports as output */
 LCD_SOFT_WAIT(50);             /* wait a few milliseconds, after power on */
 ENbit = 0; RWbit = 0; RSbit = 0;             // clear control lines
 LCD_SEND_INIT(0x38);          /* get attention */
 LCD_SEND_INIT(0x38);          /* set mode to 8 bit DATA 2 lines, 5×7 dots */
 LCD_SEND_COMMAND(0x0C);          /* Display On, Cursor Off and Blinking off */
 LCD_SEND_COMMAND(0x01);          /* Clear Display */
 LCD_SEND_COMMAND(0x06);          /* Set Entry Mode */
 }                  /* end of lcd initialisation */
```

11.6.2 4-bit mode LCD Initialisation

We have to remember that in this 4-bit mode any Data/Command writes of one-byte size are handled using:

send high-nibble, delay, send low-nibble, delay

sequence, where 1 nibble is equivalent to 4 bits.

Download free eBooks at bookboon.com

	General Initialisation	Example Initialisation
1	Wait 20ms for LCD to power up	
2	Write D7-4 = 3 hex, with RS = 0	
3	Wait 5ms	
4	Write D7-4 = 3 hex, with RS = 0, again	
5	Wait 200us	
6	Write D7-4 = 3 hex, with RS = 0, one more time	
7	Wait 200us	
8	Write D7-4 = 2 hex, to enable four-bit mode	
9	Wait 5ms	
10	Write Command "Set Interface"	Write 28 hex (4-Bits, 2-lines)
11	Write Command "Enable Display/Cursor"	Write 08 hex (don't shift display, hide cursor)
12	Write Command "Clear and Home"	Write 01 hex (clear and home display)
13	Write Command "Set Cursor Move Direction"	Write 06 hex (move cursor right)
14	--	Write 0C hex (turn on display)
Display is ready to accept data.		

Table 11-6 LCD 4-bit mode initialisation

11.6.3 The 4-bit mode LCD initialisation sample program

Here we assume that the control signals are connected to the lower 3 bits (RS to bit 0, RW to bit 1 and EN to bit 2), while the 4 data lines (D4–D7) are connected to the upper four bits of the port. D4 to port bit 4, D5 to port bit 5 and so on.

```
#define LCD_PORT      P2
sbit RSbit = LCD_PORT^0;
sbit RWbit = LCD_PORT^1;
sbit ENbit = LCD_PORT^2;
#define         LCD_EN          0x04
#define         LCD_RW          0x02
#define         LCD_RS          0x01

// The 4 data lines (D4–D7) are connected to the upper four bits of the port.
// D4 to port bit 4, D5 to port bit 5 and so on.
```


Download free eBooks at bookboon.com

```
void LCD_Wait_If_Busy (void)              /* wait for lcd if busy */
 {
// The Busy Flag is the most significant bit of the received data
 char c,d;
 LCD_PORT = 0xF0;                         // set port upper nibble to input mode
 do {
 ENbit = 1;
 RWbit = 1;                               // prepare for a Write operation
 lcd_soft_wait(5);
 c = (LCD_PORT & 0xF0);                   /* read high data nibble */
 ENbit = 0;
 RWbit = 0;
 lcd_soft_wait(5);
 ENbit = 1;
 RWbit = 1;                               // prepare for a Write operation
 lcd_soft_wait(5);
 d = ( LCD_PORT & 0xF0);                  /* read low data nibble, in Port.4 – Port.7 bits */
 ENbit = 0;
 RWbit = 0;
 d = d>>4;                                 /* move it to the lower nibble
 c = c + d;                               /* combine nibbles to form 8-bit data */
 } while (c & 0x80);                      /* wait for Busy Flag (BF) line to go low */
 LCD_PORT = 0x00;                         // set all port pins to output mode again
 }                  /*end lcd busy wait */

void LCD_Send_Data(char ch)               /* write display data to lcd */
 {
 LCD_Wait_If_Busy();
 LCD_PORT = ((ch & 0xf0) | LCD_EN | LCD_RS);          /* send character high nibble */
 ENbit = 0;
 lcd_soft_wait(3);
 LCD_PORT = (((ch & 0x0f) << 4) | LCD_EN | LCD_RS);   /* send character high nibble */
 ENbit = 0;
 }                  /* end lcd data write */
void LCD_Send_Command(char ch)        /* write display command to lcd */
```


Download free eBooks at bookboon.com

```
    {
    LCD_Wait_If_Busy();
    LCD_PORT = ((ch & 0xf0) | LCD_EN);              /* send character high nibble */
    ENbit = 0;
    lcd_soft_wait(3);
    LCD_PORT = (((ch & 0x0f) << 4) | LCD_EN ); /* send character low nibble */
    ENbit = 0;
    }                   /* end lcd write command function */
  void LCD_Send_Init_Command(char ch)                   /* write display initialization commands to lcd
*/
  // Cannot use LCD_Wait_If_Busy routine yet.
    {
    LCD_PORT = ((ch & 0xf0) | LCD_EN);              /* send character high nibble */
    ENbit = 0;
    lcd_soft_wait(5);
    LCD_PORT = (((ch & 0x0f) << 4) | LCD_EN );              /* send character low nibble */
    ENbit = 0;
    lcd_soft_wait(5);
    }                   /* end lcd write command function */

  void LCD_Init_4(void)             /* reset lcd display */
    {
    lcd_soft_wait(10);                  /* wait at least 15ms after power on*/
    LCD_Send_Init_Command (0x33);       /* get attention */
    lcd_soft_wait(5);                   /* wait */
    LCD_Send_Init_Command (0x32);       /* get attention */
    lcd_soft_wait(10);                  /* wait */
    LCD_Send_Init_Command (0x20);       /* 4 bit DATA transfer from now on */
    lcd_soft_wait(5);
  // LCD_Send_Init_Command (0x28);      /* 4 bit data, 2 lines, 5×7 dots */
    LCD_Send_Init_Command (0x2C);       /* 4 bit data, 2 lines, 5×10 dots */
    LCD_Send_Command (0x06);            /* Move Cursor to the right. Do not shift display */
    LCD_Send_Command (0x0C);            /* Display On, Cursor and Blinking off */
  // LCD_Send_Command (0x08);            /* Display, Cursor and Blinking Off */
  // LCD_Send_Command (0x0F);            /* Display, Cursor and Blinking on */
    LCD_Send_Command (0x01);            /* Clear Display */
    }                   /* end lcd initialize */
```

In any mode, in order to write a text string to the LCD, instead of writing a letter at a time we can write a routine. In the smaple program below we are assuming that the length of the text fits into the LCD display.

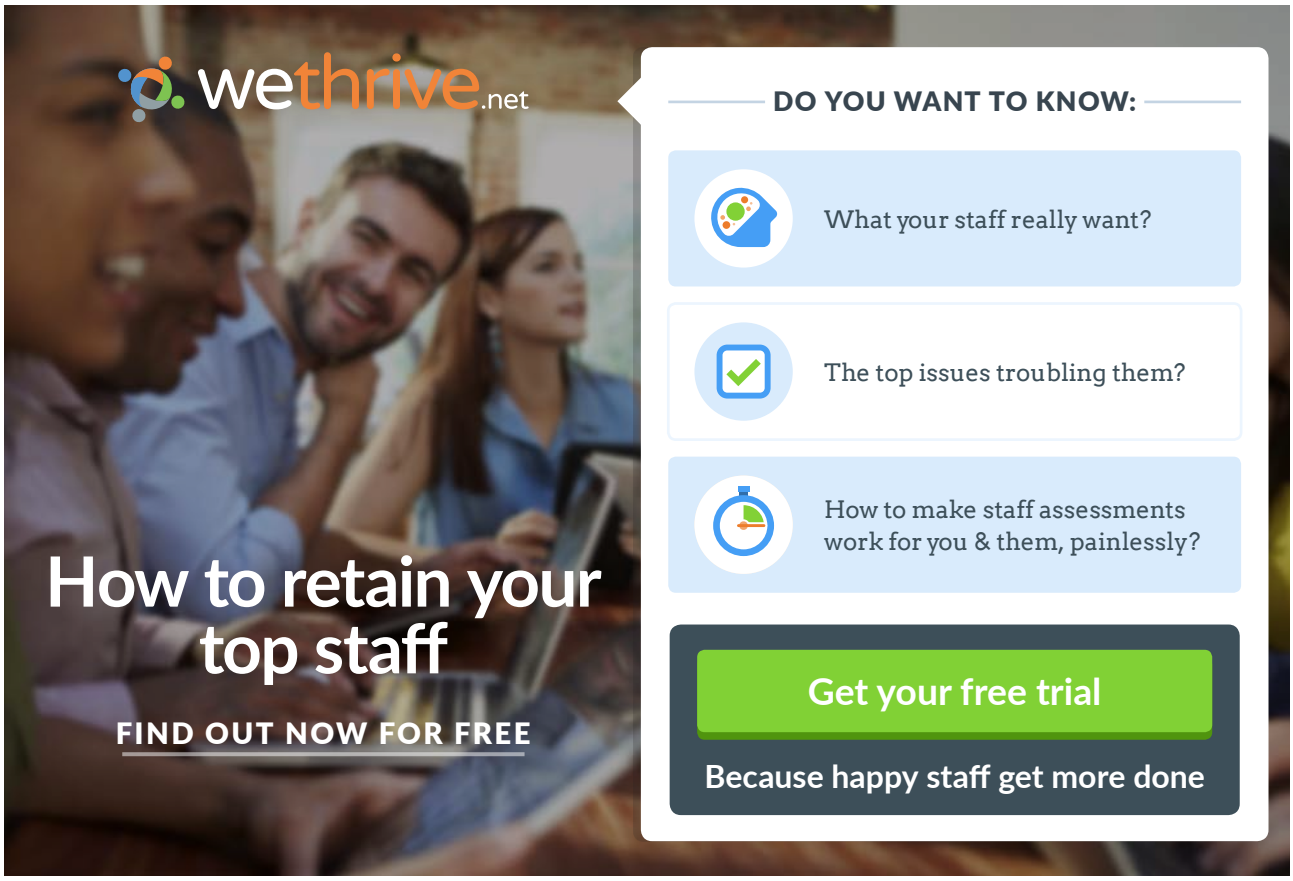
Download free eBooks at bookboon.com

```
void LCD_Write_String (char *s)
{
  while (*s) { /* Write all characters within string, checking for the end of string char /0 */
        LCD_Send_Data(*s++); /* Send character to LCD display */
     }
}
```

Similarly we can then write various other routines so that we can centre our text, write at any row or column position, display a moving text and so on.

11.7 DC Motor

A simple DC motor can be connected to the 8051 as shown in Figure 11-13. Since the motor takes some appreciable amount of current, especially when switching on, we cannot drive it directly through the port. We normally use a transistor such as the BD139 (or mechanical relay) to switch it on and off, as shown in Figure 11-13. The type of the transistor used depends on the motor specification, mainly the current that it takes. Since this current would all be passing through the transistor, Q1 must be able to handle the power without overheating. A heat-sink is also used in most case to keep the temperature of the transistor within limits. The diode across the motor is needed in order to provide a path for the back emf generated by the motor itself.



Click on the ad to read more

Download free eBooks at bookboon.com

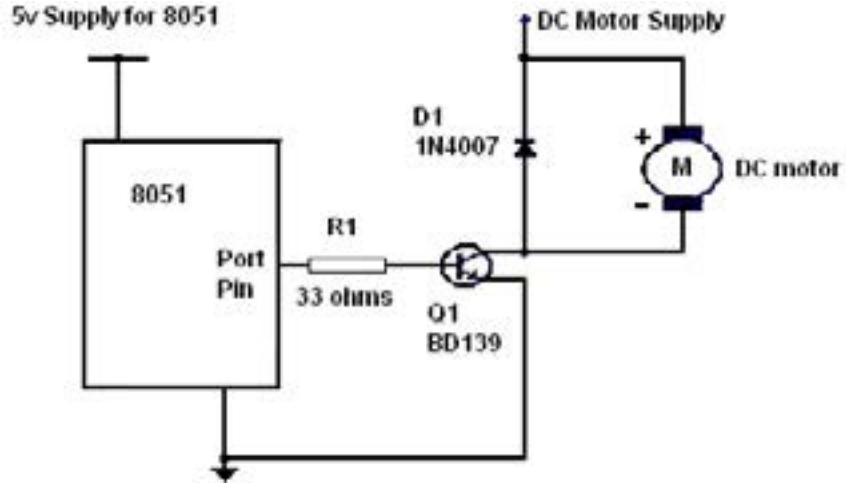


Figure 11-13 DC Motor interfacing

The supply for the dc motor is normally a separate supply which can handle the power requirements of the motor and moreover reduces the glitches on the 8051 supply rail.

Apart from just switching it ON (running at maximum speed) when we need the motor and then switching OFF when we are done with it, we can also make it run at variable speeds by switching it ON and OFF with a pulse train (or Pulse Width Modulation [PWM] signal), varying the ON pulse width relative to the OFF time. The inertia of the motor armature and whatever it is driving, will keep the motor turning even during the OFF cycle. The greater the ON time, the faster it goes, since the average voltage of the signal would be higher, as shown in Figure 11-14.

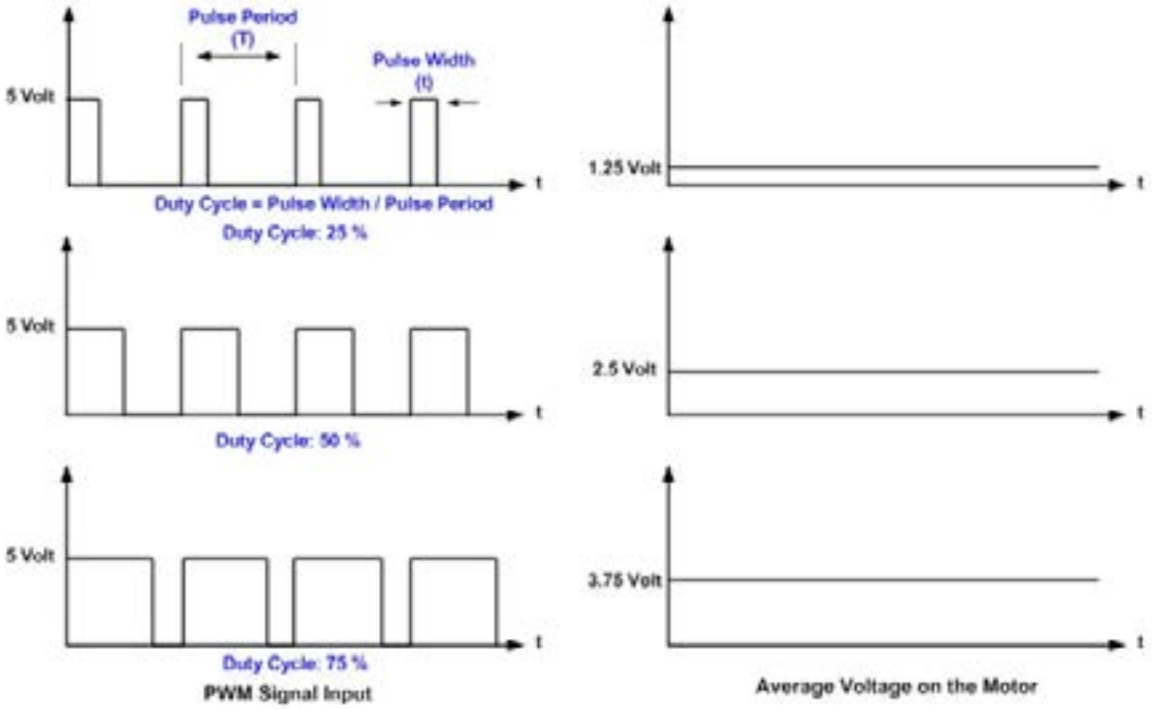


Figure 11-14 PWM used to control DC motor speed

Download free eBooks at bookboon.com

Having a very low mark (1 or ON) to space (0 or OFF) ratio could result in the motor not turning at all. It depends a lot on the type of motor and how free is the armature to rotate. So we can expect that the mark-to-space ratio would need to be above 30% for the motor to start turning and overcome friction etc.

An example which can be adapted to this setup is given in section 11.8 when discussing the H-bridge connection. The principle of using PWM to adjust and control the motor speed is still the same.

11.8 DC motor using H-Bridge

If we add an H-bridge to our circuit, we can now also change the direction of rotation of the motor, apart from controlling its speed. The H-bridge operation can be best explained with reference to the following figures which describe the operation of the dc motor. The switches shown would actually be transistor switches and they could be switched ON and OFF by means of signals coming out of the 8051 port.

Figure 11-15 shows the motor in the OFF position, where all the switches are off.

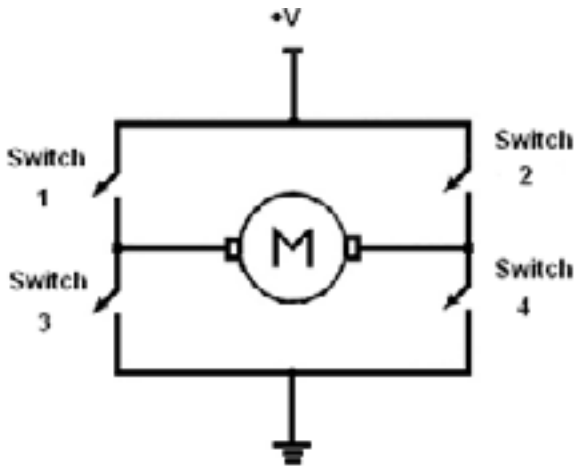


Figure 11-15 Motor Off

If now switches 1 and 4 are switched ON, leaving the others off, the motor would turn at full speed in one direction say clockwise, as shown in Figure 11-16.

Download free eBooks at bookboon.com

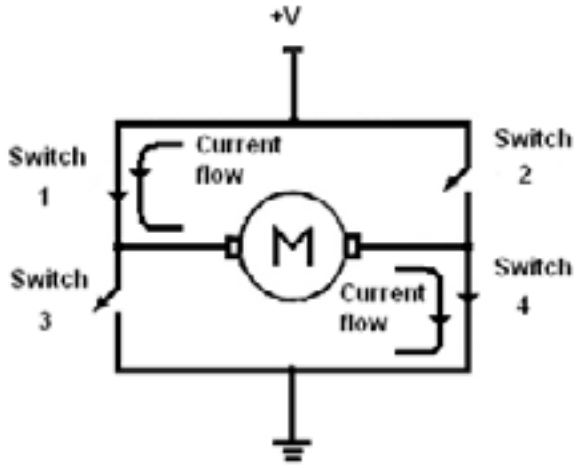


Figure 11-16 Motor Clockwise Rotation

On the other hand, if we switch ON 2 and 3, and leaving switches 1 and 4 OFF as shown in Figure 11-17 the motor would turn at full speed in the opposite direction, since the supply would now be inverted with respect to the motor terminals.

Download free eBooks at bookboon.com
Click on the ad to read more

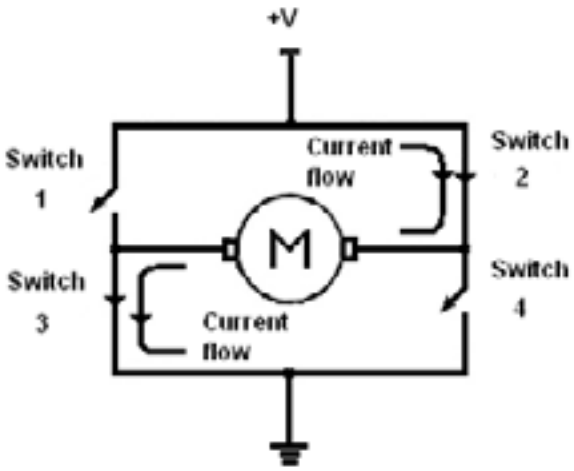


Figure 11-17 Motor Anti-Clockwise Rotation

Thus we can see that the switches normally operate in pairs since switches 1 and 4 switch on and off together and the same thing with switches 2 and 3. We would therefore require two signals which moreover are always of the opposite logic with respect to each other (one is the complement of the other). Hence theoretically we could do with one signal and its complement (which we can obtain by using an inverter). However, the need to avoid having all the switches ON at the same time (which can happen during the transition due to the propagation delay), it would be best if we use two separate signals (S1 and S2) to control the two separate pairs of switches as shown in Figure 11-18, making sure that we switch off one pair before switching on the other pair.

Once again, if instead of switching these sets of switches permanently ON, we supply them with a PWM signal, we now have the capability to control BOTH the speed and the direction of the DC motor.

A typical H-bridge, using discrete components is shown in Figure 11-18, with the transistors acting as the switches, being driven from the 8051 ports. We have to ensure by means of our software program not to have both transistors on either side of the motor ON at the same time, otherwise we would be short-circuiting the motor supply.

Download free eBooks at bookboon.com

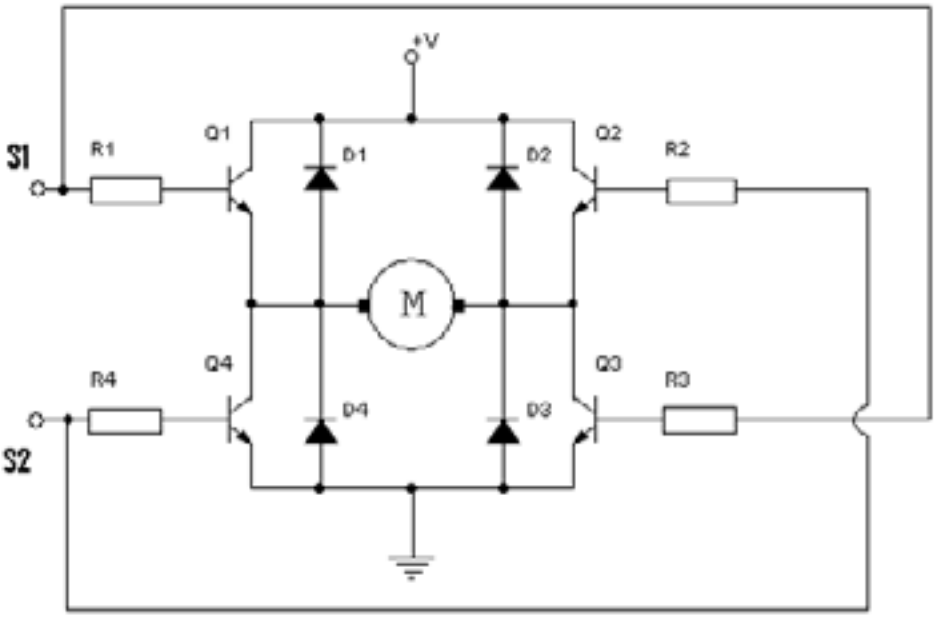


Figure 11-18 H-Bridge circuit with discrete devices

Thus, before switching from one set of transistors to the other set in order to change the direction, we must make sure to switch off ALL the transistors first. The algorithm to control the speed and direction is very simple and we describe it briefly here with the source code for a routine which controls the circuit shown in Figure 11-18. Speed can take a value between 0 and 100 representing zero (off) to 100% full speed. Direction can be either C (clockwise) or A (anti-clockwise). Duration would be the length of time in milliseconds that the motor has to be in that state.

We are assuming that we have a timer routine called ms_delay(unsigned long delay) which would wait for the specified amount of milliseconds.

Download free eBooks at bookboon.com

```
// H Bridge
#include <reg51.h>
sbit S1 = P1^0;
sbit S2 = P1^1;

void ms_delay(unsigned long);

// The following routine controls the motor, setting it at the required
// direction and speed for the specified time duration.
// The speed, although theoretically has the range 0–100, might need a value
// greater than 30 for the motor to actually start turning and overcome friction etc.
// The PWM signal has a periodic time of 100ms.
// Motor always exits the routine in the OFF condition.
```


Download free eBooks at bookboon.com

```
void MotorControl(char Direction, unsigned char Speed, unsigned long Duration) {
        unsigned long milliSeconds;

        milliSeconds = 0;

 if (Speed == 0) {
        S1 = S2 = 0;
        ms_delay(Duration);
 }                              // switch off motor completely

 else if ((Speed == 100) && ((Direction == 'A') || (Direction == 'a')) ){
        S1 = 0;
        S2 = 1;
        ms_delay(Duration);             // full speed anti-clockwise, no PWM required
        S1 = S2 = 0;
 }

 else if ((Speed == 100) && ((Direction == 'C') || (Direction == 'c')) ){
        S1 = 1;
        S2 = 0;
        ms_delay(Duration);             // full speed clockwise, no PWM equired
        S1 = S2 = 0;
 }

 else{                                  // 0 < speed < 100 hence PWM is required
        milliSeconds = 0;               // used for timing the duration of the PWM
        while(milliSeconds < Duration)
                {
                // first switch off one pair of transistors, then turn on the other pair of transistors
                // to avoid shorting the power supply
                if ((Direction == 'A') || (Direction == 'a')) {S1 = 0; S2 = 1;}
                if ((Direction == 'C') || (Direction == 'c')) {S2 = 0; S1 = 1}
                ms_delay((unsigned long)Speed);
                S1 = 0; S2 = 0;
                ms_delay((unsigned long)(100 – Speed));
                milliSeconds += 100UL; // add one PWM period to check duration
        }
 }

}
```


Download free eBooks at bookboon.com

```
void ms_delay(unsigned long delay_ms) {
// Assuming clock is 11.0592 MHz, then 921 timer counts
// would take approximately 1 millisecond
// Hence timer registers will be loaded with (65536 - 921)
// i.e. 64615, so that it will overflow after 1 millisecond
// TH0 = 64615/256 = 252
// TL0 = 64615%256 = 103

        TMOD &= 0xF0;
        TMOD |= 0x01;           // set Timer 0 in 16-bit mode 1
        ET0 = 0;                // disable Timer 0 interrupts just in case
        while (delay_ms > 0) {
          TH0 = 252;
          TL0 = 103;            // load Timer 0 registers for 1 millisecond delay
          TF0 = 0;              // clear Timer 0 overflow flag
        TR0 = 1;                // start Timer 0
          while (!TF0);         // wait for Timer 0 overflow
          delay_ms--;           // decrement 1 millisecond
        TR0 = 0;                // stop Timer 0
        }
        TF0 = 0;                // Reset flag before exit
}

void main(void) {
  S1 = S2 = 0;                          // start with motor off

  MotorControl('A',0,1000UL);           // motor stopped for 1 second
  MotorControl('C',90,4000UL);          // motor clockwise at 90%, for 4 seconds
  MotorControl('A',50,3000UL);          // motor anti-clockwise at 50%, for 3 seconds
  MotorControl('C',10,2000UL);          // motor clockwise at 10%, for 2 seconds
  MotorControl('A',100,2500UL);         // motor anti-clockwise at 100%, for 2.5 seconds

  while(1);                             // stay here when finished

}
```


Download free eBooks at bookboon.com

The H-bridge is so much in use that special ICs from a wide range of manufacturers have been designed. Shown in Figure 11-19 is a typical IC, the L292D which has the capability to drive 2 dc motors separately. Datasheets for this and similar devices are readily available on the internet, which fully describe the operation complete with examples.

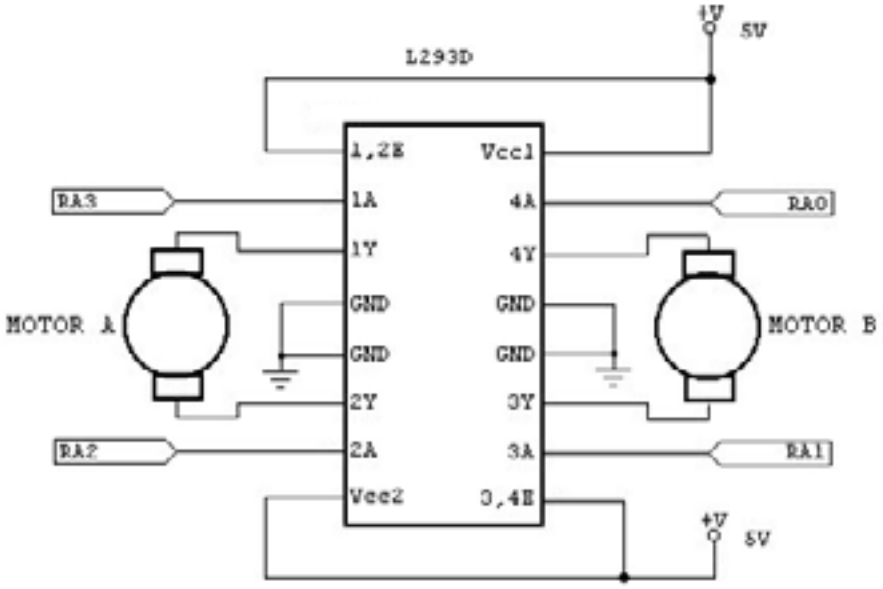


Figure 11-19 L293D H-bridge connection

Download free eBooks at bookboon.com
Click on the ad to read more

11.9 Model Servo Control

Radio Controlled (RC) model servo motors, of the type shown in Figure 11-20 can also be very easily controlled using the 8051. They are widely used in RC aero models and miniature robotics. These types of motors require a PWM signal very similar to the one explained above in sections 11.7 and 11.8. We need to have a PWM period of 20ms and we need to vary the ON time in the range of 1 to 2ms. A 1ms pulse would result in a full right movement say while a 2ms ON pulse would turn the servo arm to the full left position (a 1.5ms ON pulse would place the servo arm in the centre or neutral position).

Figure 11-20 RC Servo (www.parallaxinc.com)

Just three connections are needed as shown in Figure 11-21, two for the supply (usually around 5V, red is positive and black is ground) and the third wire (usually white or yellow) is where the PWM signal is fed from the micro-controller port pin. We should always remember to connect the ground of the servo to the ground of the micro-controller, since we would normally be feeding the servo from a separate higher capacity supply source.

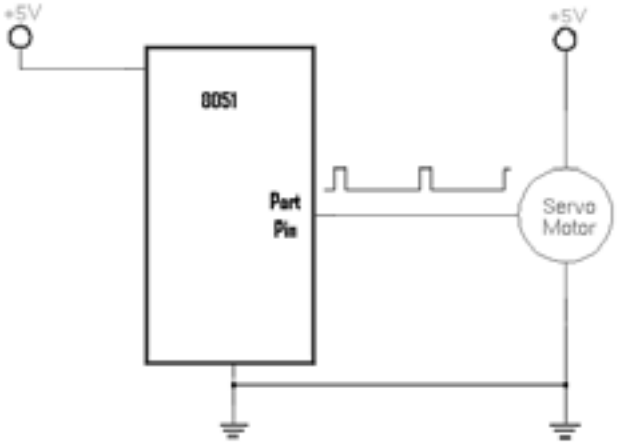


Figure 11-21 RC Servo connection

Download free eBooks at bookboon.com

Servos like all other motors, consume a lot of power especially when under load and it therefore would make sense to use a separate power supply just for the servo motor which would also reduce the interference on the 8051 supply lines.

We can also find servos which are slightly modified so that instead of just turning plus or minus 90 degrees, they are able to turn continuously. For example, a 1ms pulse would cause the servo to turn continuously clockwise and a 2ms pulse would turn it continuously anti-clockwise. In order to stop the servo, we would need to feed it with a 1.5ms pulse train, still using 20ms PWM periodicity.

11.10 Stepper Motor

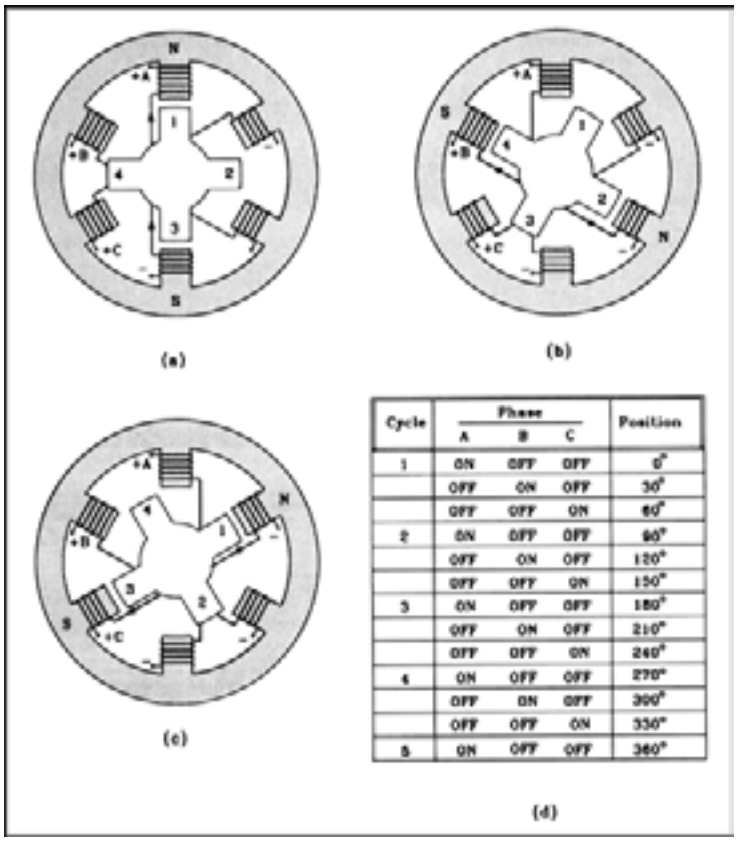
The stepper motor (see Figure 11-22) is one of the commonly used motors for precise angular movement. The advantage of using a stepper motor is that the angular position of the motor shaft can be controlled without the need of any feedback mechanism. They are widely used in industrial and commercial applications as well as in drive systems of autonomous robots.

Figure 11-22 Typical Stepper Motors

They are commonly found in dot-matrix or ink-jet printers to drive the printing head and feed forward the paper. By switching on the appropriate coils (see Figure 11-23), we can make the armature to rotate to and then stop at a specified rotation angle, so as it would align with the stator magnetic field. Moreover, if the whole 360 degree sequence is continuously repeated, the stepper motor can be made to turn at the required speed and in the required direction. The program would just have to determine which coils are to be energised and for how long.

Various ICs are available to drive these stepper motors and the L297 (or similar) stepper motor controller IC in conjunction with the L298 (or similar) dual H-bridge IC can be used.

Download free eBooks at bookboon.com



Cycle	Phase A	Phase B	Phase C	Position
1	ON	OFF	OFF	0°
	OFF	ON	OFF	30°
	OFF	OFF	ON	60°
2	ON	OFF	OFF	90°
	OFF	ON	OFF	120°
	OFF	OFF	ON	150°
3	ON	OFF	OFF	180°
	OFF	ON	OFF	210°
	OFF	OFF	ON	240°
4	ON	OFF	OFF	270°
	OFF	ON	OFF	300°
	OFF	OFF	ON	330°
5	ON	OFF	OFF	360°

Figure 11-23 Stepper Motor sequence (zone.ni.com)

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

Download free eBooks at bookboon.com

Click on the ad to read more

Index for Part I

Symbols

8032 141
  extras 141
  T2CON 146
  timer 2 144

B

baud rate
  A51 example 132
  setup 103
  timer 2 148
big endian 169

C

CALL
  ACALL 68
  LCALL 68
conditional branching 65
Control Bit Symbol
  AC 53
  C/T 50, 77
  CY 53
  EA 52, 117
  ES 52, 117
  ET0 52, 117
  ET1 52, 117
  ET2 52
  EX0 52, 117
  EX1 52, 117
  F0 53
  GATE 50, 77
  GF1 49
  GF2 49
  IDL 49
  IE0 49
  IE1 49
  INT0 47
  INT1 47
  IT0 49
  IT1 49
  M0 50, 77
  M1 50, 77
  OV 53
  P 53
  PD 49
  PS 53, 120
  PT0 53, 120
  PT1 53, 120
  PT2 53
  PX0 53, 120
  PX1 53, 120
  RB8 51, 101
  RD 47
  REN 51, 101
  RI 51, 101, 123
  RS0 53
  RS1 53
  RXD 47
  SM0 51, 101
  SM1 51, 101
  SM2 51, 101
  SMOD 49
  T0 47
  T1 47
  TB8 51, 101
  TF0 49, 82
  TF1 49, 82
  TH0 74
  TH1 74
  TI 51, 101, 123
  TL0 74
  TL1 74
  TR0 49, 82
  TR1 49, 82
  TXD 47
  WR 47

D

Development Boards
  C8051F020TB 166
  Flite-32 153
  Flite-32 IVT setup 179
  NMIY-0031 161
direct jumps 67

E

endian
  big 169
  little 169
Examples
  Big Endian and Little Endian - C 170
  PaulOS RTOS - C 220
  Traffic Lights A51 136
  UART baud rate A51 132

I

Interfacing
  4-bit mode 271
  7-Segment LEDs 250
  DC Motor 275
  H-bridge 277
  Keypad 261
  LCD 264

Download free eBooks at bookboon.com

LEDs 247
Servo Motor 283
Stepper Motor 285
Switches 258
Interrupts 69, 112, 115
common problems 128
considerations 125
IVT 152
polling sequence 118
priorities 119
sequence of events 120
serial 123
setting up 117
timer 2 151
Interrupt Vector Table 116, 152
ISR
stand-alone - PaulOS 218

J

jumps
conditional 65
direct 67

K

KEIL setup 173

L

little endian 169

M

MagnOS
description 225
OS_CHANGE_TASK_PRIORITY() 226, 232
OS_CHECK_MSG() 226, 236
OS_CHECK_TASk_PRIORITY() 232
OS_CHECK_TASK_PRIORITY() 226
OS_CHECK_TASK_SEMA4() 226, 237
OS_CLEAR_MSG() 226, 235
OS_CREATE_TASK() 226, 240
OS_GET_MSG() 226, 236
OS_INIT_RTOS() 226, 230
OS_KILL_IT() 239
OS_KILL_TASK() 226
OS_RELEASE_RES() 226, 233
OS_RTOS_GO() 226, 228
OS_RUNNING_TASK_ID() 226, 230
OS_SEMA4MINUS() 226, 238
OS_SEMA4_PLUS() 226, 238
OS_SEND_MSG() 226, 234
OS_WAIT4RES 233
OS_WAIT4RES() 226
OS_WAIT4SEM() 226, 239
OS_WAITI() 226, 231
OS_WAIT_MESSAGE() 226, 236
OS_WAITP() 212, 226, 229
OS_WAITT() 226, 231
Master-Slave 108
memory
bit-addressable 30
code area 26
external 26
internal data 27
on-chip 27
organisation 23

P

PaulOS
OS_CPU_DOWN() 218
OS_CREATE_TASK() 206, 209
OS_DEFER() 205, 206, 216
OS_INIT_RTOS() 206, 207
OS_KILL_IT() 205, 206, 216
OS_PAUSE_RTOS() 218
OS_PERIODIC() 206
OS_PERIODIC_A() 218
OS_RESUME_RTOS() 218
OS_RESUME_TASK() 206
OS_RTOS_GO() 206, 207, 209
OS_RUNNING_TASK_ID() 205, 210
OS_SCHECK() 205, 207, 210
OS_SIGNAL_TASK() 206, 207, 211
OS_WAITI() 206, 213
OS_WAITP() 205, 206
OS_WAITS() 206, 214
OS_WAITS_A() 218
OS_WAITT() 206, 215
OS_WAITT_A() 218
ready 205
running 203
stand-alone ISR 218
waiting 204
ports
P0 35
P1 40
P2 47
P3 47

R

register banks 29
RETI 123
round-robin rtos
SanctOS 191
RTOS
co-operative 189
MagnOS 225
pre-emptive 190, 225
ready state 187
round-robin 188, 191

Download free eBooks at bookboon.com

running state 187
SanctOS 191
states 187
types 188
waiting state 187

S

SanctOS
OS_CREATE_TASK() 191
OS_INIT_RTOS() 191
OS_INIT_RTOS(uchar iemask) 192
Serial Buffer 123
SFR 32
ACC 54, 56
B 54, 58
DPH 49
DPL 49
DPTR 49, 58
IE 52, 117
IP 53
P0 35
P1 40
P2 47
P3 47
PC 58
PCON 49
PSW 53
R 57
SBUF 51
SCON 51
SP 49, 59
T2CON 146
TCON 49, 81
TH0 51
TH1 51
timer 2 145
timer mode control bits 77
timer-related 74
TL0 51
TL1 51
TMOD 50, 76
Switch bounce 258

T

Timer
detecting overflow 85
initialisation 83
mode 0 77
mode 1 78
mode 2 79
mode 3 81
pulse duration 89
reading registers 84
timing events 87
Timer 2 144
auto relaod 149
capture mode 150
Timers 71

Download free eBooks at bookboon.com

Index for Part II

Examples

Buffered serial interrupt routines 80
SCC2691 UART 86
UART not under interrupt control 91
Light control using RTOS 98
Random display using RTOS 102
Master-Slave communication 105
Timer 0 Mode 3 247
Timer 1 as a baud-rate generator 247
Timer 2 as a baud-rate generator 251
XON/XOFF serial routine 253

P

programming
pitfalls 12
tips 12

S

SFR
DPTR 13

T

tips
C tips 18
DPTR 13
interrupts 15, 17
port usage 13
programming 12
ram size 12
serial 14
SFRs 13
SP setting 12
UART 14

Download free eBooks at bookboon.com

To see Part II download PaulOS Part II

Download free eBooks at bookboon.com