Unit-1

20/6/18

Embedded System:- It is a combination of Software and Hardware to perform specific task.

Hardware Contains [illegible], micro Controllers, micro processor, Sensors, input/output and Memory.

It is used for Specific task

Eg:- Mobile phones, Microwave Oven, Washing Machine.

→ Aerospace applications like flight Control, ~~Auto~~ pilots, Passenger entertainment System.

→ Medical equipment like Aneticia, ECG Monitoring Syst. MRI System

→ In defence System, Radar Systems, fighter aircraft Controll System, Radio System and Missile guidance System.

Desktop Processor

1) Runs different programs at different times depending upon the needs of user.

2) Has large amount of RAM memory and disk space.

3) Both RAM, Disk Space can be cheaply expanded if required

4) All pc's have essentially identical hardware architecture & run identical software

5) Bootup time is measured in minutes

Embedded Processor

1) Runs a single, dedicated application at all times.

2) It has sufficient memory

3) Adding memory is difficult

4) These are highly variable with different CPUs, peripherals, OS etc.

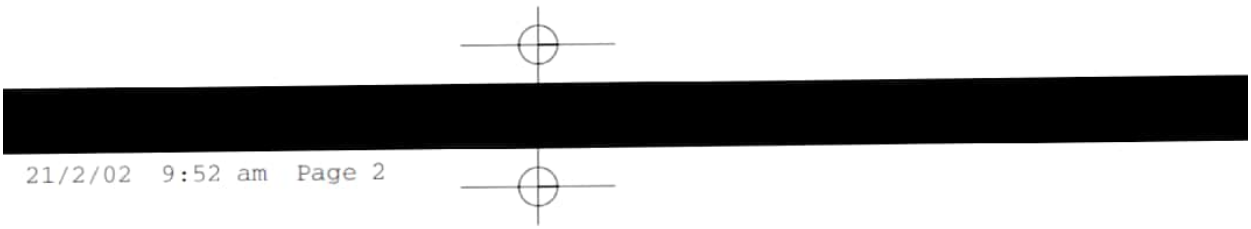
5) Bootup time is measured in seconds

BPAM

1.2 What is an embedded system?

When we talk about 'embedded systems', what do we mean? Opinions vary. Throughout this book, we will use the following loose definition:

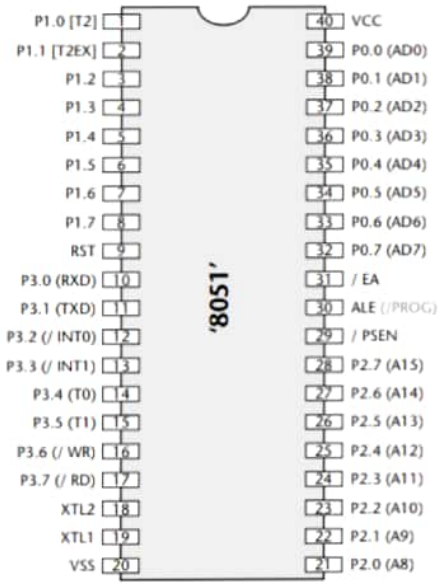
> An embedded system is an application that contains at least one programmable computer (typically in the form of a microcontroller, a microprocessor or digital signal processor chip) and which is used by individuals who are, in the main, unaware that the system is computer-based.



Typical examples of embedded applications that are constructed using the techniques discussed in this book include:

- **Mobile phone systems** (including both customer handsets and base stations).
- **Automotive applications** (including braking systems, traction control, airbag release systems, engine-management units, steer-by-wire systems and cruise-control applications).
- **Domestic appliances** (including dishwashers, televisions, washing machines, microwave ovens, video recorders, security systems, garage door controllers).
- **Aerospace applications** (including flight control systems, engine controllers, autopilots and passenger in-flight entertainment systems).
- **Medical equipment** (including anaesthesia monitoring systems, ECG monitors, drug delivery systems and MRI scanners).
- **Defence systems** (including radar systems, fighter aircraft flight control systems, radio systems and missile guidance systems).

Please note that our definition of embedded systems *excludes* applications such as 'personal digital assistants' (PDAs) running versions of Windows or similar operating systems: from a developer's perspective, these are best viewed as a cut-down version of a desktop computer system. This type of application makes up a very small percentage of the overall 'embedded' market and is not considered in this book.



Pin(s)	Function
1–8	Port 1. The bi-directional pins on this port may be used for input and output: each pin may be individually controlled and – for example – some may be used for input while others on the same port are used for output. Use of these pins is discussed in detail in Chapter 3 and Chapter 4. In 8052-based designs, Pin 1 and Pin 2 have alternative functions associated with Timer 2 (see Section 2.8).
9	The 'Reset' pin. When this pin is held at Logic 0, the chip will run normally. If, while the oscillator is running, this pin is held at Logic 1 for two (or more) machine cycles, the microcontroller will be reset. An example of simple reset hardware is given in Section 2.4.
10–17	Port 3. Another bi-directional input port (same operation as Port 1). Each pin on this port also serves an additional function. Pin 10 and Pin 11 are used to receive and transmit (respectively) serial data using the 'RS-232' protocol. See Chapter 9 for details. Pin 12 and Pin 13 are used to process interrupt inputs. We say more about interrupts in Section 2.9. Pin 14 and Pin 15 have alternative functions associated with Timer 0 and Timer 1 (see Section 2.8). Pin 16 and Pin 17 are used when working with external memory (see Section 2.6).
18–19	These pins are used to connect an external crystal, ceramic resonator or oscillator module to the microcontroller. See Section 2.5 for further details.
20	Vss. This is the 'ground' pin.
21–28	Port 2. Another bi-directional input port (same operation as Port 1). These pins are also used when working with external memory (see Section 2.6).
29	Program Store Enable (PSEN) is used to control access to external CODE memory (if used). See Section 2.6.
30	Address Latch Enable (ALE) is used when working with external memory (see Section 2.6). Note that some devices allow ALE activity to be disabled (if external memory is not used): this can help reduce the level of electromagnetic interference (EMI) generated by your product. This pin is also used (on some devices) as the program pulse input (PROG) during Flash programming.
31	External Access (EA). To execute code from internal memory (e.g. on-chip Flash, where available) this pin must be connected to Vcc. To execute code from external memory, this pin must be connected to ground. Forgetting to connect this pin to Vcc is a common error when people first begin working with the 8051.
32–39	Port 0. Another bi-directional input port (same operation as Port 1). Note that – unlike Port 1, Port 2 and Port 3 – this port does NOT have internal pull-up resistors. See Chapter 4 for further details. These pins are also used when working with external memory (see Section 2.6).
40	Vcc. This is the '5V' pin (on 5V devices; 3V on 3V devices, etc).

FIGURE 2.2 The external interface to the standard '8051' microcontroller

2.4 Reset requirements

The process of starting any microcontroller is a non-trivial one. The underlying hardware is complex and a small, manufacturer-defined, 'reset routine' must be run to place this hardware into an appropriate state before it can begin executing the user program. Running this reset routine takes time, and requires that the microcontroller's oscillator is operating.

Where your system is supplied by a robust power supply, which rapidly reaches its specified output voltage when switched on, rapidly decreases to 0V when switched off, and – while switched on – cannot 'brown out' (drop in voltage), then you can safely use low-cost reset hardware based on a capacitor and a resistor to ensure that your system will be reset correctly: this form of reset circuit is shown in Figure 2.3a.

Where your power supply is less than perfect, and / or your application is safety related, the simple RC solution will not be suitable. Several manufacturers provide more sophisticated reset chips which may be used in these circumstances: Figure 2.3b illustrates one possibility.

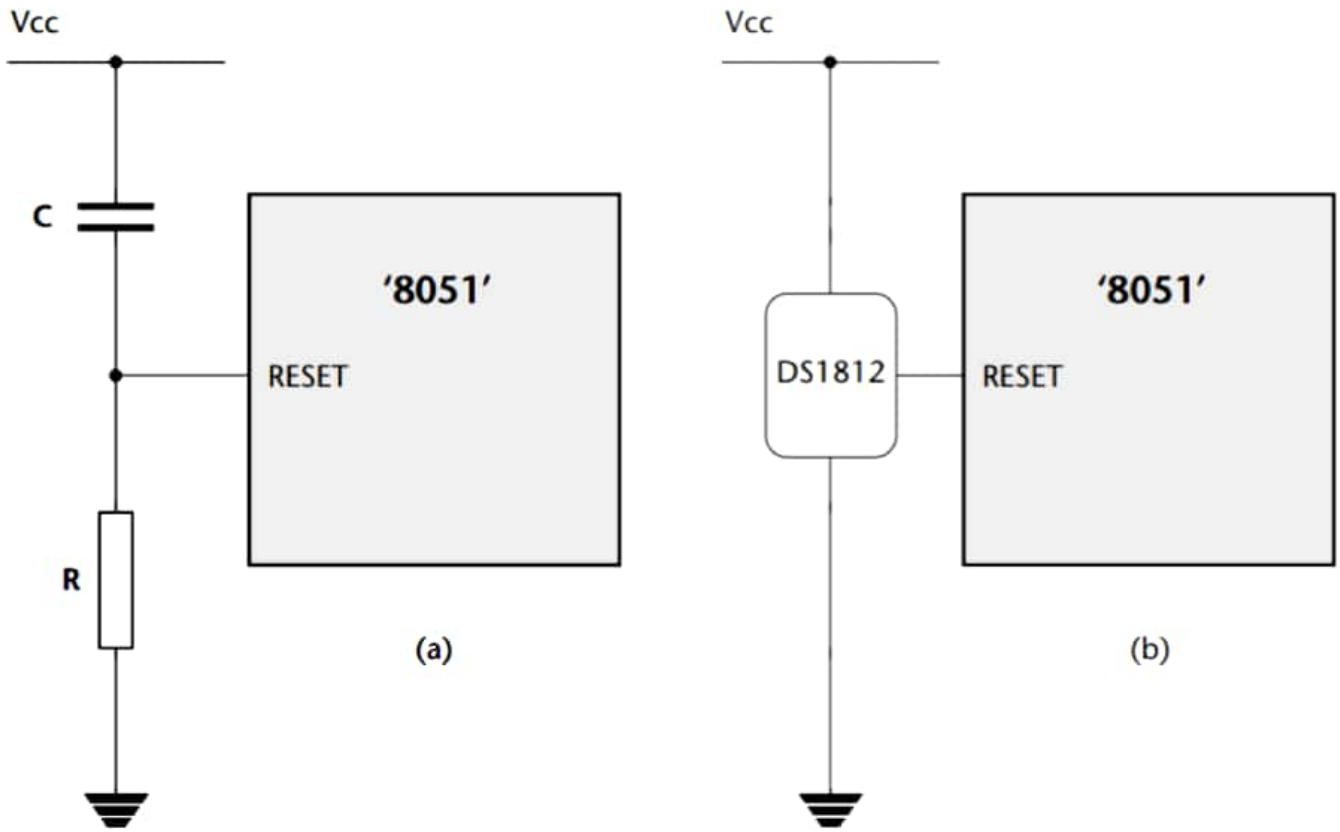


FIGURE 2.3a and b Two possible reset circuits for 8051-based designs. We will not consider hardware issues in detail in this book: please refer to Chapter 11 for sources of further information about this topic

2.5 Clock frequency and performance

All digital computer systems are driven by some form of oscillator circuit: the 8051 is certainly no exception (see Figure 2.4).

The oscillator circuit is the 'heartbeat' of the system and is crucial to correct operation. For example, if the oscillator fails, the system will not function at all; if the oscillator runs irregularly, any timing calculations performed by the system will be inaccurate.

We consider some important issues linked to oscillator frequency and performance in this section.

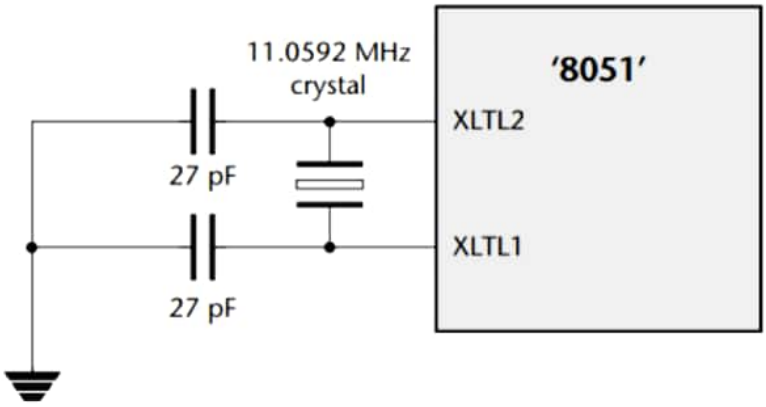


FIGURE 2.4 An example of a simple crystal oscillator circuit. We will not consider hardware issues in detail in this book: please refer to Chapter 11 for sources of further information about this topic

a) The link between oscillator frequency and machine-cycle period

One of the first questions to be asked when considering a microcontroller for a project is whether it has the required level of performance.

As a general rule, the speed at which your application runs is directly determined by the oscillator frequency: in most cases, if you double the oscillator frequency, the application will run twice as fast. When we want to compare different processors, we need a way of specifying performance in a quantitative manner. One popular measure is the number of machine instructions that may be executed in one second, usually expressed in 'MIPS' (Million Instructions Per Second). For example, in the original Intel 8051 microcontroller, a minimum of 12 oscillator cycles was required to execute a machine instruction. The original 8051 had a maximum oscillator frequency of 12 MHz and therefore a peak performance of 1 MIP.

A simple way of improving the 8051 performance is to increase the clock frequency. More modern (Standard) 8051 devices allow the use of clock speeds well beyond the 12 MHz limit of the original devices. For example, the Atmel AT89C55WD, allow clock speeds up to 33 MHz: this raises the peak performance to around 3 MIPS.

Another way of improving the performance is to make internal changes to the microcontroller so that fewer oscillator cycles are required to execute each machine instruction. The Dallas 'High Speed Microcontroller' devices (87C520, and similar) use this approach, so that only four oscillator cycles are required to execute a machine instruction. These Dallas devices also allow faster clock rates (typically up to 33 MHz). Combined, these changes give a total performance of around 8 MIPS. Similar changes are made in members of the Winbond family of Standard 8051 devices (see the Winbond W77E58, for example) resulting in performance figures of up to 10 MIPS.

Clearly, for maximum performance, we would like to execute instructions at a rate of one machine instruction per oscillator cycle. For example, the Dallas 'Ultra High Speed' 89C420 operates at this rate: as a result, it runs at 12 times the speed of the original 8051. In addition, the 89c420 can operate at up to 50 MHz, increasing overall performance to around 40–50 MIPS.

To put all these figures in perspective, a modern desktop PC has a potential performance of around 1000 MIPS. However, a good percentage of this performance (perhaps 50% or more) will be 'consumed' by the operating system. By contrast, the embedded operating system we will describe in Chapter 7 consumes less than 1% of the processor resources of the most basic 8051: this leaves sufficient CPU cycles to run a complex embedded application.

b) Why you should choose a low oscillator frequency

In our experience, many developers select an oscillator frequency that is at or near the maximum value supported by a particular device. For example, the Infineon C505/505C will operate with crystal frequency of 2–20 MHz, and many people automatically choose values at or near the top of this range, in order to gain maximum performance.

This can be a mistake, for the following reasons:

- Many applications do not require the levels of performance that a modern 8051 device can provide.
- In most modern (CMOS-based) 8051s, there is an almost linear relationship between the oscillator frequency and the power supply current. As a result, by using the lowest frequency necessary it is possible to reduce the power requirement: this can be useful, particularly in battery-powered applications.

Dynamic RAM (DRAM)

Dynamic RAM is a read-write memory technology that uses a small capacitor to store information. As the capacitor will discharge quite rapidly, it must be frequently refreshed to maintain the required information: circuitry on the chip takes care of this refresh activity. Like most current forms of RAM, the information is lost when power is removed from the chip.

Static RAM (SRAM)

Static RAM is a read-write memory technology that uses a form of electronic flip-flop to store the information. No refreshing is required, but the circuitry is more complex and costs can be several times that of the corresponding size of DRAM. However, access times may be one-third those of DRAM.

Mask Read-Only Memory (ROM)

Mask ROM is – from the software developer's perspective – read only: however, the manufacturer is able to write to the memory, at the time the chip is created, according to a 'mask' provided by the company for which the chips are being produced. Such devices are therefore sometimes referred to as 'factory-programmed ROM'. Mask programming is not cheap, and is not a low-volume option: mistakes can be very expensive, and providing code for your first mask can be a character-building process. Access times are often slower than RAM: roughly 1.5 times that of DRAM.

Many members of the 8051 family are available with on-chip, mask-programmed, ROM.

Programmable Read-Only Memory (PROM)

PROM is a form of Write-Once, Read-Many (WORM) or 'One-Time Programmable' (OTP) memory. Basically, we use a PROM programmer to blow tiny 'fuses' in the device. Once blown, these fuses cannot be repaired; however, the devices themselves are cheap.

Many modern members of the 8051 family are available with OTP ROM.

UV Erasable Programmable Read-Only Memory (UV EPROM)

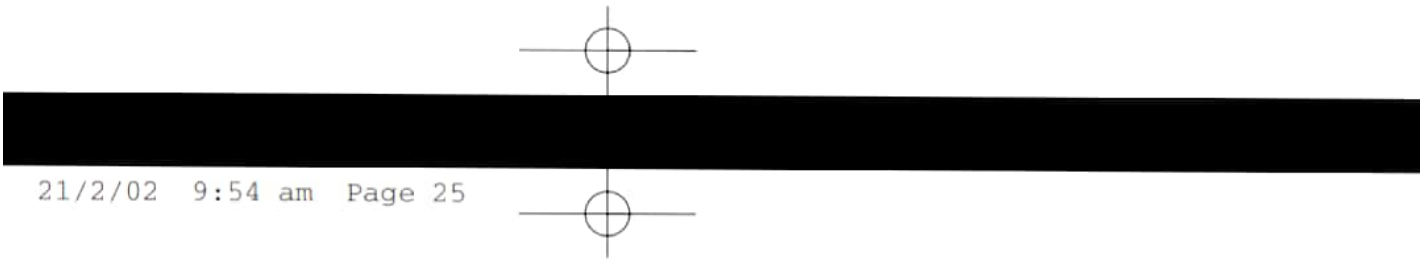
Like PROMs, UV EPROMs are programmed electrically. Unlike PROMs, they also have a quartz window which allows the memory to be erased by exposing the internals of the device to UV light. The erasure process can take several minutes and, after erasure, the quartz window will be covered with a UV-opaque label. This form of EPROM can withstand thousands of program / erase cycles.

device. Once blown, these fuses cannot be repaired; however, the devices themselves are cheap.

Many modern members of the 8051 family are available with OTP ROM.

UV Erasable Programmable Read-Only Memory (UV EPROM)

Like PROMs, UV EPROMs are programmed electrically. Unlike PROMs, they also have a quartz window which allows the memory to be erased by exposing the internals of the device to UV light. The erasure process can take several minutes and, after erasure, the quartz window will be covered with a UV-opaque label. This form of EPROM can withstand thousands of program / erase cycles.



More flexible than PROMs and once very common, UV EPROMs now seem rather primitive compared with EEPROMs (see below). They can be useful for prototyping but are prohibitively expensive for use in production.

Many older members of the 8051 family are available with on-board UV EPROM.

EEPROM and Flash ROM

Electrically-Erasable Programmable Read-Only Memory (EEPROMs) and 'Flash' ROMs are a more user-friendly form of ROM that can be both programmed and erased electrically.

EEPROM and Flash ROM are very similar. EEPROMs can usually be reprogrammed on a byte-by-byte basis, and are often used to store passwords or other 'persistent' user data. Flash ROMs generally require a block-sized 'erase' operation before they can be programmed: often the size of the block will be several kilobytes: such ROMs are often used for the storage of program code.

Many members of the 8051 family are available with on-board EEPROM or flash ROM, and some devices contain both types of memory.

The 8051 memory architecture

Having considered some of the basic memory types available and some general features of computer memory organization, we are now in a position to consider the memory architecture of the 8051.

There are two distinct memory regions in an 8051 device: the DATA area and the CODE area. We will consider each region in turn here.

DATA memory

DATA memory is used to store variables and the program stack while the program is running. The DATA area will be implemented using some form of RAM.

Most of the DATA area has a byte-oriented memory organization. However, within the DATA area is a 16-byte BDATA area which can also be accessed using bit addresses. This area can be used to store bit-sized variables (see Figure 2.6). The 8051 has machine instructions which allow bit variables to be manipulated very efficiently. These instructions can be easily utilized from C, as we will demonstrate in Chapter 3.

Byte address	Bit address							
0x2F	0x7F	0x7E	0x7D	0x7C	0x7B	0x7A	0x79	0x78
0x2E	0x77	0x76	0x75	0x74	0x73	0x72	0x71	0x70
0x2D	0x6F	0x6E	0x6D	0x6C	0x6B	0x6A	0x69	0x68
0x2C	0x67	0x66	0x65	0x64	0x63	0x62	0x61	0x60
0x2B	0x5F	0x5E	0x5D	0x5C	0x5B	0x5A	0x59	0x58
0x2A	0x57	0x56	0x55	0x54	0x53	0x52	0x51	0x50
0x29	0x4F	0x4E	0x4D	0x4C	0x4B	0x4A	0x49	0x48
0x28	0x47	0x46	0x45	0x44	0x43	0x42	0x41	0x40
0x27	0x3F	0x3E	0x3D	0x3C	0x3B	0x3A	0x39	0x38
0x26	0x37	0x36	0x35	0x34	0x33	0x32	0x31	0x30
0x25	0x2F	0x2E	0x2D	0x2C	0x2B	0x2A	0x29	0x28
0x24	0x27	0x26	0x25	0x24	0x23	0x22	0x21	0x20
0x23	0x1F	0x1E	0x1D	0x1C	0x1B	0x1A	0x19	0x18
0x22	0x17	0x16	0x15	0x14	0x13	0x12	0x11	0x10
0x21	0x0F	0x0E	0x0D	0x0C	0x0B	0x0A	0x09	0x08
0x20	0x07	0x06	0x05	0x04	0x03	0x02	0x01	0x00

FIGURE 2.6 On-chip RAM memory in the 8051: The BDATA area. See text for details

Byte address	Bit address							
0x2F	0x7F	0x7E	0x7D	0x7C	0x7B	0x7A	0x79	0x78
0x2E	0x77	0x76	0x75	0x74	0x73	0x72	0x71	0x70
0x2D	0x6F	0x6E	0x6D	0x6C	0x6B	0x6A	0x69	0x68
0x2C	0x67	0x66	0x65	0x64	0x63	0x62	0x61	0x60
0x2B	0x5F	0x5E	0x5D	0x5C	0x5B	0x5A	0x59	0x58
0x2A	0x57	0x56	0x55	0x54	0x53	0x52	0x51	0x50
0x29	0x4F	0x4E	0x4D	0x4C	0x4B	0x4A	0x49	0x48
0x28	0x47	0x46	0x45	0x44	0x43	0x42	0x41	0x40
0x27	0x3F	0x3E	0x3D	0x3C	0x3B	0x3A	0x39	0x38
0x26	0x37	0x36	0x35	0x34	0x33	0x32	0x31	0x30
0x25	0x2F	0x2E	0x2D	0x2C	0x2B	0x2A	0x29	0x28
0x24	0x27	0x26	0x25	0x24	0x23	0x22	0x21	0x20
0x23	0x1F	0x1E	0x1D	0x1C	0x1B	0x1A	0x19	0x18
0x22	0x17	0x16	0x15	0x14	0x13	0x12	0x11	0x10
0x21	0x0F	0x0E	0x0D	0x0C	0x0B	0x0A	0x09	0x08
0x20	0x07	0x06	0x05	0x04	0x03	0x02	0x01	0x00

FIGURE 2.6 On-chip RAM memory in the 8051: The BDATA area. See text for details

Note that the various locations can be accessed either via their byte addresses (0x20 to 0x2F) or via their bit addresses (0x00 to 0x7F).

Note also that there is an area where the bit addresses and the byte addresses are the same. For example, within byte address 0x24, there is a bit location with address 0x27: there is also a byte with address 0x27 in the BDATA area. It may appear that this will cause problems for the compiler. However, no conflicts will arise because the compiler can always determine from the context (that is, the type of data being manipulated) whether the bit address or byte address should be used.

CODE memory

Not surprisingly, the CODE area is used to store the program code, usually in some form of ROM ('read-only memory').

Please note:

- The CODE area may also contain read-only variables ('constants'), such as filter co-efficients or data for speech playback.
- On a desktop PC, code is copied from disk to RAM when you run the program. It is then executed from RAM. In most embedded systems, like the 8051, code is 'executed in place', from ROM.

INTERRUPTS:

In order to use any of the interrupts in the MCS-51, the following three steps must be taken.

1. Set the EA (enable all) bit in the IE register to 1.
2. Set the corresponding individual interrupt enable bit in the IE register to 1.
3. Begin the interrupt service routine at the corresponding Vector Address of that interrupt. See Table below.

Interrupt Source	Vector Address
IE0	0003H
TF0	000BH
IE1	0013H
TF1	001BH
RI & TI	0023H
TF2 & EXF2	002BH

In addition, for external interrupts, pins $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$ (P3.2 and P3.3) must be set to 1, and depending on whether the interrupt is to be level or transition activated, bits IT0 or IT1 in the TCON register may need to be set to 1.

ITx = 0 level activated

ITx = 1 transition activated

IE: INTERRUPT ENABLE REGISTER. BIT ADDRESSABLE.

If the bit is 0, the corresponding interrupt is disabled. If the bit is 1, the corresponding interrupt is enabled.

EA	—	ET2	ES	ET1	EX1	ET0	EX0

EA	IE.7	Disables all interrupts. If EA = 0, no interrupt will be acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
—	IE.6	Not implemented, reserved for future use.*
ET2	IE.5	Enable or disable the Timer 2 overflow or capture interrupt (8052 only).
ES	IE.4	Enable or disable the serial port interrupt.
ET1	IE.3	Enable or disable the Timer 1 overflow interrupt.
EX1	IE.2	Enable or disable External Interrupt 1.
ET0	IE.1	Enable or disable the Timer 0 overflow interrupt.
EX0	IE.0	Enable or disable External Interrupt 0.

*User software should not write 1s to reserved bits. These bits may be used in future MCS-51 products to invoke new features. In that case, the reset or inactive value of the new bit will be 0, and its active value will be 1.

IP: INTERRUPT PRIORITY REGISTER. BIT ADDRESSABLE.

If the bit is 0, the corresponding interrupt has a lower priority and if the bit is 1 the corresponding interrupt has a higher priority.

—	—	PT2	PS	PT1	PX1	PT0	PX0

— IP. 7 Not implemented, reserved for future use.*

— IP. 6 Not implemented, reserved for future use.*

PT2 IP. 5 Defines the Timer 2 interrupt priority level (8052 only).

PS IP. 4 Defines the Serial Port interrupt priority level.

PT1 IP. 3 Defines the Timer 1 interrupt priority level.

PX1 IP. 2 Defines External Interrupt 1 priority level.

PT0 IP. 1 Defines the Timer 0 interrupt priority level.

PX0 IP. 0 Defines the External Interrupt 0 priority level.

*User software should not write 1s to reserved bits. These bits may be used in future MCS-51 products to invoke new features. In that case, the reset or inactive value of the new bit will be 0, and its active value will be 1.

SCON: SERIAL PORT CONTROL REGISTER. BIT ADDRESSABLE.

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0 SCON. 7 Serial Port mode specifier. (NOTE 1).

SM1 SCON. 6 Serial Port mode specifier. (NOTE 1).

SM2 SCON. 5 Enables the multiprocessor communication feature in modes 2 & 3. In mode 2 or 3, if SM2 is set to 1 then RI will not be activated if the received 9th data bit (RB8) is 0. In mode 1, if SM2 = 1 then RI will not be activated if a valid stop bit was not received. In mode 0, SM2 should be 0. (See Table 9).

REN SCON. 4 Set/Cleared by software to Enable/Disable reception.

TB8 SCON. 3 The 9th bit that will be transmitted in modes 2 & 3. Set/Cleared by software.

RB8 SCON. 2 In modes 2 & 3, is the 9th data bit that was received. In mode 1, if SM2 = 0, RB8 is the stop bit that was received. In mode 0, RB8 is not used.

TI SCON. 1 Transmit interrupt flag. Set by hardware at the end of the 8th bit time in mode 0, or at the beginning of the stop bit in the other modes. Must be cleared by software.

RI SCON. 0 Receive interrupt flag. Set by hardware at the end of the 8th bit time in mode 0, or halfway through the stop bit time in the other modes (except see SM2). Must be cleared by software.

NOTE 1:

SM0	SM1	Mode	Description	Baud Rate
0	0	0	SHIFT REGISTER	Fosc./12
0	1	1	8-Bit UART	Variable
1	0	2	9-Bit UART	Fosc./64 OR Fosc./32
1	1	3	9-Bit UART	Variable

PSW: PROGRAM STATUS WORD. BIT ADDRESSABLE.

CY	AC	F0	RS1	RS0	OV	—	P

CY	PSW.7	Carry Flag.
AC	PSW.6	Auxiliary Carry Flag.
F0	PSW.5	Flag 0 available to the user for general purpose.
RS1	PSW.4	Register Bank selector bit 1 (SEE NOTE 1).
RS0	PSW.3	Register Bank selector bit 0 (SEE NOTE 1).
OV	PSW.2	Overflow Flag.
—	PSW.1	User definable flag.
P	PSW.0	Parity flag. Set/cleared by hardware each instruction cycle to indicate an odd/even number of '1' bits in the accumulator.

NOTE:
1. The value presented by RS0 and RS1 selects the corresponding register bank.

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

PCON: POWER CONTROL REGISTER. NOT BIT ADDRESSABLE.

SMOD	—	—	—	GF1	GF0	PD	IDL

SMOD	Double baud rate bit. If Timer 1 is used to generate baud rate and SMOD = 1, the baud rate is doubled when the Serial Port is used in modes 1, 2, or 3.
—	Not implemented, reserved for future use.*
—	Not implemented, reserved for future use.*
—	Not implemented, reserved for future use.*
GF1	General purpose flag bit.
GF0	General purpose flag bit.
PD	Power Down bit. Setting this bit activates Power Down operation in the 80C51BH. (Available only in CHMOS).
IDL	Idle Mode bit. Setting this bit activates Idle Mode operation in the 80C51BH. (Available only in CHMOS).

If 1s are written to PD and IDL at the same time, PD takes precedence.

*User software should not write 1s to reserved bits. These bits may be used in future MCS-51 products to invoke new features. In that case, the reset or inactive value of the new bit will be 0, and its active value will be 1.

TCON: TIMER/COUNTER CONTROL REGISTER. BIT ADDRESSABLE.

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1 TCON. 7 Timer 1 overflow flag. Set by hardware when the Timer/Counter 1 overflows. Cleared by hardware as processor vectors to the interrupt service routine.

TR1 TCON. 6 Timer 1 run control bit. Set/cleared by software to turn Timer/Counter 1 ON/OFF.

TF0 TCON. 5 Timer 0 overflow flag. Set by hardware when the Timer/Counter 0 overflows. Cleared by hardware as processor vectors to the service routine.

TR0 TCON. 4 Timer 0 run control bit. Set/cleared by software to turn Timer/Counter 0 ON/OFF.

IE1 TCON. 3 External Interrupt 1 edge flag. Set by hardware when External Interrupt edge is detected. Cleared by hardware when interrupt is processed.

IT1 TCON. 2 Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low level triggered External Interrupt.

IE0 TCON. 1 External Interrupt 0 edge flag. Set by hardware when External Interrupt edge detected. Cleared by hardware when interrupt is processed.

IT0 TCON. 0 Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low level triggered External Interrupt.

TMOD: TIMER/COUNTER MODE CONTROL REGISTER. NOT BIT ADDRESSABLE.

GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
TIMER 1				TIMER 2			

GATE When TRx (in TCON) is set and GATE = 1, TIMER/COUNTERx will run only while INTx pin is high (hardware control). When GATE = 0, TIMER/COUNTERx will run only while TRx = 1 (software control).

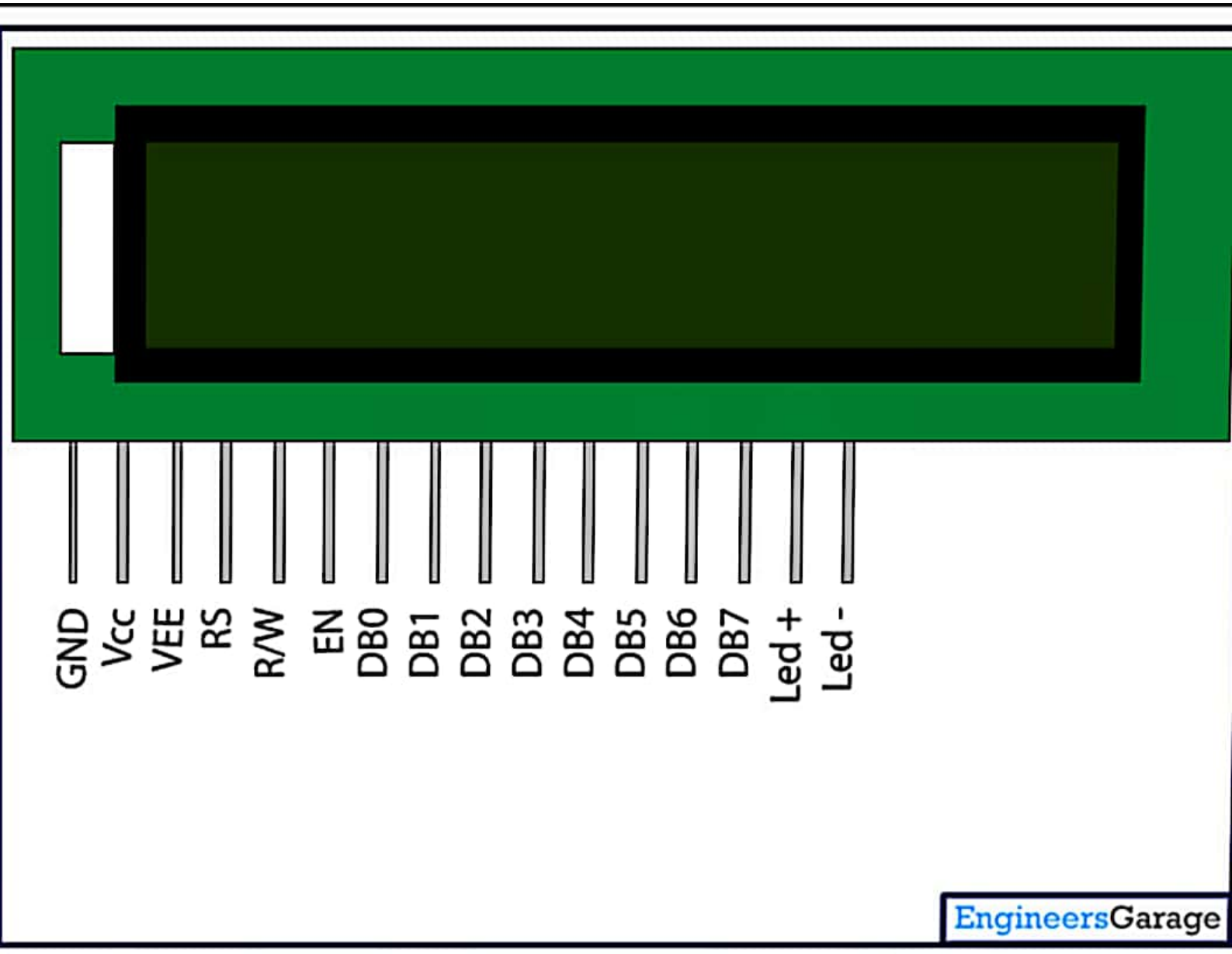
C/$\overline{T}$ Timer or Counter selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from Tx input pin).

M1 Mode selector bit. (NOTE 1)

M0 Mode selector bit. (NOTE 1)

NOTE 1:

M1	M0	Operating Mode	
0	0	0	13-bit Timer (MCS-48 compatible)
0	1	1	16-bit Timer/Counter
1	0	2	8-bit Auto-Reload Timer/Counter
1	1	3	(Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits, TH0 is an 8-bit Timer and is controlled by Timer 1 control bits.
1	1	3	(Timer 1) Timer/Counter 1 stopped.



Pin	Symbol	I/O	Description
1	GND	-	Ground
2	Vcc	-	+5V power supply
3	VEE	-	Contrast control
4	RS	I	command/data register selection
5	R/W	I	write/read selection
6	E	I/O	Enable
7	DB0	I/O	The 8-bit data bus
8	DB1	I/O	The 8-bit data bus
9	DB2	I/O	The 8-bit data bus
10	DB3	I/O	The 8-bit data bus
11	DB4	I/O	The 8-bit data bus
12	DB5	I/O	The 8-bit data bus
13	DB6	I/O	The 8-bit data bus
14	DB7	I/O	The 8-bit data bus

Figure 10-9. SCON Serial Port Control Register (Bit Addressable)

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0	SCON.7	Serial port mode specifier
SM1	SCON.6	Serial port mode specifier
SM2	SCON.5	Used for multiprocessor communication. (Make it 0)
REN	SCON.4	Set/cleared by software to enable/disable reception.
TB8	SCON.3	Not widely used.
RB8	SCON.2	Not widely used.
TI	SCON.1	Transmit interrupt flag. Set by hardware at the beginning of the stop bit in mode 1. Must be cleared by software.
RI	SCON.0	Receive interrupt flag. Set by hardware halfway through the stop bit time in mode 1. Must be cleared by software.

Note: Make SM2, TB8, and RB8 = 0.

2 We will describe how to create and use a 'Project ... lates key aspects of the hardware environment, such as the type of processor to ... used, the oscillator frequency and the number of oscillator cycles required to ex... cute each instruction. This helps to document the system, and makes it easier to ... port the code to a different processor.

3 We will describe how to create and use a 'Port Header' file. This brings together all details of the port access from the whole system. Like the Project Header, this helps during porting and also serves as a means of documenting important system features.

We will use all three of these techniques in the code examples presented in subsequent chapters.

We begin by discussing how to use object-oriented styles of programming with the C language.

5.2 Object-oriented programming with C

One way in which the different programming languages may be classified is as a series of generations (see Table 5.1).

TABLE 5.1 The classification of programming languages into different generations. Please note that some people consider O-O languages to be 5GLs: however, this distinction will not have an impact on our discussions here

Language generation	Example languages
–	Machine Code
First-Generation Language (1GL)	Assembly Language.
Second-Generation Languages (2GLs)	COBOL, FORTRAN
Third-Generation Languages (3GLs)	C, Pascal, Ada 83
Fourth-Generation Languages (4GLs)	C++, Java, Ada 95

It is often argued that object-oriented (O-O) design – and O-O programming languages – have advantages when compared with those from earlier generations. For example, as Graham notes:[15]

[The phrase] 'object-oriented' has become almost synonymous with modernity, goodness and worth in information technology circles.

15. Graham, I. (1994) *Object-Oriented Methods*, (2nd edn) Addison-Wesley, Harlow, England, p. 1.

A frequent argument is that the O-O approach is more effective than those previously used because it represents 'a more natural way' of thinking about problems. As Jalote notes:[16]

> *One main claimed advantage of using object orientation is that an OO model closely represents the problem domain, which makes it easier to produce and understand designs.*

You might reasonably ask why this book uses C, rather than a language from a later generation (such as C++ or Java). The reason is that O-O languages are not readily available for small embedded systems, primarily because of the overheads inherent in the O-O approach.

It is easy to see the source of these overheads. Suppose, for example, we have a C program with a variable Xyz that we wish to set to some value and then display. We might do so using the following code:

```
...
int Xyz;
Xyz = 3;

...
printf("%d", Xyz);
```

Now, consider the following O-O version, using C++:

```
class cClass
 {
 public:
    int  Get_Xyz(void) const;
    void Set_Xyz(const int);
 private:
    int _Xyz; // Encapsulated data
 };

...
cClass abc;

abc.Set_Xyz(3);

...

cout'<< abc.Get_Xyz();
```

16. Jalote, P. (1997) *An Integrated Approach to Software Engineering*, (2nd edn) Springer-Verlag, New York.

The C++ version has both strengths and weaknesses:

☺ **In the C++ code, the data (_Xyz) are encapsulated in the class: access to these data is controlled because it is possible only via the two member functions. By contrast, the data (Xyz) in the C version are 'global' variables and can be altered anywhere in the program. From a design perspective, the C++ code is more elegant. It may also prove easier to maintain.**

⊗ There is a CPU time overhead associated with the C++ code: this is, at least, the cost of two (member) function calls. In real applications, such overheads can be substantial: even Stroustrup, creator of the C++ programming language, has acknowledged that a C++ implementation is likely to run 25% more slowly than an equivalent application in FORTRAN.[17] This can have implications for embedded projects where speed of processing is a primary concern. For example, Sommerville cites the case of an aircraft system in which an O-O solution was abandoned due to the impact of these overheads.[18]

One solution to such performance problems is to 'in line' the member functions: this can greatly reduce the CPU overhead, but a penalty will then be paid in terms of memory usage.[19]

Neither the CPU performance load nor the alternative memory load present a significant problem on multi-megabyte desktop PCs, but – on the type of embedded projects considered in this book (where teams may struggle with code to save one or two bytes of memory) a 'pure' O-O approach is rarely practical.

Does this mean that O-O design principles need (or should) be avoided by C programmers? Fortunately it does not. For many years before O-O techniques entered the mainstream, C programmers used a 'modular' style of programming which is well supported by the language. Using this approach, it is possible to create 'file-based-classes' in C without imposing a significant memory or CPU load:

```
// BEGIN: File XYZ.C

static int Xyz;

Xyz = 3;

. . .

printf("%d", Xyz);

// END: File XYZ.C
```

17. Stroustrup, B. (1994) *The Design of C++*, University Video Communications, Stanford CA, USA. Recorded 2 March, 1994.
18. Sommerville, I. (1996) *Software Engineering*, (5th edn) Addison-Wesley, London, p. 301.
19. Pont, M.J. (1996) *Software Engineering with C++ and CASE Tools*, (1st edn) Addison-Wesley, London, pp. 191–93.

The change here is minor: we have simply used the (ISO/ANSI) C keyword `static` to ensure that only functions within the file `XYZ.C` are able to access the data `Xyz`. As a consequence, the source file becomes our 'class' and the 'static' data in that file become private data members of that class. The functions defined within the file become the member functions in our class, and our whole program may be split into a number of clearly-defined (file-based) classes (Figure 5.1).

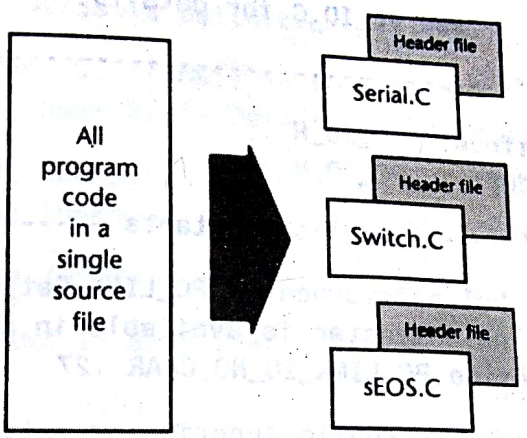


FIGURE 5.1 Turning a monolithic program into object-oriented C. See text for details

Note that we can also create 'private' member functions (which are accessible only by functions defined within a particular file) simply by including the prototypes for the function in the .C file (rather than the .H file) for the particular class.

This process is illustrated in Listing 5.1 and Listing 5.2. These files are part of a library of code designed to allow an 8051 microcontroller to use a serial interface. Please note that we will not be concerned with the operation of this code here (that will be considered in Chapter 9): at this time, we are simply concerned with the library structure.

As you look at this code, please note the presence of:

- Public 'member' functions, such as `PC_LINK_IO_Write_String_To_Buffer()`. Such functions have their prototypes in the H file.
- A private 'member' function, `PC_LINK_IO_Send_Char()`. Such `static` functions have their prototypes in the C file.
- A public constant (`PC_LINK_IO_NO_CHAR`), with a value which must be accessed by the rest of the program (see the H file).
- A limited number of public variables (e.g. `In_read_index_G`), defined in the C file (without the use of the `static` keyword).
- Numerous private constants and private variables (e.g. `RECV_BUFFER_LENGTH`), which are 'invisible' outside the C file.

5.4 The Port Header (Port.H)

In a typical embedded project, you may have a user interface created using an LCD, a keypad, and one or more single LEDs. There may be a serial (RS-485) link to another microcontroller board. There may be one or more high-power devices (say 3-phase industrial motors) to be controlled.

Each of these (software) components in your application will require exclusive access to one or more port pins. Following the structure discussed in Section 5.2, the project may include 10–20 different source files, created – perhaps – by five different people. How do you ensure that changes to port access in one component does not impact on another? How do you ensure that it is easy to adapt the application to an environment where different port pins must be used?

These issues are addressed through the use of a simple Port Header file (Figure 5.3). Using a Port Header, you pull together the different port access features for the whole project into a single (header) file. Use of this technique can ease project development, maintenance and porting.

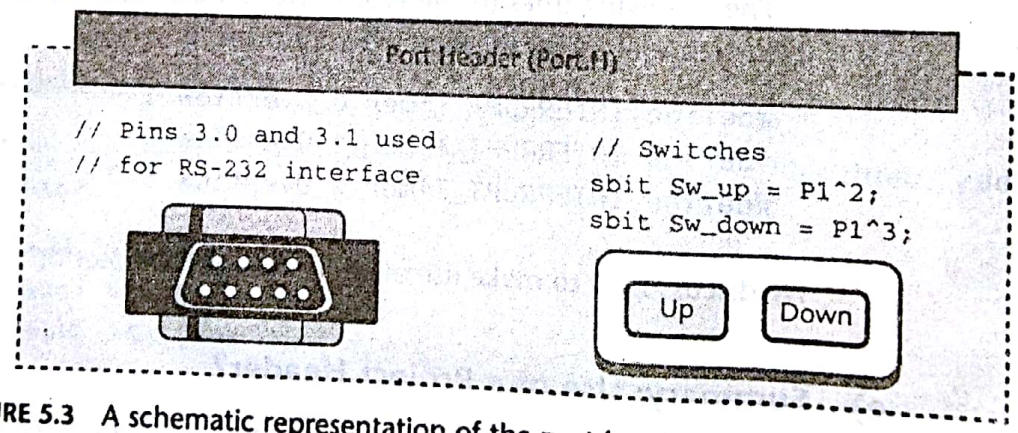


FIGURE 5.3 A schematic representation of the port header file

The Port Header file is simple to understand and easy to apply. Consider, for example, that we have three C files in a project (A, B, C), each of which require access to one or more port pins, or to a complete port.

File A may include the following:

```
// File A

sbit Pin_A = P3^2;
```

File B may include the following:

```
// File B

#define Port_B P0

...
```

File C may include the following:

```
// File C

sbit Pin_C = P2^7;

...
```

In this version of the code, all of the port access requirements are spread over multiple files. Instead of this, there are many advantages obtained by integrating all port access in a single `Port.H` header file:

```
// ----- Port.H -----

// Port access for File B
#define Port_B P0

// Port access for File A
sbit Pin_A = P3^2;

// Port access for File C
sbit Pin_C = P2^7;

...
```

Each of the remaining project files will then '`#include`' the file '`Port.H`'. Listing 5.5 shows a complete example of a `Port.H` file from a real application.

Listing 5.5 An example of a real Port Header file (Port.H) from a project using an interface consisting of a keypad and liquid crystal display

```
/*-------------------------------------------------------*-

  Port.H (v1.00)

  ------------------------------------------------------
```

```
  'Port Header' (see Chap 5) for project DATA_ACQ (see Chap
-*-------------------------------------------------------------

#ifndef _PORT_H
#define _PORT_H

#include 'Main.H'

// ------ Menu_A.C ------------------------------------------
// Uses whole of Port 1 and Port 2 for data acquisition
#define Data_Port1 P1
#define Data_Port2 P2

// ------ PC_IO.C -------------------------------------------
// Pins 3.0 and 3.1 used for RS-232 interface
#endif

/*-------------------------------------------------------------
  ---- END OF FILE ----------------------------------------
-*-------------------------------------------------------------
```

Despite its simplicity, use of a Port Header file can improve reliability and safety because it avoids potential conflicts between port pins, particularly during the maintenance phase of the project when developers (who may not have been involved in the original design) are required to make code changes.

A Port Header is itself portable: it can be used with any microcontroller, and is not linked to the 8051 family. Use of a Port Header also improves portability, by making accessible, in one location, all of the port access requirements of the application.

5.5 Example: Restructuring the Hello, Embedded World example

We present here the complete source code listing for the 'Hello, Embedded World' example introduced in Chapter 3. This time, the code is restructured to match the layout suggestions given in this chapter.

The complete code for this project is also included on the CD.

Difference between DESKTOP OS & RTOS

	Desktop OS	RTOS
1.	Large in Memory	Not Large Memory
2.	Big or Large User Interface Management	Limited No. of User Interface
3.	Network protocol are usually in-built	If required
4.	Program with definite exit loop.	Usually Infinite Loop
5.	It is always plug-n-play	For PnP in RTOS we have to make a lot changes

Summary of comparison

RTOS

- Unfair scheduling
  - Scheduling based on priority
- Kernel is preemptive either completely or up to maximum degree
- Priority inversion is a major issue
- Predictable behavior

GPOS

- Fair scheduling
  - Scheduling can be adjusted dynamically for optimized throughput
- Kernel is non-preemptive or have long non-preemptive code sections
- Priority inversion usually remain unnoticed
- No predictability guarantees