PaulOS F020: An RTOS for the C8051F020

Paul P. Debono
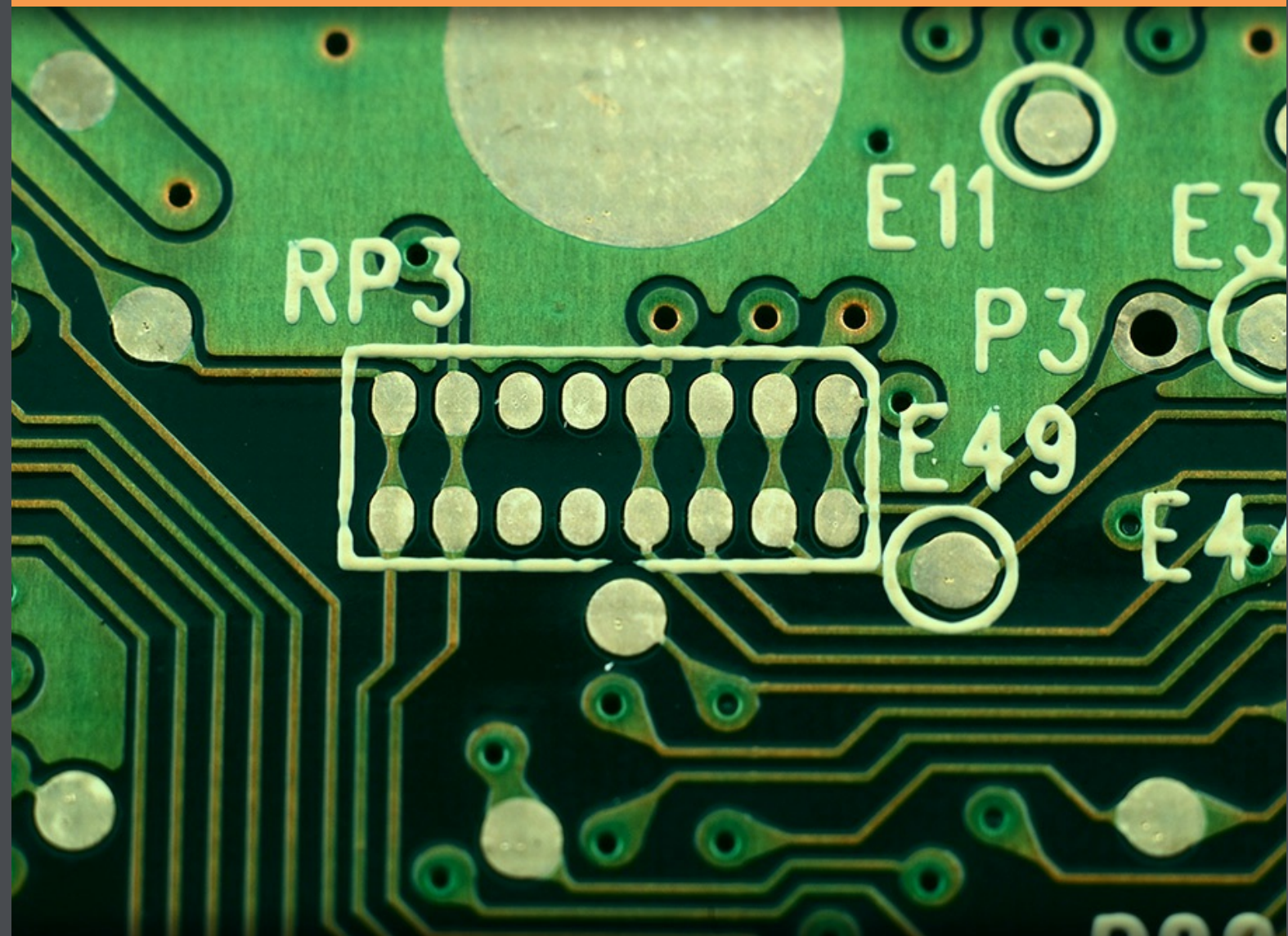
Paul P. Debono

PaulOS F020: An RTOS for the C8051F020

Contents

Preface

This text book is intended to be used as a reference book for those whose work requires familiarity with micro-controllers and real-time operating systems (RTOSs). Ideally it should be used in conjunction with my previous books (Debono, 2013a, 2013b) and with the C8051F020 data sheet (Silicon Labs, 2003b), where various simple RTOSs were fully explained. This book deals particularly with a modified version of the PaulOS co-operative RTOS so as to be able to work with the C8051F020 device, with its increased number of timers and interrupts.

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 version of the KEIL IDE (Simplicity Studio from http://www.silabs.com). The Silicon driver program SiC8051F_μVision.exe, which is also available from the Silicon Labs site should be installed so that the program would be downloaded on the development board once it is compiled for debugging.

The book is structured into 4 chapters and appendices with full source code listings of the PaulOS_F020 RTOS and a number of example programs. A brief outline of the contents of each chapter is given below:

Chapter 1:

This chapter describes the C8051F020 micro-controller and explains its internal organisation and lists the special function registers used to set the mode of operation of the various peripherals which are present on this versatile mixed-signal (Analogue and Digital) micro-controller device.

Chapter 2:

The PaulOS_F020 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 modified version, based on the PaulOS for the basic 8051 described in (Debono, 2013a) now runs on the much faster Silicon Labs C8051F020 at over 20 million instructions per second (MIPS), is ideally suited for small project applications and for getting important familiarisation with the techniques (and pit-falls) encountered when using an RTOS on such a device.

Chapter 3:

This chapter deals with multi-processor applications running under RTOS. An example is given of a serial network of boards where each Slave board runs its own PaulOS RTOS but each RTOS itself is synchronised with the other slave boards by means of serial transmissions originating from the Master.

Chapter 4:

In this 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 and understand this chapter before attempting to write the first program.

Appendices:

Finally in the appendices we can find the full program source code listings (C language format) of the C8051F020 version of the PaulOS RTOS described in chapter 2.

Whilst hoping that you will find this book useful, please feel free to contact me if you have any queries or suggestions. (e-mail: pawlu.debono@yahoo.co.uk)

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 son Luke 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 grateful to Prof. Ing. Victor Buttigieg who kindly accepted to review the final version of the book. He also put forward valuable and much appreciated suggestions.

<div align="right">PAUL DEBONO</div>

Dedications

To

my wife Maria for being so supportive and patient with me and to my two sons Neil and Luke for their continuous encouragement, and to my grand-daughter baby Mila for her adorable smile.

List of Figures

List of Tables

1   C8051F020 Basics

This chapter describes the C8051F020 micro-controller and explains its internal organisation and the way its special function registers can be used to setup the various peripherals. Many web pages, books (see bibliography list) 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. The use of the C8051F020 datasheet/manual (Silicon Labs, 2003b) and (Chew & Gupta, 2005) in conjunction with this book is highly recommended whilst following the source code examples listed throughout the book. Other good reference books are (Huang, 2009) and (Schultz, 2004) which also deal with the Silicon Labs C8051 family of micro-controllers in some of their chapters.

When using the KEIL IDE as stated in the Preface section, the Silicon Labs µV3/µV4 driver program SiC8051F_µVision.exe, which is freely available from the Silicon Labs site should be installed. This would enable the compiled program to be downloaded on the development board via the JTAG once the debug tab is pressed.

Lately, a new software package from Silicon Labs (http://www.silabs.com) is also available. The 'Simplicity Studio' application program provides one-click access to design tools, documentation, software and support resources for EFM32, EFM8, 8051, Wireless MCUs and Wireless SoCs.

At the University we use the Silicon Labs development boards shown in Figure 1-1. At the time of writing, it is one of a series of super-charged versions of the 8051 family from Silicon Labs.
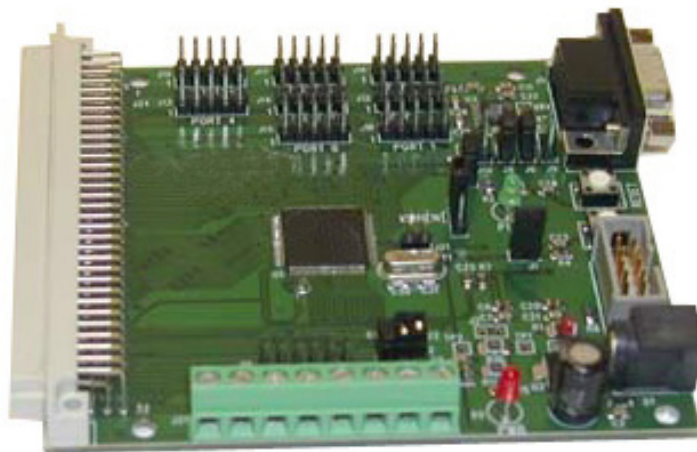
Figure 1-1 The C8051F020TB Development Board

Further details, manuals, integrated development packages and example programs can be found at the Silicon Labs site, http://www.silabs.com.

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 still an important skill for anyone who plans to develop products that will take advantage of most micro-controllers.

The various sections of the first chapter will explain such a derivative, the Silicon Laboratories C8051F020 micro-controller. The sections in these chapters are targeted at students and readers who already have some knowledge of the basic 8051 micro-controller and are attempting to move on to something more powerful and learn to program the C8051F020 multi-signal device. The appendices provide 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.

Throughout this book, it is therefore assumed that the reader has got some amount of programming knowledge in C and that he has a basic understanding of the hardware.

The C8051F020 is a 64 or 100-pin IC as shown in Figure 1-2.
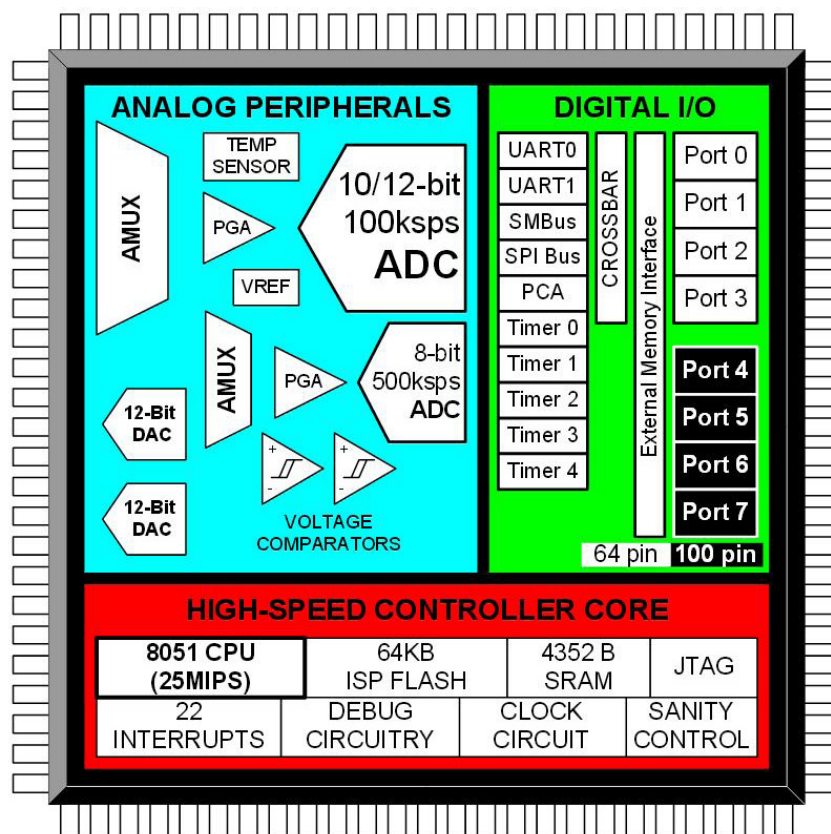
Figure 1-2 C8051F020 System Overview

We shall now deal with the internal organisation of the C8051F020 micro-controller.

1.2   Memory Types

The C8051F020 has three types of memory and each type has to be addressed in a different way. To effectively program the device it is therefore 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 C8051F020 are illustrated Table 1-1 and they are classified as the Data Memory (RAM) which is itself organised in two separate areas, namely the Internal Data Address Space which is identical with all the basic 8052/8032 devices, and External Data Address Space which has 4096 bytes present on-chip with the ability to have extra additional storage space added externally. Then there is the Program Code/Data Memory (Flash). 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) or pre-fixed with a '0x' (i.e. 0xFF) to signify that they are hexadecimal numbers.

Table 1-1 C8051F020 memory map

1.3 Program/Data Memory (Flash)

The flash memory is the part of memory that holds the actual code or program that is to be executed. This memory is limited to 64KB but being a flash memory, code can be re-written to it many times, so as to change/update the program.

Also because it is a flash memory, it can also be used to store data which needs to be updated and stored for re-use. For example you might need to store some settings (variables), which although they can be varied whilst running the program, you would still need to store their value, so that when the program runs again (after having been switched off), it would start off again using those previously updated and stored values. Example routines of writing to and reading from flash memory are widely available on the internet and these can be easily integrated in your project.

In the IDE, this memory would be denoted by the 'CODE' keyword, and apart from storing the code (program) you can/should use this memory area also to store fixed constants so as not to waste valuable RAM and XRAM.

1.4   External Data Address Space (XRAM)

As an obvious opposite of Internal RAM, the C8051F020 also supports what is called External Data Address Space (XRAM). This is accessible using the MOVX assembly code instruction. 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 assembly language instructions, namely:

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

These are executed in seven instruction cycles and in this case, external memory is seven times slower!

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 actually residing physically on the same chip, but it is still referred to as external (or XDATA) RAM and as such it is still compatible with the basic 8051 in this respect.

1.4.1   Internal Data Address Space (RAM)

The on-chip internal memory data address space can be Directly or Indirectly addressable, or both. Internal RAM is usually used to store variables, where the lower 128 bytes can be addressed in both modes (Direct and Indirect), while the upper 128 bytes, address range 80H to FFH can be only accessed by the Indirect Addressing. The Special Function Register (SFR) memory, which also lies in the address range 80H to FFH, can only be accessed by Direct addressing, so as to enable these locations to be discriminated from the other RAM bytes with the same address. It is used to store the system SFRs which control the mode of operation of the various built-in peripherals. The layout of the C8051F020's internal memory is presented in the memory map shown in Table 1-2 which is identical with that of the basic 8051 except of course that it now has more SFRs since this device is much more capable that the basic 8051, with a much larger list of peripherals.

The C8051F020 has a bank of 256 bytes of Internal RAM. This Internal RAM is found on-chip on the device itself 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 device is reset or powered off this memory is lost.

The 256 bytes of internal ram is subdivided as shown on the memory map in Table 1-2. 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, each bank having a set of 8 registers R0 to R7, 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.

The 80 bytes that remain in Direct and Indirectly addressable Internal RAM, from address 30h through address 7Fh, and the other 128 Indirectly addressable bytes 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, which is always accessed indirectly using the Stack Pointer (SP) SFR to store the address of the location to be read/written.

Hex Byte Address	Hex Bit Address								Notes
FFH 80H	Indirectly Addressable General Purpose RAM								Used as a STACK Area and to store user variables
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 1-2 C8051F020 Internal Data Address Space

The stack is used automatically by the processor in order to save the return addresses when functions and subroutines are called by the program either directly or indirectly via an interrupt in the case of an Interrupt Service Routine (ISR). It is also used to store some temporary values of registers or variables until they are retrieved again when needed. It should be noted, as illustrated in the memory map of Table 1-2, the area used for the stack is also shared with any user variables stored in 'DATA'. If more stack space is required, the variables can be moved to 'XDATA' area either when declaring the variable or by setting the global default in the 'Target Option' tab of the IDE as explained in (Debono, 2013a, pp. 174,175,180).

1.5   Register Banks

This device 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 1-2 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:

```
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 1-2).

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. However we must always remember that the register banks really reside in the first 32 bytes of Internal RAM.

1.6 Bit Memory

The C8051F020, being a communications and control oriented micro-controller, gives the user the ability to access a number of bit variables. These variables may only take the value of either a 1 or a 0.

There are 128 bit variables available to the user (see Table 1-2); individually have an address 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. For example we can use:

MOV 20h, #0FFh

This 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
```

To use bit variables in C, there is a special operator **bit** which is used to declare a bit variable. The bit type may be used for variable declarations, argument lists, and function-return values. A bit variable is declared like other C data types. For example:

```
static bit ready_flag = 0; /* bit variable */
```

Another possibility would be to declare a byte variable in the bit addressable area, and then we also have the capability to name and address the individual bits of this variable. For example:

```
char bdata Alarm;          /* Alarm is declared as a one byte variable, residing in the */
                           /*  bit addressable area (bdata). */
sbit  fire =  Alarm^0;     /* bit  0 of variable Alarm, is named 'fire' and can be used */
                           /* as a normal bit variable */
sbit smoke = Alarm^1;    /* bit 1 is named 'smoke' */
```

As illustrated in Table 1-2, 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 mainly intended for user-defined bit variables used in the program. These are not the only bit variables available on the 8051. Other bits in certain SFRs (those which have their address ending with a 0 or an 8) 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 in C:	P0 = 1;
or			
ORL P0, #01h	; logically OR P0 with 00000001 binary	or in C:	P0 |= 1;
or			
SETB 80h		or in C:	P0_1 = 1; //**
or even			
SETB P0.0	; the assembler knows that P0.0 = 80h	or in C:	P0_1 = 1; //**

** assuming that that you declare in C the following:

	sbit P0_1 = P0^0;	// name bit 0 of port P0 as P0_1
or	sbit P0^1 = 0x80;	

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.

Naturally, if no bit variables are required this bit-addressable area can be used to store other variables (bytes, integers etc). It is not restricted to storing just bits!

1.7   Special Function Register (SFR) Memory

Special Function Registers (SFRs) reside in areas of internal memory that control specific functionality of the C8051F020 chip, as shown in Table 1-3. For example, eight SFRs permit access to the 8 I/O port P0-P7. Another two SFRs (SBUF0 and SBUF1) allow a program to read from or write to its two serial ports which are called UART0 and UART1 (Universal Asynchronous Receiver/ Transmitter). Other SFRs allow the user to set the serial baud rates, control and access timers, ADC, DAC etc. and also 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
```

Similarly, if we want to write the letter 'A' to its UART0 we would write this value to the SBUF0 SFR, which has an SFR address of 99 Hex. Thus, to write the value 'A', which has an ASCII code of 65 decimal; to the serial port we would execute the instruction:

```
MOV 99h, #41h or MOV SBUF0, #41h or MOV SBUF0, 'A'
```

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. Quoting from the KEIL uV4 IDE online user manual:

…….. The Cx51 Compiler provides access to SFRs with the **sfr**, **sfr16**, and **sbit** data types.
SFRs are declared in the same fashion as other C variables. The only difference is that the type specified is sfr rather than char or int. For example:

```
sfr P0 = 0x80; /*    Port-0, address 80h */
sfr P1 = 0x90; /*    Port-1, address 90h */
sfr P2 = 0xA0; /*    Port-2, address 0A0h */
sfr P3 = 0xB0; /*    Port-3, address 0B0h */
```

…….. end of quote.

P0, **P1**, **P2**, and **P3** are the SFR name declarations. Names for sfr variables are defined just like other C variable declarations. Any symbolic name may be used in an sfr declaration.

The address specification after the equal sign ('=') must be a numeric constant. Expressions with operators are not allowed. Classic 8051 devices support the SFR address range 0x80-0xFF.

The Cx51 Compiler is a 'big endian' compiler in the sense that it stores the variables with the high byte occupying the lowest memory address. However, since the 8051 has some SFRs (such as TL2 and TH2) stored in 'little endian' format, the Keil compiler provides the **sfr16** data type to access two consecutive 8-bit SFRs as a single 16-bit SFR in 'little endian' style.

Access to 16-bit SFRs using sfr16 is possible only when the low byte immediately precedes the high byte (little endian) and when the low byte can be written last without affecting the functionality of the device. Certain devices might require the low byte to be written first when setting up the particular peripheral. The low byte is used as the address in the sfr16 declaration. For example:

```
sfr16 T2REG = 0xCC; /*        comprising the 2 SFRs TL2 at 0CCh, TH2 at 0CDh */
sfr16 RCAP2 = 0xCA; /*        RCAP2L at 0CAh, RCAP2H at 0CBh */
```

Elsewhere in the program, we can then write and execute:

```
T2REG = 0x1234;    /* resulting in TH2=0x12, TL2=0x34 with TL2 written last */
RCAP2 = 0x5678;    /* resulting in RCAP2H=0x56, RCAP2L=0x78 with RCAP2L written last */
```

The **sbit** type defines a bit within a special function register (SFR) or any variable in the bit addressable area. It is used in one of the following ways, taking an SFR as an example:

```
sbit name = sfr-name ^ bit-position;
sbit name = sfr-address ^ bit-position;
sbit name = sbit-address;
```

where:

name	is the name of the SFR bit
sfr-name	is the name of the previously-defined SFR
bit-position	is the position of the bit with the SFR
sfr-address	is the address of the SFR
sbit-address	is the address of the SFR bit.

1.7.1 SFR Addresses

The C8051F020 is a flexible micro-controller with a relatively large number of peripherals having different modes of operation. 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-controller, our program may inspect and/or change their operating mode by manipulating the values of some specific SFRs. They are accessed just 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 1-3 provides a tabular 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 6 locations which are not used. 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 at all. In fact the actual memory cell of these free locations might not be physically present on the chip. These free locations are reserved for future enhanced and upgraded versions of this family of micro-controllers, and certain versions (such as the C8051F040) need much more SFRs than can be fitted in 128 bytes. These therefore make use of more than one 128-byte page of SFRs thus having to switch SFR pages in order to set the correct SFR

	0(8) (bit addressable)	1(9)	2(A)	3(B)	4(C)	5(D)	6(E)	7(F)
F8	SPI0CN	PCA0H	PCA0CPH0	PCA0CPH1	PCA0CPH2	PCA0CPH3	PCA0CPH4	WDTCN
F0	B	SCON1	SBUF1	SADDR1	TL4	TH4	EIP1	EIP2
E8	ADC0CN	PCA0L	PCA0CPL0	PCA0CPL1	PCA0CPL2	PCA0CPL3	PCA0CPL4	RSTSRC
E0	ACC	XBR0	XBR1	XBR2	RCAP4L	RCAP4H	EIE1	EIE2
D8	PCA0CN	PCA0MD	PCA0CPM0	PCA0CPM1	PCA0CPM2	PCA0CPM3	PCA0CPM4	
D0	PSW	REF0CN	DAC0L	DAC0H	DAC0CN	DAC1L	DAC1H	DAC1CN
C8	T2CON	T4CON	RCAP2L	RCAP2H	TL2	TH2		SMB0CR
C0	SMB0CN	SMB0STA	SMB0DAT	SMB0ADR	ADC0GTL	ADC0GTH	ADC0LTL	ADC0LTH
B8	IP	SADEN0	AMX0CF	AMX0SL	ADC0CF	P1MDIN	ADC0L	ADC0H
B0	P3	OSCXCN	OSCICN			P74OUT	FLSCL	FLACL
A8	IE	SADDR0	ADC1CN	ADC1CF	AMX1SL	P3IF	SADEN1	EMI0CN
A0	P2	EMI0TC		EMI0CF	P0MDOUT	P1MDOUT	P2MDOUT	P3MDOUT
98	SCON0	SBUF0	SPI0CFG	SPI0DAT	ADC1	SPI0CKR	CPT0CN	CPT1CN
90	P1	TMR3CN	TMR3RLL	TMR3RLH	TMR3L	TMR3H	P7	
88	TCON	TMOD	TL0	TL1	TH0	TH1	CKCON	PSCTL
80	P0	SP	DPL	DPH	P4	P5	P6	PCON

Table 1-3 C8051F020 Special Function Registers (SFRs)-DIRECT addressing ONLY

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 through 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

In this section we shall only mention some special SFRs which are appreciably different from the basic 8051 SFR. The standard 8051 SFRs are still available and work in exactly the same way even on this device, and details about these 'old' SFRs can be found in an earlier book (Debono, 2013a).

In general, as mentioned in Table 1-3 itself, some 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 SCON0 controls serial port (UART0) operations.

The other SFRs can be thought of as auxiliary 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 UART0 has been configured using SCON0, the program may read or write data characters or bytes to the serial port using the SBUF0 register.

The SFRs whose address has an asterisk (*) in the Table 1-3 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 an 8h) can be accessed with bit operations, meaning that they are bit-addressable.

1.8   SFR Descriptions

As already mentioned in section 1.7.2 the basic 8051 SFRs were all fully described in a previous book (Debono, 2013a) and will not be covered here. Instead we shall mainly deal here with some important new SFRs specific to the C8051F020. For a full description of all the SFRs, such as those dealing with the ADC, DAC, etc it would be best to consult the manual/data sheet (Silicon Labs, 2003b).

1.8.1   System Clock (OSCXCN, OSCICN) registers

SFRs OSCXCN and OSCICN are used to select and configure the system clock. The routine listed in Figure 1-3 SYSCLK initialisation routine sets the system clock to use a 22.1184MHz crystal as its clock source.

```
void SYSCLK_Init (void)

{

 unsigned int i;                    // delay counter

 OSCXCN = 0x67;                     // start external oscillator with 22.1184MHz crystal

 for (i=0; i < 256; i++) ;          // wait for oscillator to start

 while (!(OSCXCN & 0x80)) ;         // Wait for crystal osc. to settle, not need if in simulator mode

 OSCICN = 0x88;                     // select external oscillator as SYSCLK

                                    // source and enable missing clock detector

 }
```

Figure 1-3 SYSCLK initialisation routine

Note that the

```
 while (!(OSCXCN & 0x80)) ;
```

is a delay instruction, waiting for the crystal oscillator to settle is not required if running the KEIL IDE in simulation mode and in certain cases the simulation would hang on this line if the simulation is not emulating the oscillator function perfectly. It could be conditionally compiled by defining SIMULATOR in the C51 Target tab when running in simulation mode and then using:

```
#ifndef SIMULATOR

        while (!(OSCXCN & 0x80)) ;

#endif
```

Thus the oscillator will be tested only when running on the actual board (without the SIMULATOR definition) and the test will be ignored when running in simulation mode.

1.8.2   Watchdog Timer (WDT)

The micro-controller has a programmable Watchdog Timer (WDT) which runs off the system clock. An overflow of the WDT forces the micro-controller into the reset state. Before the WDT overflows, using certain commands as explained in section 1.8.2.2 the application program must restart it so as the WDT starts counting again from zero.

WDT is useful in preventing the system from running out of control, especially in critical applications. If the system experiences a software or hardware malfunction which prevents the software from restarting the WDT, then the WDT will overflow and cause a controller reset. After a reset, the WDT is automatically enabled and starts running at the default maximum time interval which is 524 ms for a 2 MHz system clock or approximately 47ms in the case of a 22.1184 MHz clock.

The WDT consists of a 21-bit timer running from the programmed system clock. A WDT reset is generated when the period between specific writes to its control register exceeds the programmed limit as given in equations (1-1) and (1-2). The WDT may be enabled or disabled by software as explained in section 1.8.2.1 and in section 1.8.2.2. It may also be locked to prevent accidental disabling. Once locked, the WDT cannot be disabled until the next system reset. It may also be permanently disabled. The watchdog features are controlled by programming the Watchdog Timer Control Register (WDTCN), details of which are shown in Figure 1-4, which is taken from (Silicon Labs, 2003, p. 131).

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
								xxxxx111
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xFF

Bits7-0:   WDT Control
            Writing 0xA5 both enables and reloads the WDT.
            Writing 0xDE followed within 4 system clocks by 0xAD disables the WDT.
            Writing 0xFF locks out the disable feature.
Bit4:   Watchdog Status Bit (when Read)
            Reading the WDTCN.[4] bit indicates the Watchdog Timer Status.
            0: WDT is inactive
            1: WDT is active
Bits2-0:   Watchdog Timeout Interval Bits
            The WDTCN.[2:0] bits set the Watchdog Timeout Interval. When writing these bits, WDTCN.7 must be set to 0.

Figure 1-4 WDTCN: Watchdog Timer Control register

1.8.2.1 Disabling the WDT

This is usually disabled at the very beginning of the main program so as not to give time for the watchdog timer to overflow and reset the micro-controller. In certain cases, if there are a lot of initialisations being done in the Startup.a51 code (which is actually executed even before the main( ) function in the application program), the 'disable_watchdog' code would even need to be written directly in the Startup.a51 so as to be executed right at the beginning, immediately after a reset or switch-on. As seen in Figure 1-5 the interrupts are disabled immediately in the 'disable_watchdog' code so that nothing interferes with this process. The two write instructions to the WDTCN register should be made within 4 clock cycles of each other as suggested in (Silicon Labs, 2003, p. 130) and the interrupts are enabled before exiting the routine.

```
void DISABLE_Watchdog (void)

{

  EA = 0;

  WDTCN = 0xDE;

  WDTCN = 0xAD;

  EA = 1;

}
```

Figure 1-5 Routine used to disable the watchdog timer

1.8.2.2 Enabling and Setting WDT Interval

Bits 2-0 of WDTCN, treated as a 3-bit binary number, control the watchdog timeout interval (WDTI). The time interval is given by the following equation:

$$\text{WDTI} = 4^{3+\text{WDTCN}[2-0]} \times T_{\text{SYSCLK}} \tag{1-1}$$

where T_{SYSCLK} is the system clock period or the reciprocal of the system clock frequency SYSCLK. Hence we may also write the time interval equation as:

$$\text{WDTI} = \frac{4^{3+\text{WDTCN}[2-0]}}{SYSCLK} \tag{1-2}$$

For a 22.1184 MHz system clock, the interval range that can be programmed up to a maximum of 47.4 ms, with the lower three bits WDTCN[2-0] set to 111 binary (equivalent to 7 decimal) as shown in equation (1-3). When the watchdog timeout interval bits are written to the WDTCN register, the WDTCN.7 bit must be held at logic 0. The programmed interval may be read back by reading the WDTCN register. After a reset, WDTCN[2-0] reads 111b.

$$\text{WDTI} = \frac{4^{3+7}}{SYSCLK} = \frac{4^{10}}{22.1184 \times 10^6} = 47.4 \; ms \tag{1-3}$$

With this information, we can therefore write our routine to enable the watchdog timer, which is listed in Figure 1-6. Assuming that we are using a 22.1184MHz oscillator, then this initialisation routine would set the watchdog timer to overflow and thus causing a micro-controller reset every 47.4ms unless the watchdog timer itself is reset by 'feeding' the watchdog before overflowing.

```
// Enables the watchdog timer
//
void ENABLE_Watchdog (void)
{
    EA = 0;
    // set bit 7 to 0 in order to write count and
    // set WDTCN[2.0] to 111b giving a WDT timeout interval = ( 4(3+7) ) x Tsysclock
    // This would give the maximum timeout interval.
    // At 22.1184 MHz, this would be equal to 47.4 ms
    WDTCN = 0x07; // set the timeout interval bits
    WDTCN = 0xA5; // enable WDT
    EA = 1;
}
```

Figure 1-6 Routine to Enable the watchdog timer, with a 47.4ms interval

The watchdog timer would have to be reset to start counting up again from 0, before giving it a chance to overflow. This is done by means of the 'FeedWDT' task shown in Figure 1-7running under the PaulOS RTOS, which is a periodic function, executing every 40ms (which is less than the 47.4ms WDT interval). The PaulOS RTOS would be fully explained in chapter 2, but the commands and functions are the same as in the earlier book (Debono, 2013a).

```
/*

Task 'FeedWDT'

 – created as Task 0 to give it highest priority when RTOS sorts the 'ready' tasks

 – as explained in section 2.3.3

 – Periodicity must be less than 47.4 ms as calculated above in the

 – ENABLE_Watchdog function to avoid WDT reset

*/

void FeedWDT_Task (void)

{

    OS_PERIODIC_A(0,0,40);          /* Execute every 40 ms */

    while(1)                        // endless loop

     {

       WDTCN = 0xA5;                 // reloads WDT

       OS_WAITP( );                 // wait for the periodic interval

     }

}
```

Figure 1-7 RTOS task used to 'feed' the watchdog every 40ms

In order to give it preferential treatment, this task is allocated a task number of 0 and the priority sorting is enabled in the OS_RTOS_GO(1) command as shown in the partial listing shown in Figure 1-8. Thus if there are other tasks ready to execute, Task 0 would be selected instead of the normal first in-first out selection if no priority sorting was implemented.

```
void main (void) {
    DISABLE_Watchdog( );
    SYSCLK_Init( );
    SetUpUART(0, 115200, 1); /* Set up UART 0, at 115200 baud using Timer 1 */
    PORT_Init( );

    OS_INIT_RTOS(0);           /* initialise RTOS, variables and stack */
    OS_CREATE_TASK(0, FeedWDT_Task);   /* CREATE task0, 0 being the highest priority */
    OS_CREATE_TASK(1, Blink_Task);
    OS_CREATE_TASK(2, Clock_Task);

    ENABLE_Watchdog( );
    OS_RTOS_GO(1);                        /* Start multitasking, with priority sorting */
    while (1)
    {..
…
… }
}
```

Figure 1-8 Part of the main program showing priority allocation for the watchdog feeder task

In the example above, if the periodicity of Task 0 is set to be greater than 47.4ms, then the WDT would overflow and cause a system reset, since WDT would not be restarted in time to count from zero before overflowing.

1.8.3   Crossbar Decoder (XBR0, XBR1 and XBR2)

The user controls which digital functions are assigned to any IC pins[1] which can then be accessed by the user. This flexibility is limited only by the number of pins available on the IC. The Port pins on Port 1 can be used as Analogue Inputs to ADC1. The Priority Crossbar Decoder, or "Crossbar", allocates and assigns Port pins on Port 0 through Port 3 to the digital peripherals (UARTs, SMBus, PCA, Timers, etc.) on the device using a priority order. The Port pins are allocated in order starting with P0.0 and continue through P3.7 if necessary. The (Silicon Labs, 2003b) manual is used as a reference for this section and all the figures and tables from section 1.8.3 to section 1.8.7 are taken from it. The digital peripherals shown in Figure 1-9 are assigned Port pins in a priority order which is listed in Table 1-4, with UART0 having the highest priority and CNVSTR having the lowest priority.

Again using (Silicon Labs, 2003b), the Crossbar assigns Port pins to a peripheral if the corresponding enable bits of the peripheral are set to logic 1 in the Crossbar configuration registers XBR0, XBR1, and XBR2, shown in Figure 1-10, Figure 1-11 and Figure 1-12. For example, if the UART0EN bit (XBR0.2) is set to logic 1, the TX0 and RX0 pins will be mapped to P0.0 and P0.1 respectively. Because UART0 has the highest priority, its pins will always be mapped to P0.0 and P0.1 when UART0EN is set to logic 1. If a digital peripheral's enable bits are not set to logic 1, then its ports are not accessible at the Port pins of the device. Also note that the Crossbar assigns pins to all associated functions when a serial communication peripheral is selected (i.e. SMBus, SPI, UART). It would be impossible, for example, to assign TX0 to a Port pin without assigning RX0 as well. Each combination of enabled peripherals results in a unique device pin-out.

Once the Crossbar registers have been properly configured, the Crossbar is enabled by setting XBARE (XBR2.6) to logic 1. Until XBARE is set to logic 1, the output drivers on Ports 0 through 3 are explicitly disabled in order to prevent possible contention on the Port pins while the Crossbar registers and other registers which can affect the device pin-out are being written.

The output drivers on Crossbar-assigned input signals (like RX0, for example) are explicitly disabled; thus the values of the Port Data registers and the PnMDOUT registers have no effect on the states of these pins.
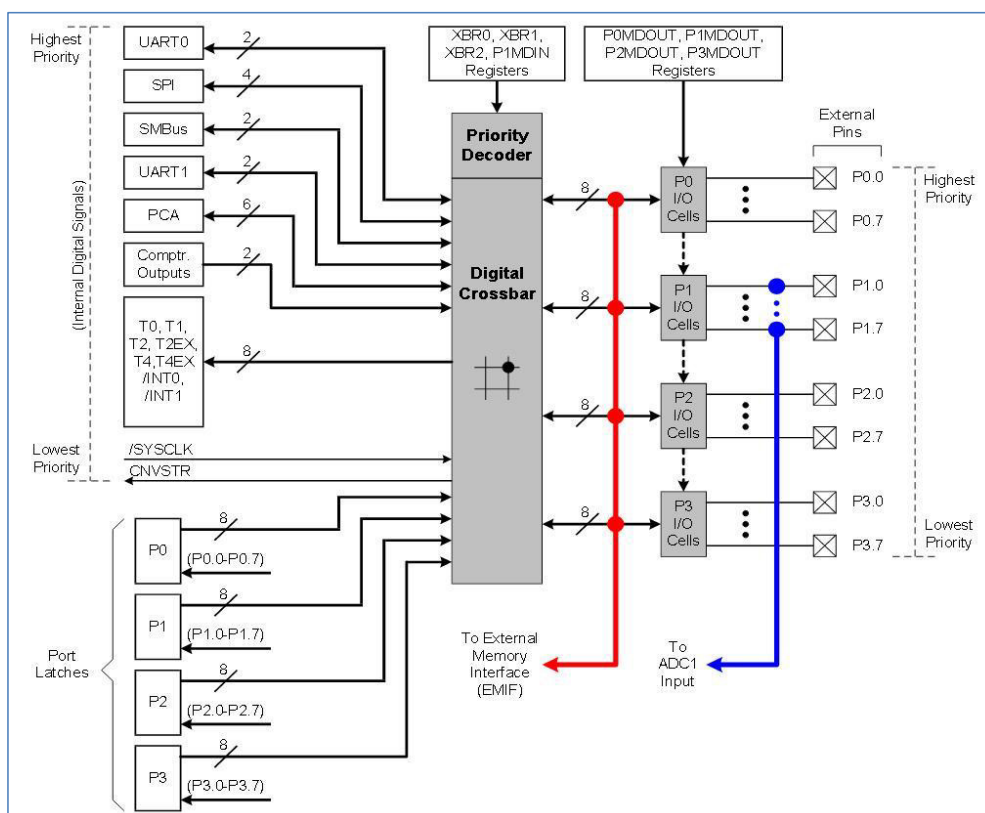
Figure 1-9 Lower Port I/O Functional Block Diagram

PIN I/O	P0.0	P0.1	P0.2	P0.3	P0.4	P0.5	P0.6	P0.7	P1.0	P1.1	P1.2	P1.3	P1.4	P1.5	P1.6	P1.7	P2.0	P2.1	P2.2	P2.3	P2.4	P2.5	P2.6	P2.7	P3.0	P3.1	P3.2	P3.3	P3.4	P3.5	P3.6	P3.7	Crossbar Register Bits
TX0	●																																UART0EN: XBR0.2
RX0		●																															
SCK	●		●																														SPI0EN: XBR0.1
MISO		●		●																													
MOSI			●		●																												
NSS				●		●																											
SDA	●		●		●		●																										SMB0EN: XBR0.0
SCL		●		●		●		●																									
TX1	●		●		●		●		●																								UART1EN: XBR2.2
RX1		●		●		●		●		●																							
CEX0	●		●		●		●		●		●																						PCA0ME: XBR0.[5:3]
CEX1		●		●		●		●		●		●																					
CEX2			●		●		●		●		●		●																				
CEX3				●		●		●		●		●		●																			
CEX4					●		●		●		●		●		●																		
ECI	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●																	ECI0E: XBR0.6
CP0	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●																CP0E: XBR0.7
CP1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●															CP1E: XBR1.0
T0	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●														T0E: XBR1.1
/INT0	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●													INT0E: XBR1.2
T1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●												T1E: XBR1.3
/INT1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●											INT1E: XBR1.4
T2	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●										T2E: XBR1.5
T2EX	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●									T2EXE: XBR1.6
T4	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●								T4E: XBR2.3
T4EX	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●							T4EXE: XBR2.4
/SYSCLK	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●						SYSCKE: XBR1.7
CNVSTR	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●					CNVSTE: XBR2.0

Bottom pin labels:
- P0.5 = ALE, P0.6 = /RD, P0.7 = /WR
- P1: AIN1.0/A8, AIN1.1/A9, AIN1.2/A10, AIN1.3/A11, AIN1.4/A12, AIN1.5/A13, AIN1.6/A14, AIN1.7/A15 — *AIN1 Inputs/Non-muxed Addr H*
- P2: A8m/A0, A9m/A1, A10m/A2, A11m/A3, A12m/A4, A13m/A5, A14m/A6, A15m/A7 — *Muxed Addr H/Non-muxed Addr L*
- P3: A0O/D0, A1O/D1, A2O/D2, A3O/D3, A4O/D4, A5O/D5, A6O/D6, A7O/D7 — *Muxed Data/Non-muxed Data*

Table 1-4 Priority Crossbar decode table (and use of XBR0, XBR1 and XBR2)

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
CP0E	ECI0E		PCA0ME		UART0EN	SPI0EN	SMB0EN	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xE1

Bit7:   CP0E: Comparator 0 Output Enable Bit.
        0: CP0 unavailable at Port pin.
        1: CP0 routed to Port pin.
Bit6:   ECI0E: PCA0 External Counter Input Enable Bit.
        0: PCA0 External Counter Input unavailable at Port pin.
        1: PCA0 External Counter Input (ECI0) routed to Port pin.
Bits5-3: PCA0ME: PCA0 Module I/O Enable Bits.
        000: All PCA0 I/O unavailable at Port pins.
        001: CEX0 routed to Port pin.
        010: CEX0, CEX1 routed to 2 Port pins.
        011: CEX0, CEX1, and CEX2 routed to 3 Port pins.
        100: CEX0, CEX1, CEX2 and CEX3 routed to 4 Port pins.
        101: CEX0, CEX1, CEX2, CEX3 and CEX4 routed to 5 Port pins.
        110: RESERVED.
        111: RESERVED.
Bit2:   UART0EN: UART0 I/O Enable Bit.
        0: UART0 I/O unavailable at Port pins.
        1: UART0 TX routed to P0.0, and RX routed to P0.1.
Bit1:   SPI0EN: SPI0 Bus I/O Enable Bit.
        0: SPI0 I/O unavailable at Port pins.
        1: SPI0 SCK, MISO, MOSL, and NSS routed to 4 Port pins.
Bit0:   SMB0EN: SMBus0 Bus I/O Enable Bit.
        0: SMBus0 I/O unavailable at Port pins.
        1: SMBus0 SDA and SCL routed to 2 Port pins.

Figure 1-10 XBR0: Port I/O Crossbar Register 0

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
SYSCKE	T2EXE	T2E	INT1E	T1E	INT0E	T0E	CP1E	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xE2

Bit7:   SYSCKE: /SYSCLK Output Enable Bit.
        0: /SYSCLK unavailable at Port pin.
        1: /SYSCLK routed to Port pin.
Bit6:   T2EXE: T2EX Input Enable Bit.
        0: T2EX unavailable at Port pin.
        1: T2EX routed to Port pin.
Bit5:   T2E: T2 Input Enable Bit.
        0: T2 unavailable at Port pin.
        1: T2 routed to Port pin.
Bit4:   INT1E: /INT1 Input Enable Bit.
        0: /INT1 unavailable at Port pin.
        1: /INT1 routed to Port pin.
Bit3:   T1E: T1 Input Enable Bit.
        0: T1 unavailable at Port pin.
        1: T1 routed to Port pin.
Bit2:   INT0E: /INT0 Input Enable Bit.
        0: /INT0 unavailable at Port pin.
        1: /INT1 routed to Port pin.
Bit1:   T0E: T0 Input Enable Bit.
        0: T0 unavailable at Port pin.
        1: T0 routed to Port pin.
Bit0:   CP1E: CP1 Output Enable Bit.
        0: CP1 unavailable at Port pin.
        1: CP1 routed to Port pin.

Figure 1-11 XBR1: Port I/O Crossbar Register 1

36

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
WEAKPUD	XBARE	-	T4EXE	T4E	UART1E	EMIFLE	CNVSTE	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xE3

Bit7: WEAKPUD: Weak Pull-Up Disable Bit.
0: Weak pull-ups globally enabled.
1: Weak pull-ups globally disabled.
Bit6: XBARE: Crossbar Enable Bit.
0: Crossbar disabled. All pins on Ports 0, 1, 2, and 3, are forced to Input mode.
1: Crossbar enabled.
Bit5: UNUSED. Read = 0, Write = don't care.
Bit4: T4EXE: T4EX Input Enable Bit.
0: T4EX unavailable at Port pin.
1: T4EX routed to Port pin.
Bit3: T4E: T4 Input Enable Bit.
0: T4 unavailable at Port pin.
1: T4 routed to Port pin.
Bit2: UART1E: UART1 I/O Enable Bit.
0: UART1 I/O unavailable at Port pins.
1: UART1 TX and RX routed to 2 Port pins.
Bit1: EMIFLE: External Memory Interface Low-Port Enable Bit.
0: P0.7, P0.6, and P0.5 functions are determined by the Crossbar or the Port latches.
1: If EMI0CF.4 = '0' (External Memory Interface is in Multiplexed mode)
P0.7 (/WR), P0.6 (/RD), and P0.5 (ALE) are 'skipped' by the Crossbar and their output states are determined by the Port latches and the External Memory Interface.
1: If EMI0CF.4 = '1' (External Memory Interface is in Non-multiplexed mode)
P0.7 (/WR) and P0.6 (/RD) are 'skipped' by the Crossbar and their output states are determined by the Port latches and the External Memory Interface.
Bit0: CNVSTE: External Convert Start Input Enable Bit.
0: CNVSTR unavailable at Port pin.
1: CNVSTR routed to Port pin.

Figure 1-12 XBR2: Port I/O Crossbar Register 2
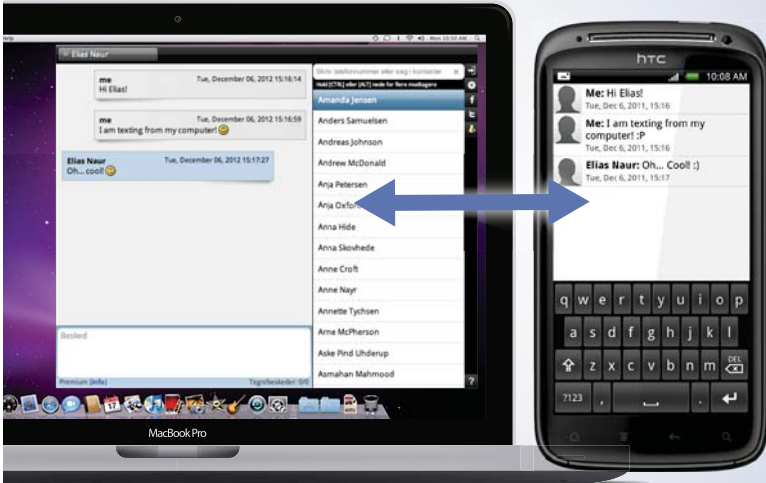
1.8.4   Input/Output ports and pin outs allocation

The C8051F020/1/2/3 devices have a wide array of digital resources which are available through the four lower I/O Ports: P0, P1, P2, and P3. As described in (Silicon Labs, 2003, pp. 162–181) each of the pins on P0, P1, P2, and P3, can be defined as a General-Purpose I/O (GPIO) pin or can be controlled by a digital peripheral or function (like UART0 or /INT1 for example), as shown in Figure 1-9. These pages of the Silicon Labs manual cover a very important section which should be fully understood by the user so as to be able to configure the device and its peripherals properly. There are also software aids which are also freely supplied by Silicon Labs where you select the devices that you want to use, and the correct register configurations are displayed, ready to copy and paste on to your program. The Simplicity Studio package, for example provides one-click access to design tools, documentation, software and support resources for EFM32, EFM8, 8051, Wireless MCUs and Wireless SoCs devices.

1.8.4.1 Configuring the Output Modes of the Port Pins

The output drivers on Ports 0 through 3 remain disabled until the Crossbar is enabled by setting XBARE (XBR2.6) to logic 1. The output mode of each port pin can be configured as either Open-Drain or Push-Pull; the default state is Open-Drain.

In the Push-Pull configuration, writing logic 0 to the associated bit in the Port Data register will cause the Port pin to be driven to GND, and writing logic 1 will cause the Port pin to be driven to VDD. In the Open-Drain configuration, writing logic 0 to the associated bit in the Port Data register will cause the Port pin to be driven to GND, and logic 1 will cause the Port pin to assume a high-impedance state.

The Open-Drain configuration is useful to prevent contention between devices in systems where the Port pin participates in a shared interconnection in which multiple outputs are connected to the same physical wire (like the SDA signal on an SMBus connection).

The output modes of the Port pins on Ports 0 through 3 are determined by the bits in the associated PnMDOUT registers (see Figure 1-14 giving the register related to Port 0. Similar registers are available for the other ports). For example, logic 1 in P0MDOUT.7 will configure the output mode of P0.7 to Push-Pull; logic 0 in P0MDOUT.7 will configure the output mode of P0.7 to Open-Drain. All Port pins default to Open-Drain output.

The PnMDOUT registers control the output modes of the port pins regardless of whether the Crossbar has allocated the Port pin for a digital peripheral or not. The exceptions to this rule are that the Port pins connected to SDA, SCL, RX0 (if UART0 is in Mode 0), and RX1 (if UART1 is in Mode 0) are always configured as Open-Drain outputs, regardless of the settings of the associated bits in the PnMDOUT registers.

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0	11111111
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
							(bit addressable)	0x80

Bits7-0: P0.[7:0]: Port0 Output Latch Bits.
(Write - Output appears on I/O pins per XBR0, XBR1, XBR2, and XBR3 Registers)
0: Logic Low Output.
1: Logic High Output (open if corresponding P0MDOUT.n bit = 0).
(Read - Regardless of XBR0, XBR1, XBR2, and XBR3 Register settings).
0: P0.n pin is logic low.
1: P0.n pin is logic high.

Note: P0.7 (/WR), P0.6 (/RD), and P0.5 (ALE) can be driven by the External Data Memory Interface.

Figure 1-13 P0

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
								00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
								0xA4

Bits7-0: P0MDOUT.[7:0]: Port0 Output Mode Bits.
0: Port Pin output mode is configured as Open-Drain.
1: Port Pin output mode is configured as Push-Pull.

Note: SDA, SCL, and RX0 (when UART0 is in Mode 0) and RX1 (when UART1 is in Mode 0) are always configured as Open-Drain when they appear on Port pins.

Figure 1-14 P0MDOUT

1.8.4.2 Configuring Port Pins as Digital Inputs

A Port pin is configured as a digital input by setting its output mode to "Open-Drain" and writing a logic 1 to the associated bit in the Port Data register. For example, P0.7 is configured as a digital input by setting P0MDOUT.7 to logic 0 and P0.7 to logic 1. If the Port pin has been assigned to a digital peripheral by the Crossbar and that pin functions as an input (for example RX0, the UART0 receive pin), then the output drivers on that pin are automatically disabled.

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
								11111111
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
								0xBD

Bits7-0: P1MDIN.[7:0]: Port 1 Input Mode Bits.
0: Port Pin is configured in Analog Input mode. The digital input path is disabled (a read from the Port bit will always return '0'). The weak pull-up on the pin is disabled.
1: Port Pin is configured in Digital Input mode. A read from the Port bit will return the logic level at the Pin. The state of the weak pull-up is determined by the WEAKPUD bit (XBR2.7).
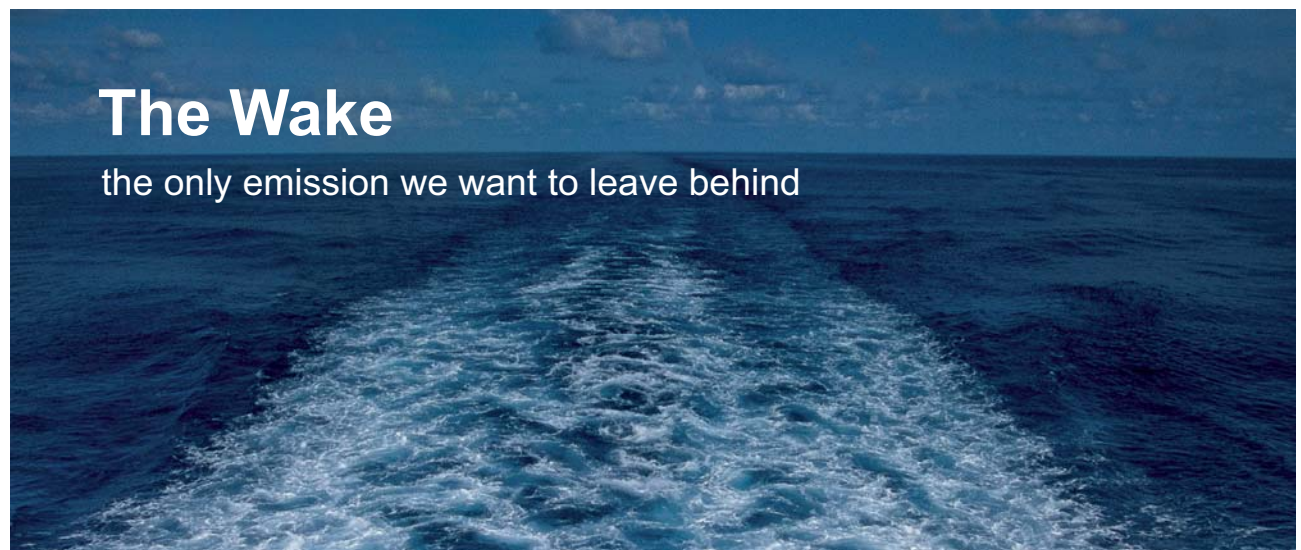
Figure 1-15 P1MDIN

1.8.4.3 Weak Pull-ups

By default, each Port pin has an internal weak pull-up device enabled which provides a resistive connection (about 100 kΩ) between the pin and VDD. The weak pull-up devices can be globally disabled by writing logic 1 to the Weak Pull-up Disable bit, (WEAKPUD, XBR2.7). The weak pull-up is automatically deactivated on any pin that is driving logic 0; that is, an output pin will not contend with its own pull-up device. The weak pull-up device can also be explicitly disabled on a Port 1 pin by configuring the pin as an Analogue Input.

1.8.5 Additional External Interrupts (IE6 and IE7)

In addition to the external interrupts /INT0 and /INT1, whose Port pins are allocated and assigned by the Crossbar, P3.6 and P3.7 can be configured to generate edge sensitive interrupts; these interrupts are configurable as falling- or rising-edge sensitive using the IE6CF (P3IF.2) and IE7CF (P3IF.3) bits. When an active edge is detected on P3.6 or P3.7, a corresponding External Interrupt flag (IE6 or IE7) will be set to logic 1 in the P3IF register (See Fig). If the associated interrupt is enabled, an interrupt will be generated and the CPU will vector to the associated interrupt vector location.

1.8.6 IE, EIE1 and EIE2 (Interrupt Enable)

The Interrupt Enable SFRs are used to enable and disable specific interrupts. Since this controller has 21 interrupt sources, apart from the reset, there are three separate registers available to handle the interrupt enabling system and these are the IE, EIE1 and EIE2 SFRs. The bits controlling the interrupts are listed in the Enable flag column of Table 1-5.

1.8.7 IP (Interrupt Priority)

The Interrupt Priority SFRs are used to specify the relative priority of each interrupt. Three SFRs (IP, EIP1 and EIP2) are available to handle the priority settings as shown in Table 1-5. 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.
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 (at a low priority setting) 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.

The priority order column is used to discriminate between interrupts with the same high/low priority setting which happen to occur exactly at the same time[2].

Interrupt Source	Interrupt Vector	Priority Order	Pending Flag	Bit addressable?	Cleared by HW?	Enable Flag	Priority Control
Reset	0x0000	Top	None	N/A	N/A	Always Enabled	Always Highest
External Interrupt 0 (/INT0)	0x0003	0	IE0 (TCON.1)	Y	Y	EX0 (IE.0)	PX0 (IP.0)
Timer 0 Overflow	0x000B	1	TF0 (TCON.5)	Y	Y	ET0 (IE.1)	PT0 (IP.1)
External Interrupt 1 (/INT1)	0x0013	2	IE1 (TCON.3)	Y	Y	EX1 (IE.2)	PX1 (IP.2)
Timer 1 Overflow	0x001B	3	TF1 (TCON.7)	Y	Y	ET1 (IE.3)	PT1 (IP.3)
UART0	0x0023	4	RI0 (SCON0.0) TI0 (SCON0.1)	Y		ES0 (IE.4)	PS0 (IP.4)
Timer 2 Overflow (or EXF2)	0x002B	5	TF2 (T2CON.7)	Y		ET2 (IE.5)	PT2 (IP.5)
Serial Peripheral Interface	0x0033	6	SPIF (SPI0CN.7)	Y		ESPI0 (EIE1.0)	PSPI0 (EIP1.0)
SMBus Interface	0x003B	7	SI (SMB0CN.3)	Y		ESMB0 (EIE1.1)	PSMB0 (EIP1.1)
ADC0 Window Comparator	0x0043	8	AD0WINT (ADC0CN.2)	Y		EWADC0 (EIE1.2)	PWADC0 (EIP1.2)
Programmable Counter Array	0x004B	9	CF (PCA0CN.7) CCFn (PCA0CN.n)	Y		EPCA0 (EIE1.3)	PPCA0 (EIP1.3)
Comparator 0 Falling Edge	0x0053	10	CP0FIF (CPT0CN.4)			ECP0F (EIE1.4)	PCP0F (EIP1.4)
Comparator 0 Rising Edge	0x005B	11	CP0RIF (CPT0CN.5)			ECP0R (EIE1.5)	PCP0R (EIP1.5)
Comparator 1 Falling Edge	0x0063	12	CP1FIF (CPT1CN.4)			ECP1F (EIE1.6)	PCP1F (EIP1.6)
Comparator 1 Rising Edge	0x006B	13	CP1RIF (CPT1CN.5)			ECP1R (EIE1.7)	PCP1F (EIP1.7)
Timer 3 Overflow	0x0073	14	TF3 (TMR3CN.7)			ET3 (EIE2.0)	PT3 (EIP2.0)
ADC0 End of Conversion	0x007B	15	AD0INT (ADC0CN.5)	Y		EADC0 (EIE2.1)	PADC0 (EIP2.1)
Timer 4 Overflow	0x0083	16	TF4 (T4CON.7)			ET4 (EIE2.2)	PT4 (EIP2.2)
ADC1 End of Conversion	0x008B	17	AD1INT (ADC1CN.5)			EADC1 (EIE2.3)	PADC1 (EIP2.3)
External Interrupt 6	0x0093	18	IE6 (P3IF.5)			EX6 (EIE2.4)	PX6 (EIP2.4)
External Interrupt 7	0x009B	19	IE7 (P3IF.6)			EX7 (EIE2.5)	PX7 (EIP2.5)
UART1	0x00A3	20	RI1 (SCON1.0) TI1 (SCON1.1)			ES1	PS1
External Crystal OSC Ready	0x00AB	21	XTLVLD (OSCXCN.7)			EXVLD (EIE2.7)	PXVLD (EIP2.7)

Table 1-5 Interrupt Summary

2 PaulOS F020: a co-operative RTOS

The PaulOS (PAUL's Operating System) F020 is the same co-operative RTOS as described in (Debono, 2013a) but with some modifications in order to work with the Silicon Laboratories C8051F020 micro-controller. This RTOS can easily be modified to accommodate other types of devices from the wide range of mixed-signal microcontroller units (MCUs) produced by Silicon Labs. Mainly it would involve checking the availability of the timers and interrupts present on the device and modifying the RTOS accordingly.

This is the main RTOS which we regularly use during the year with our students. It is also heavily used for the students' final year theses and it has therefore 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 controls when to give up the processor time to allow other tasks to run by issuing certain operating system (OS) commands to cause it to go to the WAIT state.

The idea for writing an RTOS for the 8051 had been brewing in my mind for quite some time, prompted by the desire to provide a simple RTOS for student use. It was further given a boost after coming across a book on C and the 8051 (Schultz, 1999)[3].This RTOS is a direct adaptation of my previous 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 other micro-controllers. The main task of translating it from assembly to C was undertaken years ago as a final year engineering degree thesis (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 family of micro-controllers.

Naturally there are some memory space and speed penalties to pay for the versatility obtained with an RTOS written in C rather than directly in assembly language. 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 2.3.14. The full source program can be found in appendix D.

2.1   Description of the RTOS Operation

The PaulOS_F020 RTOS is a co-operative RTOS and hence, as explained in the RTOS chapter (Debono, 2013a), 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 (or task) at a time and hence this task swapping RTOS only gives the impression of having tasks running simultaneously or concurrently. 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.
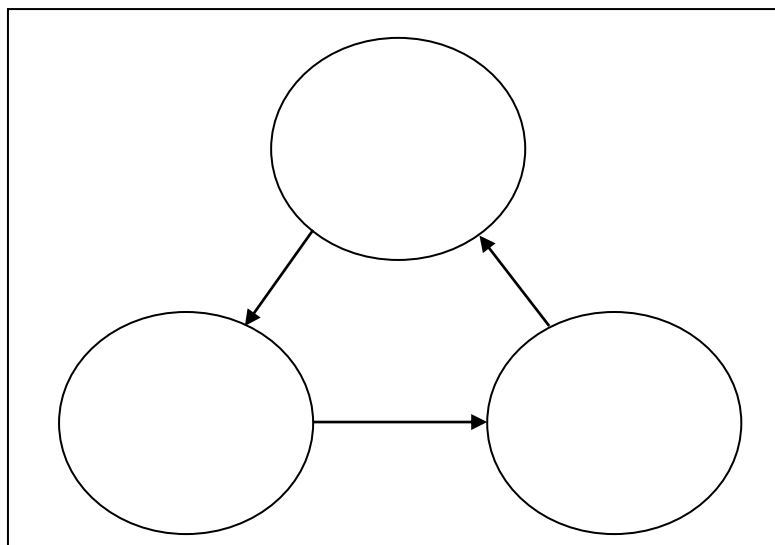
Figure 2-1 RTOS Task states diagram

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 RTOS 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 on to the Ready queue (see Figure 2-1) or whether a task swap is required 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 initiated by the RTOS. The currently running task is then usually marked as being in the Waiting queue (waiting for one or two ticks say) and the first task in the Ready queue takes over, with its stack and registers environment being copied into the working area. The environment of the old task is copied to the external memory store area for later retrieval.

The idea behind the PaulOS RTOS is that any task (a function or a routine in a program, which is normally an endless loop) can be in any one of three states as shown in Figure 2-1, 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( ) function as the running task, which in most case would be actually doing nothing, just putting the micro-controller in the idle or sleep state so as to conserve power. This will be interrupted at any time by the RTOS, taking it out of the idle mode, as soon as a task becomes ready to run and the RTOS then executes the swap.

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 (or OS_WAITT_A(min, sec, msec)) command.
- an OS_DEFER command which is actually just the normal OS_WAITT(..) with 2 ticks as the parameter, i.e. OS_WAITT(2) – wait for 2 ticks.
- a specified amount of time delay, selected by the user with OS_PERIODIC (or OS_PERIODIC_ (min, sec, msec)) 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(ticks) (or OS_WAITS_A(min, sec, msec)) 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 put in a state to wait indefinitely, effectively behaving as if the task did not exist. This is specified by the OS_KILL_IT command.

READY

It can also be in the READY QUEUE where it would be waiting for its turn to execute. This can be visualised in Figure 2-1 which shows how the tasks can move from one state to another. The RTOS, when permitted to do so, will select the top task (first in – first out) from this queue to execute instead of the currently running task, which would then be placed in the waiting queue. This RTOS also has the capability, if it is enabled, to sort the tasks in the Ready Queue according to their task number, so as to place the task with the lowest number (highest priority) at the top of the queue, so that it would be the chosen task to run at the first opportunity. This is further explained when discussion the OS_RTOS_ GO(priority) command in section 2.3.3.

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 device interrupt from other timers, UARTs, ADCs etc 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 and the stack pointer SP too have 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 F020 RTOS.

2.2   PaulOS_F020.C System Commands
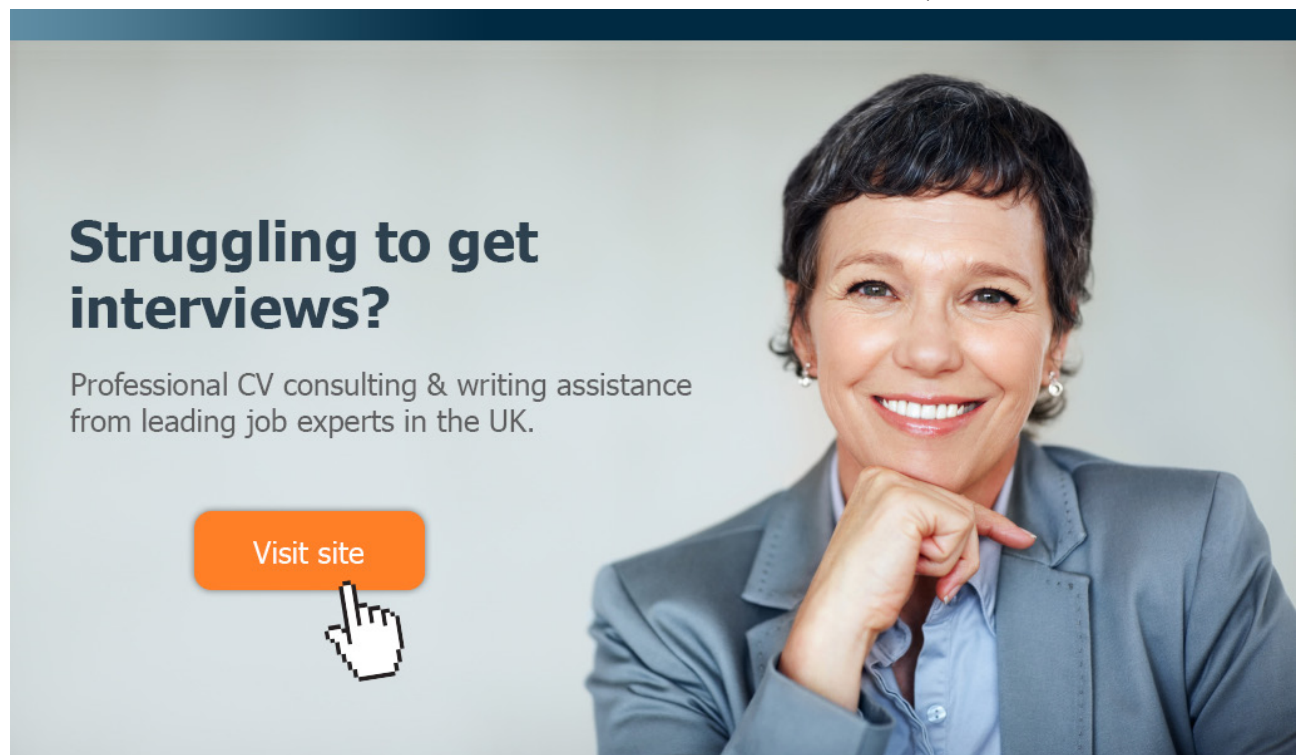
We now list and explain all the 14 PaulOS_F020 RTOS system commands. These are first listed or grouped according to whether or not they take any parameters. The list is then repeated, this time sorted according to whether the command causes a task swap or not.

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.

The following RTOS system calls do receive parameters:

- OS_CREATE_TASK (uchar tasknum, uint taskadd);   // Creates a task
- OS_INIT_RTOS (uchar blank);   // Initialises RTOS variables, parameter blank is not used at all
- 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 call *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 blank);   // Initialises all RTOS variables, parameter not
                                   // actually used
- 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_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

2.3   Descriptions of the commands

The F020 version of this RTOS provides some variations from the previous basic PaulOS RTOS, described on (Debono, 2013a, page 200). The detailed description of the commands is once again being give here, which would completely describe the PaulOS F020 RTOS. The complete source program can be found in the Appendix A and examples are given at the end of this chapter which should make it easier to understand. The variables mentioned in the explanations of the various RTOS commands can all be found in the Appendix A listings.

2.3.1 OS_INIT_RTOS(0)

This system command must be the **first** command to be issued in the main program in order to initialise the RTOS variables and parameters. It is called from the main program and takes an unsigned char parameter just for the sake of keeping the same format as that used in the previous basic PaulOS RTOS. The parameter as such is not used in the OS_INIT_RTOS function, and is therefore normally given a value of zero. An example of the syntax used for this command is:

```
OS_INIT_RTOS(0);
```

which would initialise all the required RTOS system parameters.

This system command performs the following operations:

- Clears the external memory area which is going to be used to store the stack of each task.
- Sets up the Interrupt Enable registers, depending on the TICK_TIMER parameter set in the parameter header file.
- 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 A 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.
- 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 internal 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.

2.3.2   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 MotorOn( ) procedure or function.

This system command performs the following operations:

- 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, so as to compensate for the pops.

2.3.3   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 those tasks which are ready to execute, just waiting for the currently running task to give up its place), 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 this 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 this RTOS, so as all the other tasks in the Ready Queue would be sorted above 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.

Depending on which timer is being used to generate the ticktime, this system command performs the following operations:

- 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 the reload values of the tick timer in use with the calculated value in order to obtain the required delay between timer overflow interrupts. The value used depends on the crystal frequency used on the board. Stores the reference time signal parameter in GOPARAM and TICKCOUNT.
- Starts the timer.
- Enables interrupts.
- Sets the timer overflow interrupt flag, thus causing the first interrupt immediately, and hence the timer counter registers are then loaded with the correct values in the timer ISR.

If the system is using a different clock setting, the values would be adjusted accordingly by the RTOS.

2.3.4 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 */
```

2.3.5 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 */
```

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 */

}
```

2.3.6 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.

This system command performs the following operations:

- It first checks whether the called task was already waiting for a signal.
- If the called or signalled (the intended destination task of the signal) task was not waiting, it sets its waiting for signal (SIGW) flag and exits to continue the same task.
- If the signalled 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
```

2.3.7   OS_PERIODIC(Ticks) or OS_PERIODIC_A(min, sec , msec)

This command initialises the task so as to make it repeat periodically every certain number of ticks. This number is 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 2.3.8. An example of its usage is also given in that same sub-section.

The command OS_PERIODIC_A(min, sec, msec) is a macro which makes the command OS_ PERIODIC(ticks) more user-friendly. It is explained in section 2.3.14.

We now deal with the commands that do perform a voluntary (co-operative) change of task:

2.3.8   OS_WAITP( )

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 operations:

- 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( ) or OS_PERIODIC_A(min, sec, ms) command and an example of its usage is shown below in Figure 2-2:

```
        OS_PERIODIC(50);                              // declare task as wishing to execute every 50 ticks

        // or OS_PERIODIC_A(0,0,300);;          // declare task with a periodicity of 300ms

         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

        }
```

Figure 2-2 Part listing of a periodic task

2.3.9   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 2.3.15.

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 still resume execution. A timeout of 0 would on the other hand still leave the task forever waiting for the interrupt. 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 2.3.10.

This system command performs the following operations:

- 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.

2.3.10   OS_WAITS(Timeout) or OS_WAITS_A(min, sec, msec))

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 signalled 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 force the task 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 operations:

- 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:
    o It sets its own waiting for signal (SIGW) flag.
    o It also sets the waiting for timeout variable according to the supplied parameter.
    o It then 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_WAITS(50);
// wait for a signal within 50 units or ticks, the value of the unit depends on
// the TICKTIME parameter used.
// or OS_WAITS_A(0,0,250);
// wait for a signal within 250ms
```

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.

The OS_WAITS_A(min, sec, msec) is a macro which makes the command OS_WAITS(ticks) more user friendly. It is explained in section 2.3.14.

2.3.11 OS_WAITT(Timeout) or OS_WAITT_A(min, sec, msec))

This system command is called by a task to sleep and wait for a specified timeout period. In the case of OS_WAITT(timeout) the timeout period is in units whose value depends on the TICKTIME parameter used. Valid values for the timeout period are in the range of 1 to 65535. A value of 0 is reserved for the OS_KILL_IT command, meaning permanent sleep, and therefore it 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 operations:

- 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 timeout for 60 units, the value of the unit depends on
// the TICKTIME parameter used.
// or OS_WAITT_A(0,1,0);
// Wait for 1 second
```

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.

The OS_WAITT_A(min, sec, msec) is a macro which makes the command OS_WAITT(ticks) more user-friendly by specifying the timeout in familiar minutes, seconds and milliseconds rather than in ticks. It is further explained in section 2.3.14.

2.3.12   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 exceptionally allowed timeout value of 0. The task is then placed permanently waiting and never resumes execution.

This system command performs the following operations:

- 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 (waits forever) and a new task (or main( ) would take over.*/
```

2.3.13   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 in the Waiting Queue to wait for two ticks while a new task (if ready) resumes execution.

This system command performs the following operations:

- It sets its timeout period in the TTS table to 2, which is the magic number the RTOS uses to describe any non-timing task.
- It places the task in the Waiting Queue.
- It then flows on to the QSHFT routine in order to start the task next in line.

An example of the syntax used for this command is:

```
OS_DEFER( );
/* The task simply stops execution and is placed in the Waiting Queue.*/
/* A new task would then take over. */
```

2.3.14   Enhanced event-waiting and other add-on MACROS

OS_WAITT, OS_WAITS and OS_PERIODIC functions are easily modified to make them accept absolute time, in minutes, seconds and milliseconds rather than ticks as a parameter. These macros (#define statements) perform the same functions of the OS_WAITT, OS_WAITS and OS_PERIODIC calls but rather than a tick parameter, they accept absolute time values as three parameters in terms of minutes, seconds and milliseconds, thus making the commands more user-friendly. This difference is denoted by the _A suffix (the A standing for Absolute) – e.g. 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 depending on the TICKTIME value chosen to give the required number of ticks equivalent to a 300ms delay. These macro-commands make the conversion from absolute time to ticks.

The range of possible values (65535 TICKTIMES) accepted is listed below, showing the maximum time in minutes:seconds.milliseconds:

Using a minimum TICKTIME of 1ms:

Range from 0:00.001 to 1:05.535 in steps of 1ms.

Using a TICKTIME of 10ms:

Range from 0:00.010 to 10:55.350 in steps of 10ms.

Using a maximum TICKTIME of 50 ms:

Range from 0:00.050 to 54:36.750 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(ticks) by virtue of how the OS_WAITT(ticks), OS_WAITS(ticks) and OS_PERIODIC(ticks) calls were written. In the case of the OS_WAITT(ticks) and OS_PERIODIC(ticks) 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. Can be used at the start of a stand-alone ISR
OS_RESUME_RTOS( )	// Re-enable the RTOS. Can be used at the end of a stand-alone ISR
OS_CPU_IDLE( ))	// Sets the µC in idle mode in PCON SFR. This is usually used
	// in the main program endless loop after initialising and starting the
	// RTOS. CPU wakes up at every interrupt, timers still running
OS_CPU_DOWN( )	// Sets the µC in power-down mode in PCON SFR

2.3.15   Stand-alone Interrupt Service Routines

In the PaulOS_F020 RTOS, a simple method of having one or more stand-alone interrupt service routines (ISRs) 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_F020_Parameters.H file. For example if we intend to have an ISR running under the EXT 0 (i.e. INT0) interrupt (and not under RTOS control as a task with an OS_WAITI(0) command), then we have to make sure to set to '1' the corresponding #define statements in PaulOS_F020_Parameters.H file.

```
#define STAND_ALONE_ISR_00 1 // EXT0 – set to 1 if using this interrupt as a stand alone ISR
```

Then as shown in Figure 2-3 the INT0 ISR itself also includes the commands OS_PAUSE_RTOS( ) when starting the ISR and then OS_RESUME_RTOS( ) at the end 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 also best to use register banks 2 or 3 for these ISRs.

```
void ISR_EXT0 (void) interrupt 0 using 2   // using register bank 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

 }
```

Figure 2-3 Example of a stand-alone ISR, interrupting the RTOS and executing immediately when the interrupt occurs

2.4   PaulOS_F020_Parameters.h header file

This is the RTOS parameters header file. We would mainly just need to set the TICK_TIMER, TICKTIME and NOOFTAKS parameters to reflect out particular application program. If we intend to use some stand-alone ISR, then that particular interrupt has to be selected (set to '1') in this header file as explained earlier on in section 2.3.15.

Note that the C8051F020.h used in these programs is a modified version of the standard C8051F02x header file. A complete listing is given in (A.5 C8051F020.H) which includes extra sfr16 declarations[4] and name definitions of individual bits of all the SFRs, even those that are not bit addressable. With these extra definitions, these 'non-addressable' bits can be set or cleared using bit-wise OR and bit-wise AND. Using the standard header file, certain instructions would not work since the named bit or SFR would not have been pre-defined.

```c
#ifndef _PaulOS_F020_Parameters_H_
#define _PaulOS_F020_Parameters_H_

/*
********************************************************************************
*                           RTOS KERNEL HEADER FILE
*
*                           PaulOS_F020_Parameters.H
*
* For use with PaulOS_F020.C - Co-Operative RTOS written in C
* based on PaulOS by Ing. Paul P. Debono
* for use with the 8051 family of micro-controllers
*
* File     : PaulOS_F020_Parameters.H
* Revision : 10
* Date     : Revised for C8051F020 February 2015
* By       : Paul P. Debono
*
*                    University Of Malta
*
********************************************************************************
*/


/*
********************************************************************************
*                           DATA TYPE DEFINITIONS
********************************************************************************
*/

#define TICK TIMER  2   // Set to 0,1,2 or 3, make sure not to clash with UART baud rate
timer
#define TICKTIME 1  // Length of RTOS basic tick in ms - refer to the RTOS timing definitions
// suitable values are: 1, 2, 4, 5, 8, 10, 20, 25

#define NOOFTASKS      65      // Number of tasks used in application


#define STACKSIZE     0x0F   // Number of bytes to allocate for the stack
// There is usually no need to change this parameter

/*
********************************************************************************
*/
/* Interrupt routines running as TASKS or as STAND-ALONE ISRa */
#define STAND_ALONE_ISR_00  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_01  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_02  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_03  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_04  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_05  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_06  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_07  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_08  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_09  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_10  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_11  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_12  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_13  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_14  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_15  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_16  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_17  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_18  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_19  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_20  0  // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_21  0  // set to 1 if using this interrupt as a stand alone ISR

/*
********************************************************************************
*/
#endif
```

63

2.5   Example using PaulOS_F020 RTOS

Finally, we now present an example to show the use of the RTOS in a simple program. This example using the PaulOS_F020 RTOS is partly taken from the Blinky example included with the KEIL IDE. It is implemented here with the modifications required in order to use the PaulOS_F020 RTOS. One task blinks the LED and a second task which keeps displaying the time every second via UART0.

```c
//-------------------------------------------------------------------------------
// Blinky.c
//-------------------------------------------------------------------------------
//
// AUTH: PD
// DATE: 21 FEB 15
//
// This program flashes the green LED on the C8051F020 target board
// and send Clock time via UART0
// Target: C8051F02x
//
//

//-------------------------------------------------------------------------------
// Includes
//-------------------------------------------------------------------------------
#include "C8051F020.h"          /* special function registers for C8051F020      */
#include "DualUarts.h"          /* Uarts header file      */
#include "PaulOS F020.h"        /* PaulOS system calls definitions   */
#include <stdio.h>
#include <stdlib.h>

//-------------------------------------------------------------------------------
// Global CONSTANTS
//-------------------------------------------------------------------------------

bit sio port = 0;   /* SIO port to use (0 = UART0, 1 = UART1) */

sbit  LED = P1^6;                       // green LED: '1' = ON; '0' = OFF

struct time  {                      /* structure of the time record      */
  unsigned char hour;              /* hour                              */
  unsigned char min;               /* minute                            */
  unsigned char sec;               /* second                            */
};

struct time ctime = { 12,  0,  0 };   /* storage for clock time values      */

//-------------------------------------------------------------------------------
// Function PROTOTYPES
//-------------------------------------------------------------------------------
void SYSCLK_Init (void);
void PORT_Init (void);
void DISABLE Watchdog (void);

//------------------------------------------------------------------------
// SYSCLK Init
//------------------------------------------------------------------------
//
// This routine initializes the system clock to use the 22.1184MHz crystal
// as its clock source.
//
void SYSCLK_Init (void)
{
```

```c
   unsigned int i;                          // delay counter

   OSCXCN = 0x67;                           // start external oscillator with
                                            // 22.1184MHz crystal

   for (i=0; i < 256; i++) ;                // wait for oscillator to start
#ifndef SIMULATOR
   while (!(OSCXCN & 0x80)) ;               // Wait for crystal osc. to settle
#endif                                      // not required during simulation
   OSCICN = 0x88;                           // select external oscillator as SYSCLK
                                            // source and enable missing clock
                                            // detector
}

//------------------------------------------------------------------------------
// DISABLE Watchdog
//------------------------------------------------------------------------------
//
// Disables the watchdog timer
//
void DISABLE_Watchdog (void)
{
   EA = 0;
   WDTCN = 0xDE;
   WDTCN = 0xAD;
   EA = 1;
}



//------------------------------------------------------------------------------
// PORT Init
//------------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports
//
void PORT_Init (void)
{
   XBR0    = 0x04;                    // Enable UART 0
   XBR1    = 0x00;
   XBR2    = 0x40;                    // Enable crossbar and weak pull-ups
   P0MDOUT |= 0x01;                   // enable TX0 as a push-pull output
   P1MDOUT |= 0x40;                   // enable P1.6 (LED) as push-pull output
}


/*
********************************************************************************
*/



/**********************************************************************/
/*        Task 0 'Blink'                                           */
/**********************************************************************/
void BlinkTask (void)
{
      OS_PERIODIC_A(0,0,500);                      /* Repeat every 500 ms */
while(1)
   {
      LED = !LED;
      OS_WAITP();
   }
}


/**********************************************************************/
/*        Task 1 'clock'                                           */
/**********************************************************************/
```

```
void clock (void)
{
  OS_PERIODIC_A(0,1,0);                 /* Repeat every 1 second */

  while (1)  {                          /* clock is an endless loop        */
    if (++ctime.sec == 60)  {           /* calculate the second            */
      ctime.sec = 0;
      if (++ctime.min == 60)  {         /* calculate the minute            */
        ctime.min = 0;
        if (++ctime.hour == 24)  {      /* calculate the hour              */
          ctime.hour = 0;
        }
      }
    }

    printf ("Clock Time: %02bu:%02bu:%02bu\r",      /* display time   */
                ctime.hour, ctime.min, ctime.sec);
    OS_WAITP();             /* wait for 1 second              */
  }
}

//--------------------------------------------------------------------------------
// MAIN Routine
//--------------------------------------------------------------------------------
void main (void) {

    DISABLE Watchdog ();
    SYSCLK Init ();
    SetUpUART(0, 115200, 1); /* Set up UART 0, at 115200 baud using Timer 1 */

    PORT Init ();

    OS_INIT_RTOS(0);          /* initialise RTOS, variables and stack  */

    OS CREATE TASK(0,BlinkTask);         /* CREATE  task                */
    OS_CREATE_TASK(1,clock);

    OS RTOS GO(0);                       /* Start multitasking, no priorities   */

    while (1)
      {
            #ifndef SIMULATOR
            OS_CPU_IDLE();   /* Go to idle mode if doing nothing, to conserve energy */
            #else
             ;
            #endif
      }

}
```

The LED blinking is handled by Task 0 or the 'BlinkTask' routine. Note that ant task has to be written as an endless routine. In this case, the task is declared as a periodic task with the OS_PERIODIC_A() command which is outside the endless loop and is thus executed only one at the beginning. In the 'while(1)' endless loop the LED is simply toggled on/off and then the task placed in the waiting queue waiting for the periodic interval to pass by issuing the OS_WAITP() RTOS command.

The second task is the 'clock' task which also runs periodically every one second. The endless loop handles the updating of the seconds, minutes and hours and sends the 'Clock Time:' string via UART0. The actual UART0 setup is not listed here but can be found in appendix B.2 UART0 and UART1.

The 'main' routine disables the watchdog timer, initialises the system clock, the UART0 and the RTOS. It then creates the task and then starts the multi-tasking by starting the RTOS. The main program then simply loops in the idle state.

3   Master – Slave RTOS

3.1   Multi-controller RTOSs

A number of micro-controllers can be networked together, each of them running its own RTOS but at the same time they would all be synchronised together. One controller would be acting as the master and all the others would be the slaves. The transmit pin from the master would be connected to all the receive pins of the slaves, while the transmit pins from the slaves would be connected to the master receive pin (normally via a diode). The drawing shown in Figure 3-2 represents the case where we have two slaves connected to the master. The number of slaves can be increased as required.
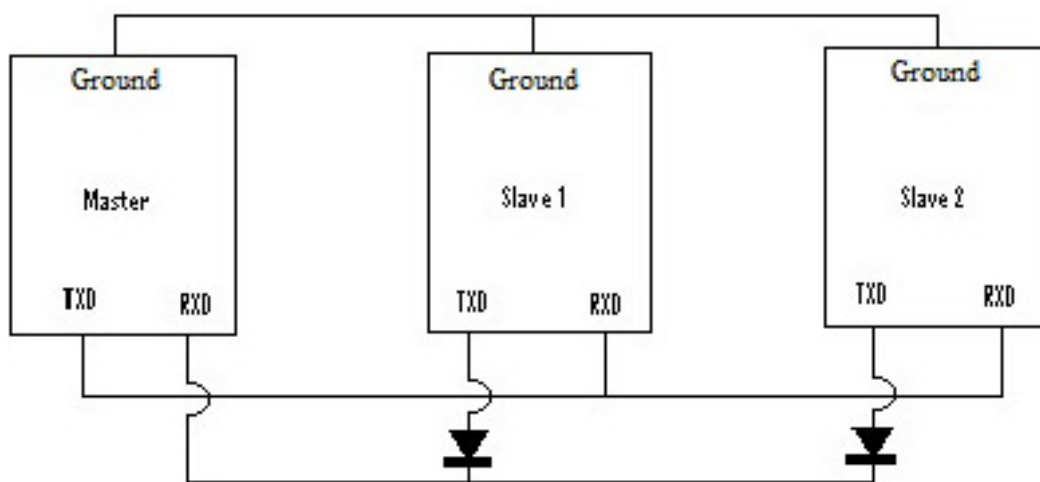
Figure 3-1 Networked micro-controllers using the UARTs to synchronise their RTOSs

Serial transmission between the master and the slave micro-controllers is used to synchronise the separate RTOSs running on the slaves with the RTOS running on a master board. The UART has a special mode dedicated for such board networking. Modes 2 (the baud rate is determined by the oscillator frequency) and 3 (the baud rate is determined by the timer overflows) of the UART provide asynchronous, full-duplex multiprocessor communications using 11 bits which are made up of 1 start bit, 8 data bits, 1 programmable additional 9th bit and a stop bit. Mode 3 is used more frequently since the baud rate can be easily programmed to one of the standard baud rates by loading the timer with the correct register reload value. For this explanation we shall use UART0 (UART1 does not support mode 2).

The 9th bit can be set to '1' or '0' by programming bit TB8 in register SCON0. Moreover, another bit in SCON0, called SM20, when set enables multiprocessor communications in modes 2 or 3. The slave boards would normally be programmed in mode 3 with UART0 interrupt enabled. When SM20 is set to '1', the receiver interrupt flag RI0 will *only* be set if the received 9th bit was a '1'. This particular behaviour is what makes this mode ideal for multiprocessor communications and in particular in our case for synchronising the RTOSs. Usually a 9th bit of 1 indicates an address byte and a 0 would indicate a data byte. The network protocol is best explained in the following points:

- The master and the slaves would all be programmed in mode 3 at the same baud rate.
- All the slaves would initially be set up so as to have their SM20 set to 1 so as to be in the so-called address listening mode and with their serial UART0 interrupt enabled. With this set up, all the slaves would have their RI0 set (meaning a serial interrupt request) only if and when they receive a character with the 9th bit set to 1.
- The master UART0 SM20 bit would be cleared to 0 and it would not be running under serial interrupt.
- Each slave would have a different address, stored in some variable.
- The master directs the activities of the slaves and therefore initiates the transmission.
- The master would set its 9th bit to 1 and starts the networking protocol by sending an address, normally of the first slave in the loop.

- All the slaves would receive this address and all would be requesting a serial interrupt. Their own serial ISR would then check and compare the received address byte with their own stored address.

- Only the slave whose address agrees with the received address would then clear its own SM20 bit to 0. From now on, its RI0 would still be set even if the received 9th bit is 0. If the serial interrupt is still left enabled, then only this particular slave would be 'serially interrupted' with any received bytes from now on, even though the received 9th bit of the data bytes would 0. The other slaves would still be expecting and address with a 9th bit set to 1.

- In most protocols, the addressed slave would normally then send an acknowledgement byte to the master as an 11-bit character with a 9th bit set to 0.

- The master can then send any required data bytes to this slave, always with the 9th bit cleared (=0) so that the other non-addressed slaves would not be getting any serial interrupt requests.

- A special pre-arranged end-of-message character, such as a '$' or '£' would be sent by the master at the end of the data transmissions, or else a previously agreed number of data bytes would be sent.

- Once the addressed slave receives the end-of-message character or the agreed number of data bytes, it would then set its SM2 back to 1 ready for the next address byte sent by the master.

- The master can then restart the sequence by sending the next slave address (again with the 9th bit set to 1).

For our particular case of RTOS synchronisation, some slight modification of the above protocol is made. The master RTOS would be running under its own timer generated ticks. Its own RTOS timer interrupt routine would have some additional code so as to send and receive bytes (or messages) to/from the slaves over a serial link. The slaves on the other hand would have their tick interrupt routine tied to the serial interrupt. As shown in the messaging sequence in Figure 3-2, this is a slightly different adaptation of the method described by (Pont, 2002, chapter 27) in the excellent book on time-triggered applications. At every master tick, the master would send an address of a slave (the slave address number changing at every tick). All the slaves would receive this address and their own UART setup would generate an interrupt which would be used as their RTOS tick generator routine. Hence all the slaves would have their own RTOSs synchronised with each other but they would be running one byte late relative to the master since the serial interrupt occurs *after* a byte is received, whereas the master's tick is generated *before* it transmits the address.

In the RTOS synchronisation protocol which could be changed as required according to the application, we also opt to disable serial interrupts in the addressed slave at this point. This is because the serial interrupts are being interpreted as the tick synchronisation pulse and only messages which could interrupt all the slaves would be treated as such. The same slave would clear its SM20 bit and then send an acknowledgement to the master (always as an 11-bit character with the 9th bit set to 0) and the slave would then stay waiting for further data transmissions from the master. The master, after receiving this acknowledgement from the addressed slave would then send a data byte or more to this same slave as required by the application (bit 9 still 0). After the agreed number of data bytes are sent, the addressed slave would revert back to the address listening mode by setting its SM20 bit back to 1. The periodic interval between the ticks must be long enough to enable all serial transmissions of the 3 bytes to take place and leave additional free time for the slave (and master) to do some other work connected with other tasks in the application in-between ticks. All these serial transmissions would be made using the 11-bit byte master-slave operation mode of the UART as described above and in (Debono, 2013a, p. 107). More detailed explanations of this protocol are given in the following sections of this chapter.

Figure 3-2 Serial communication between Master and two Slaves to synchronise the RTOSs

3.2   Master

The UART0 initialisation routine for the Master board is listed in Figure 3-3

```
/*************************************************************************/
/* Initialise the F020 8051 UART for multi-processor comms operations */
/* For the MASTER, it will not run under interrupt control */
void mp_UART_Master_init (unsigned long baudrate)
{

   ES0 = 0;                  /* Disable Serial Interrupt */

/* For baud rates 1200 to 460800 use Timer 1 as the baud rate generator */
/* We use the 22.1184 MHz Crystal */

/* Setup serial port control register SCON0 = 0xDA, not under interrupt control */
/* Mode 3: 9-bit UART, using timer 1 */
/* SM00=1, SM10=1, SM20 = 0, REN0 = 1, TB80 = 1, RB80 = 0, TI0 = 1, RI0 = 0 */
/* 1 1 0 1 1 0 1 0  = DA hex */

// A very fast baud rate might create problems when receiving data from the slaves,
// if we are using the diodes at the TxD pins of the slaves.
// Depends on diode quality. Length of cable also becomes critical.


/* For the MASTER program, TB80 will be set to 1 when sending an address,
                               and set to 0 when sending data.
*/

   SCON0  = 0xDA;                      // SCON0: mode 1, 9-bit UART, see above
   TMOD  &= 0x0F;
   TMOD  |= 0x20;                      // TMOD: timer 1, mode 2, 8-bit reload
   TH1    = -(SYSTEMCLK/baudrate/16UL); // set Timer1 reload value for baud rate
   TL1    = TH1;
   TR1    = 1;                         // start Timer1
   CKCON |= 0x10;                      // Timer1 uses SYSCLK as time base
   PCON  |= 0x80;                      // SMOD00 = 1
   #message "Using Timer 1 for F020 UART0 baud rate generator as the RTOS slave tick"
   #message "UART0 running in 9-bit multi-processor comms mode,"
   #message "baud parameter in PaulOS_F020_Master_Parameters.h"

}
/*************************************************************************/
```

Figure 3-3 Listing of the UART0 9-bit mode initialisation routine for the Master

The baud rate for the multi-processor communication serial port is usually set at a high baud rate so as not to waste much time in the transmission or reception of the address and data bytes. The Master UART0 would not be running under serial interrupt control and transmission is immediately started whenever an address or data is loaded into SBUF0.

The add-ons required in the Master RTOS tick-timer interrupt routine (executed for Timers 0, 1, 2 or 3 overflows depending on the TICK_TIMER selected) in order to handle this protocol are listed in Figure 3-4. These add-ons perform the sending of the Address (with bit 8 [meaning the 9th bit] set to 1) which causes a serial interrupt in ALL the slaves, thus triggering the RTOS interrupt routine in the slaves. This is the RTOS synchronising signal which would be received simultaneously by all the slaves. The master then waits for the acknowledgement from the addressed slave within a set timeout, marking it as dead if no acknowledgement is received. On receiving this acknowledgement, the master sends a byte of data (with bit 8 [meaning the 9th bit] set to 0) to the addressed slave which can contain any information as required by the application. This data byte will not cause any interrupt in any slave, because of the fact that the 9th bit is set to zero whenever sending data.

The sequence would then be repeated for the next slave and so on until it will loop back to the initial slave number to repeat the 'polling' ad infinitum.

```
/*
*******************************************************************************
*
* Function name : RTOS_Timer_Int
*
* Function type : Scheduler Interrupt Service Routine
*
* Description   : This is the RTOS scheduler ISR. It generates system ticks and calculates
any remaining
*                           waiting and periodic interval time for each task.
*
* Arguments     : None
*
* Returns       : None
*
*******************************************************************************
*/

#if (TICK_TIMER == 0)  /* If Timer 0 is used for the scheduler */
void RTOS_Timer_Int (void) interrupt 1 using 1
{
        uchar data k,s;                      /* Timer 0 is used, s is the slave address
variable*/
        ulong data t;            /* used for acknowledge timeout period */
        uchar data * idata q;    /* for scheduling.                    */
        bit   data On_Q;

        TH0 = BASIC_TICK / 256;      /* Timer registers reloaded    */
```

```
        TL0 = BASIC_TICK % 256;

#elif (TICK_TIMER == 1)        /* If Timer 1 is used for the scheduler */
void RTOS_Timer_Int (void) interrupt 3 using 1
{
        uchar data k,s;                   /* Timer 1 is used              */
        ulong data t;             /* used for acknowledge timeout period */
        uchar data * idata q;     /* for scheduling.                       */
        bit   data On_Q;

        TH1 = BASIC_TICK / 256;      /* Timer registers reloaded    */
        TL1 = BASIC_TICK % 256;


#elif (TICK_TIMER == 2)        /* If Timer 2 is used for the scheduler */
void RTOS_Timer_Int (void) interrupt 5 using 1
{
        uchar data k,s;                   /* Timer 2 is used */
        ulong data t;             /* used for acknowledge timeout period */
        uchar data * idata q;          /* for scheduling.                */
        bit   data On_Q;

        TF2 = 0;                          /* Timer 2 interrupt flag is cleared */

#elif (TICK_TIMER == 3)        /* If Timer 3 is used for the scheduler */
void RTOS_Timer_Int (void) interrupt 14 using 1
{
        uchar data k,s;                     /* Timer 3 is used */
        ulong data t;             /* used for acknowledge timeout period */
        uchar data * idata q;          /* for scheduling.                */
        bit   data On_Q;

        TMR3CN &= ~TF3;                  /* Timer 3 interrupt flag is cleared */

#endif

// start of Master/Slave add-ons

        // Master sends address (0 to NOOFSLAVES which is stored in parameters_master.h,
        // incremented each time)
        // Master receives ACK from the addressed slave
        // Master sends a byte of data to this slave

/*****************************************************************************/
// Add-on to send address to slaves  - reply expected within timeout, for this version
                         // s range is from 0 to (NOOFSLAVES - 1)
                         // TB80 initially set to one, when UART0 was initialised
        while(!TI0){}    // wait for any previous transmission to finish just in case.
                         // TI0 = 1, means ready to load new character in SBUF0 for Tx
        TB80 = 1;        // set bit 8 (9th bit), for address transmission
        TI0 = 0;         // clear TI0 since we are going to transmit
        SBUF0 = s;       // send slave address
                         // TB80 will be set to one, once the transmission is ready
        while(!TI0){}       // wait for transmission to finish.

/*****************************************************************************/

/*****************************************************************************/
// Add-on to receive acknowledgement from addressed slave
// If no acknowledgement is received within timeout period, then mark it with an * as dead
        t = 1000UL;
        while((!RI0) && (t != 0)){t--;}   // wait a while for acknowledgement within
                                          // timeout period
        if(RI0 == 1)                      // RI0 = 1, means received character in SBUF0
         {
           RI0 = 0;                // clear RI0
           ack[s] = SBUF0;         // read acknowledgement from slave, stored in array
         }                         // sorted and used also in application program
        else if(t==0)             // no acknowledgement received within timeout period
```

```
            ack[s] = '*';           // hence mark it as dead

/**********************************************************************/

/**********************************************************************/
// Add-on to send data to addressed slave
        TB80 = 0;                   // set bit 8 (9th bit), for data transmission
                // thus no interrupt will be generated on slaves, with their SM2 set to 1
        TI0 = 0;                    // clear TI0 since we are going to transmit
        SBUF0 = NetworkData[s]; // send slave some data, stored in the application program
                                    // TB80 will be set to one, once the transmission is ready
        s = (++s)%NOOFSLAVES;    // prepare address for next slave
/**********************************************************************/
// end of  Master/Slave add-ons
```

Figure 3-4 Part of the Master RTOS Tick Interrupt routine, showing the add-ons required for multi-board operations

A value of ack[s] equal to '*' would then be recognised by some task in the Master program to mean that the particular slave is not responding and can be acted upon accordingly.

3.3 Slave

The UART0 initialisation routine for the Slaves board is listed in Figure 3-5 which is very similar to the Master setup routine. However the Serial UART0 would in this case be running under serial interrupt control and an interrupt would be called whenever an address (with its 9th bit set to 1) is received into SBUF0.

```
/* Initialise the F020 8051 UART for multi-processor comms operations */
/* Uses timer 1 as the baud rate generator - under interrupt control for the SLAVE */

void mp_UART_Slave_init (unsigned long baudrate)
// Set up internal UART  under interrupt control
{

/* For baud rates 1200 to 460800 use Timer 1 as the baud rate generator */
/* We use the 22.1184 MHz Crystal */

/* Using Timer 1 for standard baud rate generation */
/* Setup serial port control register SCON = 0xF0 */
/* Mode 3: 9-bit UART var. baud rate */
/* SM00=1, SM10=1, SM20=1, REN0 = 1, TB80 = RB80 = TI0 = RI0 = 0 */
/*  1 1 1 1 0 0 0 0 = F0 hex, under interrupt control */

// A faster baud rate might create problems when receiving data from the slaves,
// since we are using the diodes at the TxD pins of the slaves.
// Depends on diode quality. Length of cable becomes critical.


/* For the SLAVE program, SM2 will be set to 1 when expecting an address,
                               and set to 0 when expecting data.
*/

   SCON0  = 0xF0;                      // SCON0: mode 1, 9-bit UART, see above
   TMOD  &= 0x0F;
   TMOD  |= 0x20;                      // TMOD: timer 1, mode 2, 8-bit reload
   TH1   = -(SYSTEMCLK/baud rate/16UL);   // set Timer1 reload value for baud rate
   TL1   = TH1;
   TR1   = 1;                          // start Timer1
   CKCON |= 0x10;                      // Timer1 uses SYSCLK as time base
   PCON  |= 0x80;                      // SMOD00 = 1

   ES0 = 1;                            /* Enable Serial Interrupts */

   #message "Using Timer 1 for F020 UART0 baud rate generator"
   #message "UART0 serial interrupt is used as the slave tick generator"

}
/***************************************************************************/
```

Figure 3-5 Listing of the UART0 initlisation routine for the Slaves. Note that the serial interrupt enable bit is set.

This serial interrupt would be acting as the main Tick interrupt for the slaves, and thus all the slaves would be initiating their RTOS_Interrupt routine at the same time, whenever any address is received. This routine, part of which is listed in Figure 3-6 would first check whether the address received corresponds to this particular slave. If the address is different, then the ISR would move on to the normal RTOS house-keeping chores.

If however, the address is its own, then an acknowledgement is sent to the Master and data is then received from the Master. As it is written, if the connection is lost and no data is received, the program hangs up, waiting for this never-coming data!

```
// Add-on for ticks arriving via UART0 from Master
#elif (TICK_TIMER == 999)     /* If Serial Receiver Interrupt is used for the scheduler */

void RTOS_Timer_Int (void) interrupt 4 using 1
{
        uchar data k;
        uchar data * idata q;
        bit  data On_Q;

        if ( (RI0 == 1) && (RB80 == 1))       // react to reception of an address
         {
          RI0 = 0;                             // reset the receiver interrupt flag
          RxAddress = SBUF0;      // read received address
          if (RxAddress == MyAddress) // If master is polling this slave
           {
             SM20 = ES0 = 0;        // disable serial interrupt and set SM20 to 0
        // so that although RI0 would still be set when data from Master is received
                                   // no interrupt will be generated in any other slave
        TI0 = 0;                    // Set transmitter busy
        SBUF0 = MyAck;  // send acknowledgement to Master, value declared in user program
        while(!TI0){}               // wait for transmission to finish

// now receive incoming data from Master

        while(!RI0){}       // wait for incoming data from Master
                            // RI0 = 1, means received data character in SBUF0
        RI0 = TI0 = 0;      // Clear all serial interrupts flags
        MyData = SBUF0;             // Read Data, used in application program
        SM20 = ES0 = 1;
        // Prepare for next address tick, enabling serial interrupts once again
}
#endif
```

Figure 3-6 Part of the RTOS_Timer_Int routine for the Slaves, running under Serial interrupt. This code could hang up if no data is received.

A timeout check can be added if required to eliminate this hang-up possibility, as adopted in the Master source code and shown in Figure 3-7 Part listing of the RTOS_Timer_Int slave routine showing the timeout modification during data reception Figure 3-7.

```
// now receive incoming data from Master

// If no data is received within timeout period, then mark it with an ~ as 'data not valid'
        t = 1000UL;
        while((!RI0) && (t != 0)){t--;}    // wait a while for data arrival within
                                           // timeout period
        if(RI0 == 1)                       // RI0 = 1, means received character in SBUF0
         {
            RI0 = TI0 = 0;     // clear RI0
          MyData = SBUF0;      // Read Data, used in application program
            SM20 = ES0 = 1;
          // Prepare for next address tick, enabling serial interrupts once again
         }
        else if(t==0)      // nothing received within timeout period
          MyData = '~';    // hence mark it as invalid data
```

Figure 3-7 Part listing of the RTOS_Timer_Int slave routine showing the timeout modification during data reception

A value of MyData equal to '~' would then be recognised by some other task to mean that the data received is invalid and can be acted upon accordingly.

Finally, in the RTOS Macros section of the slave header file PaulOS_F020_Slave.H, the following should be added:

```
    #elif (TICK_TIMER == 999)
    #define OS_PAUSE_RTOS()        ES0 = 0x00
    #define OS_RESUME_RTOS()       ES0 = 0x01
```

Figure 3-8 OS_PAUSE_RTOS( ) and OS_RESUME_RTOS( ) modification for the slave RTOS since it uses the serial interrupt as the tick generator.

so that the 'pause' and 'resume' OS commands will disable and enable the UART0 interrupt. Use of the OS_PAUSE_RTOS( ) command will naturally cause some temporary loss of synchronisation with the other slaves since the serial ISR would be delayed.

4   Programming Tips and Pitfalls

In this final chapter we discuss some programming tips and common pitfalls which should be avoided when programming such micro-controllers.

4.1   RAM size

The C8051F020 development target board has 64KB of flash memory (On-chip ROM) for code and constants and a 4K of RAM (On-chip XRAM). Thus the KEIL IDE should be setup as shown in Figure 4-1 to make use of this on board memory.
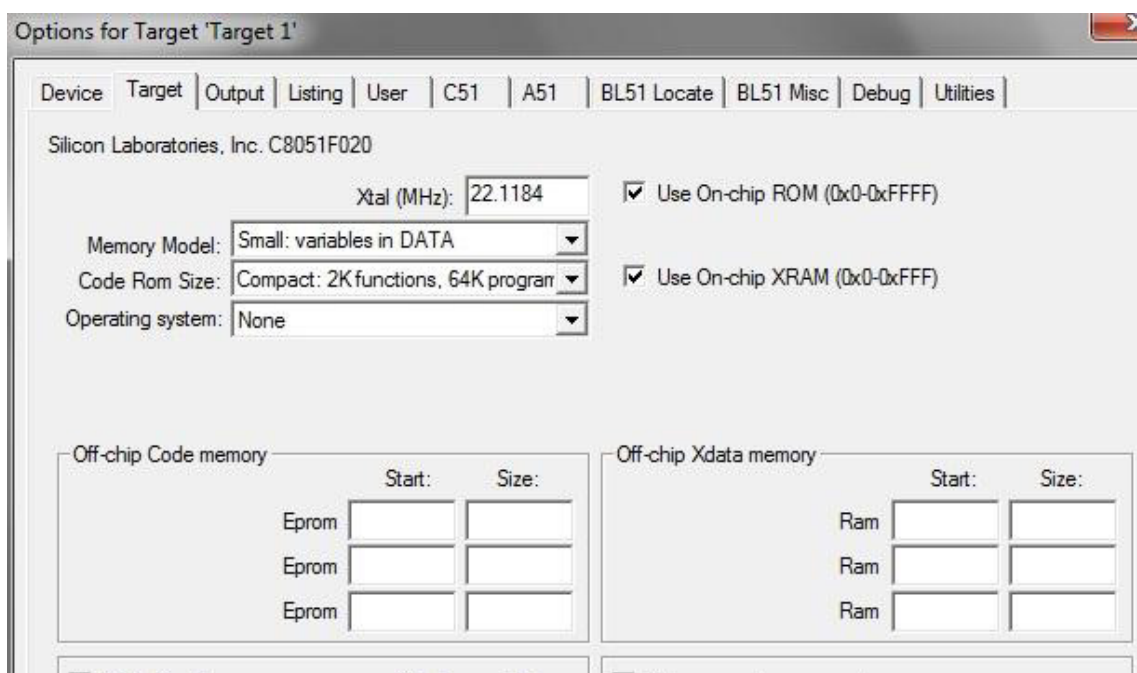
Figure 4-1 Screen shot of the Target Options setup

4.2   SFRs

SFRs are used to control the way the 8051 peripherals functions. Not all the addresses above 80h are assigned to SFRs. However, this area may not be used as additional RAM memory even if a given address has not been assigned to an SFR. Free locations are reserved for future versions of the micro-controller and if we use that area, then our program would not be compatible with future versions of the micro-controller, since those same locations might be used for special additional SFRs in the upgraded version. Moreover, certain unused locations may actually be non-existent, in the sense that the actual cells for that memory would not form part of the memory mask when being manufactured, and hence even if we do write the code to use these locations, no actual data would be stored!

78

It is therefore recommended that we do not read from or write to any SFR addresses that have not been actually assigned to an SFR. Doing so may provoke undefined behaviour and may cause our program to be incompatible with other 8051 derivatives that use those free addresses to store the additional SFRs for some new timer or peripheral included in the new derivative.

If we write a program that utilizes the new SFRs that are specific to a given derivative chip (and which therefore were not included in the standard basic 8051 SFR list), our program will not run properly on a standard 8051 where those SFRs simply did not exist. Thus, it is best to use non-standard SFRs only if we are sure that our program will only have to run on that specific micro-controller. If we happen to write code that uses non-standard SFRs and subsequently share it with a third-party, we must make sure to let that party know that our code is using non-standard SFRs and can only be used with that particular device. Good remarks, notes and warnings within the program source listing would help.

4.3   Setup faults

The setup during the initialisation is very critical and basically we would need to initialise the system clock, watchdog timer, crossbar registers, any input/output ports and whether we need to use them for digital or for analogue signals. And then of course, any timers, serial ports, ADC, DAC, SPIs etc would need to be initialised if they are going to be required in our application program. We now list some common faults which are easily made during this setup process.

4.3.1   System Clock Setup

The System clock should be setup and initialised at the start of your program. Forgetting to set it up is a common fault and also checking for the clock stabilisation during a simulation run can cause problems in cases where the simulation of the clock is not well implemented as mention in section 1.8.1.

4.3.2   Watchdog Timer Setup

Forgetting to disable the watchdog timer or disabling it late is a common fault with beginners to this device. The effect would be for the micro-controller to keep on resetting itself while executing the few initial commands in the main program.

4.3.3   Crossbar Setup

Another very common fault with newcomers to this device is setting the wrong configuration of the crossbar SFRs: XBR0, XBR1 and XBR2. Consulting the manual and reviewing the examples would help a lot to enable the user to become familiar with the initialisations required, and at which pins to expect the input or output signal to be available. (See Table 1-4, Figure 1-10, Figure 1-11 and Figure 1-12)

4.4 Serial ports (UART0 and UART1)

To use the 'printf' command, the on-board serial port or UART0 must be correctly setup at the required baud rate. It is generally necessary to initialise at least the following four SFRs: SCON0, PCON, SCON0, and TMOD. This is because SCON0 on its own does not fully control the serial port. However, in most cases the program will need to use one of the timers to establish the serial port baud rate. In this case, it would be necessary to configure Timer 1 by setting TH1, TL1 and TMOD. Another bit PCON.7 (known also as SMOD00 bit, but we should note that PCON is not a bit-addressable register), can be set to double the baud rate. In this case therefore, we would also need to program bit 7 of register PCON. This is shown in the example of Figure 4-2.

```
//--------------------------------------------------------------------------
// UART0_Init
//--------------------------------------------------------------------------
//
// Configure the UART0 using Timer1, for <baud rate> and 8-N-1.
//
void UART0_Init (void)
{
    SCON0  = 0x50;                  // SCON0: mode 1, 8-bit UART, enable RX
    TMOD  &= 0x0F;                  // clear Timer 1 control bits only
    TMOD  |= 0x20;                  // TMOD: set Timer 1: mode 2, 8-bit reload
    TH1    = -(SYSCLK/BAUDRATE/16); // set Timer1 reload value for required baud rate
    TR1    = 1;                     // start Timer1
    CKCON |= 0x10;                  // Timer1 uses SYSCLK as time base
    PCON  |= 0x80;                  // SMOD00 = 1
    TI0    = 1;                     // Indicate TX0 ready to transmit
}
```

Figure 4-2 UART0: Serial initialisation routine, not under interrupt control

Download free eBooks at bookboon.com

A common mistake is to forget to set TI0 to '1'. This indicates that the transmitter is ready to transmit. Failing to do so, the serial transmission would never start since the UART would think that it is still busy with some previous transmission. The TI0 bit would then be cleared in the putchar( ) routine.

Moreover, if the serial handling routine is to run under interrupt control, then the appropriate interrupt enable bits (ES and EA in the IE SFR) and sometimes even the interrupt priority bit (PS in the IP SFR) have also to be set. This would bring to six the number of SFRs which we may need to set in order to use the UART in interrupt mode.

Taking UART0 as an example, this time the TI0 flag is initialized to 0 if using serial interrupt routines to transmit characters stored in some software buffer. Once SBUF0 is loaded directly with the first character to be transmitted, the transmission would start, with the start bit, followed by eight bits 0 to 7 of the data, any parity bit (usually none), followed by the stop bit. TI0 would then be set to 1 automatically when this first character transmission is done and the ISR routine is then triggered which would continue to send any remaining characters in the software buffer (TI0 would need to be reset to 0 every time in the ISR code).

If however we are not using serial interrupt routines to transmit data, TI0 would be intialised to 1 in the first place, since it is usual practice to start the 'putchar( )' routine with:

```
while (TI0==0);          // wait for the transmitter to be ready (TI0=1)
SBUF0 = c;               // store character in SBUF0 and start transmitting character
                         // TI0 would be automatically set to 1 once transmission is done
```

A more thorough example is given in the serial routines in Appendix B.2 UART0 and UART1. The example in the appendix gives the option to setup any one of the two available UARTs.

4.5   Interrupts

Some common problems encountered with interrupts when using assembly language are addressed here:

Forgetting to protect the PSW register: If we write an interrupt handler routine in assembly language, it is a very good idea to always save the PSW SFR on the stack and restore it when our interrupt service routine (ISR) is complete. Many 8051 instructions modify the bits within PSW. If our ISR does not guarantee that PSW contains the same data upon exit as it had upon entry, then our program is bound to behave rather erratically and unpredictably. Moreover it will be tricky to debug since the behaviour will tend to vary depending on when and where in the execution of the program, the interrupt happened.

Forgetting to protect a Register: We must protect all our registers as explained above. If we forget to protect a register that we will use in the ISR and which might have been used in some other part of our program, very strange results may occur. If we are having problems with registers changing their value unexpectedly or having some arithmetic operations producing wrong answers, it is very likely that we have forgotten to protect some registers.

Forgetting to restore protected values: Another common error is to push registers onto the stack to protect them, and then we forget to pop them off the stack (or we pop them in the wrong order) before exiting the interrupt. For example, we may push ACC, B, and PSW onto the stack in order to protect them and subsequently pop only PSW and ACC off the stack before exiting. In this case, since the value of register B was not restored (popped), an extra value remains on the stack. When the RETI instruction is then executed at the end of the ISR, the 8051 will use that value as part of the return address instead of the correct value. In this case, the program will almost certainly crash. We must always ensure that the same number of registers are popped off the stack and in the right order:

        PUSH PSW
        PUSH ACC
        PUSH B
        …
        …
        …
        POP B
        POP ACC
        POP PSW
        RETI

Using the wrong register bank: Another common error occurs when calling another function or routine from within an ISR. Very often the called routine would have been written with a particular register bank in mind, and if the ISR is using another bank, there might be problems when referring to the registers in the called routine. If we are writing our own routine, then in the ISR we could save the PSW register, change the register bank and then restore the PSW register before exiting from the called routine. However, particularly if we are using the C compiler, we might be using functions and procedures pre-written in the compiler and which we do not have any control on, and therefore can result in program not functioning as intended.

This problem is particularly serious when using pre-emptive RTOSs, such as SanctOS or MagnOS described in (Debono, 2013a), where a forced change of task might occur, switching from task A (which was using for example using register bank 1) on to task B which uses say bank 2. For the case of co-operative RTOSs (such as PaulOS), we would be in control where the task changes occur and we would be able to take the necessary precautions.

Using RET instead of RETI: Remember that ISRs in assembly language are always terminated with the RETI instruction. It is easy to inadvertently use the RET instruction instead. However the RET instruction will not end our interrupt smoothly. Usually, using RET instead of RETI will cause the illusion of the main program running normally, but the interrupt will only be executed once. If it appears that the interrupt mysteriously stops executing, we must verify that RETI is being used.

Certain assemblers contain special features which will issue a warning if the programmer fails to protect registers or commit some other common interrupt-related errors.

The above are all taken care of by the compiler when using C as the programming language.

Common problems in C or assembly language:

Forgetting to re-start a timer: We might turn off a timer to re-load the timer register values or to read the counter in an interrupt service routine (ISR) and then forget to turn it on again before exiting from the ISR. In this case, the ISR would only execute once.

Forgetting to clear the interrupt flag: Certain interrupt are not cleared automatically when the ISR is called. For example, when using Timer 2 interrupts, the Timer 2 overflow flag TF2 is not cleared automatically when the ISR is serviced. We have to clear it in the ISR software. The same problem occurs if we forget to clear the RIx or the TIx flags when using the Serial Interrupt. In this case, the ISR keeps on being called repeatedly. Other devices may also exhibit this non-clearing flag situation and should therefore be taken care of when they are used.

4.6   RTOS pitfalls

The PaulOS_F020 co-operative RTOS is a robust and secure RTOSs which we have used extensively throughout the years with our students. This is mainly due to the fact that being a co-operative RTOS, the task changes occur when we want them since there cannot be any forced pre-emptive task changes. However there can still be hidden problems. We should take special care when handling global variables which are accessible to all the tasks. We have to make sure that these variables are allowed to be manipulated only when we want them to. Otherwise it might happen that a task starts with one value of a global variable, then it goes on to a wait state, and when it later on resumes to run, it might end up using the wrong value of the same variable simply because it was modified in the mean time by another task.

The same problem exists in the RTOS with register banks and tasks which use the same functions which are non re-entrant.

4.7   C Tips

- We should always try to keep functions (or tasks) as simple as possible.
- Use the correct required types for the variables; do not use **int** type if we really need **byte** or **bit** type. Naturally, the corresponding conversion character (%c, %bu, %d etc) should then be used with 'printf' or 'scanf' commands.
- Use signed or unsigned types correctly.
- Use specified locations for storing pointers by using declarations such as:

```
char data * xdata str;       /* pointer stored in xdata, pointing to char stored in data */

int xdata * data numtab;     /* pointer stored in data, pointing to int stored in to xdata */

long code * idata powtab;    /* pointer stored in idata, pointing to long stored in code */
```

- In order to improve the performance during code execution or to reduce the memory size requirement for our code, we should analyse the generated list files and assembly code so as to determine which routines can be improved in speed or reduced in size.

- We should always try to minimize the variable usage.

- Set the NUMBER_OF_TASKS, TICKTIME and TICK_TIMER definitions in the PaulOS_ F020_Parameter.h header file to correspond to your application program. This is often a common mistake to make.

- Ensure that if you are using interrupts, make sure that they are enabled.

- Remember that the timer used for the RTOS tick timer cannot be used also for say the baud rate generation of a UART.

- Remember to use the correct ISR parameter in PaulOS_F020_Parameter.h header file when you are using a stand-alone ISR.

Appendix A: PaulOS_F020.C Source Listing

This is the program source listing for the C version of PaulOS RTOS. It consists of:

- The header file PaulOS_F020_Parameters.h
- The header file PaulOS_F020.h
- The start-up file Startup_PaulOS_F020.A51
- The main source program PaulOS_F020.C
- The modified C8051F020 header file

A.1   PaulOS_F020_Parameters.h

```
#ifndef _PaulOS_F020_Parameters_H_

#define _PaulOS_F020_Parameters_H_


/*

**********************************************************************************

*                               RTOS KERNEL HEADER FILE

*

*                               PaulOS_F020_Parameters.H

*

* For use with PaulOS_F020.C — Co-Operative RTOS written in C based on PaulOS

* by Ing. Paul P. Debono

* for use with the 8051 family of micro-controllers

*

* File          : PaulOS_F020_Parameters.H

* Revision      : 10

* Date          : Revised for C8051F020 February 2015

* By            : Paul P. Debono

*

*                               University Of Malta

*

**********************************************************************************

*/
```

```c
/*
********************************************************************************
*                              DATA TYPE DEFINITIONS
********************************************************************************
*/


#define TICK_TIMER     2
// Set to 0, 1, 2 or 3, making sure tick timer does not to clash with UART baud rate timer


#define TICKTIME 1 // Length of RTOS basic tick in ms – refer to the RTOS timing definitions
// suitable values are: 1, 2, 4, 5, 8, 10, 20, 25


#define NOOFTASKS            65      // Number of tasks used in the application


#define STACKSIZE            0x0F    // Number of bytes to allocate for the stack
// There is usually no need to change this parameter


/*
********************************************************************************
*/


/* Interrupt routines running as TASKS or as STAND-ALONE ISRs */

#define STAND_ALONE_ISR_00 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_01 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_02 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_03 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_04 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_05 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_06 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_07 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_08 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_09 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_10 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_11 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_12 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_13 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_14 0 // set to 1 if using this interrupt as a stand alone ISR
#define STAND_ALONE_ISR_15 0 // set to 1 if using this interrupt as a stand alone ISR
```

```
#define STAND_ALONE_ISR_16 0 // set to 1 if using this interrupt as a stand alone ISR

#define STAND_ALONE_ISR_17 0 // set to 1 if using this interrupt as a stand alone ISR

#define STAND_ALONE_ISR_18 0 // set to 1 if using this interrupt as a stand alone ISR

#define STAND_ALONE_ISR_19 0 // set to 1 if using this interrupt as a stand alone ISR

#define STAND_ALONE_ISR_20 0 // set to 1 if using this interrupt as a stand alone ISR

#define STAND_ALONE_ISR_21 0 // set to 1 if using this interrupt as a stand alone ISR


/*
*********************************************************************************************
*/


#endif
```

A.2   PaulOS_F020.h

```
#ifndef _PaulOS_F020_H_

#define _PaulOS_F020_H_


/*
*******************************************************************************************
*                               RTOS KERNEL HEADER FILE
*
*                                    PaulOS_F020.H
*
* For use with PaulOS_F020.C — Co-Operative RTOS written in C based on PaulOS
* by Ing. Paul P. Debono
* for use with the 8051 family of micro-controllers
*
* File         : PaulOS_F020.H
* Revision     : 10
* Date         : Revised for C8051F020 February 2015
* By           : Paul P. Debono
*
*                               B. Eng. (Hons.) Elec.
*                               University Of Malta
*
*******************************************************************************************
*/


/*
*******************************************************************************************
*                               DATA TYPE DEFINITIONS
*******************************************************************************************
*/


typedef unsigned char uchar;

typedef unsigned int uint;

typedef unsigned long ulong;

#include "PaulOS_F020_Parameters.H"
```

```
/*
********************************************************************************
*/


/*
********************************************************************************
*                              FUNCTION PROTOTYPES
********************************************************************************
*
*
* The following RTOS system calls do not receive any parameters:
* ----------------------------------------------------------------
*/


void    OS_DEFER (void);    // Stops current task and passes control to next task in queue
void    OS_KILL_IT (void); // Kills a task – sets it waiting forever
bit     OS_SCHECK (void);   // Checks if running task's signal bit is set
void    OS_WAITP (void);    // Waits for end of task's periodic interval
uchar   OS_RUNNING_TASK_ID(void); // Returns the number of the currently executing task


/* The following commands are simply defined as MACROS below
        OS_CPU_IDLE()           Set the microprocessor into a sleep mode (awakes every interrupt)
        OS_CPU_DOWN()           Switch off microprocessor, activated again only by a hardware
                                reset
        OS_PAUSE_RTOS()         Disable RTOS, used in a stand-alone ISR
        OS_RESUME_RTOS()        Re-enable RTOS, used in a stand-alone ISR
/*


* The following RTOS system calls do receive parameters:
* -------------------------------------------------------
*/


void OS_INIT_RTOS (uchar blank);// Initialises RTOS variables, parameter is not actually used
void OS_RTOS_GO (uchar prior);          // Starts the RTOS running with priorities if required
void OS_SIGNAL_TASK (uchar task); // Signals a task
void OS_WAITI (uchar intnum); // Waits for an event (interrupt) to occur
void OS_WAITT (uint ticks);    // Waits for a timeout period given by a defined
                               // number of ticks
void OS_WAITS (uint ticks);    // Waits for a signal to arrive within a given number of ticks
void OS_PERIODIC (uint ticks); // Sets task to behave periodically every given number of ticks
```

90

```
void OS_CREATE_TASK (uchar task, uint taskadd); // Creates a task

void OS_RESUME_TASK (uchar task);     // Resumes a task which was previously killed


/* The following commands are simply defines as MACROS below

        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
*/


/*
********************************************************************************
*                              RTOS USER DEFINITIONS
********************************************************************************
*/
#define SYSCLOCK        22118400UL             // 22.1184 MHz crystal

#define CPU             5120                   // set to 8051F020 (denoted by 5120)


/* Stack variable points to the start pointer in hardware stack and */
/* should be defined in Startup_PaulOS_F020.A51 */
```

```
extern idata unsigned char MAINSTACK[STACKSIZE];
/*
********************************************************************************
*/


/*
********************************************************************************
*                                  RTOS MACROS
********************************************************************************
*/


#define OS_CPU_IDLE()            PCON |= IDLE  // Sets the microprocessor in idle mode
#define OS_CPU_DOWN()            PCON |= STOP  // Sets the microprocessor in power-down mode


#if (TICK_TIMER == 0)
#define OS_PAUSE_RTOS()       ET0 = 0
#define OS_RESUME_RTOS()      ET0 = 1


#elif (TICK_TIMER == 1)
#define OS_PAUSE_RTOS()       ET1 = 0
#define OS_RESUME_RTOS()      ET1 = 1


#elif (TICK_TIMER == 2)
#define OS_PAUSE_RTOS()       ET2 = 0
#define OS_RESUME_RTOS()      ET2 = 1


#elif (TICK_TIMER == 3)
#define OS_PAUSE_RTOS()       EIE2 ^= ET3
#define OS_RESUME_RTOS()      EIE2 |= ET3


#endif


/*
********************************************************************************
*/


/*
********************************************************************************
*                            RTOS TIMING DEFINITIONS
********************************************************************************
```

92

```
*/
/* Timers used for this RTOS use the system clock divided by 12  */
/* that is they count up once every 0.542535 micro seconds         */
/* For 1 msec, count = SYSCLK/12/1000 = 1843.2                      */
#define MSEC10 18432UL          // In theory 1843.2 counts represent 1 msec assuming an
                                // 22.1184 MHz crystal.
#define CLOCK   ((TICKTIME * MSEC10)/10)     // i.e. approx. 35 — However respecting the
                                             // condition
#define BASIC_TICK      (65535 — CLOCK + 1)   // above, max. acceptable TICKTIME = 25 msecs.
// Hence all suitable values are: 1, 2, 4, 5, 8, 10, 20, 25
// For reliable time-dependent results a value of 10 or
// above is recommended depending upon the application


#define NOT_TIMING              0
// An indefinite period of waiting time in the RTOS is given by a value of 0
#define NO_INTERRUPT 0xFF
/*
********************************************************************************************
*/


/*
********************************************************************************************
*                               COMPILE-TIME ERROR TRAPPING
********************************************************************************************
*/


#if (CPU != 5120)
#error Invalid CPU Setting
#endif


#if (NOOFTASKS > 254)
#error Number of tasks is out of range. The ReadyQ can store up to 254 tasks
#endif


#if 0 // set to one if you need the following checks to be done


#if (CPU == 5120) /* C8051F020 SiLabs processor */
#if ((MAINSTACK + STACKSIZE) > 0x100)
#error Internal RAM Space exceeded. Please recheck the MAINSTACK and STACKSIZE definitions
```

```
#endif
#elif (CPU == 8051)
#if ((MAINSTACK + STACKSIZE) > 0x80)
#error Internal RAM Space exceeded. Please recheck the MAINSTACK and STACKSIZE definitions
#endif
#endif
#endif


#if ((TICKTIME * SYSCLOCK / 12000) > 65535)
#error Tick time value exceeds valid range of the timer counter setting
#endif


#if ((TICKTIME * SYSCLOCK / 12000) < 65535) && ((1000 % TICKTIME) != 0)
#error Undesirable TICKTIME setting. Suggested: 1, 2, 4, 8, 10, 20, 25, 40, 50 ms
#endif


#if (CLOCK > 65535)
#error Timer counter setting exceeded valid range. Please recheck the TICKTIME and MSEC definitions
#endif


/*
*********************************************************************************************
*/


/*
*********************************************************************************************
*                             TASK-RELATED DEFINITIONS
*********************************************************************************************
*/
// Interrupt names
#define IE0_INT            0
#define TF0_INT            1
#define IE1_INT            2
#define TF1_INT            3
#define UART0_INT          4
#define TF2_INT             5
#define SPIF_INT           6
#define SI_INT             7
#define AD0WIN_INT         8
```

94

```
#define PCA_INT                 9

#define CP0FIF_INT              10

#define CP0RIF_INT              11

#define CP1FIF_INT              12

#define CP1RIF_INT              13

#define TF3_INT                 14

#define AD0INT_INT              15

#define TF4_INT                 16

#define AD1INT_INT              17

#define IE6_INT                 18

#define IE7_INT                 19

#define UART1_INT               20

#define XTLVLD_INT              21


#define SIGS_Flag       0x80

#define SIGW_Flag       0x40

#define SIGV_Flag       0x20


#define IDLE_TASK NOOFTASKS

// Main endless loop in application given a task number equal to NOOFTASKS
```
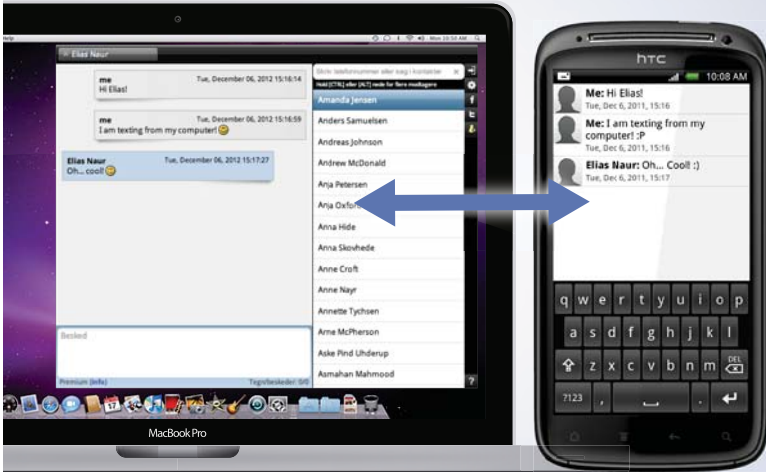
```
/*
*******************************************************************************
*/


/*
*******************************************************************************
*                    ENHANCED EVENT-WAITING ADD-ON MACROS
*******************************************************************************
*
* These macros perform the same functions of the WAITT, WAITS and PERIODIC
* calls but rather than ticks
* they accept absolute time values as parameters in terms of days, hours, minutes, seconds
* and milliseconds
* This difference is denoted by the _A suffix — eg. WAITT_A() is the absolute-time
* version of WAITT()
*
* Range of values accepted:
*
* Using a minimum TICKTIME of 1 msec: 1 msec — 1 min, 5 secs, 535 msecs
* Using a recommended TICKTIME of 10 msec: 10 msecs — 10 mins, 55 secs, 350 msecs
* Using a maximum TICKTIME of 50 msec: 50 msecs — 54 mins, 36 secs, 750 msecs


* If the conversion from absolute time to ticks results in 0 (all parameters being 0 or
* overflow) this
* result is only accepted by WAITS() by virtue of how the WAITT(), WAITS() and PERIODIC()
* calls were


* written. In the case of the WAITT() and PERIODIC() calls the tick count would automatically
* be changed to 1 meaning an interval of e.g. 50 msecs in case the TICKTIME is defined to be
* 50 msecs
*
* Liberal use of parentheses is made in the following macros in case the arguments might
* be expressions.
*
*******************************************************************************
*/
#define OS_WAITT_A(M,S,ms) OS_WAITT((uint)((60000*(##M) + 1000*(##S) + (##ms))/TICKTIME))
#define OS_WAITS_A(M,S,ms) OS_WAITS((uint)((60000*(##M) + 1000*(##S) + (##ms))/TICKTIME))
#define OS_PERIODIC_A(M,S,ms) OS_PERIODIC((uint)((60000*(##M)+1000*(##S)+(##ms))/TICKTIME))
```

96

```c
/*
********************************************************************************
*/


/*
* Other functions used internally by the RTOS:
* -------------------------------------------
*/


void QShift (void);                 // Task swapping function
void RTOS_Timer_Int (void);         // RTOS Scheduler ISR
void Xtra_Int (uchar task_intflag); // Function used by ISRs other than the RTOS Scheduler


#if (!STAND_ALONE_ISR_00)
void Xtra_Int_0 (void);             // External Interrupt 0 ISR
#endif


#if ( (TICK_TIMER != 0 ) && (!STAND_ALONE_ISR_01) )
void Xtra_Int_1 (void);             // Timer 0 ISR
#endif


#if (!STAND_ALONE_ISR_02)
void Xtra_Int_2 (void);             // External Interrupt 1 ISR
#endif


#if ( (TICK_TIMER != 1 ) && (!STAND_ALONE_ISR_03) )
void Xtra_Int_3 (void);             // Timer 1 ISR
#endif


#if (!STAND_ALONE_ISR_04)
void Xtra_Int_4 (void);             // Serial Port ISR
#endif


#if ( (TICK_TIMER != 2 ) && (!STAND_ALONE_ISR_05) )
void Xtra_Int_5 (void);             // Timer 2 ISR
#endif


#if (!STAND_ALONE_ISR_06)
void Xtra_Int_6 (void);
#endif
```

```
#if (!STAND_ALONE_ISR_07)

void Xtra_Int_7 (void);

#endif


#if (!STAND_ALONE_ISR_08)

void Xtra_Int_8 (void);

#endif


#if (!STAND_ALONE_ISR_09)

void Xtra_Int_9 (void);

#endif


#if (!STAND_ALONE_ISR_10)

void Xtra_Int_10 (void);

#endif


#if (!STAND_ALONE_ISR_11)

void Xtra_Int_11 (void);

#endif
```

```
#if (!STAND_ALONE_ISR_12)

void Xtra_Int_12 (void);

#endif


#if (!STAND_ALONE_ISR_13)

void Xtra_Int_13 (void);

#endif


#if ((TICK_TIMER != 3 ) && (!STAND_ALONE_ISR_14) )

void Xtra_Int_14 (void);                      // Timer 3 isr

#endif


#if (!STAND_ALONE_ISR_15)

void Xtra_Int_15 (void);

#endif


#if (!STAND_ALONE_ISR_16)

void Xtra_Int_16 (void);

#endif


#if (!STAND_ALONE_ISR_17)

void Xtra_Int_17 (void);

#endif


#if (!STAND_ALONE_ISR_18)

void Xtra_Int_18 (void);

#endif


#if (!STAND_ALONE_ISR_19)

void Xtra_Int_19 (void);

#endif


#if (!STAND_ALONE_ISR_20)

void Xtra_Int_20 (void);

#endif


#if (!STAND_ALONE_ISR_21)

void Xtra_Int_21 (void);

#endif
```
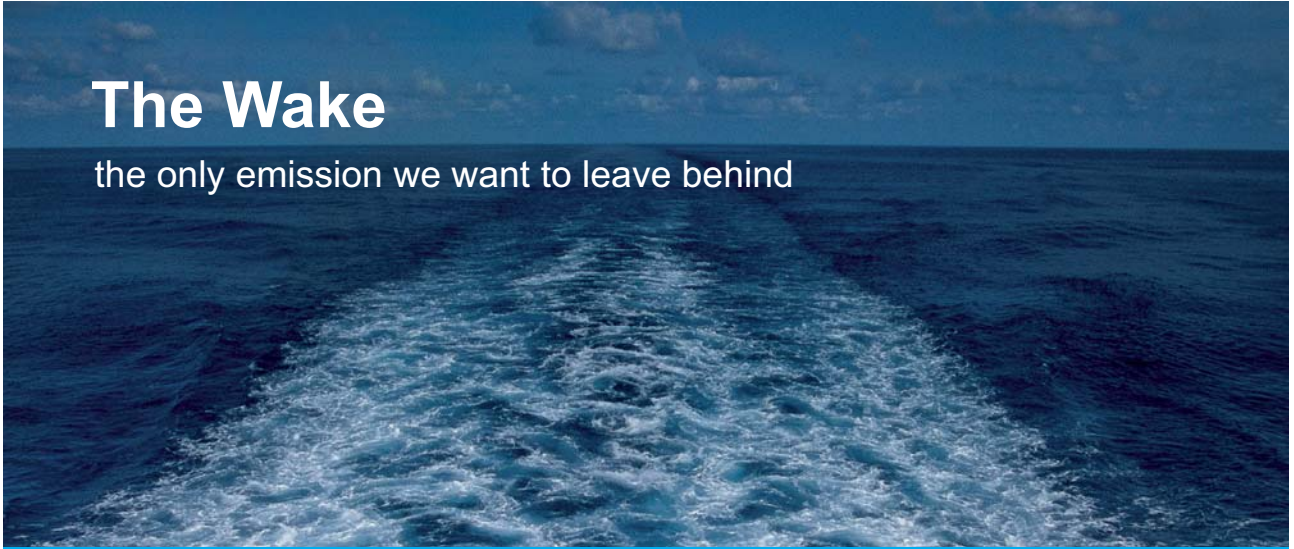
```
/*
*********************************************************************************
*/
#endif
```

A.3   Startup_PaulOS_F020.A51

```
$NOMOD51
;-------------------------------------------------------------------------------
; This file is part of the C51 Compiler package
; Copyright (c) 1988-2002 Keil Elektronik GmbH and Keil Software, Inc.
; modified by Paul Debono in order to
; handle the C8051F020 micro-controller, running the PaulOS RTOS
;-------------------------------------------------------------------------------
; Startup_PaulOS_F020.A51: This code is executed after processor reset.
;
; To translate this file use A51 with the following invocation:
;
;    A51 Startup_PaulOS_F020.A51
;
; To link the modified Startup_PaulOS_F020.OBJ file to your application use the
; following BL51 invocation:
;
;    BL51 <your object file list>, Startup_PaulOS_F020.OBJ <controls>
;
;-------------------------------------------------------------------------------
;
; User-defined Power-On Initialization of Memory
;
; With the following EQU statements the initialization of memory
; at processor reset can be defined:
;
;                         ; the absolute start-address of IDATA memory is always 0
IDATALEN            EQU  100H         ; the length of IDATA memory in bytes.
;
XDATASTART          EQU   0H          ; the absolute start-address of XDATA memory
XDATALEN            EQU  4096         ; the length of XDATA memory in bytes.
;
PDATASTART          EQU   0H          ; the absolute start-address of PDATA memory
PDATALEN            EQU   0H          ; the length of PDATA memory in bytes.
```

100

```
;
; Notes: The IDATA space overlaps physically the DATA and BIT areas of the
;           8051 CPU. At minimum the memory space occupied from the C51
;           run-time routines must be set to zero.
;-------------------------------------------------------------------------------
;
; Reentrant Stack Initilization
;
; The following EQU statements define the stack pointer for reentrant
; functions and initialized it:
;
; Stack Space for reentrant functions in the SMALL model.
IBPSTACK        EQU     0       ; set to 1 if small reentrant is used.
IBPSTACKTOP     EQU     0FFH+1 ; set top of stack to highest location+1.
;
; Stack Space for reentrant functions in the LARGE model.
XBPSTACK        EQU     0       ; set to 1 if large reentrant is used.
XBPSTACKTOP     EQU     0FFFFH+1; set top of stack to highest location+1.
;
; Stack Space for reentrant functions in the COMPACT model.
```

```
PBPSTACK        EQU    0       ; set to 1 if compact reentrant is used.

PBPSTACKTOP     EQU    0FFFFH+1; set top of stack to highest location+1.

;

;-----------------------------------------------------------------------------------

;

; Page Definition for Using the Compact Model with 64 KByte xdata RAM

;

; The following EQU statements define the xdata page used for pdata

; variables. The EQU PPAGE must conform with the PPAGE control used

; in the linker invocation.

;

PPAGEENABLE     EQU    0               ; set to 1 if pdata object are used.

;

PPAGE           EQU    0               ; define PPAGE number.

;

PPAGE_SFR       DATA   0A0H            ; SFR that supplies uppermost address byte

;               (most 8051 variants use P2 as uppermost address byte)

;

;-----------------------------------------------------------------------------------

; Standard SFR Symbols

ACC      DATA  0E0H

B        DATA  0F0H

SP       DATA  81H

DPL      DATA  82H

DPH      DATA  83H



                NAME  ?C_STARTUP



?C_C51STARTUP   SEGMENT   CODE

?STACK          SEGMENT   IDATA



#include "PaulOS_F020_Parameters.h"



                RSEG   ?STACK

MAINSTACK:      DS     STACKSIZE



                EXTRN CODE (?C_START)

                PUBLIC ?C_STARTUP

                PUBLIC MAINSTACK
```

```
                CSEG    AT 0
?C_STARTUP:     LJMP    STARTUP1


                RSEG    ?C_C51STARTUP


STARTUP1:


IF IDATALEN <> 0
                MOV     R0,#IDATALEN - 1
                CLR     A
IDATALOOP:      MOV     @R0,A
                DJNZ    R0,IDATALOOP

ENDIF


IF XDATALEN <> 0
                MOV     DPTR,#XDATASTART
                MOV     R7,#LOW (XDATALEN)
    IF (LOW (XDATALEN)) <> 0
                MOVR6,#(HIGH (XDATALEN)) +1
    ELSE
                MOV     R6,#HIGH (XDATALEN)
    ENDIF
                CLR     A
XDATALOOP:      MOVX    @DPTR,A
                INC     DPTR
                DJNZ    R7,XDATALOOP
                DJNZ    R6,XDATALOOP
ENDIF


IF PPAGEENABLE <> 0
                MOV         PPAGE_SFR,#PPAGE
ENDIF


IF PDATALEN <> 0
                MOV     R0,#LOW (PDATASTART)
                MOV     R7,#LOW (PDATALEN)
                CLR     A
PDATALOOP:      MOVX    @R0,A
                INC     R0
                DJNZ    R7,PDATALOOP
ENDIF
```

103

```
IF IBPSTACK <> 0
EXTRN DATA (?C_IBP)

            MOV     ?C_IBP,#LOW IBPSTACKTOP
ENDIF


IF XBPSTACK <> 0
EXTRN DATA (?C_XBP)

            MOV     ?C_XBP,#HIGH XBPSTACKTOP
            MOV     ?C_XBP+1,#LOW XBPSTACKTOP
ENDIF


IF PBPSTACK <> 0
EXTRN DATA (?C_PBP)

            MOV     ?C_PBP,#LOW PBPSTACKTOP
ENDIF


            MOV     SP,#?STACK-1
; This code is required if you use L51_BANK.A51 with Banking Mode 4
```
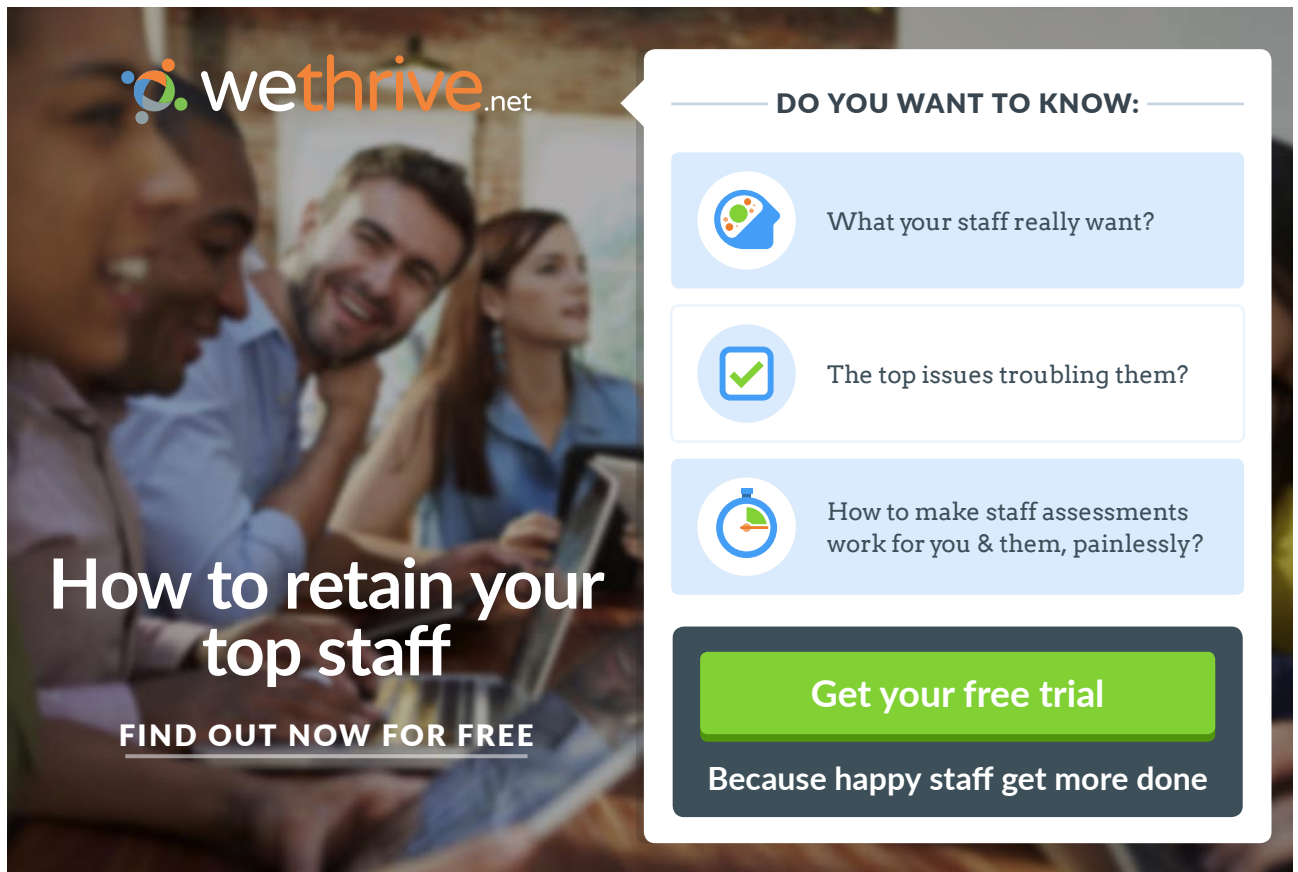
```
; EXTRN CODE (?B_SWITCH0)

;                     CALL    ?B_SWITCH0              ; init bank mechanism to code bank 0

                      LJMP    ?C_START


                      END
```

A.4   PaulOS_F020.c

```
/*
*******************************************************************************************
*       PaulOS_F020.c                        RTOS KERNEL SOURCE CODE
*
* Co-Operative RTOS written in C by Ing. Paul P. Debono:
* ------------------------------------------------------------------
*
* For use with the Silicon Labs C8051F020 family of micro-controllers
*
* Notes:
*
* Timer to use for the RTOS ticks is user selectable: Timer 0, 1, 2, or 3
*
* Assign the correct values to 'TICK_TIMER', 'CPU', 'MAINSTACK'
* and 'NOOFTASKS' in PaulOS_F020.H
*
* If it is noticed that timing parameters are not being met well — the system's TICKTIME
* can be modified by changing the value 'TICKTIME' in PaulOS_F020.H — please adhere to the
* conditions mentioned in PaulOS_F020.H
*
* File          : PaulOS_F020.C
* Revision      : 10
* Date          : FEBRUARY 2015
* By            : Paul P. Debono
*
*                              University Of Malta
*
*******************************************************************************************
*/
```

```c
/*
********************************************************************************
*                                  INCLUDES
********************************************************************************
*/


#include "C8051F020.h"        /* special function registers definitions for the C8051F020 */
#include "PaulOS_F020.h"       /* RTOS system calls definitions */


/*
********************************************************************************
*/


/*
********************************************************************************
*                            STRUCTURE DEFINITIONS
********************************************************************************
*/
struct task_param
{
     uchar stackptr;
     uchar flags;
     uchar intnum;
     uint  timeout;
     uint  interval_count;
     uint  interval_reload;
     char  stack[STACKSIZE];
};
struct task_param xdata task[NOOFTASKS + 1];


/*
********************************************************************************
*                             GLOBAL VARIABLES
********************************************************************************
*/


bit    bdata IntFlag;         // Flag indicating a task waiting for an interrupt was found
bit    bdata TinQFlag;        // Flag indicating that a task timed out
bit    bdata Priority;        // Flag indicating whether priority is enabled or disabled
```
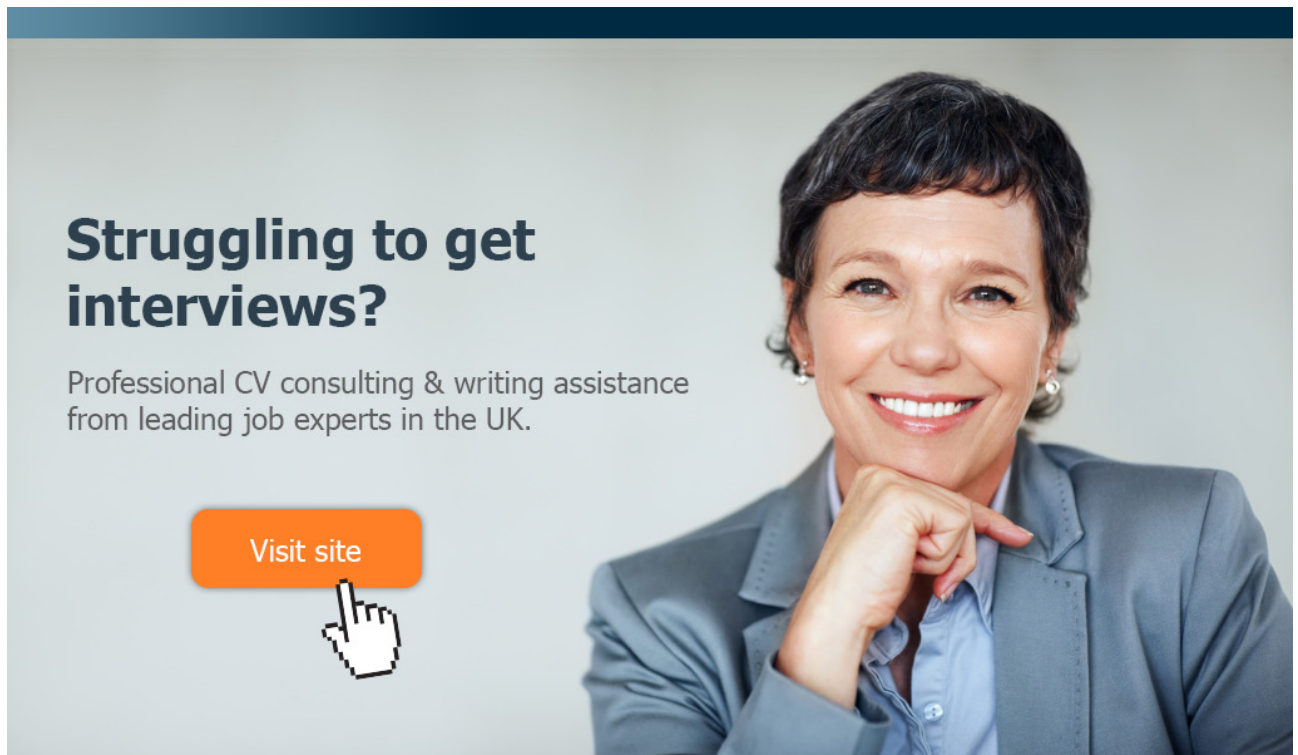
```
uchar data * data ReadyQTop;        // Address of last ready task

uchar data Running;                 // Number of current task

uchar data ReadyQ[NOOFTASKS + 2];   // Queue stack for tasks ready to run


/*
********************************************************************************
*/


/*
********************************************************************************
*                             FUNCTION DEFINITIONS
********************************************************************************
*/


/*
********************************************************************************
*
* Function name: OS_INIT_RTOS
*
* Function type: Initialisation System call
```

```
*
* Description  : This system call initialises the RTOS variables, task SPs and enables any
*                required interrupts
*
* Arguments    : 0    parameter required but not used here,
*                     just to keep compatibility with the other basic PaulOS command
*
*
* Returns      : None
*
***************************************************************************************
*/


void OS_INIT_RTOS(uchar TickTimer)
{
    uchar xdata i,j;
    TickTimer = TickTimer;
 /* parameter not used here, just to keep compatibility with basic PaulOS */
#if (TICK_TIMER == 0)
    #message "Using Timer 0 as the tick timer" // compile time message
    IE &= 0x7F;
    IE |= 0x02;                    /* Set up 8051 IE register, using timer 0 */
    IP = 0x02;                     /* Assign scheduler interrupt high priority */
#elif (TICK_TIMER == 1)
    #message "Using Timer 1 as the tick timer"
    IE &= 0x7F;
    IE |= 0x08;                    /* Set up 8051 IE register, using timer 1 */
    IP = 0x08;                     /* Assign scheduler interrupt high priority */
#elif (TICK_TIMER == 2)
    #message "Using Timer 2 as the tick timer"
    IE &= 0x7F;
    IE |= 0x20;                    /* Set up 8051 IE register, using timer 2 */
    IP = 0x20;                     /* Assign scheduler interrupt high priority */
#elif (TICK_TIMER == 3)
    #message "Using Timer 3 as the tick timer"
    EIE2 |= 0x01;                  /* Set up 8051 IE register, using timer 3 */
    EIP2 = 0x01;                   /* Assign scheduler interrupt high priority */
#endif
```

```
        Running = IDLE_TASK;     /* Set idle task as the running task */

        for (i = 0; i < NOOFTASKS; i++)

        {

                task[i].timeout = NOT_TIMING;             /* Initialise task timeouts,        */

                task[i].intnum = NO_INTERRUPT;            /* not waiting for any interrupt.  */

                task[i].interval_count = NOT_TIMING;      /* periodic interval count          */

                task[i].interval_reload = NOT_TIMING;  /* and reload variables.        */

                ReadyQ[i] = IDLE_TASK;             /* Fill the READY queue with         */

        }                                          /* with the idle task          */

        ReadyQ[NOOFTASKS] = IDLE_TASK;

        ReadyQ[NOOFTASKS + 1] = IDLE_TASK;

        ReadyQTop = ReadyQ;                        /* Pointer to last task made to point to */

                                                   /* base of the queue. */


        for (i = 0; i < NOOFTASKS + 1; i++)

        {

                task[i].stackptr = MAINSTACK + 2;       /* Initialise task SP values */

                task[i].flags = 0;                      /* Initialise task status bytes */

                for(j=0; j<STACKSIZE; j++) task[i].stack[j] = 0; /* clear all ext. stack area */

        }

}



/*
*********************************************************************************************
*/


/*
*********************************************************************************************
*
* Function name: OS_CREATE_TASK
*
* Function type: Initialisation System call
*
* Description  : This system call is used in the main program for each task to be created
*                    for use in the application.
*
* Arguments     : taskRepresents the task number (1st task is numbered as 0).
*
*                 taskadd   Represents the task's start address, which in the C
*                           environment, would simply be the name of the procedure
```

```
*
* Returns         : None
*
********************************************************************************
*/


void OS_CREATE_TASK(uchar tasknum, uint taskadd)

{

    ReadyQTop++;               /* Task is added to next available */

   *ReadyQTop = tasknum;    /* position in the READY queue.          */

    task[tasknum].stack[0] = taskadd % 256;

    task[tasknum].stack[1] = taskadd / 256;


}


/*
********************************************************************************
*/


/*
```

```
***********************************************************************************
*
* Function name: OS_RTOS_GO
*
* Function type: Initialisation System call
*
* Description  : This system calls is used to start the RTOS going such that it supervises
                 the application processes.
*
* Arguments    : prior Determines whether tasks ready to be executed are sorted
*                                prior to processing
*                                or not. If prior = 0 a FIFO queue function is implied,
*                                if prior = 1 the
*                                queue is sorted by task number in ascending order, as a
*                                higher priority is
*                                associated with smaller task number (task 0 would have
*                                the highest
*                                priority), such that the first task in the queue, which
*                                would eventually
*                                run, would be the one with the smallest task number
*                                having highest priority.
*
* Returns          : None
*
***********************************************************************************
*/

void OS_RTOS_GO(uchar prior)
{
    if (prior == 1)          /* Checks if tasks priorities    */
        Priority = 1;      /* are to be enabled                    */
    else
        Priority = 0;


#if (TICK_TIMER == 0)
    /* Configure Timer 0 in 16-bit timer mode for the 8051       */
    /* TH0 and TL0 are loaded in the TF0 RTOS Tick ISR */
    TMOD &= 0xF0;            /* Clear T0 mode control, leaving T1 untouched */
    TMOD |= 0x01;            /* Set T0 mode control */
```

```
    CKCON &= 0xF0;              /* Use sysclk/12 (T0M = 0) */

    TR0 = 1;                   /* Start timer 0 */

    TF0 = 1;                    /* Cause first interrupt immediately */


#elif (TICK_TIMER == 1)

    /* Configure Timer 1 in 16-bit timer mode for the 8051      */

    /* TH1 and TL1 are loaded in the TF1 RTOS Tick ISR */

    TMOD &= 0x0F;               /* Clear T1 mode control, leaving T0 untouched */

    TMOD |= 0x10;              /* Set T1 mode control */

    CKCON &= 0xE8;             /* Use sysclk/12 (T1M = 0) */

    TR1 = 1;                   /* Start timer 1 */

    TF1 = 1;                   /* Cause first interrupt immediately */


#elif (TICK_TIMER == 2)

    RCAP2 = BASIC_TICK;        /* Configures Timer 2 in 16-bit auto-reload mode */

    CKCON |= 0xD8;             /* Use sysclk/12 (T2M=0) */

    T2CON = 0x84;              /* TR2 = TF2 = 1, causes first interrupt immediately */


#elif (TICK_TIMER == 3)

    TMR3CN = 0x00;              // Stop timer 3, clear TF3 use SYSCLK/12 as timer base

    TMR3RL = BASIC_TICK;       /* Configures Timer 3 in 16-bit auto-reload mode     */

    TMR3 = 0xFFFF;             /* Causes immediate overflow, (reload immediately)   */

    EIE2 |= ET3;               /* Enable Timer 3 interrupts */

    TMR3CN |= TR3;             /* Start Timer 3 using sysclk/12 (TR3=1, TF3=T3M=T3XCLK=0) */


#endif


    TinQFlag = 1; /* Signals scheduler that tasks have been        */

    /* added to the queue.                                  */

    EA = 1;             /* Interrupts are enabled, starting the RTOS */


}


/*
*********************************************************************************************
*/


/*
*********************************************************************************************
```

```
*

* Function name: OS_RUNNING_TASK_ID

*

* Function type: Inter-task Communication System call

*

* Description  : This system call is used to check to get the number of the
*                                 current task.

*

* Arguments    : None

*

* Returns      : Number of currently running task from which it must be called

*

***********************************************************************************

*/

uchar OS_RUNNING_TASK_ID(void)

{

      return (Running);

}


/*
```

```
***********************************************************************************************
*
* Function name: OS_SCHECK
*
* Function type: Inter-task Communication System call
*
* Description  : This system call is used to check if the current task has its signal set.
*                      It tests whether there was any signal sent to it by some other task.
*
* Arguments    : None
*
* Returns      : 1 if its signal bit is set, 0 if not set
*
***********************************************************************************************
*/


bit OS_SCHECK(void)
{
    EA = 0;
    if (task[Running].flags & SIGS_Flag) /* If a signal is present it's cleared */
    {
        task[Running].flags &= ~SIGS_Flag;  /* and a 1 is returned. */
        EA = 1;
        return 1;
    }
    else                    /* If a signal is not present, 0 is returned */
    {
        EA = 1;
        return 0;
    }
}


/*
***********************************************************************************************
/*


***********************************************************************************************
*
* Function name: OS_SIGNAL_TASK
*
```

```
* Function type: Inter-task Communication System call
*
* Description  : This system call is used to send a signal to another task.
*
* Arguments    : task        Represents the task to which a signal is required to be sent.
*
* Returns      : None
*
*******************************************************************************************************
*/


void OS_SIGNAL_TASK(uchar tasknum)
{
    EA = 0;


    if (task[tasknum].flags & SIGW_Flag)
    {
        task[tasknum].flags &= ~SIGS_Flag;       /* If a task has been waiting      */
        task[tasknum].flags &= ~SIGW_Flag;       /* for a signal, the task no       */
        task[tasknum].timeout = NOT_TIMING;      /* longer has to wait and is */
        ReadyQTop++;                                     /* added to the READY queue. */
        *ReadyQTop = tasknum;
        TinQFlag = 1;
        EA = 1;
    }
    else                                        /* If it was not waiting, its */
        task[tasknum].flags |= SIGS_Flag;  /* signal sent bit is set     */
    EA = 1;
}


/*
*******************************************************************************************************
*/


/*
*******************************************************************************************************
*
* Function name: OS_WAITS
*
* Function type: Event-Waiting System call
```

```
*
* Description  : This system call causes a task to wait for a signal to arrive within a given
                 number of RTOS ticks. If the signal is already present, the task continues
                 to execute.
*
* Arguments    : ticks        Represents the number of ticks for which the task will wait for
                              a signal to
*                             arrive. Valid range for this argument is 0 to
*                             4294967295. A value of 0 means waiting forever for a
*                             signal to arrive.
*
* Returns            : None
*
*********************************************************************************************
*/
```

```
void OS_WAITS (uint ticks)

{

    EA = 0;


    if (task[Running].flags & SIGS_Flag)  /* If signal already sent it clears the */

    {

        task[Running].flags &= ~SIGS_Flag;         /* signal and the task */

        EA = 1;                                     /* continues to run. */

    }

    else                                   /* If signal is not present */

    {

        task[Running].flags |= SIGW_Flag;  /* the task is sent in the            */

        task[Running].timeout = ticks;     /* waiting state, by causing        */

        QShift();                          /* a task switch. */

    }

}


/*
*********************************************************************************************
*/


/*
*********************************************************************************************
*
* Function name: OS_WAITT
*
* Function type: Event-Waiting System call
*
* Description  : This system call causes a task to go in the waiting state for a timeout period
                 given
*                by a defined number of RTOS ticks.
*
* Arguments    : ticks       Represents the number of ticks for which the task will
*                            wait. Valid range for
*                            this parameter is 1 to 4294967295. A zero waiting time
*                            parameter is set to 1
*                            by the RTOS itself, since a zero effectively kills the
*                            task, making it wait forever.
*
* Returns      : None
```

117

```
*
******************************************************************************
*/


void OS_WAITT (uint ticks)

{

    EA = 0;


    if (ticks == 0)

        ticks = 1;                          /* Task's timeout variable is updated */

    task[Running].timeout = ticks;  /* and the task then enters the            */

    QShift();                           /* waiting state.                       */

}


/*
******************************************************************************
*/


/*
******************************************************************************
*
* Function name: OS_WAITP
*
* Function type: Event-Waiting System call
*
* Description  : This system call is used by a task to wait for the end of its periodic
*                interval. If the interval has already passed, the task continues to
*                execute.
*
* Arguments    : None
*
* Returns      : None
*
******************************************************************************
*/
```

```c
void OS_WAITP(void)
{
    EA = 0;

    if ((task[Running].flags & SIGV_Flag)==SIGV_Flag )/* If the periodic */
    {
        task[Running].flags &= ~SIGV_Flag;         /* interval time has elapsed, the   */
        EA = 1;                                     /* task continues to */
    }                                               /* execute.          */
    else
    {
        /* Else the task */
        task[Running].flags |= SIGV_Flag;          /* enters the waiting */
        QShift();                                   /* state.            */
    }
}


/*
*********************************************************************************************
*/
```

```
/*
********************************************************************************
*
* Function name: OS_PERIODIC
*
* Function type: Event-Waiting System call
*
* Description  : This system call causes a task to repeat its function every given number of
*                RTOS ticks.
*
* Arguments    : ticks       Represents the length of the periodic interval in terms
*                            of RTOS ticks, after
*                            which the task repeats itself. Valid range for this
*                            parameter is 1 to 4294967295.
*
* Returns              : None
*
********************************************************************************
*/


void OS_PERIODIC (uint ticks)
{
    EA = 0;

    if (ticks == 0)
        ticks = 1;                           /* at least 1 tick time is required */
        task[Running].interval_reload = ticks;   /* Task's periodic interval count */
        task[Running].interval_count = ticks;    /* and reload variables are    */
        EA = 1;                              /* initialised. */
}


/*
********************************************************************************
*/


/*
********************************************************************************
*
* Function name: OS_WAITI
*
```

```
* Function type: Event-Waiting System call

*

* Description  : This system call causes a task to wait for a given event (interrupt).

*                It identifies

*                for which interrupt the task has to wait. Once identified − the task's

*                appropriate flag is set and the task is set in the waiting state by

*                causing a task swap − the task

*                would wait indefinitely for the interrupt as its timeout

*                variable would be set to 0

*                (NOT_TIMING) either during initialisation of the RTOS or

*                after expiry of its timeout

*                period due to other prior invocations of wait-inducing system calls.

*

* Arguments    : intnum      Represents the interrupt number associated with the given

*                            interrupt for

*                            which the calling task intends to wait

*

* Returns      : None

*

*******************************************************************************************

*/



void OS_WAITI(uchar intnum)

{

    EA = 0;


    switch (intnum)

    {

#if (!STAND_ALONE_ISR_00)

    case 0:                              /* Interrupt number 0 */

        task[Running].intnum = IE0_INT;  /* Task made to wait for               */

        QShift();                         /* external interrupt 0                */

        break;

#endif



#if ( (TICK_TIMER != 0) && (!STAND_ALONE_ISR_01) )

    case 1:                              /* Interrupt number 1                  */

        task[Running].intnum = TF0_INT;   /* Task made to wait for               */

        QShift();                         /* timer 0 interrupt                   */

        break;

#endif
```

```
#if (!STAND_ALONE_ISR_02)
    case 2:                                  /* Interrupt number 2 */
        task[Running].intnum = IE1_INT;      /* Task made to wait for   */
        QShift();                            /* external interrupt 1    */
        break;
#endif


#if ( (TICK_TIMER != 1) && (!STAND_ALONE_ISR_03) )
    case 3:                                  /* Interrupt number 3      */
        task[Running].intnum = TF1_INT;      /* Task made to wait for   */
        QShift();                            /* timer 1 interrupt  */
        break;
#endif


#if (!STAND_ALONE_ISR_04)
    case 4:                                  /* Interrupt number 4      */
        task[Running].intnum = UART0_INT;    /* Task made to wait for   */
        QShift();                            /* serial port interrupt   */
        break;
#endif
```

```
#if ( (TICK_TIMER != 2) && (!STAND_ALONE_ISR_05) )
    case 5:                                 /* Interrupt number 5        */
        task[Running].intnum = TF2_INT;     /* Task made to wait for     */
        QShift();                           /* timer 1 interrupt  */
        break;
#endif


#if (!STAND_ALONE_ISR_06)
    case 6:                                 /* Interrupt number 6 */
        task[Running].intnum = SPIF_INT;    /* Task made to wait for     */
        QShift();                           /* serial peripheral interface       */
        break;
#endif


#if (!STAND_ALONE_ISR_07)
    case 7:                                 /* Interrupt number 7        */
        task[Running].intnum = SI_INT;      /* Task made to wait for     */
        QShift();                           /* SMBus interface */
        break;
#endif


#if (!STAND_ALONE_ISR_08)
    case 8:                                 /* Interrupt number 8        */
        task[Running].intnum = AD0WIN_INT;  /* Task made to wait for     */
        QShift();                           /* ADC0 Window comparator    */
        break;
#endif


#if (!STAND_ALONE_ISR_09)
    case 9:                                 /* Interrupt number 9        */
        task[Running].intnum = PCA_INT;     /* Task made to wait for     */
        QShift();                           /* Programmable Counter Array        */
        break;
#endif


#if (!STAND_ALONE_ISR_10)
    case 10:                                /* Interrupt number 10       */
        task[Running].intnum = CP0FIF_INT;  /* Task made to wait for     */
        QShift();                           /* comparator 0 Falling Edge */
        break;
#endif
```

123

```
#if (!STAND_ALONE_ISR_11)
    case 11:                                  /* Interrupt number 11 */
        task[Running].intnum = CP0RIF_INT;    /* Task made to wait for    */
        QShift();                             /* comparator 0 Rising Edge */
        break;
#endif


#if (!STAND_ALONE_ISR_12)
    case 12:                                  /* Interrupt number 12       */
        task[Running].intnum = CP1FIF_INT;    /* Task made to wait for     */
        QShift();                             /* comparator 1 Falling Edge */
        break;
#endif


#if (!STAND_ALONE_ISR_13)
    case 13:                                  /* Interrupt number 13       */
        task[Running].intnum = CP1RIF_INT;    /* Task made to wait for     */
        QShift();                             /* comparator 1 Rising Edge  */
        break;
#endif


#if ( (TICK_TIMER != 3) && (!STAND_ALONE_ISR_14) )
    case 14:                                  /* Interrupt number 14       */
        task[Running].intnum = TF3_INT;       /* Task made to wait for */
        QShift();                             /* timer 3 interrupt  */
        break;
#endif


#if (!STAND_ALONE_ISR_15)
    case 15:                                  /* Interrupt number 15       */
        task[Running].intnum = AD0INT_INT;    /* Task made to wait for */
        QShift();                             /* ADC0 end of conversion   */
        break;
#endif


#if (!STAND_ALONE_ISR_16)
    case 16:                                  /* Interrupt number 16       */
        task[Running].intnum = TF4_INT;       /* Task made to wait for     */
        QShift();                             /* timer 4 interrupt  */
        break;
#endif
```

124

```c
#if (!STAND_ALONE_ISR_17)
    case 17:                                    /* Interrupt number 17      */
        task[Running].intnum = AD1INT_INT;      /* Task made to wait for    */
        QShift();                               /* ADC1 end of conversion   */
        break;
#endif


#if (!STAND_ALONE_ISR_18)
    case 18:                                    /* Interrupt number 18      */
        task[Running].intnum = IE6_INT;         /* Task made to wait for    */
        QShift();                               /* external interrupt 6     */
        break;
#endif


#if (!STAND_ALONE_ISR_19)
    case 19:                                    /* Interrupt number 19      */
        task[Running].intnum = IE7_INT;         /* Task made to wait for    */
        QShift();                               /* external interrupt 7     */
        break;
#endif
```

```
#if (!STAND_ALONE_ISR_20)
    case 20:                                        /* Interrupt number 20        */
        task[Running].intnum = UART1_INT;       /* Task made to wait for     */
        QShift();                               /* UART1 interrupt    */
        break;
#endif


#if (!STAND_ALONE_ISR_21)
    case 21:                                        /* Interrupt number 21        */
        task[Running].intnum = XTLVLD_INT;      /* Task made to wait for     */
        QShift();                               /* External Crystal Osc. Ready interrupt */
        break;
#endif


    default:
        EA = 1;
        break;
    }
}


/*
*********************************************************************************************
*/


/*
*********************************************************************************************
*
* Function name: OS_DEFER
*
* Function type: Task Suspension System call
*
* Description  : This system call is used to stop the current task in order for the next
*                       task in the queue to execute. In the meantime
*                        the current task is placed at the end of the queue.
*
* Arguments    : None
*
* Returns      : None
*
```

```
**************************************************************************************
*/


void OS_DEFER(void)

{

    EA = 0;

    task[Running].timeout = 2;      /* Task added to the waiting                 */

                                    /* queue, for 2 tick times, prior to         */

    QShift();                       /* causing a task switch                     */

}


/*
**************************************************************************************
*/


/*
**************************************************************************************
*

* Function name: OS_KILL_IT

*

* Function type: Task Suspension System call

*

* Description  : This system call kills the current task, by putting it permanently waiting,

*                such that it never executes again. It also clears any set waiting signals

*                which the task might have.

*

* Arguments    : None

*

* Returns      : None

*
**************************************************************************************
*/


void OS_KILL_IT(void)

{

    EA = 0;

    task[Running].flags = 0;        /* Task is killed by clearing its flags      */

    task[Running].intnum = NO_INTERRUPT;

    task[Running].timeout = NOT_TIMING;    /* setting it to wait forever         */
```

127

```
        task[Running].interval_count = 0;        /* Task's periodic interval count is set to zero*/

        QShift();                                 /* and then cause a task switch.      */

}



/*
*******************************************************************************************
*/


/*
*******************************************************************************************
*
* Function name: OS_RESUME_TASK
*
* Function type: Inter-task Communication System call
*
* Description  : This system call is used to resume another KILLed task.
*
* Arguments    : task        Represents the task to which is to be restarted.
*
* Returns      : None
```

```
*
********************************************************************************
*/


void OS_RESUME_TASK (uchar tasknum)

{

    EA = 0;


    if (task[tasknum].interval_reload != 0)

                                    /* if task was a KILLed periodic task */

        task[tasknum].interval_count = 1; /* resume periodic task otherwise */

    else

        task[tasknum].timeout = 1;       /* resume normal waiting task after 1 tick */

    task[Running].timeout = 2;  /* Place the current task waiting for the */

    QShift();                    /* next 2 ticks in the waiting state, thus          */

                                 /* giving up its time for other tasks.              */

}
/*

********************************************************************************

*/


/*

********************************************************************************

*

* Function name: QShift

*

* Function type: Context Switcher (Internal function)

*

* Description  : This function is used to perform a context switch i.e. task swapping

*

* Arguments    : None

*

* Returns      : None

*

********************************************************************************

*/


void QShift (void) using 1

{
```

```
uchar data i, temp;

uchar idata * idata internal;

uchar data * idata qtask;

uchar data * idata qptr;


TinQFlag = 0;


task[Running].stackptr = temp = SP;     /* Current task's SP is saved */

internal = MAINSTACK;

                                        /* Current task's USED stack area is saved */

i = 0;

do {

     task[Running].stack[i++] = *(internal++);

}

while (internal<=temp);

qtask = ReadyQ;                    /* READY queue is shifted down */

qptr = ReadyQ + 1;

while (qtask <= ReadyQTop)          /* by one position          */

{

     *qtask++ = *qptr++;

}

ReadyQTop--;/* Pointer to last task in queue is decremented */


if (ReadyQTop < ReadyQ) /* Ensure that this pointer is never  */

     ReadyQTop = ReadyQ;        /* below the start of the READY queue  */

if (Priority == 1)             /* If task priorities are enabled */

{

                               /* the queue is sorted such that */

     qptr = ReadyQTop;         /* the highest priority task  */

     while (qptr > ReadyQ)     /* becomes the running task, i.e.              */

     {

                               /* the one having the smallest */

                               /* task number.        */

                               /* Just one scan through the list */

          qptr--;

          if (*qptr > *(qptr + 1))

          {

                 temp = *qptr;

                 *qptr = *(qptr + 1);

                 *(qptr + 1) = temp;
```

130

```
            }
        }
    }
                                /* The first task in the READY queue */
    Running = ReadyQ[0];        /* becomes the new running task       */
                                /* The new running task's stack */
                                /* area is copied to internal RAM */
    temp = task[Running].stackptr;
    internal = MAINSTACK;
    /* The new running task's USED stack area is copied to internal RAM */
    i=0;
    do {
        *(internal++) = task[Running].stack[i++];
    }
    while (internal<=temp);


    SP = task[Running].stackptr;    /* The new running task's SP is restored */
                                /* such that the new task will execute. */
    EA = 1;
}
```

```
/*
**************************************************************************************************
*/


/*
**************************************************************************************************
*
* Function name: Xtra_Int_0
*
* Function type: Interrupt Service Routine
*
* Description  : This is the external 0 interrupt ISR whose associated interrupt no. is 0.
*
* Arguments    : None
*
* Returns      : None
*
**************************************************************************************************
*/


#if (!STAND_ALONE_ISR_00)
void Xtra_Int_0 (void) interrupt 0 using 1
{
    EA = 0;
    Xtra_Int(IE0_INT); /* Passes EXT0W for identification purposes */
}
#endif


/*
**************************************************************************************************
*
* Function name: RTOS_Timer_Int
*
* Function type: Scheduler Interrupt Service Routine
*
* Description  : This is the RTOS scheduler ISR. It generates system ticks and calculates
*                any remaining waiting and periodic interval time for each task.
*
* Arguments    : None
*
```

```
* Returns     : None
*
*********************************************************************************************
*/


#if (TICK_TIMER == 0)                 /* If Timer 0 is used for the scheduler            */
void RTOS_Timer_Int (void) interrupt 1 using 1
{
     uchar data k;                /* Timer 0 is used */
     uchar data * idata q;        /* for scheduling.                      */
     bit data On_Q;


     TH0 = BASIC_TICK / 256;      /* Timer registers reloaded */
     TL0 = BASIC_TICK % 256;


#elif (TICK_TIMER == 1)      /* If Timer 1 is used for the scheduler */
void RTOS_Timer_Int (void) interrupt 3 using 1
{
     uchar data k;           /* Timer 1 is used */
     uchar data * idata q; /* for scheduling.                    */
     bit data On_Q;


     TH1 = BASIC_TICK / 256;        /* Timer registers reloaded */
     TL1 = BASIC_TICK % 256;


#elif (TICK_TIMER == 2)      /* If Timer 2 is used for the scheduler */
void RTOS_Timer_Int (void) interrupt 5 using 1
{
     uchar data k;           /* For the 8032, Timer 2 is used */
     uchar data * idata q;   /* for scheduling.                    */
     bit data On_Q;


     TF2 = 0;                  /* Timer 2 interrupt flag is cleared */


#elif (TICK_TIMER == 3)      /* If Timer 3 is used for the scheduler */
void RTOS_Timer_Int (void) interrupt 14 using 1
{
     uchar data k;           /* For the 8032, Timer 2 is used */
     uchar data * idata q;   /* for scheduling.                    */
     bit data On_Q;


     TMR3CN &= ~TF3;          /* Timer 3 interrupt flag is cleared */
```

133

```
#endif


     for (k = 0; k < NOOFTASKS; k++)

     {

          if (task[k].interval_count != NOT_TIMING) /* Updates the tasks' */

          {

                                             /* periodic intervals. */

               task[k].interval_count--;

               if (task[k].interval_count == NOT_TIMING)

               {

                    task[k].interval_count = task[k].interval_reload;

                    if (task[k].flags & SIGV_Flag)

                    {

                                                  /* If periodic interval    */

                         task[k].flags &= ~SIGV_Flag; /* has elapsed and the */

                         q = ReadyQ;                  /* task has been waiting    */

                         On_Q = 0;                    /* for this to occur, the   */

                         while (q <= ReadyQTop)       /* task is placed in the    */

                                                      /* READY queue, if it is    */

                         {
```

```
                              if (k == *q)           /* verified that the task    */
                                                     /* does not already reside   */
                        {
                                 On_Q = 1;   /* in the queue, as now      */
                                 break;      /* the task no longer        */
                                             /* requires to wait.         */
                        }
                        q++;
                }
                if (On_Q == 0)
                {
                    ReadyQTop++;
                    *ReadyQTop = k;
                    TinQFlag = 1;
                }
        }
                                              /* If however the task       */
                                              /* was not waiting for       */
                                              /* this event, the task      */
                                              /* is not place in the       */
                                              /* the ready queue.          */
        else
                task[k].flags |= SIGV_Flag;
    }
}
if (task[k].timeout != NOT_TIMING)
{
                                              /* Updates the tasks'        */
    task[k].timeout--;                        /* timeout variables.        */
    if (task[k].timeout == NOT_TIMING)
    {
        ReadyQTop++;                          /* If a waiting task's       */
        *ReadyQTop = k;                       /* timeout elapses           */
        TinQFlag = 1;                         /* the task is placed        */
        task[k].flags &= ~SIGW_Flag;  /* in the ready queue.              */
    }
}
/* If the idle task is running, when tasks are */
/* known to reside in the queue, a task switch */
/* is purposely induced so these tasks can run. */
```

```
    }

    if ((TinQFlag == 1) && (Running == IDLE_TASK))

        QShift();

}


/*
**********************************************************************************************
*/


/*
**********************************************************************************************
Here are the Interrupt Service routines handlers for ALL the C8051F020 interrupts
**********************************************************************************************
*/


/*
**********************************************************************************************
*
* Function name: Xtra_Int_1
*
* Function type: Interrupt Service Routine
*
* Description  : This is the Timer 0 ISR whose associated interrupt number is 1.
*
* Arguments    : None
*
* Returns      : None
*
**********************************************************************************************
*/


#if ( (TICK_TIMER != 0) && (!STAND_ALONE_ISR_01) )
/* Timer 0 interrupt used for RTOS on 8051 */


void Xtra_Int_1 (void) interrupt 1 using 1
{
    EA = 0;

    Xtra_Int(TF0_INT); /* Passes TF0_INT for identification purposes */
}
```

```
#endif
/*
*****************************************************************************************
*/


/*
*****************************************************************************************
*
* Function name: Xtra_Int_2
*
* Function type: Interrupt Service Routine
*
* Description  : This is the external 1 interrupt ISR whose associated int. number is 2.
*
* Arguments    : None
*
* Returns      : None
*
*/
#if (!STAND_ALONE_ISR_02)
```

```c
void Xtra_Int_2 (void) interrupt 2 using 1

{

    EA = 0;

    Xtra_Int(IE1_INT); /* Passes IE1_INT for identification purposes */

}

#endif
/*
*******************************************************************************
*/


/*
*******************************************************************************
*
* Function name: Xtra_Int_3
*
* Function type: Interrupt Service Routine
*
* Description  : This is the Timer 1 ISR whose associated interrupt number is 3.
*
* Arguments    : None
*
* Returns      : None
*
*******************************************************************************
*/
#if ( (TICK_TIMER != 1) && (!STAND_ALONE_ISR_03) )


void Xtra_Int_3 (void) interrupt 3 using 1

{

    EA = 0;

    Xtra_Int(TF1_INT); /* Passes TF1_INT for identification purposes */

}


#endif
/*
*******************************************************************************
*/


/*
```

```
/*
****************************************************************************************
*
* Function name: Xtra_Int_4
*
* Function type: Interrupt Service Routine
*
* Description  : This is the serial port ISR whose associated interrupt number is 4.
*
* Arguments    : None
*
* Returns      : None
*
****************************************************************************************
*/
#if (!STAND_ALONE_ISR_04)
void Xtra_Int_4 (void) interrupt 4 using 1
{
    EA = 0;
    Xtra_Int(UART0_INT); /* Passes UART0_INT for identification purposes */
}
#endif
/*
****************************************************************************************
*/


/*
****************************************************************************************
*
* Function name: Xtra_Int_5
*
* Function type: Interrupt Service Routine
*
* Description  : This is the Timer 2 ISR whose associated interrupt number is 5.
*
* Arguments    : None
*
* Returns      : None
*
****************************************************************************************
```

```
*/
#if ( (TICK_TIMER != 2) && (!STAND_ALONE_ISR_05) )


void Xtra_Int_5 (void) interrupt 5 using 1
{
    EA = 0;

    TF2 = 0;                  /* may be cleared in the task itself */

    Xtra_Int(TF2_INT);        /* Passes TF2_INT for identification purposes */

}
#endif


/*
********************************************************************************
*
* Function name: Xtra_Int_6
*
* Function type: Interrupt Service Routine
*
* Description  : This is the SPI interrupt ISR whose associated interrupt number is 6.
*
```

```
* Arguments    : None
*
* Returns      : None
*
*******************************************************************************
*/
#if (!STAND_ALONE_ISR_06)
void Xtra_Int_6 (void) interrupt 6 using 1
{
    EA = 0;
    Xtra_Int(SPIF_INT); /* Passes SPIF_INT for identification purposes */
}
#endif


/*
*******************************************************************************
*
* Function name: Xtra_Int_7
*
* Function type: Interrupt Service Routine
*
* Description  : This is the SI interrupt ISR whose associated interrupt number is 7.
*
* Arguments    : None
*
* Returns      : None
*
*******************************************************************************
*/


#if (!STAND_ALONE_ISR_07)
void Xtra_Int_7 (void) interrupt 7 using 1
{
    EA = 0;
    Xtra_Int(SI_INT); /* Passes SI_INT for identification purposes */
}
#endif


/*
*******************************************************************************
```

```
* Function name: Xtra_Int_8

*

* Function type: Interrupt Service Routine

*

* Description  : This is the AD0 interrupt ISR whose associated interrupt number is 8.

*

* Arguments    : None

*

* Returns      : None

*

*********************************************************************************************

*/


#if (!STAND_ALONE_ISR_08)

void Xtra_Int_8 (void) interrupt 8 using 1

{

    EA = 0;

    Xtra_Int(AD0WIN_INT); /* Passes AD0WIN_INT for identification purposes */

}

#endif


/*

*********************************************************************************************

*

* Function name: Xtra_Int_9

*

* Function type: Interrupt Service Routine

*

* Description  : This is the PCA interrupt ISR whose associated interrupt number is 9.

*

* Arguments    : None

*

* Returns            : None

*

*********************************************************************************************

*/


#if (!STAND_ALONE_ISR_09)

void Xtra_Int_9 (void) interrupt 9 using 1

{
```

```
    EA = 0;

    Xtra_Int(PCA_INT); /* Passes PCA_INT for identification purposes */

}

#endif


/*
****************************************************************************************

*

* Function name: Xtra_Int_10

*

* Function type: Interrupt Service Routine

*

* Description  : This is the CP0 interrupt ISR whose associated interrupt number is 10.

*

* Arguments    : None

*

* Returns      : None

*

****************************************************************************************

*/
```

```c
#if (!STAND_ALONE_ISR_10)
void Xtra_Int_10 (void) interrupt 10 using 1
{
    EA = 0;
    Xtra_Int(CP0FIF_INT); /* Passes CP0FIF_INT for identification purposes            */
}
#endif


/*
****************************************************************************************
*
* Function name: Xtra_Int_11
*
* Function type: Interrupt Service Routine
*
* Description  : This is the CP0 interrupt ISR whose associated interrupt number is 11.
*
* Arguments    : None
*
* Returns      : None
*
****************************************************************************************
*/


#if (!STAND_ALONE_ISR_11)
void Xtra_Int_11 (void) interrupt 11 using 1
{
    EA = 0;
    Xtra_Int(CP0RIF_INT); /* Passes CP0RIF_INT for identification purposes            */
}
#endif


/*
****************************************************************************************
*
* Function name: Xtra_Int_12
*
* Function type: Interrupt Service Routine
*
```

144

```
* Description : This is the CP1 interrupt ISR whose associated interrupt number is 12.
*
* Arguments   : None
*
* Returns     : None
*
*****************************************************************************************
*/

#if (!STAND_ALONE_ISR_12)
void Xtra_Int_12 (void) interrupt 12 using 1
{
    EA = 0;
    Xtra_Int(CP1FIF_INT); /* Passes CP1FIF_INT for identification purposes              */
}
#endif

/*
*****************************************************************************************
*
* Function name: Xtra_Int_13
*
* Function type: Interrupt Service Routine
*
* Description : This is the CP1 interrupt ISR whose associated interrupt number is 13.
*
* Arguments   : None
*
* Returns     : None
*
*****************************************************************************************
*/

#if (!STAND_ALONE_ISR_13)
void Xtra_Int_13 (void) interrupt 13 using 1
{
    EA = 0;
    Xtra_Int(CP1RIF_INT); /* Passes CP1RIF_INT for identification purposes              */
}
#endif

/*
```
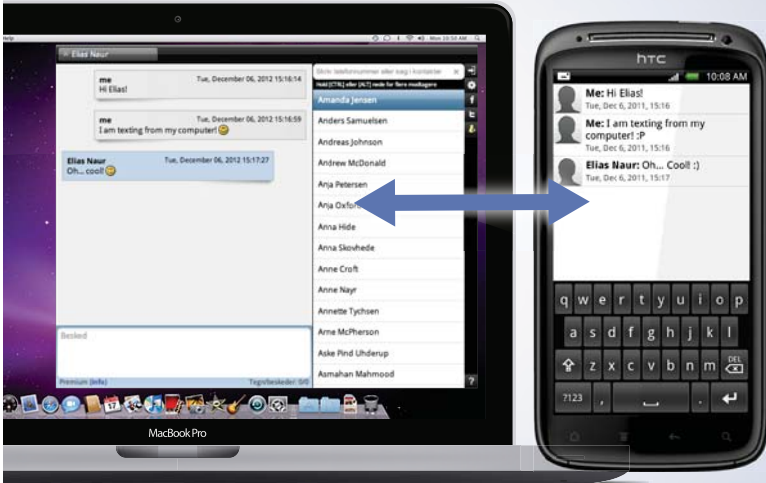
145

```
/***************************************************************************

*

* Function name: Xtra_Int_14

*

* Function type: Interrupt Service Routine

*

* Description  : This is the Timer 3 interrupt ISR whose associated interrupt number is 14.

*

* Arguments    : None

*

* Returns      : None

*

***************************************************************************

*/
```

```
#if ( (TICK_TIMER != 3) && (!STAND_ALONE_ISR_14) )

void Xtra_Int_14 (void) interrupt 14 using 1

{

    EA = 0;

    TMR3CN &= 0x7F; /* may be cleared in the task itself */

    Xtra_Int(TF3_INT); /* Passes TF3_INT for identification purposes                        */

}

#endif


/*

*********************************************************************************************

*

* Function name: Xtra_Int_15

*

* Function type: Interrupt Service Routine

*

* Description  : This is the AD0 interrupt ISR whose associated interrupt number is 15.

*

* Arguments    : None

*

* Returns      : None

*

*********************************************************************************************

**

*/


#if (!STAND_ALONE_ISR_15)

void Xtra_Int_15 (void) interrupt 15 using 1

{

    EA = 0;

    Xtra_Int(AD0INT_INT); /* Passes AD0INT_INT for identification purposes                 */

}

#endif


/*

*********************************************************************************************

*

* Function name: Xtra_Int_16

*

* Function type: Interrupt Service Routine
```

```
*
* Description  : This is the Timer 4 interrupt ISR whose associated interrupt number is 16.
*
* Arguments    : None
*
* Returns      : None
*
*********************************************************************************************
*/


#if (!STAND_ALONE_ISR_16)
void Xtra_Int_16 (void) interrupt 16 using 1
{
    EA = 0;
    Xtra_Int(TF4_INT); /* Passes TF4_INT for identification purposes                      */
}
#endif


/*
*********************************************************************************************
*
* Function name: Xtra_Int_17
*
* Function type: Interrupt Service Routine
*
* Description  : This is the AD1 interrupt ISR whose associated interrupt number is 17.
*
* Arguments    : None
*
* Returns      : None
*
*********************************************************************************************
*/


#if (!STAND_ALONE_ISR_17)
void Xtra_Int_17 (void) interrupt 17 using 1
{
    EA = 0;
    Xtra_Int(AD1INT_INT);    /* Passes AD1INT_INT for identification purposes */
}
#endif
```

```
/*
**********************************************************************************
*
* Function name: Xtra_Int_18
*
* Function type: Interrupt Service Routine
*
* Description  : This is the IE6 interrupt ISR whose associated interrupt number is 18.
*
* Arguments     : None
*
* Returns       : None
*
**********************************************************************************
*/
```

```c
#if (!STAND_ALONE_ISR_18)

void Xtra_Int_18 (void) interrupt 18 using 1

{

    EA = 0;

    Xtra_Int(IE6_INT);        /* Passes IE6_INT for identification purposes */

}

#endif


/*
*********************************************************************************************
*
* Function name: Xtra_Int_19
*
* Function type: Interrupt Service Routine
*
* Description  : This is the IE7 interrupt ISR whose associated interrupt number is 19.
*
* Arguments    : None
*
* Returns      : None
*
*********************************************************************************************
*/


#if (!STAND_ALONE_ISR_19)

void Xtra_Int_19 (void) interrupt 19 using 1

{

    EA = 0;

    Xtra_Int(IE7_INT); /* Passes IE7_INT for identification purposes                      */

}

#endif


/*
*********************************************************************************************
*
* Function name: Xtra_Int_20
*
* Function type: Interrupt Service Routine
*
* Description  : This is the UART1 interrupt ISR whose associated interrupt number is 20.
*
```

```
* Arguments   : None
*
* Returns     : None
*
********************************************************************************
*/


#if (!STAND_ALONE_ISR_20)
void Xtra_Int_20 (void) interrupt 20 using 1
{
    EA = 0;
    Xtra_Int(UART1_INT); /* Passes UART1_INT for identification purposes          */
}
#endif


/*
********************************************************************************
*
* Function name: Xtra_Int_21
*
* Function type: Interrupt Service Routine
*
* Description  : This is the XTL interrupt ISR whose associated interrupt number is 21.
*
* Arguments   : None
*
* Returns     : None
*
********************************************************************************
*/


#if (!STAND_ALONE_ISR_21)
void Xtra_Int_21 (void) interrupt 21 using 1
{
    EA = 0;
    Xtra_Int(XTLVLD_INT); /* Passes XTLVLD_INT for identification purposes          */
}
#endif


/*
********************************************************************************
```
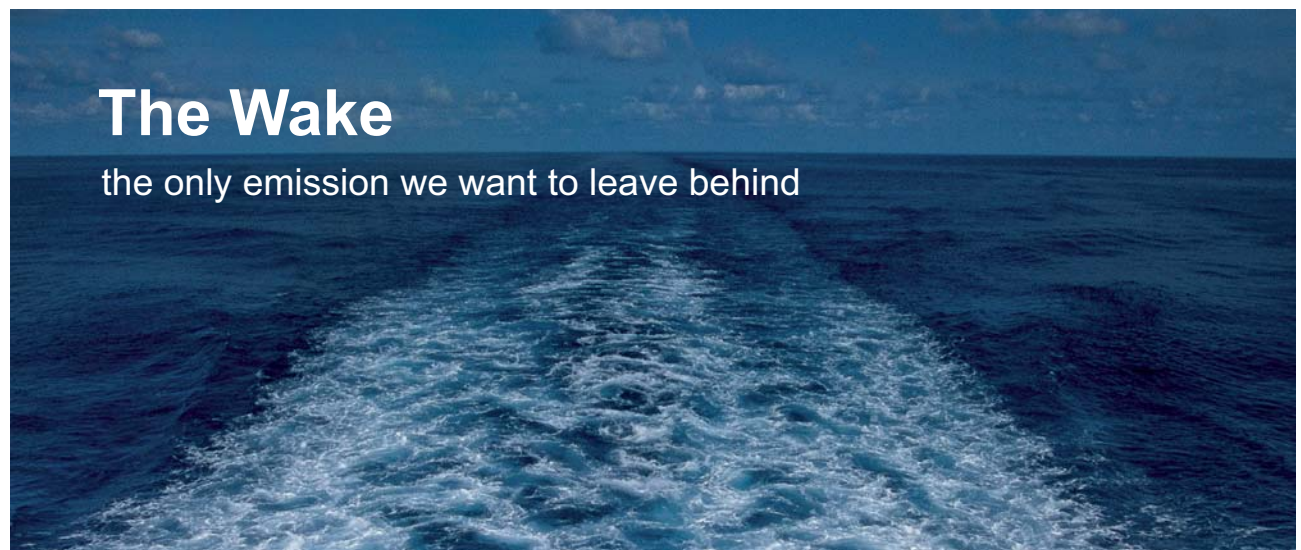
```
*/


/*
*********************************************************************************

*

* Function name: Xtra_Int

*

* Function type: Interrupt Handling (Internal function)

*

* Description  : This function performs the operations required by the previous ISRs.

*

* Arguments    : task_intflag      Represents the flag mask for a given interrupt

*                                  against which the

*                                  byte storing the flags of each task will be

*                                  compared in order to

*                                  determine whether any task has been waiting for

*                                  the interrupt in question.

*

* Returns      : None

*
```

```
*********************************************************************************************
*/


void Xtra_Int (uchar current_intnum) using 1

{

    uchar data k;

    IntFlag = 0;

    for (k = 0; k < NOOFTASKS; k ++)

    {

        if (task[k].intnum == current_intnum)

        {

            task[k].intnum = NO_INTERRUPT;

            IntFlag = 1;

            task[k].timeout = NOT_TIMING;       /* If it found that a task */

            ReadyQTop++;                        /* has been waiting for the */

            *ReadyQTop = k;                     /* given interrupt, it no    */

        }                                       /* longer requires to wait */

    }                                           /* and is therefore placed */

                                                /* on the READY queue. */


    if ((IntFlag == 1) && (Running == IDLE_TASK))

    {

        TinQFlag = 1;       /* If tasks are known to now reside in the */

        QShift();           /* READY queue while the idle task is */

    }                       /* running, a task switch is purposely    */

    /* induced, such that these tasks can run. */


    else if ((IntFlag == 1) && (Running != IDLE_TASK))

    {

                           /* Otherwise, the ISR exits after */

        TinQFlag = 1;      /* interrupts are re-enabled, */

                           /* since the RTOS cannot pre-empt task */

        EA = 1;

    }

    else EA = 1;           /* Otherwise exit normally */

}


/*
*********************************************************************************************
*/
```

A.5   C8051F020.H

```
#ifndef _C8051F020_H_

#define _C8051F020_H_


/*-------------------------------------------------------------------------------------------------------
;        Copyright (C) 2001 CYGNAL INTEGRATED PRODUCTS, INC.

;        All rights reserved.

;

;

;        FILE NAME       : C8051F020.H

;        TARGET MCUs  : C8051F020, 'F021, 'F022, 'F023

;        DESCRIPTION   : Register/bit definitions for the C8051F02x product family.

;

;        REVISION 1.1

;

;        Extra sfr16 declarations and additional bit definitions have been added to the non-bit-addressable

;                   SFRs – P. Debono

;----------------------------------------------------------------------------------------------------*/


/* BYTE Registers */


sfr P0                  = 0x80;        /* PORT 0                               */
sfr SP                  = 0x81  ;      /* STACK POINTER                    */
sfr DPL                 = 0x82;        /* DATA POINTER – LOW BYTE          */
sfr DPH                 = 0x83;        /* DATA POINTER – HIGH BYTE         */
sfr P4                  = 0x84;        /* PORT 4                          */
sfr P5                  = 0x85;        /* PORT 5                        */
sfr P6                  = 0x86; /*      PORT 6                          */
sfr PCON                = 0x87;        /* POWER CONTROL              */
sfr TCON                = 0x88;        /* TIMER CONTROL                    */
sfr TMOD                = 0x89;        /* TIMER MODE                     */
sfr TL0                 = 0x8A;        /* TIMER 0 – LOW BYTE              */
sfr TL1                 = 0x8B;        /* TIMER 1 – LOW BYTE              */
sfr TH0                 = 0x8C;        /* TIMER 0 – HIGH BYTE            */
sfr TH1                 = 0x8D;        /* TIMER 1 – HIGH BYTE            */
sfr CKCON               = 0x8E;        /* CLOCK CONTROL                 */
sfr PSCTL               = 0x8F;        /* PROGRAM STORE R/W CONTROL     */
sfr P1                  = 0x90;        /* PORT 1                        */
sfr TMR3CN              = 0x91;        /* TIMER 3 CONTROL            */
sfr TMR3RLL             = 0x92;        /* TIMER 3 RELOAD REGISTER – LOW BYTE */
```

154

```
sfr TMR3RLH        = 0x93;        /* TIMER 3 RELOAD REGISTER – HIGH BYTE */
sfr TMR3L          = 0x94;        /* TIMER 3 – LOW BYTE                 */
sfr TMR3H          = 0x95 ;       /* TIMER 3 – HIGH BYTE                */
sfr P7             = 0x96;        /* PORT 7                          */
sfr SCON0          = 0x98;        /* SERIAL PORT 0 CONTROL              */
sfr SBUF0          = 0x99 ;       /* SERIAL PORT 0 BUFFER               */
sfr SPI0CFG        = 0x9A;        /* SERIAL PERIPHERAL INTERFACE 0 CONFIGURATION  */
sfr SPI0DAT        = 0x9B;        /* SERIAL PERIPHERAL INTERFACE 0 DATA */
sfr ADC1           = 0x9C;        /* ADC 1 DATA                      */
sfr SPI0CKR        = 0x9D;        /* SERIAL PERIPHERAL INTERFACE 0 CLOCK RATE CONTROL */
sfr CPT0CN         = 0x9E;        /* COMPARATOR 0 CONTROL    */
sfr CPT1CN         = 0x9F;        /* COMPARATOR 1 CONTROL    */
sfr P2             = 0xA0;        /* PORT  2                 */
sfr EMI0TC         = 0xA1;         /* EMIF TIMING CONTROL            */
sfr EMI0CF         = 0xA3;        /* EXTERNAL MEMORY INTERFACE (EMIF) CONFIGURATION */
sfr P0MDOUT        = 0xA4;        /* PORT 0 OUTPUT MODE CONFIGURATION   */
sfr P1MDOUT        = 0xA5;        /* PORT 1 OUTPUT MODE CONFIGURATION   */
sfr P2MDOUT        = 0xA6;        /* PORT 2 OUTPUT MODE CONFIGURATION   */
sfr P3MDOUT        = 0xA7;        /* PORT 3 OUTPUT MODE CONFIGURATION   */
sfr IE             = 0xA8;        /* INTERRUPT ENABLE            */
```

```
sfr SADDR0          = 0xA9;         /* SERIAL PORT 0 SLAVE ADDRESS              */
sfr ADC1CN          = 0xAA;         /* ADC 1 CONTROL                    */
sfr ADC1CF          = 0xAB;         /* ADC 1 ANALOG MUX CONFIGURATION       */
sfr AMX1SL          = 0xAC;         /* ADC 1 ANALOG MUX CHANNEL SELECT   */
sfr P3IF            = 0xAD;         /* PORT 3 EXTERNAL INTERRUPT FLAGS      */
sfr SADEN1          = 0xAE;         /* SERIAL PORT 1 SLAVE ADDRESS MASK        */
sfr EMI0CN          = 0xAF;         /* EXTERNAL MEMORY INTERFACE CONTROL  */
sfr P3              = 0xB0;         /* PORT 3                      */
sfr OSCXCN          = 0xB1 ;        /* EXTERNAL OSCILLATOR CONTROL     */
sfr OSCICN          = 0xB2;         /* INTERNAL OSCILLATOR CONTROL    */
sfr P74OUT          = 0xB5;         /* PORTS 4 – 7 OUTPUT MODE  */
sfr FLSCL           = 0xB6;         /* FLASH MEMORY TIMING PRESCALER            */
sfr FLACL           = 0xB7;         /* FLASH ACESS LIMIT   */
sfr IP              = 0xB8;         /* INTERRUPT PRIORITY          */
sfr SADEN0          = 0xB9;         /* SERIAL PORT 0 SLAVE ADDRESS MASK       */
sfr AMX0CF          = 0xBA;         /* ADC 0 MUX CONFIGURATION          */
sfr AMX0SL          = 0xBB;         /* ADC 0 MUX CHANNEL SELECTION    */
sfr ADC0CF          = 0xBC;         /* ADC 0 CONFIGURATION              */
sfr P1MDIN          = 0xBD;         /* PORT 1 INPUT MODE             */
sfr ADC0L           = 0xBE;         /* ADC 0 DATA – LOW BYTE             */
sfr ADC0H           = 0xBF;         /* ADC 0 DATA – HIGH BYTE           */
sfr SMB0CN          = 0xC0;         /* SMBUS 0 CONTROL        */
sfr SMB0STA         = 0xC1;         /* SMBUS 0 STATUS         */
sfr SMB0DAT         = 0xC2;         /* SMBUS 0 DATA          */
sfr SMB0ADR         = 0xC3;         /* SMBUS 0 SLAVE ADDRESS            */
sfr ADC0GTL         = 0xC4;         /* ADC 0 GREATER-THAN REGISTER – LOW BYTE        */
sfr ADC0GTH         = 0xC5;         /* ADC 0 GREATER-THAN REGISTER – HIGH BYTE       */
sfr ADC0LTL         = 0xC6;         /* ADC 0 LESS-THAN REGISTER – LOW BYTE  */
sfr ADC0LTH         = 0xC7;         /* ADC 0 LESS-THAN REGISTER – HIGH BYTE */
sfr T2CON           = 0xC8;         /* TIMER 2 CONTROL            */
sfr T4CON           = 0xC9;         /* TIMER 4 CONTROL            */
sfr RCAP2L          = 0xCA;         /* TIMER 2 CAPTURE REGISTER – LOW BYTE    */
sfr RCAP2H          = 0xCB;         /* TIMER 2 CAPTURE REGISTER – HIGH BYTE   */
sfr TL2             = 0xCC;         /* TIMER 2 – LOW BYTE                */
sfr TH2             = 0xCD;         /* TIMER 2 – HIGH BYTE              */
sfr SMB0CR          = 0xCF;         /* SMBUS 0 CLOCK RATE                 */
sfr PSW             = 0xD0;         /* PROGRAM STATUS WORD          */
sfr REF0CN          = 0xD1;         /* VOLTAGE REFERENCE 0 CONTROL           */
sfr DAC0L           = 0xD2;         /* DAC 0 REGISTER – LOW BYTE              */
sfr DAC0H           = 0xD3;         /* DAC 0 REGISTER – HIGH BYTE           */
```

```
sfr DAC0CN          = 0xD4;        /* DAC 0 CONTROL                              */
sfr DAC1L           = 0xD5;        /* DAC 1 REGISTER – LOW BYTE               */
sfr DAC1H           = 0xD6;        /* DAC 1 REGISTER – HIGH BYTE              */
sfr DAC1CN          = 0xD7;        /* DAC 1 CONTROL                              */
sfr PCA0CN          = 0xD8;        /* PCA 0 COUNTER CONTROL            */
sfr PCA0MD          = 0xD9;        /* PCA 0 COUNTER MODE                      */
sfr PCA0CPM0        = 0xDA;        /* CONTROL REGISTER FOR PCA 0 MODULE 0   */
sfr PCA0CPM1        = 0xDB;        /* CONTROL REGISTER FOR PCA 0 MODULE 1   */
sfr PCA0CPM2        = 0xDC;        /* CONTROL REGISTER FOR PCA 0 MODULE 2   */
sfr PCA0CPM3        = 0xDD;        /* CONTROL REGISTER FOR PCA 0 MODULE 3   */
sfr PCA0CPM4        = 0xDE;        /* CONTROL REGISTER FOR PCA 0 MODULE 4   */
sfr ACC             = 0xE0;        /* ACCUMULATOR                               */
sfr XBR0            = 0xE1;        /* DIGITAL CROSSBAR CONFIGURATION REGISTER 0 */
sfr XBR1            = 0xE2;        /* DIGITAL CROSSBAR CONFIGURATION REGISTER 1 */
sfr XBR2            = 0xE3;        /* DIGITAL CROSSBAR CONFIGURATION REGISTER 2 */
sfr RCAP4L          = 0xE4;        /* TIMER 4 CAPTURE REGISTER – LOW BYTE    */
sfr RCAP4H          = 0xE5;         /* TIMER 4 CAPTURE REGISTER – HIGH BYTE  */
sfr EIE1            = 0xE6;        /* EXTERNAL INTERRUPT ENABLE 1      */
sfr EIE2            = 0xE7;        /* EXTERNAL INTERRUPT ENABLE 2      */
sfr ADC0CN          = 0xE8;        /* ADC 0 CONTROL                      */
sfr PCA0L           = 0xE9;        /* PCA 0 TIMER – LOW BYTE                 */
sfr PCA0CPL0        = 0xEA;         /* CAPTURE/COMPARE REG. PCA 0 MODULE 0 – LOW BYTE */
sfr PCA0CPL1        = 0xEB;         /* CAPTURE/COMPARE REG. PCA 0 MODULE 1 – LOW BYTE */
sfr RCA0CPL2        = 0xEC;         /* CAPTURE/COMPARE REG. PCA 0 MODULE 2 – LOW BYTE */
sfr PCA0CPL3        = 0xED;         /* CAPTURE/COMPARE REG. PCA 0 MODULE 3 – LOW BYTE */
sfr PCA0CPL4        = 0xEE;        /* CAPTURE/COMPARE REG.PCA 0 MODULE 4 – LOW BYTE */
sfr RSTSRC          = 0xEF;        /* RESET SOURCE                          */
sfr B               = 0xF0;        /* B REGISTER             */
sfr SCON1           = 0xF1;        /* SERIAL PORT 1 CONTROL                     */
sfr SBUF1           = 0xF2;        /* SERAIL PORT 1 DATA */
sfr SADDR1          = 0xF3;        /* SERAIL PORT 1 */
sfr TL4             = 0xF4;        /* TIMER 4 DATA – LOW BYTE */
sfr TH4             = 0xF5;        /* TIMER 4 DATA – HIGH BYTE */
sfr EIP1            = 0xF6;        /* EXTERNAL INTERRUPT PRIORITY REGISTER 1 */
sfr EIP2            = 0xF7;        /* EXTERNAL INTERRUPT PRIORITY REGISTER 2 */
sfr SPI0CN          = 0xF8;        /* SERIAL PERIPHERAL INTERFACE 0 CONTROL */
sfr PCA0H           = 0xF9;        /* PCA 0 TIMER – HIGH BYTE */
sfr PCA0CPH0        = 0xFA;        /* CAPTURE/COMPARE REG. PCA 0 MODULE 0 – HIGH BYTE */
sfr PCA0CPH1        = 0xFB;        /* CAPTURE/COMPARE REG. PCA 0 MODULE 1 – HIGH BYTE */
sfr PCA0CPH2        = 0xFC;        /* CAPTURE/COMPARE REG.PCA 0 MODULE 2 – HIGH BYTE */
```
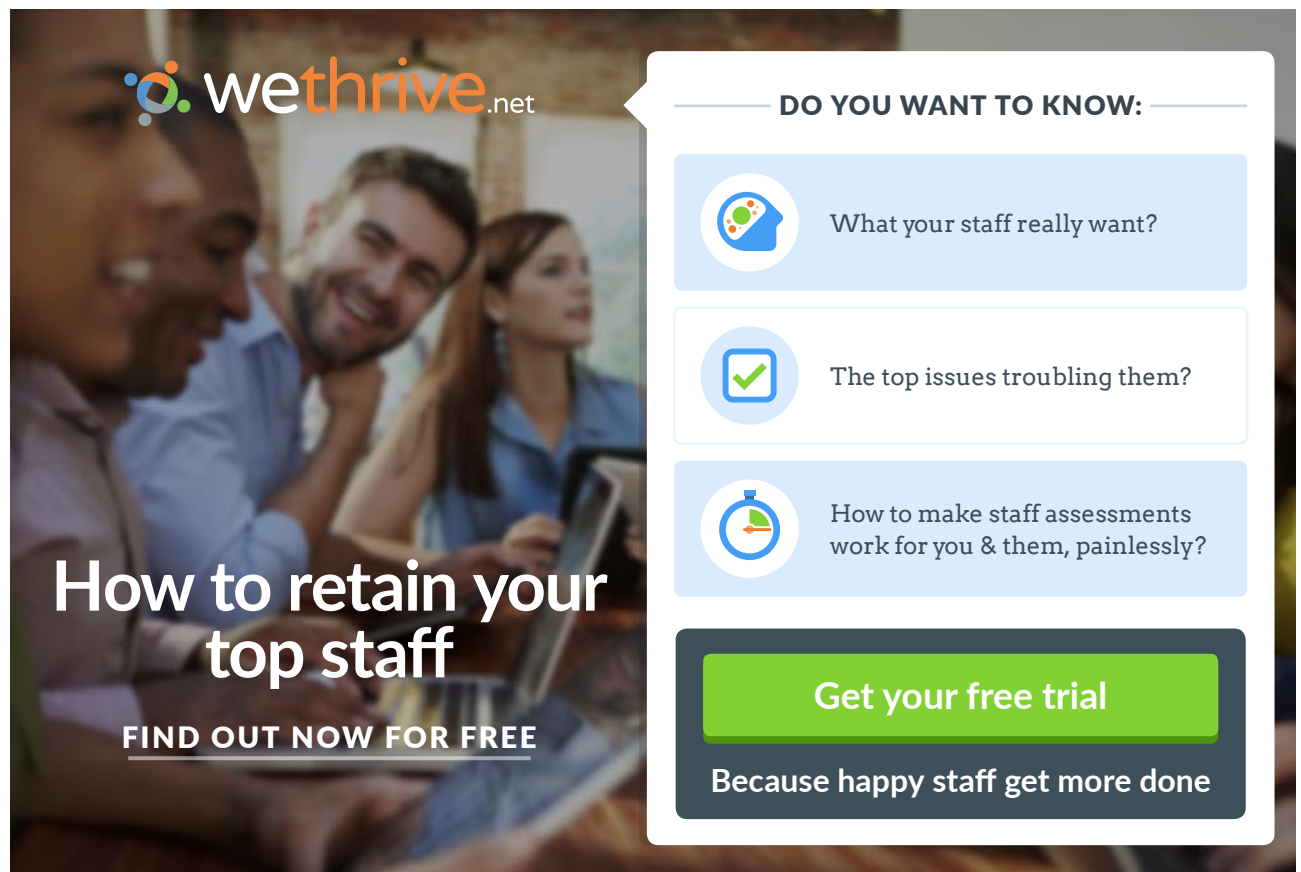
```
sfr PCA0CPH3          = 0xFD;        /* CAPTURE/COMPARE REG.PCA 0 MODULE 3 – HIGH BYTE */
sfr PCA0CPH4          = 0xFE;        /* CAPTURE/COMPARE REG.PCA 0 MODULE 4 – HIGH BYTE */
sfr WDTCN             = 0xFF;        /* WATCHDOG TIMER CONTROL */
```

```
/*
16-bit sfr Definitions – enabling 16-bit registers which have consecutive addresses for their low and high byte, to
be loaded with one command, aligning the low and high byte correctly (little endian).
*/
```

```
sfr16 DP         = 0x82;        // data pointer
sfr16 TMR3RL     = 0x92;        // Timer 3 reload value
sfr16 TMR3       = 0x94;        // Timer 3 counter
sfr16 ADC0       = 0xBE;        // ADC0 data
sfr16 ADC0GT     = 0xC4;        // ADC0 greater than window
sfr16 ADC0LT     = 0xC6;        // ADC0 less than window
sfr16 RCAP2      = 0xCA;        // Timer 2 capture/reload
sfr16 T2         = 0xCC;        // Timer 2
sfr16 RCAP4      = 0xE4;        // Timer 4 capture/reload
sfr16 T4         = 0xF4;        // Timer 4
sfr16 DAC0       = 0xD2;        // DAC0 data
sfr16 DAC1       = 0xD5;        // DAC1 data
```

```
/* BIT Registers */


/* TCON 0x88 */
sbit TF1            = TCON ^ 7;      /* TIMER 1 OVERFLOW FLAG     */
sbit TR1             = TCON ^ 6;      /* TIMER 1 ON/OFF CONTROL   */
sbit TF0            = TCON ^ 5;      /* TIMER 0 OVERFLOW FLAG     */
sbit TR0             = TCON ^ 4;      /* TIMER 0 ON/OFF CONTROL  */
sbit IE1            = TCON ^ 3;      /* EXT. INTERRUPT 1 EDGE FLAG */
sbit IT1            = TCON ^ 2;      /* EXT. INTERRUPT 1 TYPE       */
sbit IE0            = TCON ^ 1;      /* EXT. INTERRUPT 0 EDGE FLAG */
sbit IT0            = TCON ^ 0;      /* EXT. INTERRUPT 0 TYPE       */


/* SCON0 0x98 */
sbit SM00            = SCON0 ^ 7;     /* SERIAL MODE CONTROL BIT 0        */
sbit SM10           = SCON0 ^ 6;     /* SERIAL MODE CONTROL BIT 1       */
sbit SM20            = SCON0 ^ 5;     /* MULTIPROCESSOR COMMUNICATION ENABLE      */
sbit REN0           = SCON0 ^ 4;     /* RECEIVE ENABLE               */
sbit TB80           = SCON0 ^ 3;     /* TRANSMIT BIT 8                */
sbit RB80           = SCON0 ^ 2;     /* RECEIVE BIT 8                */
sbit TI0            = SCON0 ^ 1;     /* TRANSMIT INTERRUPT FLAG               */
sbit RI0            = SCON0 ^ 0;     /* RECEIVE INTERRUPT FLAG   */


/* IE 0xA8 */
sbit EA            = IE ^ 7;        /* GLOBAL INTERRUPT ENABLE         */
sbit ET2           = IE ^ 5;        /* TIMER 2 INTERRUPT ENABLE        */
sbit ES0           = IE ^ 4;        /* UART0 INTERRUPT ENABLE         */
sbit ET1           = IE ^ 3;        /* TIMER 1 INTERRUPT ENABLE        */
sbit EX1           = IE ^ 2;        /* EXTERNAL INTERRUPT 1 ENABLE     */
sbit ET0           = IE ^ 1;        /* TIMER 0 INTERRUPT ENABLE        */
sbit EX0           = IE ^ 0;        /* EXTERNAL INTERRUPT 0 ENABLE     */


/* IP 0xB8 */
sbit PT2           = IP ^ 5;        /* TIMER 2 PRIORITY                        */
sbit PS            = IP ^ 4;        /* SERIAL PORT PRIORITY                   */
sbit PT1           = IP ^ 3;         /* TIMER 1 PRIORITY                        */
sbit PX1           = IP ^ 2;        /* EXTERNAL INTERRUPT 1 PRIORITY           */
sbit PT0           = IP ^ 1;        /* TIMER 0 PRIORITY                        */
sbit PX0           = IP ^ 0;        /* EXTERNAL INTERRUPT 0 PRIORITY           */
```

```
/* SMB0CN 0xC0 */

sbit BUSY        = SMB0CN ^ 7;        /* SMBUS 0 BUSY                    */
sbit ENSMB       = SMB0CN ^ 6;        /* SMBUS 0 ENABLE                  */
sbit STA         = SMB0CN ^ 5;        /* SMBUS 0 START FLAG              */
sbit STO         = SMB0CN ^ 4;        /* SMBUS 0 STOP FLAG    */
sbit SI          = SMB0CN ^ 3;        /* SMBUS 0 INTERRUPT PENDING FLAG */
sbit AA          = SMB0CN ^ 2;        /* SMBUS 0 ASSERT/ACKNOWLEDGE FLAG */
sbit SMBFTE      = SMB0CN ^ 1;        /* SMBUS 0 FREE TIMER ENABLE   */
sbit SMBTOE      = SMB0CN ^ 0;        /* SMBUS 0 TIMEOUT ENABLE   */


/* T2CON 0xC8 */

sbit TF2         = T2CON ^ 7;   /* TIMER 2 OVERFLOW FLAG   */
sbit EXF2        = T2CON ^ 6;   /* EXTERNAL FLAG              */
sbit RCLK0       = T2CON ^ 5;   /* UART0 RX CLOCK SOURCE    */
sbit TCLK0       = T2CON ^ 4;   /* UART0 TX CLOCK SOURCE   */
sbit EXEN2       = T2CON ^ 3;   /* TIMER 2 EXTERNAL ENABLE FLAG */
sbit TR2         = T2CON ^ 2;   /* TIMER 2 ON/OFF CONTROL    */
sbit CT2         = T2CON ^ 1;   /* TIMER OR COUNTER SELECT     */
sbit CPRL2       = T2CON ^ 0;   /* CAPTURE OR RELOAD SELECT         */


/* PSW */

sbit CY          = PSW ^ 7;     /* CARRY FLAG           */
sbit AC          = PSW ^ 6;     /* AUXILIARY CARRY FLAG      */
sbit F0          = PSW ^ 5;     /* USER FLAG 0          */
sbit RS1         = PSW ^ 4;     /* REGISTER BANK SELECT 1 */
sbit RS0         = PSW ^ 3;     /* REGISTER BANK SELECT 0 */
sbit OV          = PSW ^ 2;     /* OVERFLOW FLAG       */
sbit F1          = PSW ^ 1;     /* USER FLAG 1          */
sbit P           = PSW ^ 0;     /* ACCUMULATOR PARITY FLAG         */


/* PCA0CN D8H */

sbit CF          = PCA0CN ^ 7; /* PCA 0 COUNTER OVERFLOW FLAG */
sbit CR          = PCA0CN ^ 6; /* PCA 0 COUNTER RUN CONTROL BIT */
sbit CCF4        = PCA0CN ^ 4; /* PCA 0 MODULE 4 INTERRUPT FLAG */
sbit CCF3        = PCA0CN ^ 3; /* PCA 0 MODULE 3 INTERRUPT FLAG */
sbit CCF2        = PCA0CN ^ 2; /* PCA 0 MODULE 2 INTERRUPT FLAG */
sbit CCF1        = PCA0CN ^ 1; /* PCA 0 MODULE 1 INTERRUPT FLAG */
sbit CCF0        = PCA0CN ^ 0;  /* PCA 0 MODULE 0 INTERRUPT FLAG */
```

```
/* ADC0CN E8H */

sbit AD0EN      = ADC0CN ^ 7;        /* ADC 0 ENABLE                          */
sbit AD0TM      = ADC0CN ^ 6;         /* ADC 0 TRACK MODE                     */
sbit AD0INT     = ADC0CN ^ 5;         /* ADC 0 CONVERISION COMPLETE INTERRUPT FLAG    */
sbit AD0BUSY    = ADC0CN ^ 4;        /* ADC 0 BUSY FLAG            */
sbit AD0CM1     = ADC0CN ^ 3;         /* ADC 0 START OF CONVERSION MODE BIT 1  */
sbit AD0CM0     = ADC0CN ^ 2;        /* ADC 0 START OF CONVERSION MODE BIT 0   */
sbit AD0WINT    = ADC0CN ^ 1;        /* ADC 0 WINDOW COMPARE INTERRUPT FLAG   */
sbit AD0LJST    = ADC0CN ^ 0;        /* ADC 0 RIGHT JUSTIFY DATA BIT          */


/* SPI0CN F8H */

sbit SPIF       = SPI0CN ^ 7;    /* SPI 0 INTERRUPT FLAG                 */
sbit WCOL       = SPI0CN ^ 6;    /* SPI 0 WRITE COLLISION FLAG        */
sbit MODF       = SPI0CN ^ 5;    /* SPI 0 MODE FAULT FLAG        */
sbit RXOVRN     = SPI0CN ^ 4;    /* SPI 0 RX OVERRUN FLAG        */
sbit TXBSY      = SPI0CN ^ 3;    /* SPI 0 TX BUSY FLAG           */
sbit SLVSEL     = SPI0CN ^ 2;    /* SPI 0 SLAVE SELECT          */
sbit MSTEN      = SPI0CN ^ 1;    /* SPI 0 MASTER ENABLE                 */
sbit SPIEN      = SPI0CN ^ 0;    /* SPI 0 SPI ENABLE                 */
```

```
/*
EXTRA: BIT definitions for bits held in SFRs that are not bit-addressable and hence not directly accessible
*/


/* TMOD Bits */
#define T0M0          0x01
#define T0M1          0x02
#define C_T0          0x04
#define GATE0         0x08
#define T1M0          0x10
#define T1M1          0x20
#define C_T1          0x40
#define GATE1         0x80


/* CKCON Bits */
#define T0M           0x08    /* Timer 0 clock select */
#define T1M           0x10    /* Timer 1 clock select */
#define T2M           0x20    /* Timer 2 clock select */
#define T4M           0x40    /* Timer 4 clock select */


/* PSCTL Bits */
#define PSWE          0x01    /* Program Store Write Enable */
#define PSEE          0x02    /* Program Store Erase Enable */
#define SFLE          0x04    /* Scratch pad Flash memory access enable */


/* TMR3CN Bits */
#define T3XCLK        0x01    /* Timer 3 external clock select */
#define T3M           0x02    /* Timer 3 clock select */
#define TR3           0x04    /* Timer 3 Run control */
#define TF3           0x80    /* Timer 3 overflow flag */


/* P7 Bits */
#define P7_0          0x01
#define P7_1          0x02
#define P7_2          0x04
#define P7_3          0x08
#define P7_4          0x10
#define P7_5          0x20
#define P7_6          0x40
```

```
#define P7_7              0x80


/* SPI0CFG Bits */
#define SPIFRS0          0x01    /* SPI0 Frame Size, bit 0 */
#define SPIFRS1          0x02    /* SPI0 Frame Size, bit 1 */
#define SPIFRS2          0x04    /* SPI0 Frame Size, bit 2 */
#define BC0               0x08    /* SPI0 Bit Count, bit 0 */
#define BC1              0x10    /* SPI0 Bit Count, bit 1 */
#define BC2               0x20    /* SPI0 Bit Count, bit 2 */
#define CKPOL            0x40    /* SPI0 Clock polarity */
#define CKPHA            0x80    /* SPI0 Clock phase */


/* SPI0DAT Bits, data only no bits*/


/* ADCI Bits, data word register, no bits */


/* SPI0CKR Bits */
#define SCR0             0x01    /* SPI0 Clock Rate */
#define SCR1             0x02
#define SCR2             0x04
#define SCR3             0x08
#define SCR4             0x10
#define SCR5             0x20
#define SCR6             0x40
#define SCR7             0x80


/* CPT0CN Bits */
#define CP0HYN0          0x01    /* Comparator 0 negative hysteresis control, bit 0 */
#define CP0HYN1          0x02    /* Comparator 0 negative hysteresis control, bit 1 */
#define CP0HYP0          0x04    /* Comparator 0 positive hysteresis control, bit 0 */
#define CP0HYP1          0x08    /* Comparator 0 positive hysteresis control, bit 1 */
#define CP0FIF           0x10    /* Comparator 0 Falling Edge Interrupt Flag */
#define CP0RIF           0x20    /* Comparator 0 rising Edge Interrupt Flag */
#define CP0OUT           0x40    /* Comparator 0 Output state flag */
#define CP0EN            0x80    /* Comparator 0 Enable bit */


/* CPT1CN Bits */
#define CP1HYN0          0x01    /* Comparator 1 negative hysteresis control, bit 0 */
#define CP1HYN1          0x02    /* Comparator 1 negative hysteresis control, bit 1 */
#define CP1HYP0          0x04    /* Comparator 1 positive hysteresis control, bit 0 */
```

```
#define CP1HYP1          0x08      /* Comparator 1 positive hysteresis control, bit 1 */
#define CP1FIF           0x10      /* Comparator 1 Falling Edge Interrupt Flag */
#define CP1RIF           0x20      /* Comparator 1 rising Edge Interrupt Flag */
#define CP1OUT           0x40      /* Comparator 1 Output state flag */
#define CP1EN            0x80      /* Comparator 1 Enable bit */


/* EMI0TC Bits */
#define EAH0             0x01      /* EMIF Address Hold, bit 0 */
#define EAH1             0x02      /* EMIF Address Hold, bit 1 */
#define EWR0             0x04      /* EMIF /WR and /RD Pulse Width Control, bit 0 */
#define EWR1             0x08      /* EMIF /WR and /RD Pulse Width Control, bit 1 */
#define EWR2             0x10      /* EMIF /WR and /RD Pulse Width Control, bit 2 */
#define EWR3             0x20      /* EMIF /WR and /RD Pulse Width Control, bit 3 */
#define EAS0             0x40      /* EMIF Address setup time, bit 0 */
#define EAS1             0x80      /* EMIF Address setup time, bit 1 */


/* EMI0CF Bits */
#define EALE0            0x01      /* ALE pulse width select, bit 0 */
#define EALE1            0x02      /* ALE pulse width select, bit 1 */
#define EMD0             0x04      /* EMIF operating mode select, bit 0 */
```

```
#define EMD1          0x08      /* EMIF operating mode select, bit 1 */
#define EMD2          0x10      /* EMIF Multiplex mode select */
#define PRTSEL        0x20      /* EMIF Port Select */


/* FLSCL Bits */
#define FLWE          0x01      /* Flash Read/Write Enable */
#define FRAE          0x40      /* Flash Read Always Enable */
#define FOSE          0x80      /* Flash One shot timer enable */


/* ADC1CN Bits */
#define ADC1CM0       0x02      /* ADC1 Start of conversion mode select, bit 0 */
#define ADC1CM1       0x04      /* ADC1 Start of conversion mode select, bit 1 */
#define ADC1CM2       0x08      /* ADC1 Start of conversion mode select, bit 2 */
#define AD1BUSY       0x10      /* ADC1 Busy bit */
#define AD1INT        0x20      /* ADC1 Conversion complete interrupt flag */
#define AD1TM         0x40      /* ADC1 Track mode bit */
#define AD1EN         0x80      /* ADC1 Enable */


/* ADC1CF Bits */
#define AMP1GN0       0x01      /* ADC1 Internal amplifier Gain, bit 0 */
#define AMP1GN1       0x02      /* ADC1 Internal amplifier Gain, bit 1 */
#define AD1SC0        0x08      /* ADC1 SAR Conversion clock period bit 0 */
#define AD1SC1        0x10      /* ADC1 SAR Conversion clock period bit 1 */
#define AD1SC2        0x20      /* ADC1 SAR Conversion clock period bit 2 */
#define AD1SC3        0x40      /* ADC1 SAR Conversion clock period bit 3 */
#define AD1SC4        0x80      /* ADC1 SAR Conversion clock period bit 4 */


/* SMB0STA Bits */


/* SMB0ADR Bits */
#define GC            0x01      /* General call address enable */
#define SLV0          0x02      /* Slave address, bit 0 */
#define SLV1          0x04      /* Slave address, bit 1 */
#define SLV2          0x08      /* Slave address, bit 2 */
#define SLV3          0x10      /* Slave address, bit 3 */
#define SLV4          0x20      /* Slave address, bit 4 */
#define SLV5          0x40      /* Slave address, bit 5 */
#define SLV6          0x80      /* Slave address, bit 6 */
```

```
/* T4CON Bits */

#define CP_RL4        0x01      /* Timer 4 Capture/Reload select */

#define C_T4          0x02      /* Timer 4 Counter/Timer Select */

#define TR4           0x04      /* Timer 4 Run Control */

#define EXEN4         0x08      /* Timer 4 External Enable */

#define TCLK1         0x10      /* Transmit clock flag for UART 1 */

#define RCLK1         0x20      /* Receive clock flag for UART 1 */

#define EXF4          0x40      /* Timer 4 External Flag */

#define TF4           0x80      /* Timer 4 Overflow Flag */


/* SMB0CR */
/* REF0CN Bits */

#define REFBE         0x01      /* Internal reference buffer enable bit */

#define BIASE         0x02      /* ADC/DAC Bias generator enable bit */

#define TEMPE         0x04      /* Temperature sensor enable bit */

#define AD1VRS        0x08      /* ADC1 Voltage reference select */

#define AD0VRS        0x10      /* ADC0 Voltage reference select */


/* DAC0CN Bits */

#define DAC0DF0       0x01      /* DAC0 Data format bits, bit 0 */

#define DAC0DF1       0x02      /* DAC0 Data format bits, bit 1 */

#define DAC0DF2       0x04      /* DAC0 Data format bits, bit 2 */

#define DAC0MD0       0x08      /* DAC0 Mode bits, bit 0 */

#define DAC0MD1       0x10      /* DAC0 Mode bits, bit 1 */

#define DAC0EN        0x80      /* DAC0 enable bit */


/* DAC1CN Bits */

#define DAC1DF0       0x01      /* DAC1 Data format bits, bit 0 */

#define DAC1DF1       0x02      /* DAC1 Data format bits, bit 1 */

#define DAC1DF2       0x04      /* DAC1 Data format bits, bit 2 */

#define DAC0MD0       0x08      /* DAC0 Mode bits, bit 0 */

#define DAC0MD1       0x10      /* DAC0 Mode bits, bit 1 */

#define DAC0EN        0x80      /* DAC0 enable bit */


/* PCA0MD Bits */

#define ECF           0x01      /* PCA Counter/Timer Overflow Interrupt enable */

#define CPS0          0x02      /* PCA0 Counter/Timer Pulse Select, bit 0 */

#define CPS1          0x04      /* PCA0 Counter/Timer Pulse Select, bit 1 */

#define CPS2          0x08      /* PCA0 Counter/Timer Pulse Select, bit 2 */

#define CIDL          0x80      /* PCA0 Counter/Timer Idle control */
```

```
/* PCA0CPM0 Bits */

#define EECF0       0x01    /* ECCF0 Capture/Compare Flag Interrupt enable */
#define PWM0        0x02    /* PWM0 Pulse Width Modulbation Mode enable */
#define TOG0        0x04    /* TOG0 Toggle Function enable */
#define MAT0        0x08    /* MAT0 Match Function enable */
#define CAPN0       0x10    /* CAPN0 Capture negative function enable */
#define CAPP0       0x20    /* CAPP0 Capture positive function enable */
#define ECOM0   0x40    /* ECOM0 Comparator function enable */
#define PWM160 0x80    /* PWM160 16-bit PWM enable */


/* PCA0CPM1 Bits */

#define EECF1       0x01    /* ECCF1 Capture/Compare Flag Interrupt enable */
#define PWM1        0x02    /* PWM1 Pulse Width Modulation Mode enable */
#define TOG1        0x04    /* TOG1 Toggle Function enable */
#define MAT1        0x08    /* MAT1 Match Function enable */
#define CAPN1       0x10    /* CAPN1 Capture negative function enable */
#define CAPP1       0x20    /* CAPP1 Capture positive function enable */
#define ECOM1       0x40    /* ECOM1 Comparator function enable */
#define PWM161      0x80    /* PWM161 16-bit PWM Enable */
```

```
/* PCA0CPM2 Bits */

#define EECF2        0x01    /* ECCF2 Capture/Compare Flag Interrupt enable */
#define PWM2         0x02    /* PWM2 Pulse Width Modulbation Mode enable */
#define TOG2         0x04    /* TOG2 Toggle Function enable */
#define MAT2         0x08    /* MAT2 Match Function enable */
#define CAPN2        0x10    /* CAPN2 Capture negative function enable */
#define CAPP2        0x20    /* CAPP2 Capture positive function enable */
#define ECOM2        0x40    /* ECOM2 Comparator function enable */
#define PWM162       0x80    /* PWM162 16-bit PWM enable */


/* PCA0CPM3 Bits */

#define EECF3        0x01    /* ECCF3 Capture/Compare Flag Interrupt enable */
#define PWM3         0x02    /* PWM3 Pulse Width Modulation mode enable */
#define TOG3         0x04    /* TOG3 Toggle Function enable */
#define MAT3         0x08    /* MAT3 Match Function enable */
#define CAPN3        0x10    /* CAPN3 Capture negative function enable */
#define CAPP3        0x20    /* CAPP3 Capture positive function enable */
#define ECOM3        0x40    /* ECOM3 Comparator function enable */
#define PWM163       0x80    /* PWM163 16-bit PWM enable */


/* PCA0CPM4 Bits */

#define EECF4        0x01    /* ECCF4 Capture/Compare Flag Interrupt enable */
#define PWM4         0x02    /* PWM4 Pulse Width Modulation Mode enable */
#define TOG4         0x04    /* TOG4 Toggle Function enable */
#define MAT4         0x08    /* MAT4 Match Function enable */
#define CAPN4        0x10    /* CAPN4 Capture negative function enable */
#define CAPP4        0x20    /* CAPP4 Capture positive function enable */
#define ECOM4        0x40    /* ECOM4 Comparator function enable */
#define PWM164       0x80    /* PWM164 16-bit PWM enable */


/* XBR0 bits, PORT IO Crossbar Reg 0 */
#define SMB0EN       0x01    /* SMBus 0 Bus I/O Enable bit */
#define SPI0EN       0x02    /* SPIBus 0 Bus I/O Enable bit */
#define UART0EN      0x04    /* UART0 I/O Enable bit */
#define PCA0ME0      0x08    /* PCA0 Module I/O Enable bits, bit 0 */
#define PCA0ME1      0x10    /* PCA0 Module I/O Enable bits, bit 1 */
#define PCA0ME2      0x20    /* PCA0 Module I/O Enable bits, bit 2 */
#define ECI0E        0x40    /* PCA0 External Counter Input Enable bit */
#define CP0E         0x80    /* Comparator 0 Output Enable bit */
```

```
/* XBR1 bits, PORT IO Crossbar Reg 1 */

#define CPIE           0x01     /* CP1 output enable bit */
#define T0E            0x02     /* T0 input enable bit */
#define INT0E          0x04     /* /INT0 input enable bit */
#define T1E            0x08     /* T1 input enable bit */
#define INT1E          0x10     /* /INT1 input enable bit */
#define T2E            0x20     /* T2 input enable bit */
#define T2EXE          0x40     /* T2EX input enable bit */
#define SYSCKE         0x80     /* /SYSCLK output enable bit */


/* XBR2 bits, PORT IO Crossbar Reg 2 */

#define CNVSTE         0x01     /* External Convert Start enable bit */
#define EMIFLE         0x02     /* External Memory Interface low port enable bit */
#define UART1E         0x04     /* UART1 I/O enable bit */
#define T4E            0x08     /* T4 input enable bit */
#define T4EXE          0x10     /* T4EX input enable bit */
#define XBARE          0x40     /* Crossbar enable bit */
#define WEAKPUD        0x80     /* Weak pull-up disable bit */


/* IEI1 bits, Extended Interrupt Enable 1 */

#define ESPI0          0x01     /* Enable SPI0 interrupt */
#define ESMB0          0x02     /* Enable SMBus0 interrupt */
#define EWADC0         0x04     /* Enable Window Comparison ADC0 interrupt */
#define EPCA0          0x08     /* Enable PCA0 interrupt */
#define ECP0F          0x10     /* Enable comparator 0 (CP0) Falling edge interrupt */
#define ECP0R          0x20     /* Enable comparator 0 (CP0) rising edge interrupt */
#define ECP1F          0x40     /* Enable comparator 1 (CP1) Falling edge interrupt */
#define ECP1R          0x80     /* Enable comparator 1 (CP1) Rising edge interrupt */


/* EIE2 bits, Extended Interrupt Enable 2 */

#define ET3            0x01     /* Enable timer 3 interrupt */
#define EADC0          0x02     /* Enable ADC0 End of conversion interrupt */
#define ET4            0x04     /* Enable timer 3 interrupt */
#define EADC1          0x08     /* Enable ADC1 End of conversion interrupt */
#define EX6            0x10     /* Enable External interrupt 6 */
#define EX7            0x20     /* Enable External interrupt 7 */
#define ES1            0x40     /* Enable UART1 interrupt */
#define EXVLD          0x80     /* Enable External Clock source valid interrupt */
```

169

```
/* RSTSRC Bits */

#define PINRSF        0x01    /* Hardware Pin Reset Flag */

#define PORSF         0x02    /* Power-on Reset force and flag */

#define MCDRSF        0x04    /* Missing clock detector flag */

#define WDTRSF        0x08    /* Watchdog Timer reset flag */

#define SWRSEF        0x10    /* Software Reset Force and flag */

#define C0RSEF        0x20    /* Comparator 0 (CP0) Reset enable and flag */

#define CNVRSEF       0x40    /* Convert Start Reset source enable and flag */


/* SCON1 Bits */

#define RI1           0x01    /* UART1 Receive interrupt flag */

#define TI1           0x02    /* UART1 Transmit interrupt flag */

#define RB81          0x04    /* UART1 9th bit receive */

#define TB81          0x08    /* UART1 Ninth transmit bit */

#define REN1          0x10    /* UART1 Receiver enable */

#define SM21          0x20    /* UART1 Multiprocessor communication enable */

#define TXCOL1        0x20 /* UART1 Transmit Collision bit */

#define SM11          0x40    /* UART1 mode bit */

#define RXOV1         0x40 /* UART1 Receive Overflow bit */

#define SM01          0x80    /* UART1 mode bit */

#define FE1           0x80 /* UART1 Frame Error bit */
```

```
/* EIP1 Bits */
#define PSPI0          0x01     /* SPI0 interrupt priority control */
#define PSMB0          0x02     /* SMBus0 interrupt priority control */
#define PWADC0         0x04     /* ADC0 Window comparator interrupt priority control */
#define PPCA0          0x08     /* PCA0 interrupt priority control */
#define PCP0F          0x10     /* Comparator 0 (CP0) Falling interrupt priority control */
#define PCP0R          0x20     /* Comparator 0 (CP0) Rising interrupt priority control */
#define PCP1F          0x40     /* Comparator 1 (CP1) Falling interrupt priority control */
#define PCP1R          0x80     /* Comparator 1 (CP1) Rising interrupt priority control */


/* EIP2 Bits */
#define PT3            0x01     /* Timer 3 interrupt priority control */
#define PADC0          0x02     /* ADC0 End of Conversion interrupt priority control */
#define PT4             0x04     /* Timer 4 interrupt priority control */
#define PADC1          0x08     /* ADC1 End of Conversion interrupt priority control */
#define PX6            0x10     /* External interrupt 6 Priority Control */
#define PX7            0x20     /* External interrupt 7 Priority Control */
#define EP1            0x40     /* UART1 interrupt Priority Control */
#define PXVLD          0x80     /* External Clock Source valid interrupt priority control */


/* PCON bits */
#define IDLE           0x01     /* Idle mode select */
#define STOP           0x02     /* Stop mode select */
#define SSTAT1         0x08     /* UART1 Enhanced status mode select */
#define SMOD1          0x10     /* UART1 Baud Rate doubler enable */
#define SSTAT0         0x40     /* UART0 Enhanced status mode select */
#define SMOD0          0x80     /* UART0 Baud Rate doubler enable */


#endif // _C8051F020_H_


/* ======================================================= */
```

Appendix B Further Examples

We list here some interesting examples for the 8032 microprocessor. Some of them do not use any RTOS at all, but rely solely on interrupts. Other valuable examples can be found in the application note AN122 for the C8051F02x family (Silicon Labs, 2003a).

B.1   Timer 0 in Mode 3 (split timer) and Timer 1 as a baud rate generator

The first example is a program showing how we can use Timer 0 in the split mode. This is not often found detailed in most books, probably because nowadays, most of the advanced versions of the 8051 have 4 or more timers available. However, if still using the original 8051, this mode 3 would effectively increase the number of timers available.

In this example, the two timers from Timer 0 (here labelled as Timer 00 and Timer 000) both run as an 8-bit timer, generating interrupts. The main program checks whether the required number of interrupts have been generated, and prints a statement accordingly.

Timer 1 is used as a baud rate generator and since Timer 0 is running in mode 3, the only way to switch on and off this timer 1 is by changing its mode. If timer 1 is set to mode 3, it is stopped. Thus as an example, we are starting the timer only before printing and switching it off once we are done with the printing command.

```
/* TimersMode3.c */

/*

    Timer 0 runs in mode 3 mode, thus splitting it into two timers,

                  which we shall call Timer00 and Timer000


    Assuming that we are using a 22.1184MHz clock, then if the timers    are using sysclk/12
    as their counting pulse:
    Timer 00 generates interrupts every 78.125us,

            since it is set to count 144 times before it overflows

            hence 12800 interrupts would be equivalent to 1 second (using TL0, TF0)


    Timer 000 generates interrupts every 117.1875us,

            since it is set to count 216 times before it overflows

            hence 25600 interrupts would be equivalent to 3 seconds (using TH0, TF1)
    Timer 1 is used as the baud-rate generator, switching it on and off

                  by switching it out of and into its own mode 3. No interrupts available.
```

```
*/


#include "C8051F020.h"

#include "MySystem.h"

#include <stdio.h>


void SetUp_Timer0_M3 (void);

void SetUp_Timer1_M3_and_UART0 (void);

char putchar (char c);


/* Global variables */

bit T00_FLAG, T000_FLAG; // flags to indicate timer timeouts


/* --------------------------------------------- */


/*
 * putchar: outputs character, used by the printf command
 */

char putchar (char c) {

 while (!TI0);  /* wait for transmitter to be ready */
```

```
 TI0 = 0;

 return (SBUF0 = c);

}


/* ---------------------------------------- */


/* set up Timer 0 mode 3, GATE = C/T = 0 */

/* splitting it into two timers, Timer00 and Timer000 */

/* Assuming 22.1184 MHz clock */


/* 156.25 microsecond overflow for TF0 (normal Timer 00) */

/* 78.125 microsecond overflow for TF1 (extra Timer 000) */

void SetUp_Timer0_M3 (void)

{

 CKCON &= ~T0M;         // set T0M = 0, thus using SYSCLK/12

 TMOD &= 0xF0;          // clear Timer 0 control bits only

 TMOD |= 0x03;          // mode 3 (two split timers), GATE = C/T = 0

 TL0 = 112;             // 256 − 144 = 112 ==> 78.125us for normal Timer 00

 TH0 = 40;              // 256 − 216 = 40 ==> 117.1875us for extra Timer 000

 TR0 = 1;                // Timer 00 ON

 TR1 = 1;                // Timer 000 ON

 ET0 = 1;                // Enable TF0 interrupt, from Timer 00 overflows

 ET1 = 1;                // Enable TF1 interrupt, from Timer 000 overflows

}


/* ---------------------------------------- */


/* Set up timer 1 in mode 2, 8-bit, auto re-load, GATE = C/T = 0 */

/* for 115200 baud rate generator */

/* Assuming 22.1184 MHz clock */


/* Since Timer 0 is in mode 3, then Timer 1 will be switched on and off

   by setting it to mode 2 (on) or mode 3 (off) in the application program */


/* Setup also the UART0 */


void SetUp_Timer1_M3_and_UART0 (void)

{

 CKCON &= ~T1M;                // set T1M = 0, thus using SYSCLK/12

 TMOD &= 0x0F;                 // clear timer 1 control bits only (momentarily set T1 to mode 0)
```

174

```
 TMOD |= 0x30;                 // set initially to mode 3, i.e. timer off
 TH1 = TL1 = 0xFF;             // set for 115200 or 57600 baud rate (reload value in TH1)
 PCON |= 0X80;                 // SMOD0 = 1 so as to double the baud rate to 115200 bps
 SCON0 = 0X52;                 // 8-bit UART, variable baud rate, receiver disabled,
                               // transmitter ready TI0 = 1
}


/* --------------------------------------- */


/* ----------------------------------------------- */


// Timer00 Interrupt Service Routine
void TF0_ISR (void) interrupt 1 using 1
{
static data unsigned int TF0_OVF;     // counts TF0 overflows, from Timer00
   TF0_OVF++;
   TL0 = 112;                 // reload value
   if (TF0_OVF == 12800)      // number of interrupts required for a 1 second delay
             {
                    TF0_OVF = 0;
                    T00_FLAG = 1;
             }
}


/* ----------------------------------------------- */


// Timer000 Interrupt Service Routine
void TF1_ISR (void) interrupt 3 using 2
{
static data unsigned int TF1_OVF;     // counts TF1 overflows, from Timer000
 TF1_OVF++;
 TH0 = 40;                               // reload value
 if (TF1_OVF == 25600)        // number of interrupts required for a 3 second delay
         {
                    TF1_OVF = 0;
                    T000_FLAG = 1;
         }
}
```

```
/* --------------------------------------------- */


/* --------------------------------------------- */


/* Main program */
void main(void)
{


DISABLE_Watchdog();
SYSCLK_Init ();
PORT_Init ();


SetUp_Timer0_M3 ();                 // Timer 0 mode 3 – split timer
SetUp_Timer1_M3_and_UART0();        // Timer 1 (off) mode 3,
                                    // 8-bit auto reload value as a baud rate generator
                                    // initially set in mode 3, not running.


EA = 1;
```

```
while(1)
 {
// Timer 1 is switched on and off just to show that we can still control it.
// It is switched on only for use as the baud rate generator before the 'printf' command


    if (T00_FLAG == 1)
        {
            T00_FLAG = 0;
            TMOD = 0x23; // set Timer 1 to mode 2, start it as the baud rate generator
                         // ready for the 'printf' command which follows
                         // leaving Timer 0 set to mode 3
                         // This method is used instead of:
                         //
                         // TMOD &= 0x0F;     // clear timer 1 control bits only
                         // (momentarily set T1 to mode 0)
                         // TMOD |= 0x20;     // set to mode 2, i.e. Timer 1 on
                         // TH1 = 0xFF;       // set reload value
                         //
                         // which would have placed Timer 1 momentarily in mode 0
                         // and thus possibly modifying the reload value held in TH1
                         // (and hence the baud rate) before setting it to mode 2
                         // Hence the need to set the reload value in TH1 every time.
                         // Thus TMOD = 0x23 is much quicker and neater this time!

        printf ("Timer 00: 12800 timeouts every 1 second\n");
        TMOD = 0x33;       // set Timer 1 to mode 3 to stop the baud rate generator
                           // leaving Timer 0 set to mode 3
   }


 if (T000_FLAG == 1)
     {
        T000_FLAG = 0;
        TMOD = 0x23;      // set Timer 1 to mode 2, start it as the baud rate generator
                          // ready for the 'printf' command which follows
                          // leaving Timer 0 set to mode 3
        printf ("Timer 000: 25600 timeouts every 3 seconds\n");
        TMOD = 0x33;      // set Timer 1 to mode 3 to stop the baud rate generator
                          // leaving timer 0 set to mode 3
   }
 }
}
```

```
// ================================================================


// ================================================================


/* MySystem.h */


#ifndef __MYSYSTEM_H__
#define __MYSYSTEM_H__


#define SYSCLK          22118400        // SYSCLK frequency in Hz
#define BAUDRATE        115200UL        // Baud rate of UART in bps


void DISABLE_Watchdog (void);
void PORT_Init (void);
void SYSCLK_Init (void);
void PORT_Init (void);


#endif // __MYSYSTEM_H__


// ================================================================


// MySystem.c


#include "C8051F020.h"


//-------------------------------------------------------------------
// SYSCLK_Init
//-------------------------------------------------------------------
//
// This routine initializes the system clock to use an 22.1184MHz crystal
// as its clock source.
//

void SYSCLK_Init (void)
{
    unsigned int i;                 // delay counter
    OSCXCN = 0x67;                  // start external oscillator with
                                    // 22.1184MHz crystal
```

```
    for (i=0; i < 300; i++); // wait for oscillator to start
#ifndef SIMULATOR
    while (!(OSCXCN & 0x80));// Wait for crystal osc. to settle
                            /* disable above line if using simulator */
#endif
    OSCICN = 0x88;          // select external oscillator as SYSCLK
                                // source and enable missing clock
                                // detector
}


//----------------------------------------------------------------
// DISABLE_Watchdog
//----------------------------------------------------------------
//
// Disables the watchdog timer
//
void DISABLE_Watchdog (void)
{
    EA = 0;
    WDTCN = 0xDE;
```

```
    WDTCN = 0xAD;

    EA = 1;

}



//-----------------------------------------------------------------

// PORT_Init

//-----------------------------------------------------------------

//

// Configure the Crossbar and GPIO ports, (see page 163 of manual)

//

void PORT_Init (void)

{

    XBR0 = 0x04;                    // Enable UART0, UART0EN=1

                                        // TX0=P0.0 and RX0=P0.1

    XBR2 = 0x40;                    // Enable crossbar and weak pull-ups

    P0MDOUT |= 0x01;                    // enable TX0 (P0.0) as a push-pull output

    P1MDOUT |= 0x40;                    // enable P1.6 (LED) as push-pull output

}



// =============================================================
```

B.2   UART0 and UART1

This package initialises UART0 or UART1 at the required baud rate and uses the specified timer to generate this baud rate. The function to setup the baud rate is normally called from the main program, and a routine to do this, such as 'UART_Selector( )' is included as a remarked routine in the DualUarts.c program listed below. Its usage in an application program can be seen in appendix B.3 Clock example. The UART, timer and baud rate are all defined in this 'UART_selector( )' routine. 'sio_bit' should be a global bit variable in the application program.

Note that the same 'putchar' and '_getkey' routines are used for both UARTs. Hence if the application requires the use of both UARTs, the 'sio_bit' should be set before using any 'printf' statement. It should then be reset whenever you need to print to the other UART. Moreover, another re-named copy of the 'UART_Selector( )' routine would have to be made so that each one of them would have separate UART, timer and baud rate definitions, so that both UARTs can be initialised concurrently. The XBR0, XBR2 and P1MDOUT would also have to be modified to reflect the use of both UARTs as shown below:

```
  XBR0 = 0x04;          // UART0 enabled

  XBR2 = 0x44;          // Weak pull-up, Crossbar and UART1 enabled

  P1MDOUT = 0x05;       // P0.0 (TX0) and P0.2 (TX1) configured as Push-Pull
```

The UARTs initialisation program and header are listed here for reference.

```
/*-----------------------------------------------------------------
    DualUarts.c
-------------------------------------------------------------------*/
#include "C8051F020.H"
#include <stdio.h> /* prototype declarations for I/O functions */
#include "DualUarts.h"


extern bit sio_port;


/* 'sio_port' is declared and set in the main program */
/* according to which UART one intends to use (0 = UART0, 1 = UART1) */
/* It is used here in the 'putchar' and '_getkey' routines */
/* If both UARTS are being used, then you would need to set the sio_port */
/* bit to the correct UART before issuing the 'printf' command */
/* and obviously, both UARTS must be initialised */


/*


// Copy and paste this routine in the main application program
// in order to program the UARTs.
// Moreover, you would need to declare
//
// bit sio_port;
//
// as a global variable in the main application program
//
//-----------------------------------------------------------------
// UART_Selector function
//-----------------------------------------------------------------
//
// Remember to configure XBR0, XBR2 and P1MDOUT according to which UART you use
//


void UART_Selector (void)
{
// UART0 can use Timer 1 or Timer 2 as the baud rate generator
// UART1 can use Timer 1 or Timer 4 as the baud rate generator
// set the following #define statements as required:
```

```
#define UART_IN_USE 0

#define TIMER_FOR_UART 1

#define BAUD_RATE 115200UL


    #if (UART_IN_USE == 0)

        sio_port = 0;

          // SIO port to use (0 = UART0, 1 = UART1), used in DualUarts.c

        SetUpUART(UART_IN_USE, BAUD_RATE, TIMER_FOR_UART);


        #if (TIMER_FOR_UART == 1)

            #message "Set up UART0, at BAUD_RATE bps using Timer 1"

          #elif (TIMER_FOR_UART == 2)

            #message "Set up UART0, at BAUD_RATE bps using Timer 2"

          #else

            #error "Wrong Timer for UART0"

          #endif

    #endif
```

```
        #if (UART_IN_USE == 1)

            sio_port = 1;

            // SIO port to use (0 = UART0, 1 = UART1), used in DualUarts.c

            SetUpUART(UART_IN_USE, BAUD_RATE, TIMER_FOR_UART);


            #if (TIMER_FOR_UART == 1)

               #message "Set up UART1, at BAUD_RATE bps using Timer 1"

            #elif (TIMER_FOR_UART == 4)

               #message "Set up UART1, at BAUD_RATE bps using Timer 4"

            #else

               #error "Wrong Timer for UART1"

            #endif

        #endif


}
*/


/*------------------------------------------------------------------*/


/*-----------------------------------------------------------------
The following putchar function replaces the one in the library.
-------------------------------------------------------------------*/
char putchar (char c)
{
char d;


if (sio_port == 0)
 {                      /* UART0 */
 while (!TI0);
 TI0 = 0;
 SBUF0 = c;
 }
else               /* UART1 */
 {
 while (!(SCON1 & TI1)); /* While TI1 = 0 */
 SCON1 &= ~TI1; /* TI1 = 0 */
 SBUF1 = c;
 }


for(d=0; d<10; d++){;} /* just a delay if needed, depending on receiving device requirement */
```

183

```
return (c);

}


/*------------------------------------------------------------------

The following _getkey function replaces the one in the library.

------------------------------------------------------------------*/

char _getkey (void)

{

char c;


if (sio_port == 0)

  {

 while (!RI0);

 c = SBUF0;

 RI0 = 0;

  }

else

  {

 while (!(SCON1 & RI1)); /* While RI1 = 0 */

 c = SBUF1;

 SCON1 &= ~RI1; /* RI1 = 0 */

  }


return (c);

}


/*--------------------------------------------


 Set up any UART in mode 1, 8-bit, variable baud rate

 UART 0 can use Timers 1 or 2 as the baud rate generator

 UART 1 can use Timers 1 or 4 as the baud rate generator


---------------------------------------------*/


void SetUpUART(unsigned char UART, unsigned long BaudRate, unsigned char Timer)

{

#message "Remember to set XBR0, XBR2 and P1MDOUT correctly"

if (UART == 1) /* Set up UART 1 */

  {

      switch (Timer)

        {
```

184

```
//-----------------------------------------------------------------
// Setup RS232 UART1 in the Silicon Labs chip using Timer 1 or Timer 4
// For UART0 to use timer 1, set it in mode 1, 8-BIT AUTO-RELOAD.
// to generate baud rate. SMOD1 = 0 (divisor 32) and T0M = 1 (use SYSCLK)
// so baud rate formula is
// Baud rate = 22.1184 MHz /(32 * (256 − TH1))
// TH1 = 256 − [22118400/(32 * BR)]
//     = 256 − (691200/BR)
// For 9600 baud TH0        = 184 = 0xB8
// For 115.2K baud TH0      = 250 = 0xFA
//
// Similarly for Timer 4
// Serial interrupt is NOT enabled
//-----------------------------------------------------------------

        case 1:
                CKCON |= T1M;        // T1M = 1, use SYSCLK for timer 1
                PCON &= ~SMOD1;      // SMOD1 = 0, baud rate divide by 2 disabled for UART 1
                SCON1 = 0x50;        // 8-bit UART variable baud rate, mode 1, REN1 enabled
                TMOD &= 0x0F;        // Clear Timer 1 control bits
```

```
                    TMOD |= T1M1;        // Set to mode 2

                    TH1 = -(SYSCLK/32UL/BaudRate);

                    TR1 = 1;             // start timer 1

                    SCON1 |= TI1;        // Indicate TX1 ready for UART 1

                    break;


           case 4:

                    CKCON |= T4M;        // T4M = 1, use SYSCLK for timer 4

                    SCON1 = 0x50;        // SCON1: Mode 1, 8-bit UART, enable receiver

                    T4CON = TCLK1 + RCLK1; // T4CON: Use T4 for Baud Rate Tx and Rx on UART1

                    RCAP4 = -(SYSCLK/32UL/BaudRate); // set Timer reload value for baud rate

                    T4 = RCAP4;          // initialise Timer value

                    T4CON |= TR4;        // TR4: T4 Run

                    SCON1 |= TI1;        // TI1: Set TI1 to send first char of UART1

                    break;


       }

     }
else /* Set up UART 0 */


  {

      switch (Timer)

         {
//-----------------------------------------------------------------
// Setup RS232 UART0 in the Silicon Labs chip using Timer 1 or Timer 2

// For UART0 to use timer 1, then set it in mode 1, 8-BIT AUTO-RELOAD.

// to generate baud rate. SMOD1 = 0 (divisor 32) and T0M = 1 (use SYSCLK)

// so baud rate formula is

// Baud rate = 22.1184 MHz /(32 * (256 − TH1))

// TH1 = 256 − [22118400/(32 * BR)]

//         = 256 − (691200/BR)

// For 9600 baud TH1 = 184 = 0xB8

// For 115.2K baud TH1 = 250 = 0xFA

//

// Similarly if using Timer 2

// Serial interrupt is NOT enabled
//-----------------------------------------------------------------
         case 1:

                    CKCON |= T1M;   // T1M = 1, use SYSCLK for timer 1

                    PCON &= ~SMOD0; // SMOD0 = 0, baud rate divide by 2 disabled for UART 0
```

186

```
                SCON0 = 0x50; // 8-bit UART variable baud rate, REN0 enabled

                TMOD &= 0x0F; // Clear Timer 1 control bits

                TMOD |= T1M1; // Set to mode 2

                TH1  = -(SYSCLK/32UL/BaudRate);

                TR1  = 1;    // start timer 1

                TI0 = 1;     // Indicate TX0 ready, for UART 0

                break;


        case 2:

                SCON0 = 0x50; // SCON: Mode 1, 8-bit UART, enable receiver

                T2CON = 0x34;        // T2CON: Use T2 for Baud Rate on UART0

                RCAP2 = -(SYSCLK/32UL/BaudRate); // set Timer reload value for baud rate

                T2 = RCAP2;                 // initialise Timer value

                TR2 = 1;                    // TR2: T2 Run

                TI0 = 1;                    // TI0: Set TI0 to send first char of UART0

                break;
        }


    }
}
```

Header file to be include in the main program when using DualUARTS.c file

```
/*----------------------------------------------------------------
     DualUarts.h
----------------------------------------------------------------*/
#ifndef _DUAL_UARTS_H_
#define _DUAL_UARTS_H_


#define SYSCLK (22118400UL)


char putchar (char c);
char _getkey (void);


void SetUpUART(unsigned char UART, unsigned long BaudRate, unsigned char Timer);


#endif // _DUAL_UARTS_H_


/*----------------------------------------------------------------/
```

Or else we may write separate routines for UART0 and UART1 such as:

```
/*----------------------------------------------------------------
The following putcharU0 function uses UART0
-----------------------------------------------------------------*/
char putcharU0 (char c)
{
 char d;

 while (!TI0);
 TI0 = 0;
 SBUF0 = c;
 for (d=0; d<2; d++){;} /* just a delay if needed, since no hand shaking */
 return (c);
}


/*----------------------------------------------------------------
The following putcharU1 function uses UART1
-------------------------------------------------------------------*/
```

```
char putcharU1 (char c)

{

 char d;


 while (!(SCON1 & TI1)); /* While TI1 = 0 */

 SCON1 &= ~TI1; /* TI1 = 0 */

 SBUF1 = c;

 for (d=0; d<2; d++){;} /* just a delay if needed, since no hand shaking */

 return (c);

}


// ***********************************************

// TEXT STRING, TERMINATED WITH A NULL, IS TRANSMITTED THROUGH UART0

void TX_STRING_U0(char *text)

{

 while(*text != '\0')

    putcharU0(*text++);

}


// ***********************************************

// TEXT STRING, TERMINATED WITH A NULL, IS TRANSMITTED THROUGH UART1

void TX_STRING_U1(char *text)

{

 while(*text != '\0')

    putcharU1(*text++);

}
/*---------------------------------------------------------------/
```

With these routines, both UARTs can be setup using the 'SetUpUART( )' routine for each UARTx. Then we can easily print a string of text to whichever UART we want by using the corresponding 'TX_STRING_Ux( )' routine, without the need to use the standard 'printf' (which uses 'putchar( )'). We can of course opt to use the 'printf' commands with one UART and 'TX_SRING_Ux( )' with the other UART; we are completely flexible to do so.

B.3   Clock

This example is a clock with a blinking LED. It uses 5 tasks, one task keeping track of the seconds, running periodically every second. It sends a signal to the minute task (which is set waiting for a signal) every sixty seconds. The minute task itself then signals the hour task every sixty minutes. Another task, the 'clock_reset' task waits for an External 0 (/INT0) interrupt. When this falling edge triggered interrupt happens, the task resets the clock to 23:58:50. The last task simply blinks the LED connected to pin 3.6 every 500ms.

Note the 'PORT_Init' task where the UART0 Tx and Rx signals and the /INT0 external input signals are routed to Port 0. The bits to setup can be verified by looking at Figure 1-10, Figure 1-11 and Table 1-4. The 'UART_Selector( )' routine is used to initialise the required UART as previously explained in appendix B.2 UART0 and UART1.

```
//-----------------------------------------------------------------
// Clock.c
//-----------------------------------------------------------------
// Copyright (C) 2015
//
// AUTH: PD
// DATE: 21 FEB 15
//
// This program flashes the green LED on the C8051F020 target board, 500ms on, 500ms off
// Example program to demonstrate the use of various PaulOS_F020 commands
// Target: C8051F02x
//
//


//-----------------------------------------------------------------
// Includes
//-----------------------------------------------------------------
#include "C8051F020.h"          /* special function registers 8051F020 */
#include "DualUarts.h"          /* UARTS functions header file */
#include "PaulOS_F020.h"        /* PaulOS_F020 version system calls definitions */
#include <stdio.h>
#include <stdlib.h>


//-----------------------------------------------------------------
// Enumerates
//-----------------------------------------------------------------
```

```
enum eTasks {CLOCK_SEC, CLOCK_MIN, CLOCK_HOUR, CLOCK_RESET, BLINK};


//------------------------------------------------------------------
// Global CONSTANTS
//------------------------------------------------------------------


bit sio_port; /* SIO port to use (0 = UART0, 1 = UART1) */


sbit LED = P1^6;                // green LED: '1' = ON; '0' = OFF
sbit INT0 = P0^2;               // EXT0 (/INT0) input pin, routed to this port pin
                                // in the port initialisation routine


struct time { /* structure of the time record */
 unsigned char hour;                /* hour                         */
 unsigned char min;                 /* minute                       */
 unsigned char sec;                 /* second                       */
};


struct time ctime = { 12, 58, 30 }; /* storage for clock time values */
```

```c
//-------------------------------------------------------------------
// Function PROTOTYPES
//-------------------------------------------------------------------
void SYSCLK_Init (void);

void PORT_Init (void);

void DISABLE_Watchdog (void);

void UART_Selector (void);


//-------------------------------------------------------------------
// SYSCLK_Init
//-------------------------------------------------------------------
//
// This routine initializes the system clock to use an 22.1184MHz crystal
// as its clock source.
//
void SYSCLK_Init (void)
{
    unsigned int i;                 // delay counter


    OSCXCN = 0x67;                  // start external oscillator with
                                    // 22.1184MHz crystal


    for (i=0; i < 256; i++) ;       // wait for oscillator to start
#ifndef SIMULATOR                   // SIMULATOR defined in the C51 Target Tab
    while (!(OSCXCN & 0x80)) ;      // Wait for crystal osc.to settle
#endif
    OSCICN = 0x88;                  // select external oscillator as SYSCLK
                                    // source and enable missing clock
                                    // detector
}


//-------------------------------------------------------------------
// DISABLE_Watchdog
//-------------------------------------------------------------------
//
// Disables the watchdog timer
//
void DISABLE_Watchdog (void)
{
```

```
    EA = 0;

    WDTCN = 0xDE;

    WDTCN = 0xAD;

    EA = 1;

}


//-----------------------------------------------------------------

// PORT_Init

//-----------------------------------------------------------------

//

// Configure the Crossbar and GPIO ports

//

void PORT_Init (void)

{

    XBR0 = 0x04;                    // Enable UART 0, UART0EN = 1

                                    // TX0 => P0.0 and RX0 => P0.1

    XBR1 = 0x04;                    // Route INT0 to port pins, INT0E = 1

                                    // INT0 => P0.2

    XBR2 = 0x40;                    // Enable crossbar and weak pull-ups

    P0MDOUT |= 0x01;                // enable TX0 as a push-pull output

    P1MDOUT |= 0x40;                    // enable P1.6 (LED) as push-pull output

}


//-----------------------------------------------------------------

// UART_Selector

//-----------------------------------------------------------------

//

// Remember to configure XBR0, XBR2 and P1MDOUT according to which UART you use

//


void UART_Selector (void)

{

// UART0 can use Timer 1 or Timer 2 as the baud rate generator

// UART1 can use Timer 1 or Timer 4 as the baud rate generator

#define UART_IN_USE 0

#define TIMER_FOR_UART 1

#define BAUD_RATE 115200UL
```

```
#if (UART_IN_USE == 0)

    sio_port = 0;

    /* SIO port to use (0 = UART0, 1 = UART1), used in DualUarts.c */

    SetUpUART(UART_IN_USE, BAUD_RATE, TIMER_FOR_UART);


    #if (TIMER_FOR_UART == 1)

        #message "Set up UART0, at BAUD_RATE bps using Timer 1"

    #elif (TIMER_FOR_UART == 2)

        #message "Set up UART0, at BAUD_RATE bps using Timer 2"

    #else

        #error "Wrong Timer for UART0"

    #endif

#endif


#if (UART_IN_USE == 1)

    sio_port = 1;

    /* SIO port to use (0 = UART0, 1 = UART1), used in DualUarts.c */

    SetUpUART(UART_IN_USE, BAUD_RATE, TIMER_FOR_UART);
```

Download free eBooks at bookboon.com

```
              #if (TIMER_FOR_UART == 1)

                    #message "Set up UART1, at BAUD_RATE bps using Timer 1"

              #elif (TIMER_FOR_UART == 4)

                    #message "Set up UART1, at BAUD_RATE bps using Timer 4"

              #else

                    #error "Wrong Timer for UART1"

              #endif

        #endif

}


/****************************************************************/
/*        Task 0 'clock_sec' */
/****************************************************************/
void clock_sec (void)

{

 OS_PERIODIC_A(0,1,0);          /* Repeat every 1 second */

 while (1)                      /* clock is an endless loop */

 {

    if (++ctime.sec == 60)

          {   /* calculate the second */

        ctime.sec = 0;

           OS_SIGNAL_TASK(CLOCK_MIN);

          }


    else printf ("Clock Time: %02bu:%02bu:%02bu\r", /* display time */

                    ctime.hour, ctime.min, ctime.sec);

    OS_WAITP(); /* wait for 1 second */

 }

}


/****************************************************************/
/* Task 2 'clock_min' */
/****************************************************************/
void clock_min (void)

{

 while (1) /* clock is an endless loop */

 {

    OS_WAITS(0); /* wait for 1 second */
```

195

```
    if (++ctime.min == 60)

          {  /* calculate the second */

       ctime.min = 0;

           OS_SIGNAL_TASK(CLOCK_HOUR);

          }


    else printf ("Clock Time: %02bu:%02bu:%02bu\r", /* display time */

                    ctime.hour, ctime.min, ctime.sec);

 }

}


/****************************************************************/
/*      Task 2 'clock_hour' */
/****************************************************************/
void clock_hour (void)
{
 while (1)            /* clock is an endless loop */
 {
    OS_WAITS(0); /* wait for 1 second */


    if (++ctime.hour == 24)
          { /* calculate the second */
          ctime.hour = 0;
           }


    printf ("Clock Time: %02bu:%02bu:%02bu\r", /* display time */

                    ctime.hour, ctime.min, ctime.sec);

 }

}


/****************************************************************/
/*     Task 3 'rest clock' */
/****************************************************************/
```

```
void clock_reset (void)

{

while(1)

    {

        OS_WAITI(0);                           // wait for /INT0

                        ctime.hour = 23;

                        ctime.min = 59;

                        ctime.sec = 50;

    }

}


/****************************************************************/
/*      Task 4 'Blink' */
/****************************************************************/
```

```c
void BlinkTask (void)
{
        OS_PERIODIC_A(0,0,500);                  /* Repeat every 500 ms */
while(1)
    {
        LED = !LED;
        OS_WAITP(); // wait for the periodic interval
    }
}
/*****************************************************************/


/*
********************************************************************
*/
//-------------------------------------------------------------------
// MAIN Routine
//-------------------------------------------------------------------
void main (void) {

    DISABLE_Watchdog ();
    SYSCLK_Init ();
    UART_Selector (); /* Set up UART */


    PORT_Init ();


    OS_INIT_RTOS(TICK_TIMER); /* initialise RTOS (Timer 0 interrupt), */
                                /* variables and stack */


        /* CREATE the 5 tasks */
    OS_CREATE_TASK(CLOCK_SEC,clock_sec);
    OS_CREATE_TASK(CLOCK_MIN,clock_min);
    OS_CREATE_TASK(CLOCK_HOUR,clock_hour);
    OS_CREATE_TASK(CLOCK_RESET,clock_reset);
    OS_CREATE_TASK(BLINK,BlinkTask);


        IT0 = 1; // fallling edge triggered
        EX0 = 1; // enable external 0 (/INT0) interrupt
```

```
OS_RTOS_GO(0); /* Start the RTOS */


while (1)

   {
            #ifndef SIMULATOR

            OS_CPU_IDLE();        /* Go to idle mode if doing nothing, to conserve energy

            */

      #else

                ;

                #endif

   }


}
```

Bibliography

Blaut, J. (2004). *8051 RTOS*. B.Sc. Electrical Engineering Thesis, University of Malta.

Chew, M.T., & Gupta, G.S. (2005). *Embedded Programming with Field-Programmable Mixed-Signal Microcontrollers*. Silicon Laboratories.

Debono, P.P. (2013a). *PaulOS: Part I – An 8051 Real-Time Operating System* (1st ed.). bookboon.com.

Debono, P.P. (2013b). *PaulOS: Part II – An 8051 Real-Time Operating System* (1st ed.). bookboon.com.

Huang, H. (2009). *Embedded System Design with the C8051*. Stanford, CT, USA: Cengage Learning.

Pont, M.J. (2002). *Patterns for Time-Triggered Embedded Systems: Building reliable applications with the 8051 family of microcontrollers*. Boston, Ma, USA: Addison-Wesley Longman Publishing Co., Inc.

Schultz, T.W. (1999). *C and the 8051 (volume II): building efficient applications.* Upper Saddle River, NJ, USA: Prentice Hall PTR.

Schultz, T.W. (2004). *C and the 8051*. Pagefree Publishing.

Silicon Labs. (2003a). AN122 – Annotated "C" Examples for the "F02x" Family. Austin, TX, USA: Silicon Laboratories Inc.

Silicon Labs. (2003b). C8051F020 Data Sheet. Austin, TX, USA: Silicon Laboratories Inc.

Index

Endnotes

1   This resource assignment flexibility is achieved through the use of a Priority Crossbar Decoder. Note that the state of a Port I/O pin can always be read from its associated Data register regardless of whether that pin has been assigned to a digital peripheral or behaves as GPIO.

2   When the interrupts are recognised by the micro-controller at the same time (i.e. simultaneous), a decision has to be made on which interrupt is to be serviced first. If all these interrupts have the same high/low priority setting, the controller will follow the fixed priority order column shown in Table 1-5 to determine which one should run first. If the interrupts are not simultaneous, then the priority order column does not come into play at all. An interrupt occurring while another interrupt of the same high/low priority setting is running, will not be allowed to interrupt this running ISR.

3   The main structure for this RTOS came from the book "C and the 8051 – Building Efficient Applications – Volume II" by Thomas W. Schultz and published by Prentice Hall (0-13-521121-2). In this book, Prof. Schultz discusses the development of two real-time kernels. The first one is the RTKS which I corrected and developed into PaulOS co-operative RTOS. The second one is the RTKB which I also corrected, modified and developed into MagnOS pre-emptive RTOS. Both operating systems, RTKS and RTKB as written in the book are not fully functional, contain some errors and lack some essential components. I did correspond with Prof. Schultz and sent him my modifications and final versions of the programs which he later acknowledged in the 3rd edition of the book "C and the 8051", again published by Prentice-Hall (0-58961-237-X). So I am particularly grateful to Prof. Schultz for being the catalyst of my increased interest in RTOSs.

4   16-bit SFR declarations: Some 8051 derivatives have 16-bit SFRs that are created using consecutive addresses in SFR memory to specify 16-bit values. For example, the C8051F020 uses addresses 0xCC and 0xCD for the low and high bytes of timer/counter 2 respectively. The Cx51 Compiler provides the sfr16 data type to access two 8-bit SFRs as a single 16-bit SFR (see also section 1.7).

Access to 16-bit SFRs using sfr16 is possible only when the low byte address location immediately precedes the high byte (little endian) and when the low byte is written last. The low byte is used as the address in the sfr16 declaration. For example:

sfr16 ADC0 = 0xBE;   /* ADC0L 0BEh, ADC0H 0BFh */

sfr16 T2 = 0xCC; /* TL2 0CCh, TH2 0CDh */

sfr16 RCAP2 = 0xCA;   /* RCAP2L 0CAh, RCAP2H 0CBh */

sfr16 RCAP4 = 0xE4;   /* RCAP4L 0E4h, RCAP4H 0E5h */

In this example, ADC0, T2, RCAP2 and RCAP4 are declared as 16-bit SFRs and can b used as for example:

T2 = 0x1234; // equivalent to TH2 = 0x12 and TL2 = 0x34

Whilst hoping that you found this book useful, please feel free to contact me if you have any queries or suggestions.

If there is a great demand for porting the RTOS to another family of micro-controllers, I would be willing to attempt to do so.

Paul Debono
e-mail: pawlu.debono@yahoo.co.uk