

- Introduction

This is a book about the Intel 8051 microcontroller and its large family of descendants. It is intended to give you, the reader, some new techniques for optimizing your 8051 projects and the development process you use for those projects. It is not the purpose of this book to provide various recipes for different types of embedded projects.

Wherever possible, I have included code examples to make the discussion clearer. There are points in the book where projects are discussed as a means of illustrating the point of the given chapter. Much of this code is available on the companion disk, to use it you will need to be familiar with C and 8051 assembler since this book is not intended to be a tutorial in C or 8051 assembler. There are many fine books you can buy to learn about ANSI C. As for 8051 assembler, the best source is the Intel data book which is free from your 8051 vendor or the manual that comes with your particular assembler.

The code on the companion diskette contains the code I wrote and compiled for the book you hold in your hands. It is fully functional and has been tested. This is not to say that the code on the diskette is ready to go into your system and be delivered as part of your projects. Some of it will require change before it can be integrated into your system.

This book will help you learn how to make the best out of the tools you have. If you only have an 8051 assembler, you can still learn from this book and use the examples, but you will have to decide for yourself how to implement the C language examples in assembler. This is not a difficult task for anyone who understands the basics of C and the 8051 assembler set.

If you have a C compiler for the 8051, then I congratulate you. You have made an excellent decision in your use of C. You will find that your project development time using C is lower and that your maintenance time using C is also lower. If you have the Keil C51 package, then you have made an excellent decision in 8051 development tools. I have found that the Keil package for the 8051 provides the best support. The code in this book directly supports the Keil C extensions. If you have one of the other development packages such as Archimedes or Avocet, you will find that this book is still of great service to you. The main thing to be aware of is that you may have to change some of the Keil specific directives to the appropriate ones for your development tools.

In many places in this book are diagrams of the hardware on which the example code runs. These are not intended to be full schematics, but are merely block diagrams that have enough information to allow you to understand how the software must interface to the hardware.

You should look upon this book as a learning tool rather than a source of various system designs. This is not an 8051 cookbook, but rather an exploration of the capabilities of the 8051 given proper hardware and software design. I prefer to think that you will use this book as a source of ideas from which your designs springboard and grow in a marvelous world of sunshine and roses! Seriously, though, I think you will gain useful knowledge from this book that will help you greatly improve your designs and make you look like your company's 8051 guru.

- The Hardware

Overview

The 8051 family of micro controllers is based on an architecture which is highly optimized for embedded control systems. It is used in a wide variety of applications from military equipment to automobiles to the keyboard on your PC. Second only to the Motorola 68HC11 in eight bit processors sales, the 8051 family of microcontrollers is available in a wide array of variations from manufacturers such as Intel, Philips, and Siemens. These manufacturers have added numerous features and peripherals to the 8051 such as I²C interfaces, analog to digital converters, watchdog timers, and pulse width modulated outputs. Variations of the 8051 with clock speeds up to 40MHz and voltage requirements down to 1.5 volts are available. This wide range of parts based on one core makes the 8051 family an excellent choice as the base architecture for a company's entire line of products since it can perform many functions and developers will only have to learn this one platform.

The basic architecture consists of the following features:

- an eight bit ALU
- 32 discrete I/O pins (4 groups of 8) which can be individually accessed
- two 16 bit timer/counters
- full duplex UART
- 6 interrupt sources with 2 priority levels
- 128 bytes of on board RAM
- separate 64K byte address spaces for DATA and CODE memory

One 8051 processor cycle consists of twelve oscillator periods. Each of the twelve oscillator periods is used for a special function by the 8051 core such as op code fetches and samples of the interrupt daisy chain for pending interrupts. The time required for any 8051 instruction can be computed by dividing the clock frequency by 12, inverting that result and multiplying it by the number of processor cycles required by the instruction in question. Therefore, if you have a system which is using an 11.059MHz clock, you can compute the number of instructions per second by dividing this value by 12. This gives an instruction frequency of 921583 instructions per second. Inverting this will provide the amount of time taken by each instruction cycle (1.085 microseconds).

Memory Organization

The 8051 architecture provides the user with three physically distinct memory spaces which can be seen in Figure A - 1. Each memory space consists of contiguous addresses from 0 to the maximum size, in bytes, of the memory space. Address overlaps are resolved by utilizing instructions which refer specifically to a given address space. The three memory spaces function as described below.

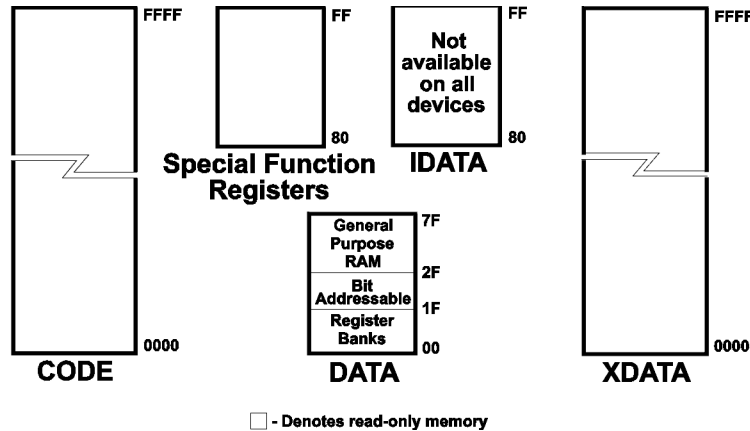


Figure A - 1 - 8051 Memory Architecture

The CODE Space

The first memory space is the CODE segment in which the executable program resides. This segment can be up to 64K (since it is addressed by 16 address lines). The processor treats this segment as read only and will generate signals appropriate to access a memory device such as an EPROM. However, this does not mean that the CODE segment must be implemented using an EPROM. Many embedded systems these days are using EEPROM which allows the memory to be overwritten either by the 8051 itself or by an external device. This makes upgrades to the product easy to do since new software can be downloaded into the EEPROM rather than having to disassemble it and install a new EPROM. Additionally, battery backed SRAMs can be used in place of an EPROM. This method offers the same capability to upload new software to the unit as does an EEPROM, and does not have any sort of read/write cycle limitations such as an EEPROM has. However, when the battery supplying the RAM eventually dies, so does the software in it. Using an SRAM in place of an EPROM in development systems allows for rapid downloading of new code into the target system. When this can be done, it helps avoid the cycle of programming/testing/erasing with EPROMs, and can also help avoid hassles over an in circuit emulator which is usually a rare commodity.

In addition to executable code, it is common practice with the 8051 to store fixed lookup tables in the CODE segment. To facilitate this, the 8051 provides instructions which allow rapid access to tables via the data pointer (DPTR) or the program counter with an offset into the table optionally provided by the accumulator. This means that oftentimes, a table's base address can be loaded in DPTR and the element of the table to access can be held in the accumulator. The addition is performed by the 8051 during the execution of the instruction which can save many cycles depending on the situation. An example of this is shown later in this chapter in

Listing A - 5.

The DATA Space

The second memory space is the 128 bytes of internal RAM on the 8051, or the first 128 bytes of internal RAM on the 8052. This segment is typically referred to as the DATA segment. The RAM locations in this segment are accessed in one or two cycles depending on the instruction. This access time is much quicker than access to the XDATA segment because memory is addressed directly rather than via a memory pointer such as DPTR which must first be initialized. Therefore, frequently used variables and temporary scratch variables are usually assigned to the DATA segment. Such allocation must be done with care, however, due to the limited amount of memory in this segment.

Variables stored in the DATA segment can also be accessed indirectly via R0 or R1. The register being used as the memory pointer must contain the address of the byte to be retrieved or altered. These instructions can take one or two processor cycles depending on the source/destination data byte.

The DATA segment contains two smaller segments of interest. The first subsegment consists of the four sets of register banks which compose the first 32 bytes of RAM. The 8051 can use any of these four groups of eight bytes as its default register bank. The selection of register banks is changeable at any time via the RS1 and the RS0 bits in the Processor Status Word (PSW). These two bits combine into a number from 0 to 3 (with RS1 being the most significant bit) which indicates the register bank to be used. Register bank switching allows not only for quick parameter passing, but also opens the door for simplifying task switching on the 8051.

The second sub-segment in the DATA space is a bit addressable segment in which each bit can be individually accessed. This segment is referred to as the BDATA segment. The bit addressable segment consists of 16 bytes (128 bits) above the four register banks in memory. The 8051 contains several single bit instructions which are often very useful in control applications and aid in replacing external combinatorial logic with software in the 8051 thus reducing parts count on the target system. It should be noted that these 16 bytes can also be accessed on a "byte-wide" basis just like any other byte in the DATA space.

Special Function Registers

Control registers for the interrupt system and the peripherals on the 8051 are contained in internal RAM at locations 80 hex and above. These registers are

referred to as special function registers (or SFRs for short). Many of them are bit addressable. The bits in the bit addressable SFRs can either be accessed by name, index or bit address. Thus, you can refer to the EA bit of the Interrupt Enable SFR as EA, IE.7, or 0AFH. The SFRs control things such as the function of the timer/counters, the UART, and the

+	0	1	2	3	4	5	6	7
F8								
F0	B							
E8								
E0	ACC							
D8								
D0	PSW							
C8	T2CON		RCAP2L	RCAP2H	TL2	TH2		
C0								
B8	IP							
B0	P3							
A8	IE							
A0	P2							
98	SCON	SBUF						
90	P1							
88	TCON	TMOD	TL0	TL1	TH0	TH1		
80	P0	SP	DPL	DPH				PCON

 - Denotes bit addressable Special Function Registers in this table and all following diagrams

Table A - 1

interrupt sources as well as their priorities. These registers are accessed by the same set of instructions as the bytes and bits in the DATA segment. A memory map of the SFRs indicating the registers which are bit addressable is shown in Table A - 1.

The IDATA Space

Certain 8051 family members such as the 8052 contain an additional 128 bytes of internal RAM which reside at RAM locations 80 hex and above. This segment of RAM is typically referred to as the IDATA segment. Because the IDATA addresses and the SFR addresses overlap, address conflicts between IDATA RAM and the SFRs are resolved by the type of memory access being performed, since the IDATA segment can only be accessed via indirect addressing modes.

The XDATA Space

The final 8051 memory space is 64K in length and is addressed by the same 16 address lines as the CODE segment. This space is typically referred to as the external data memory space (or the XDATA segment for short). This segment usually consists of some sort of RAM (usually an SRAM) and the I/O devices or external peripherals to which the 8051 must interface via its bus. Read or write operations to this segment take a minimum of two processor cycles and are performed using either DPTR, R0, or R1. In the case of DPTR, it usually takes two processor cycles or more to load the desired address in addition to the two cycles required to perform the read or write operation. Similarly, loading R0 or R1 will take minimum of one cycle in addition to the two cycles imposed by the memory access itself. Therefore, it is easy to see that a typical operation with the XDATA segment will, in general, take a minimum of three processor cycles. Because of this, the DATA segment is a very attractive place to store any frequently used variables.

It is possible to fill this segment entirely with 64K of RAM if the 8051 does not need to perform any I/O with devices in its bus or if the designer wishes to cycle the RAM on and off when I/O devices are being accessed via the bus. Methods for performing this technique will be discussed in chapters later in this book.

Bit processing and Boolean logic

The 8051 contains a single bit Boolean processor which can be used to perform logical operations on any of the 128 addressable bits in the BIT segment, the 128 addressable bits in the SFRs, and any of the 32 I/O lines (port 0 through port 3). The 8051 can perform OR, AND, XOR, complement, set, and clear operations on bits as well as moving bit values as one would normally move byte values.

Listing A - 1

```
MOV    C, 22H          ; move the bit value at address
                        ; 22H to the carry bit
ORL     C, 23H          ; or the bit value at address
                        ; 23H into the carry bit
ANL     24H, C          ; and the carry bit into bit
                        ; address 24H
```

There are also conditional branches which use addressed bits as the condition. One such branch which is especially useful is the "jump if bit is set and clear bit" instruction. This "branch and clear" can be performed in two processor cycles and saves a cycle or two over splitting the jump and the clear into two separate op codes. As an example, suppose that you had to write a routine which waited for pin P0.0 to set, but could not wait indefinitely. This routine would have to decrement a timeout value and exit the polling loop when this timeout is exceeded. When pin P0.0 sets, the processor must force it back to 0 and exit the polling loop. With normal logic flow, the routine would look like the following.

Listing A - 2

```
MOV     timeout, #TO_VAL ; set the timeout value
L2:     JB      P0.0, L1   ; check the bit
        DJNZ    timeout, L2 ; decrement the timeout counter
                        ; and sample again
L1:     CLR     P0.0       ; force P0.0 to logic level 0
        RET                      ; exit the routine
```

Using the JBC instruction, the same routine would be coded as follows.

Listing A - 3

```
MOV     timeout, #TO_VAL ; set the timeout value
L2:     JBC     P0.0, L1   ; check the bit and force P0.0
                        ; to logic level 0 if set
        DJNZ    timeout, L2 ; decrement the timeout counter
L1:     RET                      ; exit the routine
```

While the second routine may not offer a huge amount of savings in the code, it does make the code a little simpler and more elegant. There will be many situations in your use of assembly code on the 8051 controller where this instruction will come in handy.

Addressing Modes

The 8051 is capable of performing direct and indirect memory accesses on its various memory spaces. These are the typical methods through which processor systems access memory. Direct accesses are characterized by presence of the address of the accessed variable in the instruction itself. These accesses can only be performed on the DATA segment and the SFRs. Examples of direct memory accesses are shown below.

Listing A - 4

```
MOV    A, 03H                ; move the value at address 03H to
                                ; the accumulator
MOV    43H, 22H              ; move the value at address 22H to
                                ; address 43H
MOV    02H, C                ; move the value of the carry bit to
                                ; bit address 02H
MOV    42H, #18              ; load address 42H with the value 18
MOV    09H, SBUF             ; load the value in SBUF into
                                ; address 09H
```

Indirect accesses involve another register (DPTR , PC, R0, or R1 on the 8051) which contains the address of the variable to be accessed. The instruction then refers to the pointing register rather than the address itself. This is how all accesses to CODE, IDATA, and XDATA segments are performed. The DATA segment may also be accessed in this manner. Bits in the BDATA segment can only be accessed directly.

Indirect memory accesses are quite useful when a block of data must be moved, altered or operated on with a minimum amount of code since the pointer can be incremented through the memory area via a looping mechanism. Indirect accesses to the CODE segment can have a base address in either the DPTR or the PC register and an offset in the accumulator. This is useful for operations involving lookup tables. Examples of indirect memory accesses are shown below.

Listing A - 5

DATA and IDATA accesses

```
MOV    R1, #22H           ; set R1 to point at DATA
                                ; address 22H
MOV    R0, #0A9H          ; set R0 to point at IDATA
                                ; address A9H
MOV    A, @R1             ; read the value at DATA
                                ; address 22H
                                ; into the accumulator
MOV    @R0, A             ; write the value in the accumulator
                                ; to IDATA address A9H
INC    R0                 ; set R0 to point at IDATA
                                ; address AAH
INC    R1                 ; set R1 to point at DATA
                                ; address 23H
MOV    34H, @R0           ; write the value at IDATA
                                ; address AA
                                ; to DATA address 34H
MOV    @R1, #67H          ; write 67H to DATA address 23H
```

XDATA accesses

```
MOV    DPTR, #3048H       ; set DPTR to point at XDATA
                                ; address 3048H
MOVX   A, @DPTR           ; read the data at XDATA
                                ; address 3048H
                                ; into the accumulator
INC    DPTR               ; set DPTR to point at XDATA
                                ; address 3049H
MOV    A, #26H            ; set the accumulator to 26H
MOVX   @DPTR, A           ; write 26H to XDATA address 3049H

MOV    R0, #87H           ; set R0 to point at XDATA
                                ; address 87H
MOVX   A, @R0             ; read the data at XDATA
                                ; address 87H into the accumulator
```

CODE accesses

```
MOV    DPTR, #TABLE_BASE ; set DPTR to point at the base
                                ; of a lookup table
MOV    A, index           ; load the accumulator with an
                                ; index into the table
MOVC   A, @A+DPTR         ; read the value from the table
                                ; into the accumulator
```

Processor Status

Processor status is kept in a bit addressable SFR called PSW (Processor Status Word). This register contains the carry bit, an auxiliary carry bit which is used with BCD operations, the Accumulator parity flag and overflow flag, two general purpose flags, and two bits which select the register bank to use as the default. As mentioned before, the register bank selection bits make a two bit number from 0 to 3 which indicates the bank to be used. Bank 0 begins at the base of the DATA segment (address 00H), bank 1 begins at address 08H, bank 2 at address 10H and bank 3 at address 18H. Any of these memory locations are always available for direct and indirect memory accesses via their addresses regardless of the register bank selection. The layout of PSW is shown below.

Power Control

The CHMOS versions of the 8051 feature two power saving modes that can be activated by software: idle mode and power down mode. These modes are accessed via the PCON (Power CONTROL) SFR which is shown in Table A - 2. The idle mode is activated by setting the IDLE bit high. The idle mode causes all program execution to stop. Internal RAM contents are preserved and the oscillator continues to run but is blocked from the CPU. The timers and the UART continue to function as normal. Idle mode is terminated by the activation of any interrupt. Upon completion of the interrupt service routine, execution will continue from the instruction following the instruction which set the IDLE bit.

Processor Status Word (PSW) - Bit Addressable

CY	AC	F0	RS1	RS0	OV	USR	P
CY	Carry Flag						
AC	Auxiliary Carry Flag						
F0	General Purpose Flag						
RS1	Register Bank Selector 1. MSB of selector.						
RS0	Register Bank Selector 0. LSB of selector.						
OV	Overflow Flag						
USR	User Definable Flag						
P	Accumulator Parity Flag						

Table A - 2

The power down mode is activated by setting the PDWN bit high. In this mode, the on chip oscillator is stopped. Thus, the timers and the UART as well as software execution are halted. As long as a minimum of 2 volts are applied to the chip (assuming a five volt part) the contents of the internal RAM will be preserved. The only way to force the processor out of power down mode is by applying a power on reset.

The SMOD (Serial MODE) bit can be used to double the baud rates of the serial port whether generated by the timer 1 overflow rate or the oscillator frequency. Setting SMOD high causes a doubling of the baud rate for the UART when operated in mode 1, 2, or 3. When Timer 2 is used to generate baud rates, the value of SMOD will have no effect on the UART.

Power Control Register (PCON) - Not Bit Addressable

SMOD	-	-	-	GF1	GF0	PDWN	IDLE
SMOD	Serial baud rate generator mode. If SMOD=1 the baud rate of the UART is doubled.						
-	Reserved.						
-	Reserved.						
-	Reserved.						
GF1	General Purpose flag.						
GF0	General Purpose flag.						
PDWN	Power Down flag. Setting this bit causes activation of power down mode.						
IDLE	Idle flag. Setting this bit causes activation of idle mode.						

Table A - 3

Interrupts on the 8051

The basic 8051 supports six interrupt sources: two external interrupts, two timer/counter interrupts, and a serial byte in/out interrupt. These interrupt sources force the processor to vector to one of five locations in the lowest part of the CODE address space (serial input and serial output interrupts share the same vector). The interrupt service routine must either reside there or be branched to from there. A map of the interrupt vector for the 8051/8052 is shown below in Table A - 4.

Interrupt Source	Vector Address
Power On Reset	0000H
External Interrupt 0	0003H
Timer 0 Overflow	000BH
External Interrupt 1	0013H
Timer 1 Overflow	001BH
Serial Receive/Transmit	0023H
Timer 2 Overflow	002BH

Table A - 4

The 8015 supports two interrupt priority levels: low and high. The nature of the interrupt mechanism is very standard and thus, a low level interrupt service routine can only be interrupted by a high level interrupt and a high level interrupt service routine cannot be interrupted.

Interrupt Priority Register

Each interrupt source can be individually set to one of two priority levels by altering the value of the IP (Interrupt Priority) SFR. If an interrupt source's corresponding bit in this register is set, it will have high priority. Similarly, if the corresponding bit is cleared the interrupt will be of low priority and subject to being interrupted by any high priority interrupts. If two levels of priority seems a little limited, hang on - later I'll discuss how to raise the number of priority levels as high as you want. Table A - 5 shows the IP register and its bit assignment. Note that this register is bit addressable.

Interrupt Priority Register (IP) - Bit Addressable

-	-	PT2	PS	PT1	PX1	PT0	PX0
-	Reserved						
-	Reserved						
PT2	Timer 2 overflow interrupt priority level						
PS	Serial receive and transmit complete interrupt priority						
PT1	Timer 1 overflow interrupt priority						
PX1	External interrupt 1 priority						
PT0	Timer 0 overflow interrupt priority						
PX0	External interrupt 0 priority						

Table A - 5

Interrupt Enable Register

All interrupts are enabled or blocked by setting or clearing the EA bit (Enable All) of the IE (Interrupt Enable) register. Each interrupt source can be individually enabled and disabled at any time by the software by altering the value of the corresponding enable bit in the IE SFR. Table A - 6 shows the IE register and its bit assignment. Like the IP register, the IE SFR is bit addressable.

Interrupt Enable Register (IE) - Bit Addressable

EA	-	ET2	ES	ET1	EX1	ET0	EX0
EA	Enable Flag. If EA=1, each interrupt can be enabled via its enable bit. If EA=0, no interrupts are allowed.						
-	Reserved						
ET2	Timer 2 overflow interrupt enable						
ES	Serial receive and transmit complete interrupt enable						
ET1	Timer 1 overflow interrupt enable						
EX1	External interrupt 1 enable						
ET0	Timer 0 overflow interrupt enable						
EX0	External interrupt 0 enable						

Table A - 6

Interrupt Latency

The 8051 samples the interrupt flags once every processor cycle to determine if any interrupts are pending. An interrupt is requested by the appropriate signal being set for the processor core to recognize in its next sampling period. Thus, the time between an interrupt being requested and recognized by the processor is one instruction cycle. At this point, the hardware will generate a call to the interrupt service routine in the vector which takes two cycles. Thus, the overall process takes three cycles total. Under ideal conditions (where nothing is blocking the interrupt call) and no instruction is in the works, an interrupt will be responded to in three instruction cycles. This response time is excellent and provides the user with very fast response time to system events.

There will inevitably be times that an interrupt is not responded to within the three cycles discussed above. The most significant of these is when an interrupt of equal or higher priority is being serviced. In this case, the latency to service the pending interrupt depends entirely on the ISR currently being executed.

Another situation in which the latency will be more than three cycles occurs when the processor is executing a multi-cycle instruction and detects a pending interrupt during this instruction. The pending interrupt will not be serviced until the current instruction is completed. This situation will add a minimum of one cycle to the latency (assuming that a two cycle instruction such as a MOVX is executing) to a maximum of three cycles (assuming the interrupt is detected after the first cycle of a MUL). The maximum condition gives a worst case latency of six instruction cycles (the three cycles due to the architecture itself and the three cycles due to the completion of the instruction) when the pending interrupt is not blocked by a currently executing interrupt.

The final case in which an interrupt will not be vectored to in three cycles is when the interrupt was recognized during a write to IE, IP, or during a RETI (return from interrupt) instruction. This prevents very odd real time conditions from occurring in your system unexpectedly.

External Interrupt Signals

The 8051 supports two external interrupt signals. These inputs allow external hardware devices to request interrupts and thus some sort of service from the 8051. The external interrupts on the 8051 are caused by either a low logic level on the interrupt pin (P3.2 for interrupt 0 and P3.3 for interrupt 1) or by a high to low level transition in the interrupt pin. The mode of the interrupt (level triggered or edge triggered) is selected by altering the ITx (interrupt type) bit corresponding to the interrupt in the TCON (Timer CONtrol) register. The layout of the TCON register is shown below in Table A - 7.

In level mode, the interrupt will be fired any time the processor samples the input signal and sees a 0. For the low to be detected, it must be held for at least one processor cycle (or 12 oscillator cycles) since the processor samples the input signal once every instruction cycle. In edge mode, the interrupt is fired when a one to zero transition is detected during back to back samples. Therefore, the zero state of the input must be held for at least one processor cycle to ensure that it is sampled.

On-Board Timer/Counters

The standard 8051 has two timer/counters (other 8051 family members have varying amounts), each of which is a full 16 bits. Each timer/counter can function as a free running timer (in which case they count processor cycles) or can be used to count falling edges on the signal applied to their respective I/O pin (either T0 or T1). When used as a counter, the input signal must have a frequency equal to or lower than the instruction cycle frequency divided by 2 (ie: the oscillator frequency /24) since the incoming signal is sampled every instruction cycle, and the counter is incremented only when a 1 to 0 transition is detected (which will require two samples). If desired, the timer/counters can force a software interrupt when they overflow.

The TCON (Timer CONtrol) SFR is used to start or stop the timers as well as hold the overflow flags of the timers. The TCON SFR is detailed below in Table A - 7. The timer/counters are started or stopped by changing the timer run bits (TR0 and TR1) in TCON. The software can freeze the operation of either timer as well as restart the timers simply by changing the TRx bit in the TCON register. The TCON register also contains the overflow flags for the timers. When the timers overflow, they set their respective flag (TF0 or TF1) in this register. When the processor detects a 0 to 1 transition in the flag, an interrupt occurs if it is enabled. It should be noted that the software can set or clear this flag at any time. Therefore, an interrupt can be prevented as well as forced by the software.

Timer Control Register (TCON) - Bit Addressable

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
TF1	Timer 1 overflow flag. Set when timer 1 overflows. Cleared by processor upon vectoring to the interrupt service routine.						
TR1	Timer 1 control bit. If TR1=1, timer 1 runs. If TR1=0, timer 1 stops.						
TF0	Timer 0 overflow flag. Set when timer 0 overflows. Cleared by processor upon vectoring to the interrupt service routine.						
TR0	Timer 0 control bit. If TR0=1, timer 1 runs. If TR0=0, timer 1 stops.						
IE1	External interrupt 1 edge flag. Set when a valid falling edge is detected at pin P3.3. Cleared by hardware when the interrupt is serviced.						
IT1	Interrupt 1 type control bit. If IT1=1, interrupt 1 is triggered by a falling edge on P3.3. If IT1=0, interrupt 1 is triggered by a low logic level on P3.3						
IE0	External interrupt 0 edge flag. Set when a valid falling edge is detected at pin P3.2. Cleared by hardware when the interrupt is serviced.						
IT0	Interrupt 0 type control bit. If IT0=1, interrupt 1 is triggered by a falling edge on P3.2. If IT0=0, interrupt 0 is triggered by a low logic level on P3.2						

Table A - 7

The timers are configured by altering the value in the TMOD (timer mode) SFR. By changing TMOD, the software can control the mode of both timers as well as the source they use to count (the signal at their I/O pin or the processor cycles). The upper nibble of TMOD controls the operation of timer 1 and the low nibble controls the operation of timer 0. The layout of the TMOD register (which is not bit addressable) is shown below.

Timer Mode Register (TMOD) - Not Bit Addressable

GATE	C/T	M1	M0	GATE	C/T	M1	M0
Timer One				Timer Zero			
GATE	If GATE=1, timer x will run only when TRx=1 and INTx=1. If GATE=0, timer x will run whenever TRx=1.						
C/T	Timer mode select. If C/T=1, timer x runs in counter mode taking its input from Tx pin. If C/T=0, timer x runs in timer mode taking its input from the system clock.						
M1	Mode selector bit 1. MSB of selector.						
M0	Mode selector bit 0. LSB of selector.						

Table A - 8

The source for the timer can be configured by altering the C/T bit in TMOD. Setting this bit to true will force the timer to count pulses on the I/O pin assigned to it. Setting this bit false will force counting of processor cycles. When a timer is forced to count processor cycles it can do this either under hardware or software control. Software control is commanded by setting the GATE bit of TMOD to 0. In this case, the timer will count any time its TRx bit in the TCON register is high. In the hardware control mode, both the TRx bit and the INTx pin on the chip must be high for the timer to count. When a low is detected at the INTx pin, the timer will stop. This is useful for measuring pulse widths of signals on the INTx pin if one does not mind surrendering an external interrupt source to the incoming signal.

Timer Mode 0 and Mode 1

The timer/counters can be operated in one of four modes, under software control. In mode 0, the timer/counter will behave like a 13 bit counter. When the counter overflows, the TF0 or TF1 (timer flag) bit in the TCON (timer control) SFR is set. This will cause the appropriate timer interrupt (assuming it is enabled). Both timer 0 and timer 1 operate in the same way for mode 0. The operation of the timers in mode 1 is the same as it is for mode 0 with the exception that all sixteen bits of the timer are used instead of only thirteen.

Timer Mode 2

In mode 2, the timer is set up as an eight bit counter which automatically reloads whenever an overflow condition is detected. The low byte of the timer (TL0 or TL1) is used as the counter and the high byte of the timer (TH0 or TH1) holds the reload value for the counter. When the timer/counter overflows, the value in THx is loaded into TLx and the timer continues counting from the reload value. Both timer 0 and timer 1 function identically in mode 2. Timer 1 is often used in this mode to generate baud rates for the UART.

Timer Mode 3

In mode 3, timer 0 becomes two eight bit counters which are implemented in TH0 and TL0. The counter implemented in TL0 maintains control of all the timer 0 flags, but the counter in TH0 takes over the control flags in TCON from timer 1. This implies that timer 1 can no longer force interrupts, however, it can be used for any purpose which will not require the overflow interrupt such as a baud rate generator for the UART, or as a timer/counter which is polled by the software. This is useful when an application must use a UART mode which requires baud rate generation from timer 1 and also requires two timer/counters. When timer 1 is placed in mode 3 it simply freezes.

Timer 2

Many 8051 family members, such as the 8052 also have a third on board timer referred to as timer 2. Timer 2 is controlled through the T2CON (Timer 2 CONTROL) SFR. The T2CON SFR is bit addressable. Its layout is shown below.

Timer 2 Control Register (T2CON) - Bit Addressable

TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2	CP/RL2
TF2	Timer 2 overflow flag. Set when timer 2 overflows. Will not be set if RCLK=1 or TCLK=1.						
EXF2	Timer 2 external flag. EXF2 is set when a falling edge is detected on T2EX and EXEN2=1. This causes an interrupt, if the timer 2 interrupt is enabled.						
RCLK	Receive clock flag. When RCLK=1, the UART (if in mode 1 or 3) will use the timer 2 overflow frequency for the receive clock.						
TCLK	Transmit clock flag. When TCLK=1, the UART (if in mode 1 or 3) will use the timer 2 overflow frequency for the transmit clock.						
EXEN2	External enable flag. If EXEN2=1, a capture or reload will be caused by a falling edge on T2EX. If EXEN2=0, external events on T2EX are ignored.						
TR2	Timer run control bit. If TR2=1, the timer will run. If TR2=0, the timer will stop.						
C/T2	Timer mode select. If C/T2=1, timer 2 will act as an external event counter. If C/T2=0, timer 2 will count processor clock cycles.						
CP/RL2	Capture/Reload flag. If CP/RL2=1, detection of a falling edge on T2EX causes a capture if EXEN2=1. If CP/RL2=0, detection of a falling edge on T2EX or an overflow causes a timer reload if EXEN2=1.						

Table A - 9

Via T2CON the software can configure timer/counter 2 to operate in one of three basic modes. The first of these modes is referred to as Capture mode. In Capture Mode the timer can be operated just as timer 0 or timer 1 in the 16 bit timer/counter mode (mode 1). This operation is selected by clearing the EXEN2 bit. When the EXEN2 bit is set, the timer/counter will latch its current count in two other SFRs (RCAP2H and RCAP2L) when the signal at P1.1 (referred to as T2EX) exhibits a 1 to 0 transition. This event can also be linked to an interrupt from T2.

A second mode of timer 2 is called auto reload. In this mode there are also two sub functions which are selected via the EXEN2 bit. When EXEN2 is cleared, the rollover of the 16 bit timer fires an interrupt and loads the value set in RCAP2H and RCAP2L into the timer. When EXEN2 is set, the timer/counter will react the same way to a rollover and in addition it will also reload the timer given a 1 to 0 transition at the T2EX pin.

In its final mode, timer 2 can be used to generate a baud rate for the UART. This is done by setting either RCLK or TCLK or both. In its baud rate generator mode, the timer increments once every other oscillator cycle instead of once every 12th oscillator cycle as timer 0 and timer 1 do meaning that the maximum UART baud rate is higher. Additionally, the entire 16 bits of the timer are reloaded from RCAP2H and RCAP2L every overflow.

On-Board UART

The 8051 features an on board, full duplex UART which is under software control. The UART is configured via the SCON (Serial CONTROL) SFR. The SCON register allows the user to select the UART mode, enable reception, and check UART status. SCON is illustrated in Table A - 10.

Serial Control Register (SCON) - Bit Addressable

SM0	SM1	SM2	REN	TB8	RB8	TI	RI
SM0	Serial Port Mode Specifier 0. MSB						
SM1	Serial Port Mode Specifier 1. LSB.						
SM2	Multiprocessor Mode enable. In mode 0, this bit should be 0. In mode 1, if SM2=1, RI will not be set unless a valid stop bit was received. In modes 2 and 3 if SM2=1, RI will not be set unless the ninth data bit is 1.						
REN	Receive Enable Flag. Must be 1 to allow UART to receive data.						
TB8	The ninth data bit that will be sent in mode 2 and 3.						
RB8	In mode 0 this bit is unused. In mode 1 if SM2=0, RB8 is the stop bit that was received. In modes 2 and 3 RB8 is the ninth data bit that was received.						
TI	Transmit interrupt flag. Must be cleared by software.						
RI	Receive interrupt flag. Must be cleared by software.						

Table A - 10

The UART features a one byte buffer for incoming data so that another byte can be ringing into the UART before the last byte has been read. However, after one byte time, the buffer will be overwritten as the next incoming byte is completed. Therefore, the software must be capable of responding to an incoming byte within one serial byte time. This is also true for outgoing data assuming that it is required to be back to back.

The 8051 supports standard ten bit frames as well as an eleven bit frame designed for inter processor communications and a high speed 8 bit shift register mode. The baud rate is variable for all modes except the eight bit shift mode and one of the inter processor modes.

UART Mode 0

In mode 0 the UART acts as an eight bit shift register clocking data in and out at a baud rate of 1/12th of the oscillator frequency. Data is sent LSB first and enters and exits the UART via the RXD pin. Therefore mode 0 does not support full duplex since the RXD pin is used for all incoming and outgoing data and the TXD pin is used to echo the shift clock. This mode is useful for situations in which the micro controller is used to interface to a serial device such as an EEPROM which has a serial eight bit interface format.

Transmission of a byte begins when the SBUF SFR is the destination register of a move instruction. At this point, the eight bits are clocked out and the TI bit is set when the transmission of the eighth bit is complete. Reception begins when the REN bit of the SCON register is set true. The RI bit is set when the eighth bit is clocked in.

UART Mode 1

In mode 1 of the UART, 8 data bits are transmitted in a ten bit frame: one start bit, eight data bits, and one stop bit. This mode is suitable for communications to most serial devices including personal computers. The baud rate is variable and is controlled by the overflow rate of timer 1 in the auto reload mode or, optionally, by timer 2 in the baud rate generating mode on an 8052. Overflow interrupts should be disabled for the timer being used to generate the baud rate. The SMOD bit in the PCON (power control) SFR can be set high to double the baud rate implemented by the UART.

The TI and RI interrupt signals are activated halfway through the transmission or reception of the stop bit. Typically, this will allow the software time to respond to the interrupt and load SBUF with the next

byte in back to back during data block transfers. The amount of processing time available depends on the baud rate in use and the oscillator frequency being used to drive the 8051.

If timer 1 is going to be used to generate the desired baud rate of the UART, you must compute the reload value for TH1 using the following equation:

```
TH1=256-(K*OscFreq)/(384*BaudRate)
K=1 if SMOD=0
K=2 if SMOD=1
```

Any baud rate which does not give a positive reload value less than 256 can not be generated by the 8051 at the given clock frequency. Reload values which are not integers must be very close to the next integer. Oftentimes the resultant baud rate may be close enough to allow the system to work. This evaluation must be made by the developer.

Thus, if you have an 8051 which is using a 9.216MHz oscillator, and you want to generate a baud rate of 9600 baud you must go through these steps. First, run the equation for K=1 then later try it for K=2. For K=1, the numerator becomes 9216000 and the denominator becomes 3686400. Dividing these two values gives a result of 2.5. From this it is obvious that the reload value given by this function will not be an integer. Rerunning the equation with K=2 gives a numerator of 18432000 and a denominator of 3686400. Dividing these two values gives an answer of 5 which you subtract from 256. This gives a reload value of 251 or 0FBH for TH1.

For an 8052 using timer 2 to generate the baud rate, the reload value for RCAP2H and RCAP2L must be computed. Again, you must start from the desired baud rate and solve the following equation to obtain the reload values.

```
[RCAP2H, RCAP2L]=65536-OscFreq/(32*BaudRate)
```

Assume that you again have a system with an oscillator at 9.216MHz, and you want to generate a baud rate of 9600 baud. For this to be possible, the resultant 16 bit answer of the above equation must be both positive and "near integer." You end up dividing 9216000 by 307200 and getting an intermediate result of 30. Subtracting this from 65536 gives an answer of 65506 or FFE2H. You should then use a reload value of 255 or FFH for RCAP2H and a reload value of 226 or E2H for RCAP2L.

UART Mode 2

Mode 2 of the UART causes an eleven bit frame to be transmitted: one start bit, eight data bits, a ninth (or stick) bit, and one stop bit. The value of the ninth bit is determined by the TB8 bit of SCON for transmissions and the RB8 bit of SCON for receptions. The ninth bit is typically used for inter processor communications. To facilitate this, the UART can be initialized to fire a receive interrupt only when the ninth bit (or stick bit) is set. This feature is referred to as the multiprocessor communication feature by Intel and is controlled by the SM2 bit in the SCON register. When SM2 is set, the ninth data bit must be set for the UART to fire an interrupt. When it is cleared, the UART will fire a receive interrupt whenever a valid eleven bit frame rings in.

The stick bit is used to lower the amount of unnecessary interrupts during serial communications across a multidrop serial bus. In such situations an address or command byte is sent first with the stick bit set. All processors on the bus are interrupted and check the incoming byte to see if it is necessary for them to receive the message. If it is, the SM2 bit is cleared to remove the restriction of having the stick bit set, and the rest of the message is received. Otherwise, the SM2 bit is left set and the normal processing continues without constantly being disturbed by a string of interrupts for the incoming byte stream.

The baud rate for mode two is 1/64th of the oscillator frequency when the SMOD bit is cleared and it is 1/32nd of the oscillator frequency when the SMOD bit is set. Therefore, very high baud rates (over 345K baud) are achievable using this mode and a relatively common oscillator frequency such as 11.059MHz. Mode 3 of the UART is exactly the same as mode two in terms of the data format and the use of the ninth bit. However, the baud rates in mode 3 are variable and are controlled in the same manner as in mode 1.

Other Peripherals

Many 8051 derivatives have additional devices integrated onto the chip to make them a more attractive product for your embedded application. Some of the more common peripherals are discussed below.

I²C

A new form of inter-device communication becoming popular is the I²C (inter-integrated circuit) interface created and popularized by Phillips. I²C is a serial format data link which uses two wires (one for data and one for clock) and can have many drops to varying devices. Each device has its own ID on the link to which it will respond, data transfers are bi-directional, and the bus can have more than one master. Phillips has been a leader in adding I²C capability to the 8051. Hardware wise, two I/O pins are taken from port 1 for the I²C interface and a set of SFRs are added to control the I²C and aid in implementing the protocol of this interface. Specifics on the I²C interface can be obtained in the Phillips 8051 Family data book.

Analog to Digital Converters

Analog to digital converters are peripherals which are not available on every 8051 family member, but are common enough that they were worth discussing in this overview. A/D converters are usually controlled via some master register (usually called ADCON) which is given one of the empty locations in the SFR memory segment. The ADCON register allows the user to select the channel to be used for A/D conversion, to start a new conversion and to check the status of a current conversion. Typical A/D converters are taking 40 instruction cycles or less to complete the conversion, and they can be configured to fire an interrupt upon completion which causes the processor to vector to a location specific for the A/D. The drawback to this is that often times the A/D conversion requires that the processor be active rather than entering idle mode to wait for the completion interrupt. Results of a conversion are read from another SFR or pair of SFRs depending on the resolution of the converter.

Watchdog Timers

Watchdog timers are available on an expanding group of 8051 family members. The purpose of a watchdog timer is to reset the controller if the timer is not fed by a specific sequence of operations within a specified amount of time. This prevents coincidental reloading of the watchdog by runaway software. To use a watchdog, the timing in the software must be understood well enough for the designer to determine where the calls to the feed routine should be placed in the system. If the watchdog is fed too often, some amount of processing power is wasted. However, if the watchdog is not fed often enough, it may reset the system even though the software is still functioning as expected.

In the 8051 family, the watchdog is usually implemented as another on board timer which scales the system oscillator down and then counts the divided clock. When the timer rolls over, the system resets. The watchdog can be configured for its rollover rate and often times can be used as another timer, albeit a low resolution one.

Design Considerations

The 8051 family of processors contains a wide variety of members with a wide range of peripherals and features and is suitable for a high amount of applications. Given such a multitude of choices, the greatest difficulty in using this controller may just be selecting the appropriate derivative! When circuit board space and cost are a consideration (as they often are) it will be desirable to keep the parts count as low as possible. The 8051 family provides many options with controllers that have up to 512 bytes of on board RAM and up to 32K bytes of on board EPROM. Often times a system can be done using just the internal RAM and EPROM on the 8051. The advantages of this in terms of parts count are significant. From the start, you eliminate the need for an EPROM (which is typically a 28 pin part), an address latch for the lower eight bits of the bus (typically a 20 pin part), and an external RAM (which is also a 28 pin part). In addition to these parts savings, you have increased the available I/O capability of the 8051 by 16 pins (port 0 and port 2). This can be used to easily interface other devices to the 8051 without having any sort of bus interface for them which would typically involve a decoder and possibly more data latches. When the extra I/O pins are not needed and the application code will be suitably small, a 28 pin version of the 8051 can be used to save even more circuit board space. A drawback to approaches like this is that there may not be sufficient program or RAM space for larger applications. When this is the case, the designer has little choice but to go with the full 8051 core and whatever support chips (SRAM, EPROM, etc) are required. Many components such as A/D, PWM, hardware triggers and timers can be replaced by the correct 8051 family member and the appropriate software control routines which will be discussed later.

Oftentimes power consumption of an embedded product is of great concern. It may be that the software has so many chores to do that the processor does not get to spend much time in sleep or idle mode. In these cases, the designer has the option of going to a low voltage (3.6 volts or below) system to reduce power consumption. Additionally, if there is sufficient spare processing time available, the designer can consider lowering the oscillator frequency which will provide small gains in power consumption.

The designer must carefully choose the oscillator frequency for applications that must communicate serially at standard baud rates (1200, 4800, 9600, 19.2K, etc.). It is very beneficial to generate tables of the possible baud rates for readily available crystals and then select your frequency based upon required baud rates, required processing power, and availability. Oftentimes crystal availability can be given a much lower priority in this equation due to the fact that the set up cost to manufacture custom crystals is usually not overwhelming. When selecting an oscillator frequency greater than 20MHz, the designer must be careful to ensure that any parts placed in the bus of the 8051 can handle the access times that will be required by the core. Typically, parts such as EPROMs and SRAMs which can handle the access speeds are readily available. Address latches such as the 74C373 are also available in HC versions that can support all 8051 frequencies. In addition, the designer must consider that as the crystal frequency is increased, the power consumption of the system will also be increased. This trade off, as discussed above, must be carefully considered for applications that must run off batteries for any length of time.

Implementation Issues

After the appropriate 8051 family member is selected and the necessary peripherals are chosen, the next issue to be decided is typically the memory map for system I/O. It is a given that the CODE space will start at address 0 and will increase upward in a contiguous block. This could be altered, but in my experience I have never seen a real need to justify it. The XDATA memory space is usually composed of some combination of RAM and I/O devices. Again, the RAM is typically one contiguous block of memory and starts at address 0000 or address 8000. It is oftentimes useful to put the SRAM at address 0000 and use A15 to enable the RAM in conjunction with the RD' and WR' signals generated by the micro controller. This approach allows for a RAM of up to 32K to be used which is usually more than sufficient for an embedded application. Additionally, 32K of address locations (from 8000 to FFFF) can be given to external I/O devices. For the most part, the number of I/O devices present in an 8051 system is low, and therefore the higher order address lines can be run through a decoder to provide enable signals for the peripherals. An example of a base core implementing such a memory map for its system I/O is shown in Figure A - 2 - 8051 Bus I/O. As can easily be seen, this approach simplifies the hardware by reducing the amount of address decoding required to access a given I/O device. It can also simplify the software since it will not be necessary to load the lower half of DPTR when performing I/O with these devices.

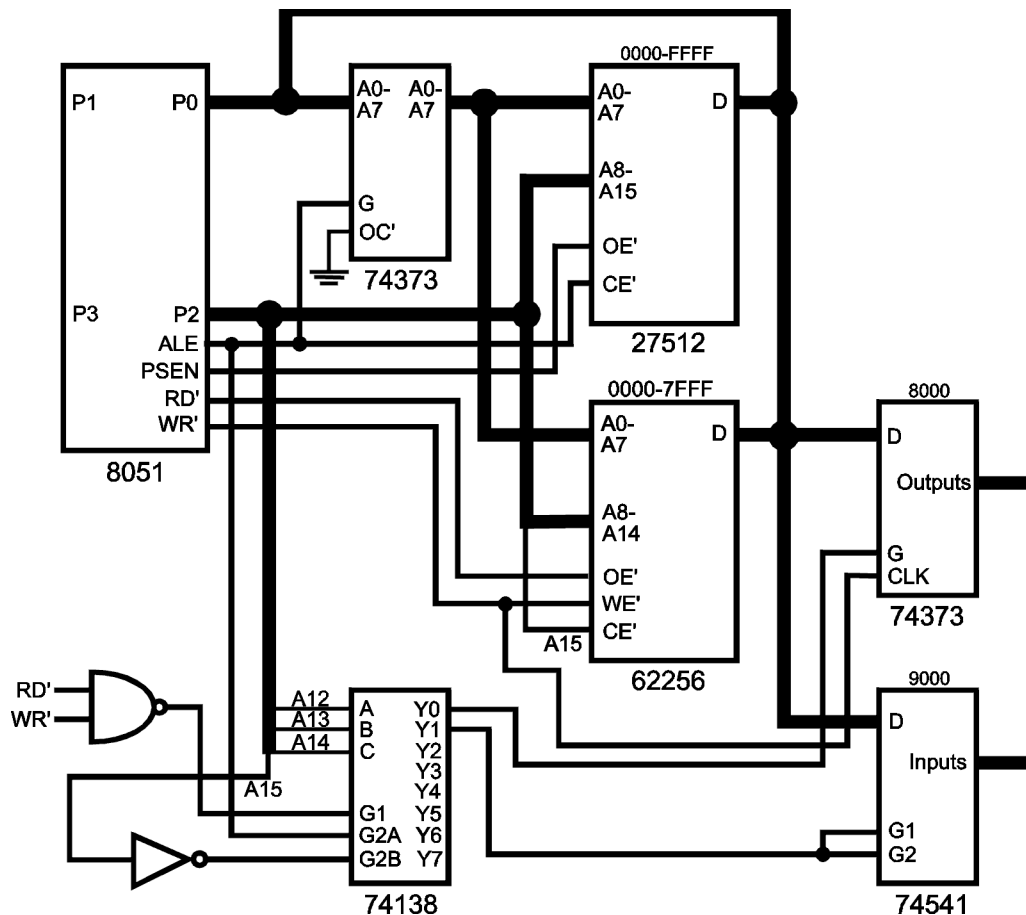


Figure A - 2 - 8051 Bus I/O

Sample accesses to the input and output latch for this circuit are shown below.

Listing A - 6

```
MOV    DPTR, #09000H      ; set DPTR to point at the input
                                ; latch
MOVX   A, @DPTR           ; read the value of the latch
MOV    DPH, #080H         ; set DPTR to point at the output
                                ; latch
MOVX   @DPTR, A           ; write the input value to the
                                ; output latch
```

It can be seen that sequential I/O will be simplified by the architecture laid out in the above circuit since the software does not need to concern itself with the lower half of the data pointer. The first instruction could just as easily been "MOV DPH, #090H" since it does not matter what value is on the lower order 8 bits.

Conclusion

I hope that this brief review of 8051 basics has been enlightening. It is not intended to replace the data book that the manufacturers of 8051 family members provide you with. These books are always an invaluable source of information regarding chip specifics and operation details. They have a permanent place on my desk. The next chapter will explore general software design issues for the 8051, including the use of the C programming language.

■ *Using C with the 8051*

Why Use a High Level Language?

When designing software for a smaller embedded system with the 8051, it is very commonplace to develop the entire product using assembly code. With many projects, this is a feasible approach since the amount of code that must be generated is typically less than 8 kilobytes and is relatively simple in nature. If a hardware engineer is tasked with designing both the hardware and the software, he or she will frequently be tempted to write the software in assembly language. My experience has been that hardware engineers are usually not familiar with a high level language like C nor do they care to be.

The trouble with projects done with assembly code can be that they can be difficult to read and maintain, especially if they are not well commented. Additionally, the amount of code reusable from a typical assembly language project is usually very low. Use of a higher level language like C can directly address these issues.

A program written in C is easier to read than an assembly program. Since a C program possesses greater structure, it is easier to understand and maintain. Because of its modularity, a C program can better lend itself to reuse of code from project to project. The division of code into functions will force better structure of the software and lead to functions that can be taken from one project and used in another, thus reducing overall development time.

A high order language such as C allows a developer to write code which resembles a human's thought process more closely than does the equivalent assembly code. The developer can focus more time on designing the algorithms of the system rather than having to concentrate on their individual implementation. This will greatly reduce development time and lower debugging time since the code is more understandable.

By using a language like C, the programmer does not have to be intimately familiar with the architecture of the processor. This means that someone new to a given processor can get a project up and running quicker, since the internals and organization of the target processor do not have to be learned. Additionally, code developed in C will be more portable to other systems than code developed in assembly. Many target processors have C compilers available which support ANSI C.

All of this is not to say that assembly language does not have its place. In fact, many embedded systems (particularly real time systems) have a combination of C and assembly code. For time critical operations, assembly code is frequently the only way to go. It has been my experience, however, that the remainder of the project (including much of the hardware interface) can and should be developed in C. One of the great things about the C language is that it allows you to perform low level manipulations of the hardware if need be, yet provides you with the functionality and abstraction of a higher order language.

Sticking Points with C

This text is not intended to teach you how to program using the C language. Numerous books are available to help you learn the C language. The most widely regarded book is [The C Programming Language](#) by Kernighan and Ritchie. Their book is generally considered to be the final authority on C. Keil's C51 fully supports the C standard set forth in the Kernighan and Ritchie book as well as many C language extensions which are specifically designed to optimize use of the 8051's architecture.

There are a few issues regarding the C language that many users of C still shy away from. Even though this book is not a C tutorial, it is worth it to review the concepts of structures, unions, pointers and type definitions. These three topics seem to cause the new and occasional C programmer the most grief.

Structures

A structure is a user defined type in C which allows the programmer to group together several variables into a single collection. This feature is very handy when you have variables which are closely related. For example, assume that you have a set of variables which keep track of the time of day. To do this, you have defined an hour, minute, and second variable to hold each portion of the time as follows.

```
unsigned char hour, min, sec;
```

This set of variables is further augmented by a variable which keeps track of the current day of the year (from 0 to 364). This variable is defined as follows.

```
unsigned int days;
```

Taken together, you have four variables which together hold the time of the day. This is certainly workable, but can be written to be much cleaner using a structure. The structure will allow you to group together these four variables and give them a common name. The syntax for declaring the structure is as follows.

```
struct time_str {  
    unsigned char hour, min, sec;  
    unsigned int days;  
} time_of_day;
```

This code tells the compiler to define a structure type called `time_str` and create a variable called `time_of_day` of this type. The members of `time_of_day` are accessed by using the variable name (`time_of_day`) followed by a `.` and then the member variable name:

```
time_of_day.hour=XBYTE[HOURS];  
time_of_day.days=XBYTE[DAYS];  
time_of_day.min=time_of_day.sec;  
curdays=time_of_day.days;
```

The members of the structure are treated as any other variable with the exception that they must have the parent name of the structure in front of them. The nice thing about structures is that you can create as many of them as you need, since the compiler treats it as a new type. For example, you could have the following definition later in your code:

```
struct time_str oldtime, newtime;
```

This creates two new structures called “oldtime” and “newtime.” These new structures are independent of any of the other instances of the type “struct time_str” just like multiple copies of an “int” variable are.

Structures of the same type can be copied easily using the C assignment operator:

```
oldtime=time_of_day;
```

This makes the code very easy to read and saves you from typing several lines of code to copy the four variables. Of course, individual members of a structure can be copied to another structure simply by using the assignment operator:

```
oldtime.hour=newtime.hour;  
oldtime.days=newtime.days-1;
```

In Keil C (and most other C compilers) the structure is implemented as a contiguous block of memory and the member names serve as indices into that block for the compiler. Thus, the `time_str` would be implemented as a block of memory consisting of five bytes.

The order in which the members are declared in the structure is the order in which they are placed in the block of memory. Therefore, an instance of `time_str` would have the map shown in Table 0-1.

Offset	Member	Bytes
0	hour	1
1	min	1
2	sec	1
3	days	2

Table 0-1

Once you have created a structure type, it can be treated just like any other C variable type. For example, you can have an array of structures, a structure as a member of another structure and pointers to structures.

Unions

A C union is very similar to a structure in that it is a collection of related variables, each of which is a member of the union. However, the union can only hold one of the members at a time. The members of a union can be of any valid type in the program. This includes all built in C types such as `char`, `int` or `float` as well as user defined types such as structures and other unions. An example of a definition of a union is shown below.

```
union time_type {
    unsigned long secs_in_year;
    struct time_str time;
} mytime;
```

In this case a long is defined to hold the number of seconds since the start of the year and as an alternative format of determining how far into the current year time has gone is the `time_str` from the above discussion.

Any member field of the union can be accessed at any time no matter what the contents of the union are. As an illustration, consider the following code:

```
mytime.secs_in_year=JUNE1ST;
mytime.time.hour=5;
curdays=mytime.time.days;
```

While the data may not make sense to be used in the format you're requesting of the union, C lets you perform the access anyway.

A union is implemented as a contiguous block of memory just as a structure was. However, the block of memory is only as large as the largest member of the union. Thus, the above union has the following memory map.

Offset	Member	Bytes
0	secs_in_year	4
0	mytime	5

Table 0-2

Since the largest member (`mytime`) is allocated a total of five bytes, the structure size becomes five bytes. When the union holds the `secs_in_year`, the fifth byte is unused.

Oftentimes, a union is used to provide a program with differing views of the same data. For example, suppose you had a variable defined as an unsigned long which really held the value of four hardware registers. You could give your program two simple views of this data (on a per byte basis and an all-at-once basis) by combining an array of bytes and an unsigned long in a union.

```
union status_type {
    unsigned char status[4];
    unsigned long status_val;
} io_status;

io_status.status_val=0x12345678;
if (io_status.status[2] & 0x10) {
    ...
}
```

Pointers

A pointer is a variable which contains a certain memory address. Typically, the address in a pointer is the base address of another variable. Since the address of some other variable is stored in the pointer, the pointer can be used to indirectly access the variable to which it points, just like one of the 8051 registers can be used to access another address in the DATA segment or like the DPTR is used to access external memory spaces. Pointers are very convenient to use because they are easily moved from one variable to the next and thus can allow you to write very generic routines which operate on a variety of different variables.

A pointer is defined to point at objects of a certain type. For example, if you define a pointer with the long keyword, C treats the memory location being pointed at by the pointer as the base address of a variable of type long. This is not to say that the pointer can not be coerced to point at another type, it just implies that C believes there be a long at the location pointed at. Some sample pointer definitions are shown below.

```
unsigned char *my_ptr, *another_ptr;
unsigned int *int_ptr;
float *float_ptr;
time_str *time_ptr;
```

Pointers can be assigned to any variable or memory location that you have defined in your system.

```
my_ptr=&char_val;
int_ptr=&int_array[10];
time_ptr=&oldtime;
```

Pointers can be incremented and decremented to allow them to move through memory as well as being assigned to a given location. This is especially useful when a pointer is being used to pass through an array. When C increments a pointer, it adds the size of the type being pointed at to the pointer. Consider the following code as an example.

```
time_ptr=(time_str *) (0x10000L);    // set pointer to address 0
timeptr++;                          // pointer now aims at
                                   // address 5
```

Pointers can be assigned to each other just like any other variable. The object that a pointer is aimed at can be assigned to also by dereferencing the pointer.

```
time_ptr=oldtime_ptr;           // make time_ptr and
                                // oldtime_ptr point to the
                                // same thing
*int_ptr=0x4500;                // assign 0x4500 to the
                                // variable pointed at by
                                // int_ptr
```

When a pointer is used to access the members of a structure or union the dot notation is no longer used. Instead an arrow is used to reference the members of a structure or a union. However, if the pointer is dereferenced the standard structure/union syntax can be used.

```
time_ptr->days=234;
*time_ptr.hour=12;
```

One of the places in which pointers are very heavily used is dynamic data structures such as linked lists and trees. For example, suppose that you needed to create a data structure in which you could insert names and then later check to see if a given name is valid. One of the simplest ways to implement this efficiently is to use a binary search tree. You could then declare a node of the tree as follows.

```
struct bst_node {
    unsigned char name[20];           // storage for the name
    struct bst_node *left, *right;    // a pointer to the left
                                      // and right subtrees
};
```

The binary search tree can shrink and grow as needed by allocating new nodes and assigning their addresses to the left or right pointer of the correct node. The pointers greatly add to the ability to treat the binary search tree in a generic manner.

Type Definitions

A type definition (or typedef) in C is nothing more than a way to create a synonym for a given type. In other words, its a way to avoid some typing when more complex types are involved. For example, you could create a synonym for the time_str as follows.

```
typedef struct time_str {
    unsigned char hour, min, sec;
    unsigned int days;
} time_type;
```

A variable of type “time_type” can then be defined just like any other variable.

```
time_type time, *time_ptr, time_array[10];
```

Type definitions can also be used to rename standard types in C and also make definitions using standard types simpler to read and type.

```
typedef unsigned char UBYTE;
typedef char * STRPTR;
UBYTE status[4];
STRPTR name;
```

Remember, the main point behind using a typedef is to make your code easier to read and save you a little bit of typing. However, you should make sure that you do not go overboard with typedef's and make your code unreadable. Many programmers tend to use a lot of typedef's to rename unsigned char, unsigned int, etc into names that mean something to them. However, when someone else picks up this code the names chosen to do not mean a thing and this makes the code harder to deal with.

Keil C versus ANSI C

This section will present the key features of Keil C and its differences from ANSI C. Additionally, it will provide some hints for effectively using this package on the 8051.

The Keil compiler provides the user with a superset of ANSI C with a few key differences. For the most part these differences allow the user to take advantage of the architecture of the 8051. Additional differences are due to limitations of the 8051.

Data Types

Keil C has all the standard data types available in ANSI C plus a couple of specific data types which help maximize your use of the 8051's architecture. The following table shows the standard data types and the number of bytes they take on the 8051. It should be noted that integers and longs are stored with the most significant byte in the lower address (MSB first).

In addition to these standard data types the compiler supports a bit data type. A variable of type 'bit' is allocated from the bit addressable segment of internal RAM and can have a value of either one or zero. Bit scalars can be operated on in a manner similar to other data types. Their type is promoted for operations with higher data types such as char or int. Arrays of bits and pointers to bit variables are not allowed.

Data Type	Size
char / unsigned char	8 bits
int / unsigned int	16 bits
long / unsigned long	32 bits
float / double	32 bits
generic pointer	24 bits

Table 0-3

Special Function Registers

The special function registers of the 8051 are declared using the type specifier 'sfr' for an eight bit register or 'sfr16' for a 16 bit register such as DPTR. In these declarations, the name and the address of the SFR is provided in the code. The address must be greater than 80 hex. Bits of the bit addressable SFRs can be declared by using the 'sbit' type. This type cannot be applied to any SFR which is not normally bit addressable. Examples of SFR declarations are shown in Listing 0-1. For most of the 8051 family members, Keil provides a header file which defines all the SFRs and their bits. Headers for new derivatives can easily be created by using one of the existing header files as a model.

Listing 0-1

```
sfr SCON = 0x98;                // declare SCON

sbit SM0 = 0x9F;                // declare sbit members of SCON
sbit SM1 = 0x9E;
sbit SM2 = 0x9D;
sbit REN = 0x9C;
sbit TB8 = 0x9B;
sbit RB8 = 0x9A;
sbit TI = 0x99;
sbit RI = 0x98;
```

Memory Types

Keil C allows the user to specify the memory area that will be used to hold program variables. This allows for user control over the utilization of certain memory areas. The compiler recognizes the following memory areas.

Memory Area	Description
DATA	The lower 128 bytes of internal RAM. All locations can be accessed directly and within one processor cycle.
BDATA	The 16 bytes of bit addressable locations in the DATA segment.
IDATA	The upper 128 bytes of internal RAM available on devices such as the 8052. All locations in this segment must be accessed indirectly.
PDATA	The 256 bytes of external memory which are accessed by an address placed on P0. Any access to this segment takes two cycles and is done via a MOVX @Rn command.
XDATA	External memory which must be accessed via the DPTR.
CODE	Program memory which must be accessed via the DPTR.

Table 0-4

DATA Segment

The DATA segment will provide the most efficient access to user variables and for this reason should be used to hold frequently accessed data. This segment must be used sparingly because of the limited amount of space. The 128 bytes of the DATA segment hold your program variables as well as other key information such as the processor stack and the register banks. Examples of data declarations are shown in Listing 0-2.

Listing 0-2

```
unsigned char data system_status=0;
unsigned int data unit_id[2];
char data inp_string[16];
float data outp_value;
mytype data new_var;
```

You should note that an object of any basic type or any user defined type can be declared in the DATA segment as long as the size of the type does not exceed the maximum available block size in the DATA segment. Since C51 uses the default register bank for parameter passing, you will lose at least eight bytes of the DATA segment. Additionally, sufficient space must be allowed for the processor's stack. The stack size will peak when your program is at the deepest point in its function calling tree, including

any and all interrupt routines that can be active at that time. If you overflow the internal stack, you will notice that your program mysteriously restarts itself. The real problem is that the 8051 family of microcontrollers does not have any sort of hardware error reporting mechanism and thus any errors that crop up, such as stack overflow manifest themselves in odd ways.

BDATA Segment

The BDATA segment allows you declare a variable that will be placed in the bit addressable segment of DATA memory. Following such a declaration, the variable becomes bit addressable and bit variables can be declared which point directly into it. This is particularly useful for things such as status registers where use of individual bits of a variable will be necessary. Additionally, the bits of the variable can be accessed without using previously declared bit names. The following listing shows sample declarations with the bdata keyword and accesses into bits of a BDATA object.

Listing 0-3

```
unsigned char bdata status_byte;
unsigned int bdata status_word;
unsigned long bdata status_dword
sbit stat_flag = status_byte^4;

if (status_word^15) {
    ...
}

stat_flag = 1;
```

You should note that the compiler will not allow you declare a variable of type float or double to exist in the BDATA segment. If you want to access a float bit by bit, one trick that can be done is to declare a union of a float and a long and place that in the BDATA segment as follows.

Listing 0-4

```
typedef union {                // create a type for the union
    unsigned long lvalue;      // the long value in the union
                                // (32 bits)
    float fvalue;              // the float in the union
                                // (also 32 bits)
} bit_float;                   // name the type 'bit_float'

bit_float bdata myfloat;       // declare the union in bit
                                // addressable memory
sbit float_ld = myfloat^31;     // give the most significant bit
                                // a name
```

The following code compares accesses of a specific bit within a status register. As a baseline, the code to access a byte declared in DATA memory is shown and is compared to code to access the same bit in a bit addressable byte via a bit name and via a bit number. Note that the assembly code generated for an access to a bit in this variable is better than the code generated to check a bit in a status byte declared as just DATA memory. One interesting thing to keep in mind about the bit addressable variables is that if you specify a bit offset into a BDATA object in your code instead of using a

predeclared bit name, the code emitted will be worse. In the following example observe that the assembly code for 'use_bitnum_status' is larger than the code for 'use_byte_status.'

Listing 0-5

```
1      // declare a byte wide status register
2      unsigned char data byte_status=0x43;
3
4      // declare a bit addressable status register
5      unsigned char bdata bit_status=0x43;
6      // set a bit variable to use bit 3 of bit_status
7      sbit status_3=bit_status^3;
8
9      bit use_bit_status(void);
10
11     bit use_bitnum_status(void);
12
13     bit use_byte_status(void);
14
15     void main(void) {
16 1      unsigned char temp=0;
17 1      if (use_bit_status()) {    // if third bit is set
18 2          temp++;                // increment temp
19 2      }
20 1      if (use_byte_status()) {  // if third bit is set
21 2          temp++;                // increment temp again
22 2      }
23 1      if (use_bitnum_status()) { // if third bit is set
24 2          temp++;                // increment temp again
25 2      }
26 1  }
27
28     bit use_bit_status(void) {
29 1      return (bit) (status_3);
30 1  }
31
32     bit use_bitnum_status(void) {
33 1      return (bit) (bit_status^3);
34 1  }
35
36     bit use_byte_status(void) {
37 1      return byte_status&0x04;
38 1  }
39
```

THE FINAL WORD ON THE 8051

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```
                ; FUNCTION main (BEGIN)
                                ; SOURCE LINE # 15
                                ; SOURCE LINE # 16
0000 E4          CLR      A
0001 F500      R      MOV      temp,A
                                ; SOURCE LINE # 17
0003 120000    R      LCALL     use_bit_status
0006 5002          JNC      ?C0001
                                ; SOURCE LINE # 18
0008 0500      R      INC      temp
                                ; SOURCE LINE # 19
000A          ?C0001:
                                ; SOURCE LINE # 20
000A 120000    R      LCALL     use_byte_status
000D 5002          JNC      ?C0002
                                ; SOURCE LINE # 21
000F 0500      R      INC      temp
                                ; SOURCE LINE # 22
0011          ?C0002:
                                ; SOURCE LINE # 23
0011 120000    R      LCALL     use_bitnum_status
0014 5002          JNC      ?C0004
                                ; SOURCE LINE # 24
0016 0500      R      INC      temp
                                ; SOURCE LINE # 25
                                ; SOURCE LINE # 26
0018          ?C0004:
0018 22          RET
                ; FUNCTION main (END)

                ; FUNCTION use_bit_status (BEGIN)
                                ; SOURCE LINE # 28
                                ; SOURCE LINE # 29
0000 A200      R      MOV      C,status_3
                                ; SOURCE LINE # 30
0002          ?C0005:
0002 22          RET
                ; FUNCTION use_bit_status (END)

                ; FUNCTION use_bitnum_status (BEGIN)
```

The compiler obtains the desired bit by using the entire byte instead of using a bit address.

```
                                ; SOURCE LINE # 32
                                ; SOURCE LINE # 33
0000 E500    R    MOV    A,bit_status
0002 6403            XRL    A,#03H
0004 24FF            ADD    A,#0FFH
                                ; SOURCE LINE # 34
0006        ?C0006:
0006 22            RET
                                ; FUNCTION use_bitnum_status (END)

                                ; FUNCTION use_byte_status (BEGIN)
                                ; SOURCE LINE # 36
                                ; SOURCE LINE # 37
0000 E500    R    MOV    A,byte_status
0002 A2E2            MOV    C,ACC.2
                                ; SOURCE LINE # 38
0004        ?C0007:
0004 22            RET
                                ; FUNCTION use_byte_status (END)
```

You should bear this example in mind when you are dealing with the bit addressable variables. Declare bit names into the BDATA object you wish to use instead of accessing them by number.

IDATA Segment

The IDATA segment is the next most popular segment for frequently used variables since it is accessed by using a register as the pointer. Setting an eight bit address in a register and then doing an indirect move is more attractive in terms of processor cycles and code size when compared with doing any sort of access to external memory.

```
unsigned char idata system_status=0;
unsigned int idata unit_id[2];
char idata inp_string[16];
float idata outp_value;
```

PDATA and XDATA Segments

Declarations of variables in either of these two segments follows the same syntax as the other memory segments did. You will be limited to 256 bytes of allocation in the PDATA segment, but you will not reach the limits of the XDATA segment until you have declared 65536 bytes worth of variables! Some sample declarations are shown below.

```
unsigned char xdata system_status=0;
unsigned int pdata unit_id[2];
char xdata inp_string[16];
float pdata outp_value;
```

The PDATA and XDATA segments provide similar performance. If you can use PDATA accesses to external data, do it because the setup of the eight bit address is shorter than the setup of a sixteen bit address required for variables declared to be xdata. Both accesses will be implemented using a MOVX op code which will consume two processor cycles.

Listing 0-6 shows some sample code in which an access to PDATA is compared to an access to XDATA.

Listing 0-6

```
1      #include <reg51.h>
2
3      unsigned char pdata inp_reg1;
4
5      unsigned char xdata inp_reg2;
6
7      void main(void) {
8  1      inp_reg1=P1;
9  1      inp_reg2=P3;
10 1      }

ASSEMBLY LISTING OF GENERATED OBJECT CODE

      ; FUNCTION main (BEGIN)

                                   ; SOURCE LINE # 7
                                   ; SOURCE LINE # 8

Note that the assignment 'inp_reg1=P1' takes 4 processor cycles

0000 7800    R    MOV     R0,#inp_reg1
0002 E590          MOV     A,P1
0004 F2          MOVX    @R0,A
                                   ; SOURCE LINE # 9

Note that the assignment 'inp_reg2=P3' takes 5 processor cycles

0005 900000    R    MOV     DPTR,#inp_reg2
0008 E5B0          MOV     A,P3
000A F0          MOVX    @DPTR,A
                                   ; SOURCE LINE # 10
000B 22          RET

      ; FUNCTION main (END)
```

Oftentimes, the external memory segment will contain a combination of variables and input/output devices. Accesses to I/O devices can be done by casting addresses into void pointers or using macros provided by the C51 package. I prefer to use the macros provided for memory accesses because they are easier to read. These macros make any memory segment look as if it is an array of type char or int. Some sample absolute memory accesses are shown below.

Listing 0-7

```
inp_byte=XBYTE[0x8500];           // read a byte from address 8500H
inp_word=XWORD[0x4000];           // read a word from address 2000H
                                   // and 2001H
c=((char xdata *) 0x0000);        // read a byte from address
                                   // 0000H
XBYTE[0x7500]=out_val;           // write out_val to address 7500H
```

Absolute accesses as shown above can take place to and from any memory segment other than the BDATA and BIT segments. The macros are be defined in the system include file "absacc.h" into your program.

CODE Segment

The CODE segment should only be used for data which will not change, since the 8051 does not have the capability to write to the CODE segment. Typically the CODE segment is used for lookup tables, jump vectors, and state tables. An access into this segment will take a comparable amount of time to an XDATA access. Objects declared in the CODE segment must be initialized at compile time, otherwise they will not have the value you desire when you go to use them. Examples of CODE declarations are shown below.

```
unsigned int code unit_id[2]=1234;
unsigned char hex2bcd[16]={
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15
};
```

Pointers

C51 implements generic memory pointers as a structure of three bytes. The first of these bytes is a selector which indicates the memory space the pointer refers to. The remaining two bytes of the pointer hold a sixteen bit offset into the memory space. In cases such as the DATA, IDATA, and PDATA segments, only eight address bits are needed and thus part of the pointer is doing nothing but taking up space.

Pointer Type	Size
generic pointer	3 bytes
XDATA pointer	2 bytes
CODE pointer	2 bytes
DATA pointer	1 byte
IDATA pointer	1 byte
PDATA pointer	1 byte

Table 0-5

To enhance the pointer support provided by C51, Keil allows the user to specify which memory segment a given pointer will deal with. Such a pointer is called a memory specific pointer. Part of the advantage to a memory specific pointer is the reduced amount of storage required (see Table 0-5). Additionally, the compiler does not have to generate code to use the selector and determine the correct op code for memory access. This will make your code that much smaller and more efficient. The limitation to this, of course, is that you must guarantee that a memory specific pointer will never be used to access a space other than the one declared. Such an access will fail, and may prove to be very difficult to debug.

The following example demonstrates the efficiency gained by using a memory specific pointer to move through a string rather than using a generic pointer. The first while loop using the generic pointer takes a total of 378 processor cycles as compared to a total of 151 processor cycles for the second while loop which uses a memory specific pointer.

Listing 0-8

```

1      #include <absacc.h>
2
3      char *generic_ptr;
4
5      char data *xd_ptr;
6
7      char mystring[]="Test output";
8
9      main() {
10  1      generic_ptr=mystring;
11  1      while (*generic_ptr) {
12  2          XBYTE[0x0000]=*generic_ptr;
13  2          generic_ptr++;
14  2      }
15  1
16  1      xd_ptr=mystring;
17  1      while (*xd_ptr) {
18  2          XBYTE[0x0000]=*xd_ptr;
19  2          xd_ptr++;
20  2      }
21  1      }

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION main (BEGIN)
; SOURCE LINE # 9
; SOURCE LINE # 10
0000 750004 R    MOV    generic_ptr,#04H
0003 750000 R    MOV    generic_ptr+01H,#HIGH mystring
0006 750000 R    MOV    generic_ptr+02H,#LOW mystring
0009      ?C0001:
; SOURCE LINE # 11
0009 AB00 R    MOV    R3,generic_ptr
000B AA00 R    MOV    R2,generic_ptr+01H
000D A900 R    MOV    R1,generic_ptr+02H
000F 120000 E    LCALL   ?C_CLDPTR
0012 FF      MOV    R7,A
0013 6011      JZ     ?C0002
; SOURCE LINE # 12
0015 900000      MOV    DPTR,#00H
0018 F0      MOVX   @DPTR,A
; SOURCE LINE # 13
0019 7401      MOV    A,#01H
001B 2500 R    ADD    A,generic_ptr+02H

```

```

001D F500    R    MOV    generic_ptr+02H,A
001F E4      CLR    A
0020 3500    R    ADDC    A,generic_ptr+01H
0022 F500    R    MOV    generic_ptr+01H,A
                                ; SOURCE LINE # 14
0024 80E3      SJMP    ?C0001
0026      ?C0002:
                                ; SOURCE LINE # 16
0026 750000    R    MOV    xd_ptr,#LOW mystring
0029      ?C0003:
                                ; SOURCE LINE # 17
0029 A800    R    MOV    R0,xd_ptr
002B E6      MOV    A,@R0
002C FF      MOV    R7,A
002D 6008      JZ     ?C0005
                                ; SOURCE LINE # 18
002F 900000      MOV    DPTR,#00H
0032 F0      MOVX    @DPTR,A
                                ; SOURCE LINE # 19
0033 0500    R    INC    xd_ptr
                                ; SOURCE LINE # 20
0035 80F2      SJMP    ?C0003
                                ; SOURCE LINE # 21
0037      ?C0005:
0037 22      RET
                                ; FUNCTION main (END)

```

Anytime I can get a 2:1 improvement in execution time by being a little more careful with how I use my pointers, I will be sure to take advantage of it.

Interrupt Routines

Most 8051 projects are interrupt driven which places emphasis on the interrupt service routines. The C51 compiler allows you to declare and code interrupt routines completely in C (as well as using assembly if you desire...more on that later). An interrupt procedure is declared by using the 'interrupt' keyword with the interrupt number (0 to 31) in the function declaration. The interrupt numbers indicate to the compiler where in the interrupt vector the ISRs address belongs. The numbers directly correspond to the enable bit number of the source in the IE SFR. In other words, bit 0 of the IE register enables external interrupt zero. Accordingly, the interrupt number for external interrupt zero is 0. Table 0-6 illustrates the correlation between the IE bits and the interrupt numbers.

IE Bit Number and C Interrupt Number	Interrupt Source
0	External Interrupt 0
1	Timer 0 Overflow
2	External Interrupt 1
3	Timer 1 Overflow
4	Serial Port Interrupt
5	Timer 2 Overflow

Table 0-6

An interrupt routine must not take any parameters, and can have no return value. Given these constraints, the compiler does not have to worry about use of the register bank for parameters and writes your interrupt routine to push the accumulator, the processor status word, the B register, the data pointer and the default registers onto the stack only if they are used in the ISR. At the end of the routine, the compiler pops whatever registers it pushed in the processor stack at the start and inserts a RETI op code just as you would do in assembly. The address of the ISR is placed in the interrupt vector by the compiler. The C51 supports all five standard 8051/8052 interrupts (which are numbered 0 to 4), as well as up to 27 more interrupt sources which may be used on later 8051 derivatives. A sample ISR is shown below.

Listing 0-9

```
1      #include <reg51.h>
2      #include <stdio.h>
3
4      #define RELOADVALH  0x3C
5      #define RELOADVALL  0xB0
6
7      extern unsigned int tick_count;
8
9      void timer0(void) interrupt 1 {
10     1      TR0=0;          // stop T0 while it is reloaded
11     1      TH0=RELOADVALH; // set T0 to overflow in 50ms
12     1      TL0=RELOADVALL; // given a 12MHz clock
13     1      TR0=1;          // restart T0
14     1      tick_count++;    // increment a time counter
15     1      printf("tick_count=%05u\n", tick_count);
16     1      }
```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```
          ; FUNCTION timer0 (BEGIN)
0000 C0E0      PUSH    ACC
0002 C0F0      PUSH    B
0004 C083      PUSH    DPH
0006 C082      PUSH    DPL
0008 C0D0      PUSH    PSW
000A C000      PUSH    AR0
000C C001      PUSH    AR1
000E C002      PUSH    AR2
0010 C003      PUSH    AR3
0012 C004      PUSH    AR4
0014 C005      PUSH    AR5
0016 C006      PUSH    AR6
0018 C007      PUSH    AR7
                                ; SOURCE LINE # 9
                                ; SOURCE LINE # 10
001A C28C      CLR     TR0
                                ; SOURCE LINE # 11
001C 758C3C    MOV     TH0,#03CH
                                ; SOURCE LINE # 12
001F 758AB0    MOV     TL0,#0B0H
                                ; SOURCE LINE # 13
0022 D28C      SETB    TR0
                                ; SOURCE LINE # 14
```

THE FINAL WORD ON THE 8051

```
0024 900000 E    MOV    DPTR,#tick_count+01H
0027 E0          MOVX   A,@DPTR
0028 04          INC    A
0029 F0          MOVX   @DPTR,A
002A 7006        JNZ    ?C0002
002C 900000 E    MOV    DPTR,#tick_count
002F E0          MOVX   A,@DPTR
0030 04          INC    A
0031 F0          MOVX   @DPTR,A
0032           ?C0002:
                                ; SOURCE LINE # 15
0032 7B05        MOV    R3,#05H
0034 7A00 R      MOV    R2,#HIGH ?SC_0
0036 7900 R      MOV    R1,#LOW ?SC_0
0038 900000 E    MOV    DPTR,#tick_count
003B E0          MOVX   A,@DPTR
003C FF          MOV    R7,A
003D A3          INC    DPTR
003E E0          MOVX   A,@DPTR
003F 900000 E    MOV    DPTR,#?_printf?BYTE+03H
0042 CF          XCH    A,R7
0043 F0          MOVX   @DPTR,A
0044 A3          INC    DPTR
0045 EF          MOV    A,R7
0046 F0          MOVX   @DPTR,A
0047 120000 E    LCALL   _printf
                                ; SOURCE LINE # 16
004A D007        POP    AR7
004C D006        POP    AR6
004E D005        POP    AR5
0050 D004        POP    AR4
0052 D003        POP    AR3
0054 D002        POP    AR2
0056 D001        POP    AR1
0058 D000        POP    AR0
005A D0D0        POP    PSW
005C D082        POP    DPL
005E D083        POP    DPH
0060 D0F0        POP    B
0062 D0E0        POP    ACC
0064 32          RETI
                                ; FUNCTION timer0 (END)
```

In the above example, the call to 'printf' forces the compiler to save all the registers since the call itself uses the register bank and the non-reentrant function 'printf' uses several of the other registers. If the function is rewritten to remove the call to 'printf' the code emitted for the ISR is reduced because less registers will be saved to the stack.

Listing 0-10

```
1      #include <reg51.h>
2
3      #define RELOADVALH  0x3C
4      #define RELOADVALL  0xB0
5
6      extern unsigned int tick_count;
7
8      void timer0(void) interrupt 1 using 0 {
9  1      TR0=0;           // stop T0 while it is reloaded
10  1      TH0=RELOADVALH;  // set T0 to overflow in 50ms
11  1      TL0=RELOADVALL;  // given a 12MHz clock
12  1      TR0=1;           // restart T0
13  1      tick_count++;    // increment a time counter
14  1      }

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION timer0 (BEGIN)
0000 C0E0      PUSH      ACC

Push and pop of register bank 0 and the B register is eliminated because printf was using  
the registers for parameters and using B internally.

0002 C083      PUSH      DPH
0004 C082      PUSH      DPL

; SOURCE LINE # 8
; SOURCE LINE # 9
0006 C28C      CLR       TR0

; SOURCE LINE # 10
0008 758C3C    MOV       TH0,#03CH

; SOURCE LINE # 11
000B 758AB0    MOV       TL0,#0B0H

; SOURCE LINE # 12
000E D28C      SETB      TR0

; SOURCE LINE # 13
0010 900000    E        MOV       DPTR,#tick_count+01H
0013 E0        MOVX      A,@DPTR
0014 04        INC       A
0015 F0        MOVX      @DPTR,A

```

```
0016 7006      JNZ      ?C0002
0018 900000    E      MOV      DPTR,#tick_count
001B E0        MOVX     A,@DPTR
001C 04        INC      A
001D F0        MOVX     @DPTR,A
001E          ?C0002:
                                ; SOURCE LINE # 14
001E D082      POP      DPL
0020 D083      POP      DPH
0022 D0E0      POP      ACC
0024 32        RETI
                ; FUNCTION timer0 (END)
```

Specifying the ISR Register Bank

Some of the overhead associated with pushing and popping the SFRs saved by C51 can be avoided by declaring that the ISR will use a given register bank. This is done with the 'using' keyword and a number from 0 to 3 to indicate the register bank desired. When a register bank is specified the default register bank will not be pushed onto the stack. This will result in a savings of 32 cycles in the ISR since each push takes two processor cycles and each pop takes two cycles. The drawback to specifying a register bank for your ISR is that any function called by the ISR must use this same register bank, or erroneous parameters and return values will result. The listing below shows the sample ISR for timer 0, but this time I have told the compiler that this ISR will use register bank 0.

Listing 0-11

```
1      #include <reg51.h>
2      #include <stdio.h>
3
4      #define RELOADVALH  0x3C
5      #define RELOADVALL  0xB0
6
7      extern unsigned int tick_count;
8
9      void timer0(void) interrupt 1 using 0 {
10  1      TR0=0;           // stop T0 while it is reloaded
11  1      TH0=RELOADVALH;  // set T0 to overflow in 50ms
12  1      TL0=RELOADVALL;  // given a 12MHz clock
13  1      TR0=1;           // restart T0
14  1      tick_count++;    // increment a time counter
15  1      printf("tick_count=%05u\n", tick_count);
16  1      }
```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

 ; FUNCTION timer0 (BEGIN)

0000 C0E0 PUSH ACC

0002 C0F0 PUSH B

Push and pop of register bank 0 has been eliminated because the compiler assumes that this ISR 'owns' RB0.

0004 C083 PUSH DPH

0006 C082 PUSH DPL

0008 C0D0 PUSH PSW

000A 75D000 MOV PSW,#00H

 ; SOURCE LINE # 9

 ; SOURCE LINE # 10

000D C28C CLR TR0

 ; SOURCE LINE # 11

000F 758C3C MOV TH0,#03CH

 ; SOURCE LINE # 12

0012 758AB0 MOV TL0,#0B0H

 ; SOURCE LINE # 13

0015 D28C SETB TR0

 ; SOURCE LINE # 14

0017 900000 E MOV DPTR,#tick_count+01H

001A E0 MOVX A,@DPTR

001B 04 INC A

```

001C F0          MOVX    @DPTR,A
001D 7006        JNZ     ?C0002
001F 900000    E      MOV     DPTR,#tick_count
0022 E0          MOVX    A,@DPTR
0023 04          INC     A
0024 F0          MOVX    @DPTR,A
0025            ?C0002:
                                ; SOURCE LINE # 15
0025 7B05        MOV     R3,#05H
0027 7A00    R    MOV     R2,#HIGH ?SC_0
0029 7900    R    MOV     R1,#LOW ?SC_0
002B 900000    E      MOV     DPTR,#tick_count
002E E0          MOVX    A,@DPTR
002F FF          MOV     R7,A
0030 A3          INC     DPTR
0031 E0          MOVX    A,@DPTR
0032 900000    E      MOV     DPTR,#?_printf?BYTE+03H
0035 CF          XCH     A,R7
0036 F0          MOVX    @DPTR,A
0037 A3          INC     DPTR
0038 EF          MOV     A,R7
0039 F0          MOVX    @DPTR,A
003A 120000    E      LCALL   _printf
                                ; SOURCE LINE # 16
003D D0D0        POP     PSW
003F D082        POP     DPL
0041 D083        POP     DPH
0043 D0F0        POP     B
0045 D0E0        POP     ACC
0047 32          RETI
                                ; FUNCTION timer0 (END)

```

Reentrant Functions

Due to the limited amount of internal stack space available to the 8051, the C51 package does not implement a calling stack in the sense that we are used to on larger systems. Normally, each function call in a C program causes a "frame" to be pushed onto a stack which contains the function's parameters and allocation for the locals of the current invocation of the function. For efficiency's sake, the C51 compiler does not provide such a stack. Instead, a "compiled stack" is used. In this block of memory, a function is given space for its local variables and the function is coded by the compiler to assume that its locals are at a fixed address in the compiled stack. Thus, recursive calls to a function will cause the data in the local variables to be corrupted.

In certain real time applications, non-reentrancy is not acceptable because a function call may be interrupted by an ISR which in turn calls that same function. For cases like this, C51 allows a function to be declared as reentrant. A reentrant function has a stack implemented in the classic sense. It can be called recursively or called by multiple processes without fear of locals being clobbered because a separate copy of the locals exists for each function invocation. Because the stack is simulated, reentrant functions are often larger and slower than their normal counterparts. Thus, you should use them with

care. The simulated stack also requires you to eliminate any bit parameters, locals and return values from reentrant functions.

Things to do and things to avoid with Keil C

The Keil compiler has the capability to take your C source code and produce highly optimized object code from it. However, there are things that you, as the designer, can do to help the compiler generate better code. This section will discuss some of these tips and tricks to help you get the most of using C on the 8051.

Downsizing Your Variables

One of the most basic things you can do to improve your code is pay careful attention to the size of your variables. For anyone used to coding in C on a machine such as a mainframe or a PC, it is very commonplace to declare things like loop counters as integers, even for variables whose values will never exceed 255. On an eight bit machine like the 8051, wide use of data types whose size is greater than eight bits will be a large waste of processing power and memory. You must carefully consider the potential range of values for any variable that you declare, and then choose the smallest type that will meet these needs. Obviously, the most preferred type for variables will be unsigned char since it only uses one byte.

Use Unsigned Types

At this point, you may be wondering why I specified the preferred type above to be unsigned char instead of char. The reasoning behind this is that the 8051 does not support signed arithmetic, and the extra code required by a signed value as opposed to an unsigned value will take away from your overall processor resources. Thus, in addition to choosing variable types according to range of value, you must also consider if the variable will be used for any operation which will require negative numbers. If not, then make sure that you specify that the variable is to be unsigned. If you can eliminate negative numbers from a function or entirely from your application, your project will be that much leaner for it.

Stay Away from Floating Point

Along the same vein, you should avoid floating point math like the plague. Performing floating point operations on 32 bit values with an eight bit controller is like mowing your lawn with a pair of hedge clippers. You may be able to do it, but it will waste a hell of a lot of time. Any time you are using floating point in an application you should ask yourself if it is absolutely necessary. Many times, floating point numbers can be eliminated by promoting all the values by a couple of orders of magnitude and using integer arithmetic. You will be much better off dealing with ints and longs than you will be with doubles and floats. Your code will execute faster, and the floating point routines will not be linked into your application. Additionally, if you must use floating point, you should consider using an 8051 derivative which is optimized for math operations such as the Siemens 80517, 80537 or the Dallas Semiconductor 80320.

There may be times when you are forced to incorporate the use of floating point numbers into your system. You have already read of the disadvantages in code size and speed you face. Additionally, you should be aware that if you use floating point math in a routine which can be interrupted you must either ensure that the interrupting routine does not use floating point math anywhere in its calling tree, or save the state of the floating point system at the beginning of the ISR using 'fpsave' put it back to its original state at the end of the ISR using 'fprestore'. Another approach is to wrap your calls to floating point routines such as sin() with a function which disables interrupts before the call to the math function and reenables them after the call.

Listing 0-12

```
#include <math.h>

void timer0_isr(void) interrupt 1 {
    struct FPBUF fpstate;
    ...                               // any initialization code or
                                     // non-floating point code
    fpsave(&fpstate);                 // save floating point system
                                     // state
    ...                               // any ISR code, including ALL
                                     // floating point code
    fprestore(&fpstate);               // restore floating point
                                     // system state
    ...                               // any non-floating point
                                     // ISR code
}

float my_sin(float arg) {
    float retval;
    bit old_ea;
    old_ea=EA;                        // save current interrupt state
    EA=0;                             // kill interrupts
    retval=sin(arg);                  // make the floating point call
    EA=old_ea;                        // put the interrupt state back
    return retval;
}
```

If you must use floating point in your program, then you should go to every effort to determine the maximum precision that you need. Once you have computed the maximum number of bits of precision needed in your floating point computations, enter the number in the workbench compiler options dialog box under the “Bits to round for float compare” control. This will allow the floating point routines to limit the amount of work they do to only the degree of precision that is meaningful to your system.

Make Use of bit Variables

When you are using flags which will only contain a one or a zero, use the bit type instead of an unsigned char. This will help make your memory reserves go farther, since you will not be wasting seven bits. Additionally, bit variables are always in internal RAM and therefore will be accessed in one cycle.

Use Locals instead of Globals

Variables which are declared to be global data will be less efficient than use of local variables. The reason for this is that the compiler will attempt to assign any local variables to internal registers. In comparison, global data may or may not be in internal data depending on your declaration. In cases where the globals are assigned to XDATA by default (such as large memory model programs) you have given up speedy access. Another reason to avoid globals is that you have to coordinate access to such variables between processes in your system. This will become a problem in interrupt systems or multitasking systems where it is possible that more than one process will be attempting to use the global. If you can avoid the globals and use locals, the compiler can manage that most efficiently for you.

Use Internal Memory for Variables

Globals and locals can be forced into any memory area you wish. Given the previous discussions of the tradeoffs between the segments, it should be apparent that you can optimize the speed of your program by placing the most frequently accessed variables in internal RAM. Additionally, your code will be smaller since it takes less instructions and setup to access variables in internal RAM than it does to access variables in external RAM. In terms of speeding up your program, you will want to fill the memory segments in the following order: DATA, IDATA, PDATA, XDATA. Again, be careful to leave enough space in the DATA segment for the processor's internal stack.

Use Memory Specific Pointers

If your program is using pointers for any sort of operations, you may want to examine their usage and confine them to a specific memory area such as the XDATA or CODE space. If you can do this, you can then use memory specific pointers. As discussed previously, the memory specific pointers will not require a selector and code which uses them will be tighter because the compiler can write it to assume a reference to a given memory segment rather than having to determine to which segment the pointer aims.

Use Intrinsics

For simple functions such as bit-wise rotation of variables, the compiler provides you with intrinsic functions which can be called. Many of the intrinsics correspond directly to assembler instructions while others are more involved and provide ANSI compatibility. All of the intrinsic functions are reentrant, and thus can be safely called from anywhere in your code.

For rotation operations on a single byte the intrinsic functions `_crol_` (rotate left) and `_cror_` (rotate right) directly correspond to the assembly instructions 'RL A' and 'RR A.' If you wish to perform a bit-wise rotation on larger objects such as an integer or a long, the intrinsic will be more complicated and will therefore take longer. These rotations can be called using the `_lrol_`, `_lror_` intrinsics for integers and `_lrol_`, `_lror_` intrinsics for longs.

The "jump and clear bit if set" (JBC) op code from Chapter 2 is also implemented as an intrinsic for use in C. It is called by `_testbit_`. This intrinsic returns true if the parameter bit was set and false otherwise. This is frequently useful for checking flags such as RI, TI, or the timer overflow flags, and results in more readable C code. The function directly translates to a JBC instruction.

Listing 0-13

```
#include <intrins.h>

void serial_intr(void) interrupt 4 {
    if (!_testbit_(TI)) {                // if this is a transmit interrupt
        P0=1;                           // toggle P0.0
        _nop_();                         // wait one cycle
        P0=0;
        ...                             // do any other TI things...
    }
    if (!_testbit_(RI)) {
        test=_crot_(SBUF, 1);           // rotate the value in SBUF
                                         // right one bit
        ...                             // do any other RI things
    }
}
```

Use Macros Instead of Functions

In addition to using intrinsics, you can also make your code more readable by implementing small operations such as a read from a latch or code to enable a certain circuit as macros. Instead of duplicating one or two lines of code all over the place, you can isolate duplicate code into a macro which will then look like a function call. The compiler will place the code inline when it emits object code and will not generate a call. The code will be much easier to read and maintain since the macro can be given a name which describes its operation. When this operation needs to change because of some system change or hardware flaw (both are inevitable in most projects), you only need to change the software in one place, rather than hunting through your code and making the same change in many places.

Listing 0-14

```
#define led_on() {\
    led_state=LED_ON;      \
    XBYTE[LED_CNTRL] = 0x01;}

#define led_off() {\
    led_state=LED_OFF;     \
    XBYTE[LED_CNTRL] = 0x00;}

#define checkvalue(val) \
    ( (val < MINVAL || val > MAXVAL) ? 0 : 1 )
```

Macros will also make code which must access many levels into structures or arrays easier to deal with. Oftentimes, you can implement complex accesses which occur frequently as macros to simplify your job of typing in the code, and later, reading and maintain the code.

Memory Models

The C51 package implements three basic memory models for storage of variables, parameter passing, and stack for reentrant functions. You should always use the small memory model. There are very few systems which require use of other memory models such as systems with large reentrant stacks. In general, a system which uses less than the amount of internal RAM available will always be compiled with the small memory model. In this model, the DATA segment is the default segment for location of any global and local variables. All parameter passing occurs in the DATA segment as well and if any functions are declared to be reentrant, the compiler will attempt to implement the invocation stack in internal RAM. The advantage to this model is that all data will be accessed as quickly as possible. However, the disadvantage to this is that you will only have 120 bytes to work with (128 bytes total space minus 8 bytes for the default register bank). You will also have to allocate some of this space to the 8051's calling stack. The space allocated must be equal to the deepest branch of your calling tree at the worst case nesting of interrupts.

If your system has 256 bytes of external RAM or less, you can use the compact memory model. By default, variables will be allocated to the PDATA segment unless you declare them to be in another memory segment. This memory model will expand the amount of RAM you can use, and will still provide improvement in access time over allocating variables out of the XDATA memory segment. Parameters to variables will be passed via internal RAM unless you have more parameters than can be passed in one register bank, and thus function invocations will still be quick. In addition to the rules which applied to the DATA segment for the small memory model, the compact memory model allows for allocation of the 256 bytes of PDATA to user variables. The PDATA segment will be accessed indirectly via R0 and R1 and thus accesses will be somewhat shorter than they are for XDATA.

In the large memory model, all variables default to the XDATA segment. Keil C will attempt to pass parameters in internal RAM using the default register bank. The rules for the number of parameters which can be passed in the register bank are the same as they are for the compact memory model. If any functions are declared as reentrant, the invocation stack will be allocated from the XDATA segment. Access to XDATA variables will be the slowest, and thus the large memory model will require good justification for use and careful analysis of variable placement during use to assure that your systems average access time to its data is optimized.

Mixed Memory Models

The Keil compiler allows you to mix and match the memory models within a single program to help you optimize your use of memory. I have found this feature to be most useful in large memory model systems. In the case of a large memory model program, I will declare functions which must deliver the utmost in speed as small memory model segments. This forces the compiler to generate code for the function which places its locals in internal RAM and guarantees that all parameters are passed via internal RAM. While this may not seem like a huge gain in performance, when one compares the size of the code generated for a function which must frequently access its locals which are in XDATA to the size of the code for the same function using locals in DATA memory, you will agree that it is worth the effort to mix the models this way.

Just as you can declare functions to be small within a program which is compiled as large, you can declare a small program's functions to be large or compact. Typically this is done with functions that are using a large amount of local storage such as a buffer or computation table. In this case, the function can be compiled as compact or large, and the locals will be allocated out of an external RAM. However, you can also compile the function using the small memory model but force the large local variables to the XDATA segment in their definition.

Run time Library

The run time routines provided with the Keil C51 package offer both high performance and small code size. You will have no problem using these routines, and in most cases will have a difficult time writing a better routine yourself. The main thing to be warned about is that some of the functions in the library are not reentrant. Thus, if you call such a function and are then interrupted by an ISR that calls the same function, the first call will invariably generate unexpected results and will be a bug that is very difficult to find. Table 0-7 shows the library functions which are not reentrant. My suggestion for dealing with these routines is to block any interrupt that can call the library routine you are about to call during its execution.

gets	atof	atan2
printf	atol	cosh
sprintf	atoi	sinh
scanf	exp	tanh
sscanf	log	calloc
memcpy	log10	free
strcat	sqrt	init_mempool
strncat	srand	malloc
strncmp	cos	realloc
strncpy	sin	ceil
strspn	tan	floor
strcspn	acos	modf
strpbrk	asin	pow
strrchr	atan	

Table 0-7

Dynamic Memory Allocation

The Keil C compiler includes the capability to perform dynamic memory allocation via the standard C functions 'malloc' and 'free'. For most applications, it is better to determine how much memory will be needed and perform as much of the allocation at compile time as is possible. However, for applications which utilize dynamic structures such as trees and linked lists, this is not usually practical. For these situations, the Keil run time routines provide the necessary support.

The dynamic allocation routines given require that the user declare a byte array to be used as the heap. Selection of the size of this array will depend upon the estimated usage of dynamic memory by the application. The array declared for the heap must reside in the XDATA segment since the library routines use memory specific pointers to access this memory. Besides, it does not make much sense to dynamically allocate memory out of the DATA segment, since there is such a small amount of memory there to work with.

Once the XDATA memory block has been declared, a pointer to this block and its size must be given to an initialization routine (init_mempool) which will set some internally owned variables and prepare the block for use as the heap as well as initializing allocatable memory. Once this function is complete, the dynamic memory allocation routines can be called as they would be in any other system. Support routines for dynamic memory allocation include malloc (which takes an unsigned integer size argument and returns a pointer to the block), calloc (which takes an unsigned integer to indicate the number of items, and an unsigned integer size argument and returns a pointer to the block), realloc (which takes a pointer to a block and an unsigned integer argument indicating the new size and returns a pointer to the object with a new size specified by the size argument), and free (which takes a pointer to block and adds it back into the heap as unallocated memory). All functions which return a pointer to a block of memory will return NULL if the allocation operation requested fails.

An example of using the dynamic memory allocation routines is given in Listing 0-15.

Listing 0-15

```
#include <stdio.h>
#include <stdlib.h>

// this code uses memory specific pointers for improved efficiency

typedef struct entry_str {           // define a queue entry type
    struct entry_str xdata *next;    // pointer to the next element
}
```

```
char text[33];                // string to place in queue
} entry;

void init_queue(void);
void insert_queue(entry xdata *);
void display_queue(entry xdata *);
void free_queue(void);
entry xdata *pop_queue(void);

entry xdata *root=NULL;      // set the queue to empty

void main(void) {
    entry xdata *newptr;

    init_queue();            // set up the queue
    ...
    newptr=malloc(sizeof(entry));    // allocate a queue entry
    sprintf(newptr->text, "entry number one");
    insert_queue(newptr);        // put it in the queue
    ...
    newptr=malloc(sizeof(entry));
    sprintf(newptr->text, "entry number two");
    insert_queue(newptr);        // insert another entry in
                                // the queue
    ...
    display_queue(root);        // traverse the queue
    ...
    newptr=pop_queue();          // pop the leading entry
    printf("%s\n", newptr->text);
    free(newptr);               // deallocate it
    ...
    free_queue();               // free the entire queue
}

void init_queue(void) {
    static unsigned char memblk[1000]; // this block of memory
                                      // will be used as the heap
    init_mempool(memblk, sizeof(memblk)); // run the routine to set
                                      // up the heap
}

void insert_queue(entry xdata *ptr) { // add an entry to the tail
entry xdata *fptr, *tptr;
    if (root==NULL) {
```

```
    root=ptr;
} else {
    fptr=tptr=root;
    while (fptr!=NULL) {
        tptr=fptr;
        fptr=fptr->next;
    }
    tptr->next=ptr;
}
ptr->next=NULL;
}

void display_queue(entry xdata *ptr) { // traverse the queue
    entry xdata *fptr;
    fptr=ptr;
    while (fptr!=NULL) {
        printf("%s\n", fptr->text);
        fptr=fptr->next;
    }
}

void free_queue(void) { // free all queue entries
    entry xdata *temp;
    while (root!=NULL) {
        temp=root;
        root=root->next;
        free(temp);
    }
}

entry xdata *pop_queue(void) { // remove the top queue entry
    entry xdata *temp;
    if (root==NULL) {
        return NULL;
    }
    temp=root;
    root=root->next;
    temp->next=NULL;
    return temp;
}
```

Note that the usage of the dynamic memory allocation routines looks just like the usage of these same routines in ANSI C. Thus, there should be no difficulty in learning to use the Keil functions in your code.

Conclusion

Use of C on the 8051 will make development of your project speedier and more enjoyable. You will find that using C does not inhibit your ability to control the hardware at a low level and does not add a high amount of unwanted overhead. If you follow good design practices and the tips given in this chapter, you will have an efficient system with a fair amount of reusable code. Additionally, you will find that your project is easier to maintain over the long haul.

- Using Software to Complement the Hardware

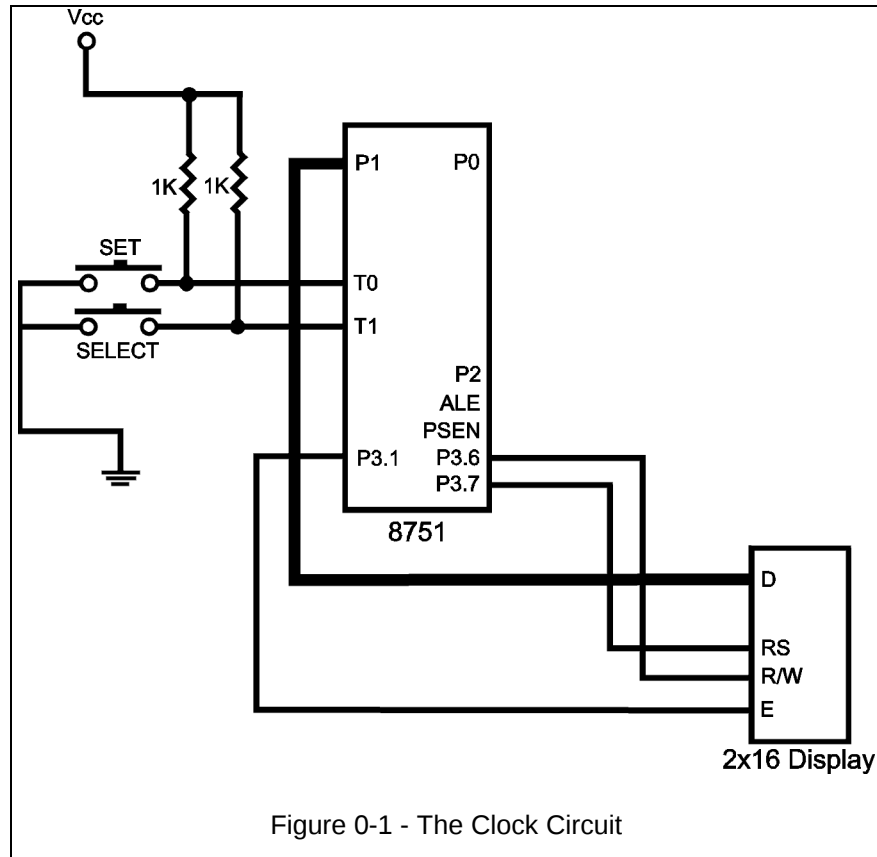
Introduction

This chapter will present several ways in which you can improve your overall system by using software techniques. Through these techniques you will see how to easily implement a user interface, time system events, and eliminate hardware components using software. A simple clock based on the 8051 processor will be developed as an example of these concepts. The clock will display the time of day using a standard 2 by 16 character LCD display which will be mapped to port one. A simple interface of two buttons will allow the user to set the time of day indicated by the clock. Pressing the first button will activate set mode and place the cursor under the current field. Pressing the second button during set mode will increment the selected field. If no buttons are pressed during set mode, the clock will return to normal mode in 15 seconds.

Since one of the goals of the clock project is to produce it as inexpensively as possible, the processor will be used to emulate a real time clock chip and the LCD panel will be directly connected to port one of the processor. By having software perform the function of the RTC and directly controlling the interface lines of the LCD panel, the need for an address decoder, and an RTC chip is eliminated. To further reduce parts count, the clock will use only internal memory, eliminating the need for an external RAM. To do this, the project must use no more than 128 bytes of RAM for the stack space and all necessary variables.

Once the software has been designed the internal RAM usage will be analyzed to ensure that the space available there will be sufficient. The system as described above is shown in Figure 0-1. This project will also use an 8051 with internal EPROM to eliminate the external EPROM and the 74373 that is used to interface to it. However, ports zero and two will be left open in case the system memory will later have to be expanded to include an external EPROM.

For comparison, the following block diagram of the system in Figure 0-2 assumes that a conventional design approach is taken to the system architecture. This design puts the display and the RTC chip in the system memory map. These changes require that an address decoder be inserted into the system as well as a NAND gate and an inverter. Additionally, an external SRAM has been designed in. Note the difference in parts count between the two block diagrams.



Using the Small Memory Model

To accomplish the elimination of the static RAM, the clock project will have to use the small memory model. It will be limited to 128 bytes of total RAM usage including the processor's internal stack, the compiled stack, and all variables used by the program and the library routines that get invoked.

Because the linker should have the ability to optimize RAM usage by overlaying the locations, as many of the variables as possible be locals. Using overlay analysis, the linker will decide which variables need to be allocated together and which ones do not exist at the same time. This analysis will tell L51 how to use the memory locations. Many times, one memory location will be used to hold different locals depending on the calling tree. Therefore, it is beneficial to have most variables be local to the function which uses them, if possible. There will naturally be a couple of variables that will have to be globals, such as some "in use" flags, and a structure which keeps track of the time of day. It is possible to make such variables static to a given function, but the compiler will implement these the same as it would a global declaration.

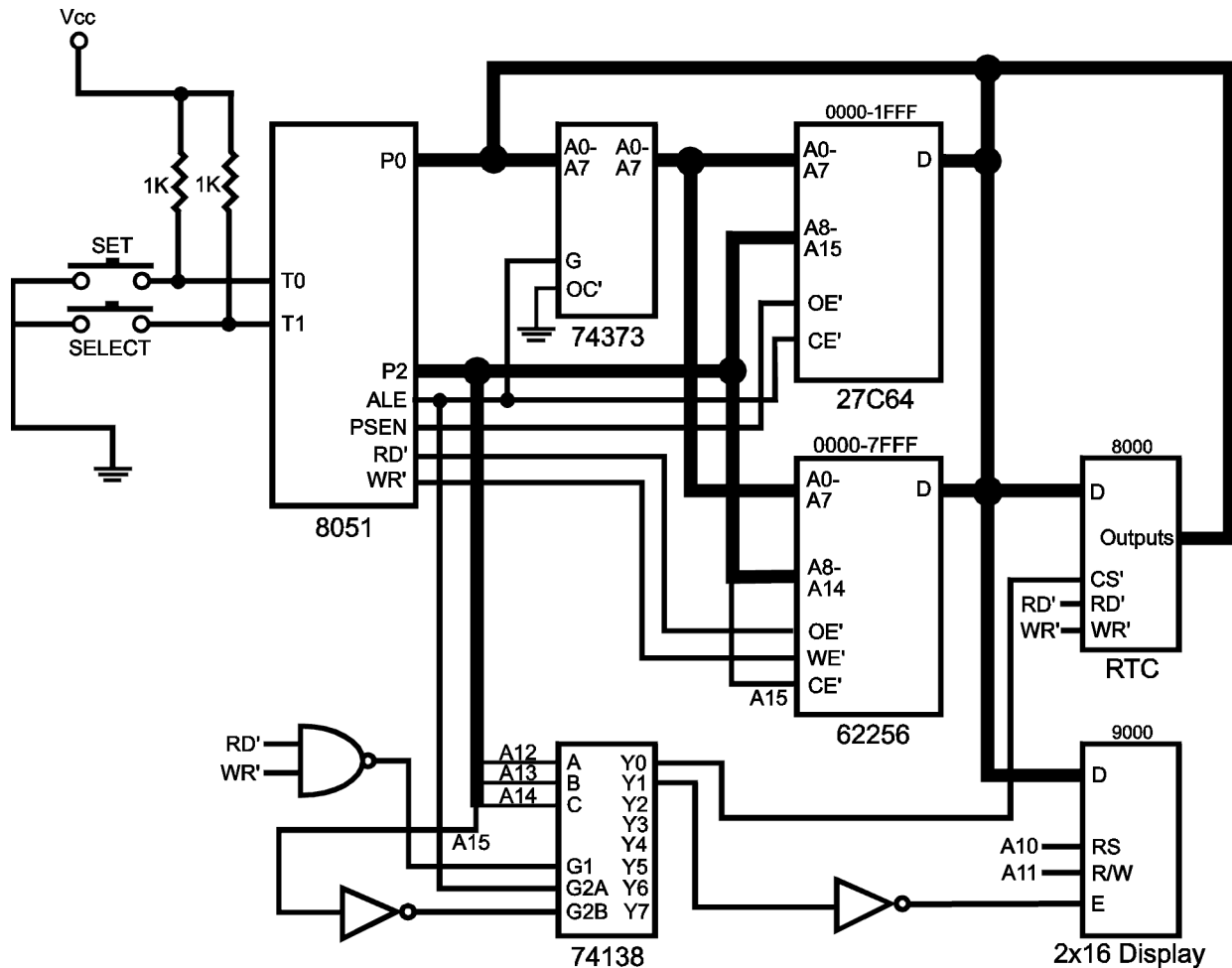


Figure 0-2 - The Extended Clock Circuit

To help minimize RAM usage, calls to the run time library routines will be kept to a minimum. Many of the larger routines use a fair amount of RAM (when one considers that the clock only has 128 bytes) and may be more general in scope and function than is required. One such function to keep a close eye on is the 'printf' function. Initially, the clock will use the 'printf' function to format and output a string for the display. The 'printf' function includes a lot of formatting capabilities that the clock won't need or use. Once the initial version of the project is completed, it will be run with the standard 'printf' and will be

analyzed to determine if it will be worth it to write a complete replacement for this function. Initially, it will be a lot simpler to use printf and hope that it does not consume too many system resources.

Instruction	Description	RS	R/ W	D7	D6	D5	D4	D3	D2	D1	D0
Clear display	Clears entire display and sets DD RAM address 0 in address counter	0	0	0	0	0	0	0	0	0	1
Return home	Sets DD RAM address 0 in address counter. Returns display from shifted to original position. DD RAM contents remain unchanged.	0	0	0	0	0	0	0	0	1	*
Entry mode set	Sets cursor move direction (I/D) and specifies shift of display (S). These operations are performed during data write and data read.	0	0	0	0	0	0	0	1	I/D	S
Display on/off	Sets on/off of entire display (D), cursor on/off (C), and blink of cursor position character (B).	0	0	0	0	0	0	1	D	C	B
Cursor/display shift	Moves cursor and shifts display without changing DD RAM contents.	0	0	0	0	0	1	S/C	R/L	*	*
Function set	Sets interface data length (DL), number of display lines (N), and character font (F).	0	0	0	0	1	DL	N	F	*	*
Set CG RAM address	Sets CG RAM address. CG RAM data are sent and received after this setting.	0	0	0	1	ACG					
Set DD RAM address	Sets DD RAM address. DD RAM data are sent and received after this setting.	0	0	1	ADD						
Read busy flag and address	Reads busy flag (BF) indicating internal operation and reads address counter contents.	0	1	BF	AC						
Write data to CG of DD RAM	Writes data into DD RAM or CG RAM.	1	0	Write data							
Read data from CG or DD RAM	Reads data from DD RAM or CG RAM.	1	1	Read data							
I/D: 1 = Increment, 0 = Decrement S: 1 = Accompanies display shift S/C: 1 = Display shift, 0 = Cursor move R/L: 1 = Shift to the right, 0 = Shift to the left DL: 1 = 8 bits, 0 =4 bits N: 1 = 2 lines, 0 = 1 line F: 1 = 5 x 10 dots, 0 = 5 x 7 dots BF: 1 = Internally operating, 0 = Ready for next instruction											

Table 0-1

Using the LCD panel

The LCD panel chosen for the clock project is a Stanley GMD16202 display. It has 2 rows of 16 characters each. The interface to this display is simple; it has a small set of commands as shown in Table 0-1. Upon power on, the display must be put through an initialization routine in which it is told the width of its data bus, the number of lines it has, its entry mode, etc. In between each command, the software must sample the display's status register so it can determine when the display is ready for the next command. The display typically takes about 40 microseconds to execute each command, although some may take as long as 1.64 milliseconds to execute.

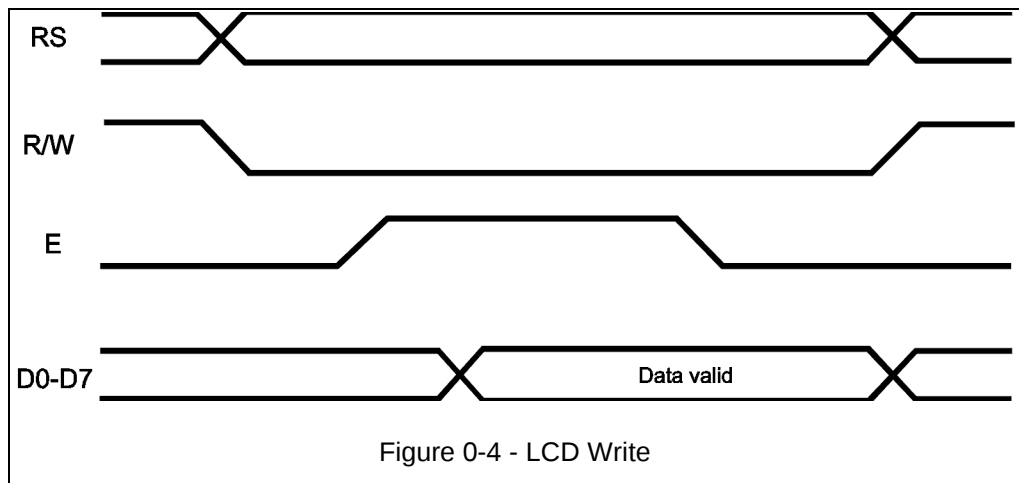
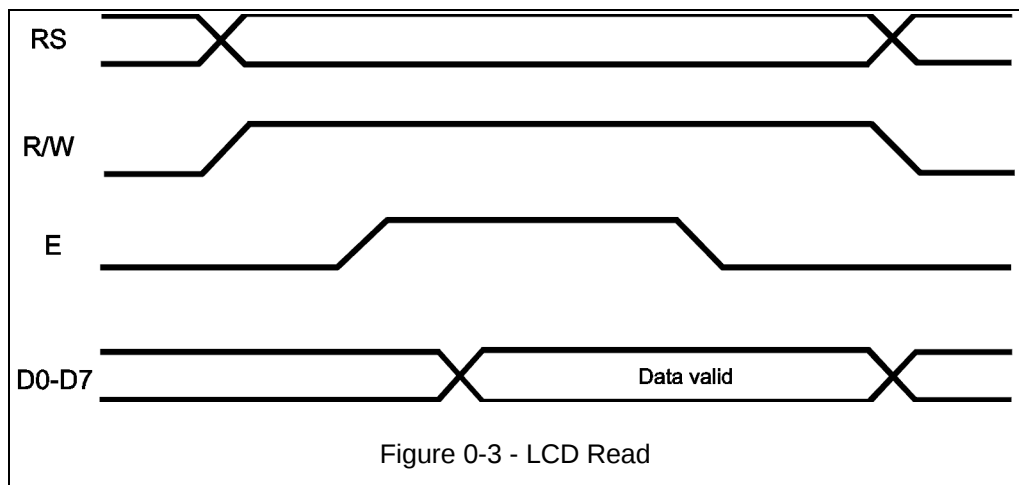
Interfacing to the LCD Panel

One of the main design goals of the clock project is to eliminate as many parts as possible in an effort to reduce overall cost of the project. This approach is strongly evident in the interface to the LCD panel. In this case, the panel has an 8 bit 'bus' set up between itself and port 1 of the 8051. The software will be responsible for correctly controlling the display and generating the correct sequence of enable signals to latch data in and out of the display. In a typical system, the LCD panel would be interfaced via the 8051's bus and the software would have to do nothing but an `XBYTE[]` call to access the display. However, by shifting some of the work into the software, the clock will eliminate an address decoder and several support parts while trading off system speed. Since the software will now have to perform more work to move data between the 8051 and the LCD panel, the code size and the execution time will naturally increase. In the clock project, there will be plenty of spare EPROM space left over, so the increased amount of executable code will not be an issue. The increase in execution time will also not be an issue as will be seen later when the performance of the system is analyzed.

The low level I/O routines for the display are simple to write once one understands the signals and timing required by the LCD panel. Examining the data sheets for the panel reveals that there are only three basic functions required of the software: write a command to the display, write the next character to the display, and read the status register of the display. The timing relationships for these operations is shown below in Figure 0-3 and Figure 0-4.

Since the display panel will accept long delays in the relationship of the above signals, it will be safe to

have these signals go active or inactive on a microsecond type time base instead of the nanosecond type time base we would see in the system bus. Given this, the display I/O functions simply follow the timing diagrams shown above to perform their operation with the display.



Listing 0-1

```
void disp_write(unsigned char value) {  
    DISPDATA=value;           // latch the data  
    REGSEL=1;                 // select the data register  
    RDWR=0;                   // select write mode  
    ENABLE=1;                 // latch the data into the  
                             // LCD panel  
    ENABLE=0;  
}
```

The responsibility of the 'disp_write' function is to send the next character to the LCD panel for displaying. It enforces the above signal relationships and returns to the caller. It is the caller's responsibility to check if the display is busy before sending the next character or command to the display.

Listing 0-2

```
void disp_cmd(unsigned char cmd) {  
    DISPDATA=cmd;             // latch the command  
    REGSEL=0;                 // select the command register  
    RDWR=0;                   // select write mode  
    ENABLE=1;                 // latch the command into the  
                             // LCD panel  
    ENABLE=0;  
    TH1=0;                    // start a timer for 85ms  
    TL1=0;  
    TF1=0;  
    TR1=1;  
    while (!TF1 && disp_read() & DISP_BUSY); // wait for the display  
                                         // finish the command  
    TR1=0;  
}
```

The 'disp_cmd' function enforces the same timing relationships as the 'disp_write' routine. However, the 'disp_cmd' routine will not return until the display panel is ready for the next command. There is no time-out used for waiting for the display panel in this project because if the display fails, the entire product is no good anyway!

Listing 0-3

```
unsigned char disp_read(void) {
    unsigned char value;
    DISPDATA=0xFF;           // set the port for all inputs
    REGSEL=0;                // select the command register
    RDWR=1;                  // select read mode
    ENABLE=1;                // enable the LCD output
    value=DISPDATA;          // read in the data
    ENABLE=0;                // disable the LCD output
    return(value);
}
```

The job of the 'disp_read' routine is to latch in the value currently held in the status register of the display panel. Again, the code enforces the signal timing relationships shown above and reads P1 at the correct time. The value read is saved and returned as the result of the function. Listing 0-3 shows this function.

As you can see, it is a simple matter to control the display directly from the port of the processor instead of using the system bus. The drawback is that it will take longer to access the display using this method than it would to access it through the bus. Additionally, the amount of code required for this method is larger than the amount of code it takes to interface to the display via the system bus. These things must be traded off with decreased parts count and product cost in the long run to determine which approach will be better suited to your application.

Writing Data to the Display

Once the initialization sequence has been executed, the display is ready for use. To write a character to the display is a simple matter. The display is told what address it should be reading characters for, and the character is then sent to the display. The display will automatically increment the address to the next character cell, so that every character does not need to be preceded by an address specifier.

To properly display messages and interact with the user, the clock project will need a function that is capable of performing the tasks listed above and perhaps clearing the display panel on command. The 'putchar' function will be redefined to output characters to the LCD panel. As such it must understand how to write a character to the display using the low level I/O routines defined above. In addition to writing its argument to the LCD panel, a couple of other changes have been made to 'putchar'. When the routine sees character 255, it issues a command to the display to clear itself and return to the home position. Additionally, 'putchar' keeps track of the number of characters written to the display since the last home command so that it knows when it should start writing characters on the second line of the display. Listing 0-4 shows the 'putchar' routine as it will be used in this project.

Listing 0-4

```
char putchar(char c) {
    static unsigned char flag=0;
    if (!flag || c==255) {           // check if the display
                                     // should be moved to home

        disp_cmd(DISP_HOME);
        flag=0;
        if (c==255) {
            return c;
        }
    }
}
```

```
}
if (flag==16) {                                // check if its time to use
                                                // the next display line
    disp_cmd(DISP_POS | DISP_LINE2); // move the display to ln 2
}
disp_write(c);                                // write the character to
                                                // the display
while (disp_read() & DISP_BUSY); // wait for the display
flag++;                                       // increment the line flag
if (flag>=32) { flag=0; }                    // when the whole display is
                                                // written, clear it
return(c);                                   // for compatibility
}
```

As you can see, the replacement for the 'putchar' routine ends up to be fairly simple. It calls some low-level I/O routines written to pass data back and forth with the display, but other than that there is no great trick to it. If it is successful, 'putchar' returns the character it was passed. In this case, it is assumed that the display is there and functional, and therefore the character that was written is always returned.

Customizing 'printf' to suit your needs

The C51 run time library support includes a fully functioning 'printf' routine. The 'printf' function formats strings and outputs them to the standard output device. In the case of a PC this would be your display, in the case of the 8051 it is the serial port. However, in this project, there is only a display. Internally, the 'printf' function calls putchar to output the string character by character. Therefore, by redefining 'putchar', such as has already been done above, the output from 'printf' can be redirected in any manner needed. When the project is linked together, the linker will use the 'putchar' routine in the clock source code, instead of the one in the run time library. The following function will be responsible for calling 'printf' to format the time string and send it to the display panel.

Listing 0-5

```
void disp_time(void) {
    // use the holder to display the current time.
    // note that the holder use flag is not cleared until we
    // are finished with the holder's contents. this will
    // prevent the contents from changing in the middle of use
    printf("\xFFTIME OF DAY IS: %B02u:%B02u:%B02u      ",
           timeholder.hour, timeholder.min, timeholder.sec);
    disp_update=0;                                // clear the display update flag
}
```

Using the Timer/Counters as a System Tick

Many embedded systems, particularly those which have small size or low cost as a goal do not have parts such as real time clocks or multivibrators to provide any sort of timing tick to them. Yet, many of these same systems have tasks which must occur on a periodic time schedule or in a specific amount of time from some system event. These tasks may vary from controlling a display which must present new data at a rate a human can deal with to polling a certain input at a given frequency. Many times, the person designing the system will use a timing loop to perform all the system timing. This loop usually consists of a given number of cycles which will cause the processor to waste one second, for example. The drawback to such a system is that the system often ends up having several of these loops to allow

for different time delays needed by the system. Additionally, a lot of processor time is wasted simply executing NOP and DJNZ instructions. In situations where the embedded system must run off a battery, this will severely limit the life of the battery which is powering the system.

A much better approach is to dedicate one of the on board timers as a system tick generator. The timer is reloaded with a value that will cause an overflow interrupt in a given amount of time, for example 50 milliseconds. An interrupt service routine is then written which maintains the timer and reloads it for the next tick as well as dispatching any scheduled events and executing any processes that should occur. The advantages to this sort of a system are manifold. In the first place, the processor no longer has to execute timing loops. It can now spend any time between tick interrupts in idle mode or executing background type tasks. Secondly, all the control for the timing becomes centrally located in one ISR. Therefore, if the crystal speed changes or if the resolution of the tick needs to be changed, the software only has to change in one place in the code. Thirdly, the entire timing code can be written in C. If you desire, you can find out the exact code delay from the time the timer overflows to the time your ISR reloads the timer and restarts it by examining the assembly code emitted by the compiler. The computed code delay can then be taken into account when you calculate the reload value for the timer.

Any embedded system I have written that requires any sort of systematic timing but does not have an external tick interrupt has used this approach to system timing. This section will show you how to develop a tick which occurs every 50 milliseconds. This routine will later be put to use in the clock project. When writing a tick routine for the 8051, the first thing you must know is your required resolution. If the fastest you must perform any task is once every 3 milliseconds, then pick this time to be your tick. Things that must occur at much slower rates can simply divide down the ticks to obtain the correct resolution. If this method prevents you from accurately timing the slow events, then choose the lowest common denominator between the fastest task and the other tasks and make that your tick rate. If you find that your system times are completely incompatible, you may want to consider using two timers for system ticks and dividing the work between them.

Once you have determined the resolution of your system tick, you must then figure out the timer reload value that will generate a tick at the desired frequency. To do this, you must take your oscillator frequency and use it to obtain the time it takes for each processor cycle since the timer will be counting these cycles. Assuming that you want to generate a 50 millisecond tick, and that your system is running at 12 MHz, you get the following answers. First, divide the 12MHz by 12 to obtain the instruction cycle frequency. Doing this yields 1MHz. Inverting this number gives you 1 microsecond per instruction cycle.

Now that you have this number, you must calculate the number of instruction cycles per system tick. In this case, the tick is to be every 50ms and the instruction cycle takes 1 μ s. This means that it will take 50000 instruction cycles to obtain a tick rate of 50ms. Since the timer counts upward from 0 to 10000 hex, you must subtract the 50000 cycles from 65536 (10000 hex) to get the initial value for the timer. Performing this subtraction yields a result of 15536 (3CB0) for the reload value. If you do not care about losing a few microseconds every tick, you can simply insert this value into your ISR and be done with the timing. For right now, that is how the example will be coded. Later, this timing error will be corrected.

Now that the reload value is known, the skeleton tick ISR can be written. In the following example, timer 0 is used as the system tick timer which leaves timer 1 to function as the baud rate divisor for the on board UART or as any other sort of timing/counting function required.

Listing 0-6

```
#define RELOAD_HIGH    0x3C
#define RELOAD_LOW     0xB0

void system_tick(void) interrupt 1 {
    TR0=0;                // temporarily stop timer 0
    TH0=RELOAD_HIGH;      // set the reload value
    TL0=RELOAD_LOW;
    TR0=1;                // restart the timer

    // perform system tick operations here.
}
```

The above routine will serve as the basic structure for this ISR. Once the timer is reset and sent on its way, the ISR performs system tick operations such as maintenance of tick counters, execution of events, and setting of flags. You must ensure that these operations do not take longer than the tick rate, or you will find yourself losing ticks. Later, this section will discuss ways to handle this situation.

Once you have gotten to this point, you have a routine to which you can add tick counters and such as need be. You can typically use this as an easy way to force the system to take some action a specified amount of ticks later. This is done by setting a variable to the number of ticks you wish to delay. This global variable will be decremented by the tick routine. When it has a value of zero, the action associated with this variable will be taken. For example, if you have an LED attached to a pin you have called 'LED' that you want to stay on for two seconds and then turn off, at the point you enable the LED set the associated tick counter to 40 ($40 * 50\text{ms} = 2\text{ s}$). The code in the tick routine will look like this:

```
if (led_timer) {          // if the LED timer is active
    led_timer--;           // decrement its count
    if (!led_timer) {      // if the LED on time has
                           // expired...
        LED=OFF;           // turn off the LED
    }
}
```

While the above segment of code may seem simple, it will serve the purpose for most embedded systems. In cases where more complex functions must be performed, the code can be placed in a function and then called from the tick routine. All system functions that use the tick can be placed in this ISR in sequential fashion. Thus, after one timer is checked, the ISR will continue to check the next timer and perform the appropriate functions until it has completed the routine. Any timers which share a common rate of division of the tick can be placed in an area that is only executed when the tick has been divided by a specific amount.

Suppose that you needed to perform several actions on at resolutions which are never less than one second, and that you are using the tick routine shown above. In this case, you maintain a counter (most likely in the DATA segment) that divides the ticks for you and only look at the timers which are based on seconds when the dividing counter reaches zero. This will save the system from wasting a lot of time looking at counters which do not require 50ms resolution.

```
second_cnt--;                // decrement the tick divisor
if (!second_cnt) {          // if one second has passed...
    ...                     // check all second based timers
    second_cnt=20;          // reset the tick divisor to
                            // one second
}
```

You must pay careful attention to the amount of time your ISR is taking to execute. If the execution time becomes greater than the time between ticks you will find that you begin to lose system ticks, and your timing will be off. In this situation, you can typically take things out of the tick routine and execute them from the main loop. The tick routine tells the main loop that it must execute the associated actions by setting a flag. The actions which are removed from the tick routine must be ones that you are willing to give up some accuracy on. The actions that must still have the accuracy should stay in the ISR where their frequency of execution can be assured.

This tick routine developed above will serve as the engine of the clock. In here, it will keep track of the time of day and set a request flag when the display panel must be updated. A part of the main loop (or background process) will monitor this flag and update the display with the new time of day whenever it is set. Sections of the timer 0 interrupt will be responsible for timing the switch delays to make the user interface more usable.

Using the system tick for the user interface

The user interface for this product is relatively simple, but this does not mean that the concepts used here can not be applied to much larger systems. The SET switch is used to activate the set mode in which the user can change the current time of day in the clock. Once the clock is in set mode, the SET switch will increment the field under which the cursor is. The SELECT switch will advance the cursor to the next position. Once the cursor is advanced past the final field (seconds) the set mode is ended. Each time SET or SELECT is active, the set mode timer is reset to the maximum value. This timer is decremented each system tick. When it reaches zero, set mode is terminated.

The interface implemented here will poll the two user switches once every 50ms in the tick routine. This sampling rate will be more than sufficient for humans. It has been my experience that a human will be happy with a user interface that samples switches and knobs as slowly as once every .2 seconds! Compared to an 8051, a human is a slow I/O device. When the tick routine notes that a switch has been depressed, it will set a counter to a debounce value. This counter will be decremented in the tick routine which will not sample the switches again until the timer has reached zero.

When set mode is active, the software must control the position of the cursor on the display panel to aid the user in knowing which field is being set. The current field is pointed to by the 'cur_field' variable. The 'set_cursor' function will turn the cursor on or off and move it to the location currently being altered. To simplify the user's job of setting and possibly synchronizing the clock, the section of the system tick which is responsible for computing the time of day is halted when set mode is active. This will also prevent the main loop from trying to use 'printf' to update the display at the same time the interrupt routine is using it to update the display. To further guarantee that this will not happen the set mode cannot be activated during a display update by the main routine. Again, this will prevent the 'printf' routine from being called by more than one interrupt level at a time.

The 'system_tick' routine as it currently exists is shown below. For most systems, this routine will be more than sufficient. It can easily be used as a model for system tick routines in your projects.

Listing 0-7

```
void system_tick(void) interrupt 1 {
    static unsigned char second_cnt=20; // counter to divide system
                                        // ticks into one second

    TR0=0;                             // temporarily stop timer 0
    TH0=RELOAD_HIGH;                   // set the reload value
    TL0=RELOAD_LOW;
    TR0=1;                             // restart the timer

    if (switch_debounce) {             // debounce user switches
        switch_debounce--;
    }
    if (!switch_debounce) {
        if (!SET) {                    // if set switch is pressed...
            switch_debounce=DB_VAL;    // set switch debounce
            if (!set_mode && !disp_update) { // if the clock is not
                                        // in set mode

                set_mode=1;             // enter set mode
                set_mode_to=TIMEOUT;    // set the idle timeout value
                cur_field=HOUR;          // select the first field
                set_cursor(ON, HOUR);    // enable the cursor
            } else {
                cur_field++;             // advance the current field
                if (cur_field>SEC) {     // if its greater than SEC
                                        // then set mode is complete

                    set_mode=0;         // exit set mode
                    set_mode_to=0;
                    set_cursor(OFF, HOME); // disable the cursor
                } else {
                    set_cursor(ON, cur_field); // move the cursor to the next field
                    set_mode_to=TIMEOUT;
                }
            }
        }
    }
    if (set_mode && !SELECT) {          // if the select switch is pressed
        set_mode_to=TIMEOUT;
        incr_field();                  // increment selected field
        disp_time();                   // display the updated time
    }
}
```

```
if (!set_mode) {                // when in set mode, stop the clock
    second_cnt--;                // decrement the tick divider
    if (!second_cnt) {          // if one second has passed...
        second_cnt=20;          // reset the divider
        second_tick();          // perform the functions
                                // which take place every second
    }
}
}
```

Improving the Clock Software

At this point, you can take action to eliminate the error in the system tick. As you will remember the error is due to the code delays involved from the time the timer overflows to the time the tick routine reloads and restarts the timer. To eliminate the error, simply compile this routine using the C51 'code' option (this is done by selecting the "Include assembly assembly code" checkbox in the "C51 Compiler Options" dialog box of the workbench) and then count the cycles that the ISR takes to restart the timer. Once you get this count, you must add in two cycles for the processor to vector to your ISR. You could argue that the processor could have taken up to three more cycles to recognize the interrupt if it were in the middle of a DIV or MUL operation, but it is usually safe to assume that it has entered this point in the code from idle mode. There is no quick and reliable way to determine the exact latency for the processor to recognize the interrupt. Compiling the existing system tick routine with the 'code' option yields the following output fragment to which I have added an instruction count.

Listing 0-8

```
                ; FUNCTION system_tick (BEGIN)
0000 C0E0        PUSH    ACC                2, 2
0002 C0F0        PUSH    B                  2, 4
0004 C083        PUSH    DPH                2, 6
0006 C082        PUSH    DPL                2, 8
0008 C0D0        PUSH    PSW                2, 10
000A C000        PUSH    AR0                2, 12
000C C001        PUSH    AR1                2, 14
000E C002        PUSH    AR2                2, 16
0010 C003        PUSH    AR3                2, 18
0012 C004        PUSH    AR4                2, 20
0014 C005        PUSH    AR5                2, 22
0016 C006        PUSH    AR6                2, 24
0018 C007        PUSH    AR7                2, 26
                                ; SOURCE LINE # 332
                                ; SOURCE LINE # 335
001A C28C        CLR     TR0                1, 27
                                ; SOURCE LINE # 336
001C 758C3C      MOV     TH0,#03CH          2, 29
                                ; SOURCE LINE # 337
001F 758AAF      MOV     TL0,#0AFH          2, 31
                                ; SOURCE LINE # 338
```

```
0022 D28C          SETB    TR0          1, 32
                                           ; SOURCE LINE # 340
```

According to the instruction count, reload value of the timer must be altered by 34 (32 + 2) cycles to allow for all the calculable loss in the system. It is interesting to note that the greatest amount of loss is due to all the pushing to the stack. Since each of these operations must have a corresponding pop, this ISR will spend 52 cycles working with the stack. A big part of this is because the compiler is attempting to save the code from any damage that called functions may cause. This source of waste can be eliminated by specifying the register bank for the ISR.

Another source of processing waste is the 'printf' function. Simulation runs show that it will take 6039 cycles to print the entire time string to the display when the time is 00:00:00. This simulation was done assuming that the display panel indicates immediately that it is ready for the next command. We can therefore conclude that the 6039 cycles is due entirely to the 'printf' and 'putchar' routines. The 6039 cycles will amount to the processor working 6.039 milliseconds in between executions of the system tick routine. It is a safe assumption that this amount of processor usage will not cause any harm to system stability. Just to make sure of this, another simulation was run to determine the total execution time of the 'system_tick' routine when the time structure must be incremented from 23:59:59 to 00:00:00. This should represent the worst case execution time for this routine when the system is not in set mode. The total number of cycles taken by the system_tick routine under this condition is 207. This means that the worst case execution for a non-set mode system_tick will be .207 milliseconds. For cases where no time increment is performed, the ISR will take 76 cycles of .076 milliseconds.

In the case of the clock, this is all a mute point since the system tick is set at 50 milliseconds. This means that the clock can update the time structure, and update the LCD panel **every system tick** and still spend 43.754 milliseconds (or 87.508% of the time) in idle mode before it has to deal with the next system tick interrupt. However, let's assume that you want to get all the performance out of this product that you can. Perhaps it will be a battery powered product that must conserve the power cell supplying it as much as possible. The best bet to lower execution time for the processor is to replace the 'printf' routine with a more specific and smaller routine.

Since you know that this system does not need to display any messages other than the time message you can greatly simplify the 'printf' routine. It does not need to deal with string formatting, character formatting, any kind of integers, longs, or floats. You can also assume that only fixed locations in the string will be changing. The remainder of the locations will not change at all. The replacement for 'printf' will deal with a buffer that has the fixed text already preformatted and will simply insert the right characters to represent the time in the correct locations. To speed execution of this routine, the characters will be obtained by using the magnitude of the time field in use to access a lookup table. This is a trade off of execution time for EPROM space. Since this program is relatively small so far (less than 2000 bytes), there is plenty of EPROM space to use. If you were in the opposite position, you should force the 'system_tick' routine to keep the time in BCD with an ASCII bias. Therefore, when the system is ready to display the time, the structure which holds it is already be in character representation. This approach would certainly cause the 'system_tick' ISR to take more than its current average execution time of 76 cycles.

The 'disp_time' function itself becomes the replacement for the 'printf' routine. Instead of calling 'printf' with a format string and the three bytes as parameters, it will now build up the string itself in a local buffer and pass the whole thing to 'putchar' one byte at a time. The complexity of the user generated code has increased, but the overall size of the system has decreased from 1951 bytes to 1189 bytes even though a 120 byte table has been added to the EPROM. The 'printf' routine was taking 811 total bytes and the new 'disp_time' now takes 105 bytes. Listing 0-9 shows the new 'disp_time' routine.

Listing 0-9

```
void disp_time(void) {
    // use the holder to display the current time.
    // note that the holder use flag is not cleared until we
    // are finished with the holder's contents.  this will
    // prevent the contents from changing in the middle of use
    static char time_str[32]="TIME OF DAY IS: XX:XX:XX      ";
    unsigned char I;

    time_str[T_HOURT]=bcdmap[timeholder.hour][0];
    time_str[T_HOUR]=bcdmap[timeholder.hour][1];
    time_str[T_MINT]=bcdmap[timeholder.min][0];
    time_str[T_MIN]=bcdmap[timeholder.min][1];
    time_str[T_SECT]=bcdmap[timeholder.sec][0];
    time_str[T_SEC]=bcdmap[timeholder.sec][1];
    putchar(0xFF);
    for (i=0; i<32; i++) {
        putchar(time_str[i]);
    }
    disp_update=0;                      // clear the display update lag
}
```

The new 'disp_time' routine takes a total of 2238 processor cycles or 2.238 milliseconds on a 12MHz system. Note again that the times presented do not allow for the delays caused by the display processor, for which there is nothing we can do. The worst case latency for the display will be one delay of 1.64 milliseconds to clear it and a total delay of 1.280 milliseconds to write 32 characters to the display panel. The total delay due to the LCD panel is thus 2.920 milliseconds per refresh. Assuming that the display panel is refreshed every second, the processor is executing 6.866 milliseconds out of every second. This was obtained by taking the non-set mode time of system tick (76 cycles) and multiplying it by 19, since there will be 19 ticks like this per second. 207 cycles were added to allow for the one system tick per second in which the time structure is updated. 2238 cycles were added to allow for the time it takes the disp_time routine to function, and 2920 cycles were added to allow for the time it takes the LCD panel to receive and display 32 characters. As you can see, this system will be spending most of its time in idle mode and will be very easy on its power supply.

Optimizing the Internal RAM Usage

One drawback to the clock software that has not been discussed is that it does not effectively use its internal RAM. The DATA segment memory map from the M51 file shows this:

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME

* * * * *	D A T A	M E M O R Y	* * * * *	
REG	0000H	0008H	ABSOLUTE	"REG BANK 0"
DATA	0008H	0002H	UNIT	"DATA_GROUP"
	000AH	0016H		*** GAP ***
BIT	0020H.0	0000H.2	UNIT	?BI?CH4CLOCK
BIT	0020H.2	0000H.1	UNIT	"BIT_GROUP"
	0020H.3	0000H.5		*** GAP ***
DATA	0021H	002BH	UNIT	?DT?CH4CLOCK
IDATA	004CH	0001H	UNIT	?STACK

While not immediately obvious, the problem is that there is a 22 byte gap in the DATA segment between address 0A and the start of the bit addressable segment (address 20). This is because the linker could not fit the DATA segment for the CH4CLOCK module in this area, and thus left a gap. The effects of the gap are that it leaves some amount of the DATA segment unused, and pushes the base of the stack closer to the start of the ISR segment, meaning that the system is that much closer to overflowing the stack and having a mysterious reset problem.

This has happened because the clock code defined all its variables in one file, ch4clock.c. The easiest way to correct this problem is to use the precede directive with the linker when you link your modules. The precede directive will allow you specify which data segments of your program are stored at the bottom of the DATA segment. The easiest solution is to use this command and tell the linker that the data variables of the clock's code be placed at the bottom of the DATA segment. This is done by entering the linker options dialog box in the workbench, selecting the "Segments" tab and filling in the "Precede" edit control with the following string.

```
?DT?ch4clock
```

This will work great as long as the data usage of the module(s) you specify does not exceed the space between the highest used register bank and the top of the bit addressable segment assuming that you have bit variables. Once you exceed this memory limitation, you will find that your BIT segment disappears and you get linker warnings and errors. The way around this is to move a group of internal data variables to another file. This will create two smaller data segments and allow you to properly use the precede directive. By moving the correct amount of allocation to another file, you can completely eliminate the gap.

The extra file is compiled separately and linked together with the main file (or files). In this case, 22 bytes worth of variables need to move to another file. The clock program, being a small system has only nine bytes which can be moved to another file. Performing this operation and re linking yields the following result.

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME

* * * * *	* * * *	D A T A	M E M O R Y	* * * * *
REG	0000H	0008H	ABSOLUTE	"REG BANK 0"
DATA	0008H	0022H	UNIT	?DT?CH4NEW
BIT	002AH.0	0000H.2	UNIT	?BI?CH4NEW
BIT	002AH.2	0000H.1	UNIT	_BIT_GROUP_
	002AH.3	0000H.5		*** GAP ***
DATA	002BH	0009H	UNIT	?DT?VARS
DATA	0034H	0004H	UNIT	_DATA_GROUP_
IDATA	0038H	0001H	UNIT	?STACK

According to the above table, the linker has left the project with 72 bytes of stack space (80 hex - 38 hex) and that unsightly gap is gone. You must now make sure that 72 bytes will be enough stack space for the program at the deepest point in the calling tree. It turns out that this point is during the 'disp_time' call in the system tick ISR. From earlier discussions, you know that the ISR is pushing 13 bytes worth of registers onto the stack. The processor pushes the PC on the stack when the interrupt is caused and the call to the ISR takes two more bytes. This means that the total is 17 bytes. The 'disp_time' call will take two more bytes on the stack for the PC as will its calls to 'putchar'. This brings us to 21 bytes. 'putchar' calls 'disp_cmd' which in turn will call 'disp_read'. This adds four more bytes of data from the PC to bring the total to 25 bytes. This is the deepest the stack will grow, and there are still 47 bytes of stack space free. It is safe to assume that this program will not exceed the stack space that the linker has given to it.

Ship it!

The clock project is complete at this point. The software timing has been checked and there is a substantial amount of processor bandwidth left. The design goals of eliminating most of the hardware using software routines was realized and the system is basically down to an 8751 and an LCD panel. The software as it stands at this point is shown in Listing 0-10. The following sections will present some other ways to improve your system using software techniques.

Listing 0-10

```
#include <reg51.h>
#include <stdio.h>

// define the reload values to generate a 50ms tick in timer 0
#define RELOAD_HIGH    0x3C
#define RELOAD_LOW     0xD2

// define the switch debounce time value
#define DB_VAL         6
// define the time period of maximum inactivity in set mode
// before set mode is terminated
#define TIMEOUT         200

// define constants for cur_field
#define HOME            0
#define HOUR            1
```

```
#define MIN          2
#define SEC          3

// define constants for the cursor state
#define OFF          0
#define ON           1

// define constants for the display commands
#define DISP_BUSY    0x80
#define DISP_FUNC    0x38
#define DISP_ENTRY   0x06
#define DISP_CNTL    0x08
#define DISP_ON      0x04
#define DISP_CURSOR  0x02
#define DISP_CLEAR   0x01
#define DISP_HOME    0x02
#define DISP_POS     0x80
#define DISP_LINE2   0x40

sbit SET = P3^4;           // set switch input pin
sbit SELECT = P3^5;       // select switch input pin
sbit ENABLE = P3^1;       // display enable output
sbit REGSEL = P3^7;       // display register select output
sbit RDWR = P3^6;        // display access mode output

sfr DISPDATA = 0x90;      // eight bit data bus to the
                          // display panel

typedef struct {           // define a type to hold
    unsigned char hour, min, sec; // the time of day
} timestruct;

bit          set_mode=0,   // set if we are in set mode
            disp_update=0; // set when the display
                          // needs refreshing

unsigned char set_mode_to=0, // timer used to count time
                          // from last user operation
                          // to the termination of set mode
            switch_debounce=0, // switch debounce timer
            cur_field=HOME;    // currently selected field
                          // for set mode

timestruct   curtime,       // holds the current time
```

```
        timeholder;                // holds the time to go on
                                    // the display panel

unsigned char code fieldpos[3]={    // tells set_cursor what
                                    // physical position the
                                    // field argument refers to

        DISP_LINE2 | 0x01,
        DISP_LINE2 | 0x04,
        DISP_LINE2 | 0x07
};

#define T_HOURLT      16
#define T_HOUR        17
#define T_MINT         19
#define T_MIN          20
#define T_SECT         22
#define T_SEC          23

char code bcdmap[60][2]={
    "00", "01", "02", "03", "04", "05", "06", "07", "08", "09",
    "10", "11", "12", "13", "14", "15", "16", "17", "18", "19",
    "20", "21", "22", "23", "24", "25", "26", "27", "28", "29",
    "30", "31", "32", "33", "34", "35", "36", "37", "38", "39",
    "40", "41", "42", "43", "44", "45", "46", "47", "48", "49",
    "50", "51", "52", "53", "54", "55", "56", "57", "58", "59"
};

// declarations of program functions
void disp_cmd(unsigned char);
void disp_init(void);
unsigned char disp_read(void);
void disp_time(void);
void disp_write(unsigned char);
void incr_field(void);
void second_tick(void);
void set_cursor(bit, unsigned char);

/*****
Function:      main
Description:   This is the entry point of the program. This
               function initializes the 8051, enables the correct
               interrupt source and enters idle mode. The idle mode loop checks
               after each interrupt to see if the LCD panel must be updated.
Parameters:   None.
Returns:      Nothing.
*****/
```

```

*****/
void main(void) {
    disp_init();                // set up display
    TMOD=0x11;                  // set both timers in 16 bit mode
    TCON=0x15;                  // start timer 0. both ext
                                // ints are edge
    IE=0x82;                    // enable the timer 0 int
    for (;;) {
        if (disp_update) {
            disp_time();        // display new time;
        }
        PCON=0x01;             // enter idle mode
    }
}

/*****
Function:    disp_cmd
Description: This routine writes a given command to the LCD
             panel and waits to assure that the command was
             completed by the panel.
Parameters:  cmd - unsigned char. Holds the command to be
to the display.
Returns:     Nothing.
*****/
void disp_cmd(unsigned char cmd) {
    DISPDATA=cmd;               // latch the command
    REGSEL=0;                   // select the command reg
    RDWR=0;                     // select write mode
    ENABLE=1;                   // latch the command into
                                // the LCD panel

    ENABLE=0;
    TH1=0;                      // start a timer for 85ms
    TL1=0;
    TF1=0;
    TR1=1;
    while (!TF1 && disp_read() & DISP_BUSY); // wait for the display
                                                // to finish the command
    TR1=0;
}

/*****
Function:    disp_init
Description: Sends the correct data sequence to the display
to initialize it for use.
Parameters:  None.
Returns:     Nothing.
*****/
```

```

*****/
void disp_init(void) {
    TH1=0;                // start a timer for 85ms
    TL1=0;
    TF1=0;
    TR1=1;
    while (!TF1 && disp_read() & DISP_BUSY); // wait for the display
                                           // to finish the command
    TR1=0;
    disp_cmd(DISP_FUNC);   // set the display for an 8
                           // bit bus, 2 display lines,
                           // and a 5x7 dot font
    disp_cmd(DISP_ENTRY);  // set the character entry
                           // mode to increment display
                           // address for each
                           // character, but not to scroll
    disp_cmd(DISP_CNTL | DISP_ON); // turn the display on, cursor off
    disp_cmd(DISP_CLEAR);   // clear the display
}
/*****
Function:    disp_read
Description: This routine reads from the LCD panel status
             register.
Parameters:  None.
Returns:     The value read from the LCD panel.
*****/
unsigned char disp_read(void) {
    unsigned char value;
    DISPDATA=0xFF;        // set the port for all inputs
    REGSEL=0;             // select the command reg
    RDWR=1;               // select read mode
    ENABLE=1;             // enable the LCD output
    value=DISPDATA;        // read in the data
    ENABLE=0;             // disable the LCD output
    return(value);
}
/*****
Function:    disp_time
Description: This routine takes the time in the holder buffer
             and
             formats it for display on the LCD panel. No
             consideration is given to the position of the
             cursor.
Parameters:  None.
Returns:     Nothing.
*****/

```

```
void disp_time(void) {
    // use the holder to display the current time.
    // note that the holder use flag is not cleared until we
    // are finished with the holder's contents.  this will
    // prevent the contents from changing in the middle of use
    static char time_str[32]="TIME OF DAY IS: XX:XX:XX      ";
    unsigned char i;
    time_str[T_HOURT]=bcdmap[timeholder.hour][0];
    time_str[T_HOUR]=bcdmap[timeholder.hour][1];
    time_str[T_MINT]=bcdmap[timeholder.min][0];
    time_str[T_MIN]=bcdmap[timeholder.min][1];
    time_str[T_SECT]=bcdmap[timeholder.sec][0];
    time_str[T_SEC]=bcdmap[timeholder.sec][1];
    putchar(0xFF);
    for (i=0; i<32; i++) {
        putchar(time_str[i]);
    }
    disp_update=0;                                // clear the display update flag
}

/*****
Function:      disp_write
Description:   This routine writes a data byte to the LCD panel.
Parameters:   value - unsigned char. Holds the data byte to be
              written to the display.
Returns:      Nothing.
*****/
void disp_write(unsigned char value) {
    DISPDATA=value;                                // latch the data
    REGSEL=1;                                       // select the data reg
    RDWR=0;                                         // select write mode
    ENABLE=1;                                       // latch the data into the
                                                    // LCD panel

    ENABLE=0;
}

/*****
Function:      incr_field
Description:   This routine increments the time field indicated by
cur_field. No rollover from seconds to minutes or minutes to
hours is performed.
Parameters:   None.
Returns:      Nothing.
*****/
void incr_field(void) {
```

```
if (cur_field==SEC) {
    curtime.sec++;
    if (curtime.sec>59) {
        curtime.sec=0;
    }
}
if (cur_field==MIN) {
    curtime.min++;
    if (curtime.min>59) {
        curtime.min=0;
    }
}
if (cur_field==HOUR) {
    curtime.hour++;
    if (curtime.hour>23) {
        curtime.hour=0;
    }
}
}

/*****
Function:    putchar
Description: This routine replaces the standard putchar
             function. Its job is to redirect output to the LCD panel.
Parameters:  c - char. This is h\next character to write to the
             display.
Returns:     The character just written.
*****/
char putchar(char c) {
    static unsigned char flag=0;
    if (!flag || c==255) {                // check if the display
                                           // should be moved to home
        disp_cmd(DISP_HOME);
        flag=0;
        if (c==255) {
            return c;
        }
    }
    if (flag==16) {                      // check if its time to use
                                           // the next display line
        disp_cmd(DISP_POS | DISP_LINE2); // move the display to ln 2
    }
    disp_write(c);                       // write the character to
                                           // the display
    while (disp_read() & DISP_BUSY);     // wait for the display
}
```

```

    flag++;                                // increment the line flag
    if (flag>=32) { flag=0; }              // when the whole display is
                                           // written, clear it
    return(c);                             // for compatibility
}
/*****

Function:      second_tick
Description:   This function performs all functions which must be done
every second. In this system that involves incrementing the time
by one second and requesting a display panel refresh.
Parameters:    None.
Returns:       Nothing.
*****/

void second_tick(void) {
    curtime.sec++;                          // advance the seconds
    if (curtime.sec>59) {                  // check for rollover
        curtime.sec=0;
        curtime.min++;                    // advance the minutes
        if (curtime.min>59) {              // check for rollover
            curtime.min=0;
            curtime.hour++;                 // advance the hours
            if (curtime.hour>23) {          // check for rollover
                curtime.hour=0;
            }
        }
    }
    if (!disp_update) {                   // make sure the holder
                                           // isn't in use
        timeholder=curtime;                // set the new time in the holder
        disp_update=1;                     // the time will change, so
                                           // update the LCD panel
    }
}

/*****

Function:      set_cursor
Description:   This routine enables or clears the cursor and moves
it to a specified point.
Parameters:    new_mode - bit. Set if the cursor should be
               visible.
               field - unsigned char. Indicates the field to move
the cursor to.
Returns:       Nothing.
*****/

void set_cursor(bit new_mode, unsigned char field) {
    unsigned char mask;

```

```
mask=DISP_CNTL | DISP_ON;
if (new_mode) {
    mask|=DISP_CURSOR;
}
disp_cmd(mask);
if (field==HOME) {
    mask=DISP_HOME;
} else {
    mask=DISP_POS | fieldpos[field-1];
}
disp_cmd(mask);
}

/*****
Function:      system_tick
Description:   This is the ISR for timer 0 overflows. It
               maintains the timer and reloads it with the
               correct value
               for a 50ms tick. The time is
               counted in this routine and
               the user interface is maintained.
Parameters:   None.
Returns:      Nothing.
*****/

void system_tick(void) interrupt 1 {
    static unsigned char second_cnt=20; // counter to divide system
                                       // ticks into one second
    TR0=0;                             // temporarily stop timer 0
    TH0=RELOAD_HIGH;                   // set the reload value
    TL0=RELOAD_LOW;
    TR0=1;                             // restart the timer
    if (switch_debounce) {             // debounce user switches
        switch_debounce--;
    }
    if (!switch_debounce) {
        if (!SET) {                   // if set switch is pressed...
            switch_debounce=DB_VAL;   // set switch debounce
            if (!set_mode && !disp_update) { // if the clock is not in set mode
                set_mode=1;            // enter set mode
                set_mode_to=TIMEOUT;   // set the idle timeout value
                cur_field=HOUR;        // select the first field
                set_cursor(ON, HOUR);  // enable the cursor
            } else {
                cur_field++;           // advance the current field
                if (cur_field>SEC) {    // if its greater than SEC
                                        // then set mode is complete
                    set_mode=0;        // exit set mode
                }
            }
        }
    }
}
```

```
        set_mode_to=0;
        set_cursor(OFF, HOME);          // disable the cursor
    } else {
        set_cursor(ON, cur_field);      // move the cursor to the
                                         // next field

        set_mode_to=TIMEOUT;
    }
}

if (set_mode && !SELECT) {              // if the select switch is pressed
    set_mode_to=TIMEOUT;
    incr_field();                        // increment selected field
    disp_time();                        // display the updated time
}

if (!set_mode) {                        // when in set mode, stop the clock
    second_cnt--;                       // decrement the tick divider
    if (!second_cnt) {                  // if one second has passed...
        second_cnt=20;                 // reset the divider
        second_tick();                 // perform the functions which
                                         // take place every second
    }
}
}
```

Using a watchdog timer

Many embedded systems perform tasks which involve waiting for a certain device or spending a lot of time in loops processing data. It is crucial for a system to consistently be available to perform the function it was designed for. This means that an embedded system should never find itself in an infinite loop which is preventing it from carrying out its job. Infinite loops in an embedded system may be due to the failure of an I/O device, the reception of unexpected or bad input, or a software bug. Whatever the source, they will pop up at the worst times and will make your product look unreliable and poorly designed no matter how well the system functions otherwise.

As a protection against such infinite loops or system tie ups, many designers use a watchdog timer. The timer counts from a specific value and must be reloaded by the software within a specific amount of time. If the timer overflows, the system is reset on the assumption that since the software did not reload the timer, it must be hung up in a loop or unexpected condition. Presumably, resetting the system is better than having it lock up in some unknown state. The software engineer must deal with the watchdog timer by writing a function which reloads the watchdog and calling this function at a rate faster than the overflow rate of the timer. For most watchdog timers this routine is relatively simple to write. Usually there is some basic order of operations that must be observed to reload the timer. This is done to prevent accidental reloads by runaway programs. An initialize function and a reload function for the watchdog timer used on the Philips 80C550 are shown in Listing 0-11.

Listing 0-11

```
void wd_init(unsigned char prescale) {  
    WDL=0xFF;                // set watchdog reload value to  
                             // the maximum  
    WDCON=(prescale & 0xE0) | 0x05; // set the prescale value of the  
                             // timer to the slowest count  
    wd_reload();             // rate and start the watchdog  
}  
  
void wd_reload(void) {  
    EA=0;                    // block any interrupts  
    WFEED1=0xA5;             // first step of feed sequence  
    WFEED2=0x5A;             // second step of feed sequence  
    EA=1;                    // allow interrupts now  
}
```

Typically, you can place the reload function in your main loop in systems which have a periodic interrupt such as a system tick from a timer or from an RTC. This approach will work fine unless you have a system which also receives interrupts from another source which can fire many interrupts in a row and thus prevent the processor from executing the main loop often enough to reload the timer. In this case, you can put the reload function call in both the main loop and the ISR which is being executed so frequently. However, the reload call in the ISR is not made every interrupt. Instead, the number of interrupts are counted and the reload call is only made when the count reaches a certain value. Assuming that a system used this approach, the main routine and the ISR would have code fragments as such:

Listing 0-12

```
void main(void) {
    ...                // system initialization
    for (;;) {         // main loop to wait for and
                        // service interrupts
        ...            // any background code needed
        wd_reload();   // feed the watchdog timer
        PCON=0x80;     // enter idle mode and wait for
                        // next interrupt
    }
}

void my_ISR(void) interrupt 0 {
    static unsigned char int_count=0; // the interrupt count divider
    int_count++;                     // increment the interrupt
                                    // counter
    if (int_count > MAXINTS) {       // if the max threshold is
                                    // broken...
        int_count=0;                // reset the counter
        wd_reload();                // and feed the watchdog
    }
    ...                             // the rest of the ISR
}
```

The reset by the watchdog timer is indiscernible from a normal power on reset to the software. Thus, if your system has any data that is learned or programmed in during the execution of the software, you may want to save a backup copy of that data in external RAM or make sure that you don't initialize it at the beginning of every execution run of the system. To do this means that the system will need to understand how and when to prevent itself from clobbering data from the previous run that is still good.

Saving system data

Your embedded system may need to treat resets differently depending on the previous status of the unit. For example, if your system had already been executing properly, but was reset by a watchdog timer or a user pressing a reset switch, you may want to take different initialization actions than you would if your system had just been powered on. Typically, situations like the watchdog reset and user reset are referred to as warm boots. In 8051 systems which do not have any sort of battery backup for the internal and external RAM, these warm boots are easily detected by using a flag.

When the system first starts executing the code, this flag is checked for a specific value. If the value is not good, the power on initialization continues as if the unit is being cold started. If the value is good, the cold start specific code is by passed and only necessary initialization code is executed. Once the system is initialized, the warm boot flag is set to the value desired. The value chosen should be one that will not typically occur in the RAM should all power be lost. This will prevent a cold start from looking like a warm boot. Therefore values such as 00 and FF should be avoided. A value like AA or CC which has a definite pattern to it is a good choice. For systems which must keep data in the internal RAM from restart to restart, the boot flag must be checked in the startup code of the compiler. This means you will have to modify startup.a51. For systems which keep learned data in the external RAM or only need to perform different actions on a warm boot, the flag can be checked in the main routine assuming that the flag is not kept in internal RAM. This is because the default startup code for the compiler zeroes out all the memory locations in the internal RAM of the 8051 but will not zero the external memory spaces

unless you change startup.a51 to do so. Thus, if your boot flag is kept in internal RAM and not checked before the RAM is zeroed, your system will always believe it should cold boot.

As an example, consider the following main routine which checks the boot flag and performs extra initialization functions if the flag does not hold a good value.

Listing 0-13

```
unsigned char xdata bootflag;

void main(void) {

    ...

    if (bootflag!=0xAA) {                // if the system is being cold
                                        // started...

        init_lcd();                    // initialize the display panel
        init_rtc();                    // initialize the RTC
        setup_hw();                    // setup I/O ports
        reset_queue();                 // reset data structures
        bootflag=0xAA;                 // set the bootflag to a good
                                        // value

    } else {

        clear_lcd();                    // clear the message from the
                                        // display panel

    }

    ...

}
```

A user who must maintain the data in internal RAM in the case of a warm boot must alter the startup.a51 file and ensure that the code only clears those areas of the RAM which will be used by the compiler and which do not have to be remembered by the system. Such a modified startup.a51 will look like the one below.

Listing 0-14

```
-----
; This file is part of the C-51 Compiler package
; Copyright (c) KEIL ELEKTRONIK GmbH and Keil Software, Inc.,
; 1990-1992
;-----; STARTUP.A51: This
code is executed after processor reset.
;
; To translate this file use A51 with the following invocation:
;
; A51 STARTUP.A51
;
; To link the modified STARTUP.OBJ file to your application use
; the following L51 invocation:
```

```
;
;   L51 <your object file list>, STARTUP.OBJ <controls>
;
;-----
;
;   User-defined Power-On Initialization of Memory
;
;   With the following EQU statements the initialization of memory
;   at processor reset can be defined:
;
EXTRN          DATA      (bootflag)

;
; the absolute start-address of IDATA memory is always 0
IDATALEN      EQU  80H    ; the length of IDATA memory in bytes.
;
XDATASTART    EQU   0H    ; the absolute start-address of XDATA
                        ; memory
XDATALEN      EQU   0H    ; 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.
;
; Notes:  The IDATA space overlaps physically the DATA and BIT
;         areas of the 8051 CPU.  At minimum the memory space
;         occupied from the C-51 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.
;
;-----

                NAME      ?C_STARTUP
?C_C51STARTUP   SEGMENT    CODE
?STACK         SEGMENT    IDATA

                RSEG      ?STACK
                DS        1

                EXTRN CODE (?C_START)
                PUBLIC ?C_STARTUP

                CSEG      AT      0
?C_STARTUP:    LJMP      STARTUP1

                RSEG      ?C_C51STARTUP

STARTUP1:
                MOV       A, bootflag    ; check if RAM is good
                CJNE      A, #0AAH, CLRMEM
                SJMP      CLRCOMP        ; RAM is good, clear only
                                           ; compiler owned locations

CLRMEM:
                                           ; RAM was not good,
                                           ; zero it all

IF IDATALEN <> 0
```

```
        MOV     R0,#IDATALEN - 1
        CLR     A
IDATALOOP:  MOV     @R0,A
            DJNZ  R0,IDATALOOP
            JMP   CLRXDATA
ENDIF

CLRCOMP:   CLR     A                ; zero out compiler owned
            ; areas

            MOV     20H, A
            MOV     R0, #3EH
L1:         MOV     @R0, A
            INC     R0
            CJNE    R0, #76H, L1
CLRXDATA:
IF XDATALEN <> 0
            MOV     DPTR,#XDATASTART
            MOV     R7,#LOW (XDATALEN)
            IF (LOW (XDATALEN)) <> 0
                MOV     R6,#(HIGH XDATALEN) +1
            ELSE
                MOV     R6,#HIGH (XDATALEN)
            ENDIF
ENDIF

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

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

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

IF PPAGEENABLE <> 0
        MOV     P2, #PPAGE
ENDIF

        MOV     SP, #?STACK-1
        LJMP    ?C_START

        END
```

This routine checks the boot flag and if it is good, clears only the addresses used by the compiler's run time routines. All program owned locations must be explicitly handled in the user generated code. The addresses of the library routines were determined by examining the linker output file and clearing those segments of memory. As you can see, the bit variable at address 20H and the block from 3EH to 75H must be zeroed. The linker output file used with the above startup.a51 is shown below.

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME

* * * * *	* * * * *	D A T A	M E M O R Y	* * * * *
REG	0000H	0008H	ABSOLUTE	"REG BANK 0"
DATA	0008H	0012H	UNIT	?DT?VARS
DATA	001AH	0001H	UNIT	?DT?PUTCHAR
	001BH	0005H		*** GAP ***
DATA	0020H	0001H	BIT_ADDR	?C_LIB_DBIT
BIT	0021H.0	0000H.5	UNIT	?BI?COINOP
BIT	0021H.5	0001H.2	UNIT	"BIT_GROUP"
	0022H.7	0000H.1		*** GAP ***
DATA	0023H	001BH	UNIT	?DT?COINOP
DATA	003EH	000FH	UNIT	?C_LIB_DATA
DATA	004DH	0029H	UNIT	"DATA_GROUP"
IDATA	0076H	001EH	UNIT	?ID?COINOP
IDATA	0094H	0001H	UNIT	?STACK

Another way to save your internal data variables without going to the hassle of having to determine the addresses which are safe and unsafe to zero out is to store them in external RAM. This of course assumes that you have an external RAM that is battery backed up. If you don't, an EEPROM or flash memory can be used in its place and will be more reliable. Most of my systems use the RAM method because it is quicker to write data to an SRAM than it is to write it to an EEPROM. The reason this matters is that the system saves all the valid internal variables to a block of external RAM when the processor receives a shut down interrupt. Once this interrupt is activated, the system has plenty of time to write data to the SRAM and put itself in power down mode. However, EEPROM's are much slower devices and could not be accessed quickly enough for this purpose. If the data you need to save is not frequently changed, then a back up copy of it could be saved in non-volatile storage. The back up must then be refreshed each time the main copy of the data is altered. If the data is altered in many places in the program the added work and code of this method may prove to be prohibitive.

Regardless of the data backup method employed, when the system is re powered, a data valid byte (much like the boot flag discussed above) is checked. If it shows that the data is good, the internal variables are restored from the data in the non-volatile storage. This is done during system initialization in an if loop much like the one used with the boot flag.

Conclusion

This chapter has presented ways to eliminate hardware components and ease the job of the hardware in your systems. The methods discussed here are not the only methods available, they are meant to show you techniques that you can learn and build upon. Obviously, there will be many more situations in which software can assume some hardware functions and make the hardware design simpler. To cover them all would take a lifetime. Use the tools that have been presented here to build new tools and improve each project that you undertake.

- Assembly and C on the 8051

Introduction

At some point in your embedded systems career you will be forced to leave the cozy world of high level language development and use assembly language in your system. For the most part, you will find that the portions of the program which must be done in assembly can be easily integrated with the remainder of the program which should be written in C. This chapter will present methods to help you integrate your assembly segments into your C program. In addition, it will discuss ways in which you can alter the assembly code emitted by the compiler and control time critical sections accurately.

Creating Segments and Local Variables

To link them into your C code you should design your assembler functions to closely resemble the design of C functions. That is, you will want them to have clearly defined boundaries, parameters, return values and local variables as is the case with C functions.

Often times assembler routines are written to pass parameters in a wide array of registers usually dependent upon which locations are available the first place the function is called. This leads to a tangled mess of parameter passing methods amongst the functions in an assembler program which quickly becomes difficult to maintain. Designing your assembler functions to look like C functions and adhere to the C51 conventions of passing data between modules will make your projects which are purely in assembler easier to read and maintain. Later, you will see that the code you write in assembler following such guidelines will be easy to interface to C code. If you generate your functions in the same style as the C compiler, you will reap another benefit: the linker will be able to perform overlay analysis on your DATA segments and will be able to link your segments together in the code space in the best manner.

In assembler, each of your functions can exist in its own segment in the code memory space. If you have local variables, then they can exist in a segment in the appropriate memory space (DATA, XDATA, etc.). For example, if you used some local variables that you needed quick access to, you could declare a DATA segment containing these variables. Similarly, if you have a lookup table that only this function needs access to, you could declare it in the function's CODE segment. The point is that only the current segment should have visibility to the temporary variables it uses as locals. In the following example, a function is defined which has several local variables allocated in the DATA memory space.

Listing 0-1

```
; declare the code segment for the function
?PR?IDCNTL?IDCNTL      SEGMENT CODE

; declare the data segment for local storage
; this segment is overlayable for linker optimization
; of memory usage
?DT?IDCNTL?IDCNTL      SEGMENT DATA OVERLAYABLE

                        PUBLIC  idcntl
                        PUBLIC  ?idcntl?BYTE

; define the layout of the local data segment
                        RSEG     ?DT?IDCNTL?IDCNTL
?idcntl?BYTE:
TEMP:                  DS       1
COUTNT:                DS       1
VAL1:                  DS       2
VAL2:                  DS       2

                        RSEG     ?PR?IDCNTL?IDCNTL
idcntl:                ...      ; function code begins here

                        RET
```

The labels assigned to the DATA segment for the function are used just like any other variable is in assembler. The linker will assign them physical addresses during the link phase. The overlayable attribute on the segment will allow the linker to perform overlay analysis on the internal memory with this segment. Without this attribute, the variables in the ?idcntl?BYTE segment will always be allocated, i.e: they will act like C static variables. This is not efficient usage of critical data memory unless the variables do not need to be statics.

Setting the Address of a Variable

Sometimes, it is desirable to fix a variable at a certain location in memory. This is especially useful in systems where an SRAM is initialized by a master device before the slave 8051 system is allowed to execute. In this case, both systems must agree upon a memory map and adhere to it or else mayhem will result when the 8051 attempts to use the initialized data improperly. Thus, the 8051 must assume that its variables are mapped to the correct locations. Keil C51 provides you with the capability to set the address of any variable in the system provided that you do not want to initialize the variable at compile time. For example, if you wanted to define an integer variable and initialize it with the value 0x4050 then you would not be able to force this variable to a certain location using C. Additionally, you cannot force bit variables which return a bit to a predetermined address. However, for variables which you do not need to initialize at definition time you can use the `'_at_'` keyword to specify the address of the variable.

The syntax for using the `'_at_'` keyword in C adds on to the existing syntax for declaring a C variable:

```
type [memory_space] variable_name _at_ constant;
```

As is normal for Keil C, if the memory space specifier is missing the default memory space as determined by the selected memory model is used. Thus if you are compiling using the small memory model the variable will be allocated in the DATA segment. An example of a C51 declaration using the `_at_` keyword is shown below.

```
unsigned char data byteval _at_ 0x32;
```

One of the nice things about the `_at_` keyword is that you can assign a variable name to your hardware input / output devices by setting the variable name at the address of your i/o device. For example, suppose that you had an input register at address 0x4500 in the XDATA segment. You could declare a variable name for it using the following statement in your C source code.

```
unsigned char xdata inpreg _at_ 0x4500;
```

Reading the input register is then a simple matter of using `'inpreg'` in a C statement as usual. Of course, you can still access the input register using the memory access macros provided by Keil. An example is shown in Listing 0-2.

When you want to force the address of a variable that has to be initialized, you can use more traditional methods in assembler code. Many times, you have a lookup table for which you can optimize the accesses by fixing the base location of the table to some address but which is initialized at compile time because it is in the CODE segment. For example, if you had a table of 256 bytes and wanted to access it quickly, you could fix the base address of the table to a multiple of 256 and then load DPH with the high byte of this address and DPL with the index of the desired element. This eliminates adding a base address to an offset and dealing with carry from the lower eight bits to the upper eight bits. Such an access can be seen in Listing 0-2

```
void myfunc(void) {
```

```
    unsigned char inpvai;
```

```
    inpvai=inpreg;                // this line and the next do
    inpvai=XBYTE[0x4500];         // the same thing
    ...
    if (inpreg & 0x40) {          // make a decision based on
        ...                      // the value of inpreg
    }
}
```

Listing 0-3.

Listing 0-2

```
void myfunc(void) {
    unsigned char inpval;
    inpval=inpreg;                // this line and the next do
    inpval=XBYTE[0x4500];        // the same thing
    ...
    if (inpreg & 0x40) {          // make a decision based on
        ...                      // the value of inpreg
    }
}
```

Listing 0-3

```
                ; get the high order byte of the table's
                ; base address
MOV     DPH, #HIGH mytable
MOV     DPL, index    ; add in the index
CLR     A
MOVC    A, @A+DPTR    ; read in the byte
```

Fixing a variable's address in a given segment is a simple matter of defining a segment which can not be relocated by the linker and specifying its starting address. The table referenced in the above code example can be forced to address 8000H by the following code. Additionally, an example of fixing a variable in the DATA space is shown.

Listing 0-4

```
                ; define this to be a code segment
CSEG     AT      8000H

mytable:    DB      1, 2, 3, 4, 5, 6, 7, 8
            ...      ; remainder of table definition

            DSEG     AT      70H
cur_field:  DS      1

            ...

            END      ; end of file
```

With this type of setup, a variable can be forced to any address by changing the CSEG keyword to the correct one for the memory space in which you wish to allocate your variable. These variables will be completely accessible by any C code you link with it provided that you give the correct 'extern' declaration in your C code. Given this information, the linker can correctly patch in the address of the variable.

This same sort of scheme is used to install interrupt service routines in the interrupt vector. An absolute segment can be used to place one function in the vector or place them all in it. Thus, if most of your code and ISRs are written in C, but you have one ISR that must be in assembler, it is a simple matter to force this ISR into the vector at the correct location. Listing 0-5 shows that the method to do this is very similar to assigning a variable to a fixed address location.

Listing 0-5

```
CSEG    AT    023H
LJMP    serial_intr
```

The ISR called from the vector is then defined to exist in a CODE segment just as any other function would be:

Listing 0-6

```
                ; define a relocatable segment
RSEG    ?PR?serial_intr?SERIAL

                USING    0        ; specify register bank
                ; 0 as active
serial_intr:    PUSH    ACC        ; start ISR
                ...            ; ISR code
                RETI            ; exit interrupt
```

Integrating C and Assembly Together

Let's assume that you have a project which has to perform an operation which does not easily lend itself to coding in C. It may be that this function must use packed BCD numbers efficiently and you feel that you can get more performance out of it by coding the function in assembly; or it may be that the function is time critical and you do not trust that you can time your C code properly. You decide that this function must be coded in assembly, however, you do not want to write the entire application in assembly just because one function must be coded in assembler. The answer to this is to code the one function in assembler and link it into the rest of your application just like you would do with any C function.

The assembler function should be given a segment name and definition which will make it compatible with the segments defined for the C functions. If you expect to pass data in and out of this function then you will have to ensure that the memory space used by your assembler function for receiving and returning values matches the space that the compiler and linker expect to use. A typical assembler function which can be called from C with no parameters has the following general format.

Listing 0-7

```
; declare a segment in the code space
?PR?clrmem?LOWLVL      SEGMENT CODE

; export the name of the function
                        PUBLIC clrmem

; this segment can be placed by the linker wherever it wants
                        RSEG      ?PR?clrmem?LOWLVL

;*****
;
; Function:      CLRMEM
; Description:   This routine will clear memory of the internal
;               RAM
; Parameters:    none
; Returns:       nothing.
; Side Effects:  none.
;*****
clrmem:                MOV     R0,#7FH
                        CLR     A
IDATALOOP:             MOV     @R0,A
                        DJNZ    R0,IDATALOOP
                        RET
                        END
```

The format of the assembler file is very simple. The segment which will contain the function is given a name. Since the segment will be in the CODE memory space, the convention used by the compiler is to have the name begin with ?PR. These first two characters maintain compatibility with the internal naming convention established by the C51 compiler. This convention is shown in Table 0-1.

The segment name is given the RSEG attribute. This means that the segment is relocatable which allows the linker to assign it to any physical address it needs to in the CODE memory space. Once the segment name is defined, the file must declare any public symbols and then define the function with the code. This is all that is involved with writing a simple assembly language function.

Memory Space	Naming Convention
CODE	?PR, ?CO
XDATA	?XD
DATA	?DT
BIT	?BI
PDATA	?PD

Table 0-1

Functions which receive parameters from the caller or return a value to the caller must observe certain rules for the passing of these values. For the most part Keil C passes parameters in internal RAM using the current register bank. This is also true of return values. When you write a function which receives more than three parameters, however, it is guaranteed that another segment in the default memory space must be created to pass the remainder of the parameters. Register assignment for the incoming parameters adheres to the rules in the following table.

Parameter #	Parameter Type			
	char, 1 byte ptr	int, 2 byte ptr	long, float	generic ptr
1	R7	R6, R7	R4...R7	R1, R2, R3
2	R5	R4, R5	R4...R7	R1, R2, R3
3	R3	R2, R3	N/A	R1, R2, R3

Table 0-2

The assembly function simply accesses these registers when it needs to use the value of the parameter. If these values are used and saved elsewhere or no longer needed, the register locations used to pass them to the function can be used as general purpose storage locations by the function. An example of a function invocation in C and the code which implements the function in assembler is shown below. You should note that functions which pass parameters via internal RAM have an understrike placed before their name by the compiler. Assembler functions should be named accordingly. Functions which take more than three parameters have some of the parameters passed to them in a parameter block defined in the default memory segment.

Listing 0-8

C code

```
// C declaration of the assembly function
bit devwait(unsigned char ticks, unsigned char xdata *buf);

// invocation of assembly function
if (devwait(5, &outbuf)) {
    bytes_out++;
}
```

Listing 0-9

Assembler code

```
; declare a segment in the code space
?PR?_devwait?LOWLVL SEGMENT    CODE

; export the name of the function
        PUBLIC _devwait

; this segment can be placed by the linker wherever it wants
        RSEG    ?PR?_devwait?LOWLVL

;*****
;
; Function:      _devwait
; Description:   This function waits a specified amount of timer 0
;               overflows for an external device to signal that
;               the data at P1 is valid. If the timeout is not
;               reached the data is written to a specified XDATA
;               location.
```

```

; Parameters:    R7 - holds the number of ticks to wait.
;               R4|R5 - holds the XDATA address to write to.
; Returns:      1 if the read was successful, 0 if it timed out.
; Side Effects: none.
;*****
_devwait:       CLR    TR0            ; set up timer 0
                CLR    TF0
                MOV    TH0, #00
                MOV    TL0, #00
                SETB   TR0
                JBC    TF0, L1        ; check for a timer tick
                JB     T1, L2        ; check if the data
                                    ; is ready
L1:             DJNZ   R7, _devwait   ; decrement the tick counter

                CLR    C            ; the unit timed out,
                                    ; clear the return bit

                CLR    TR0          ; stop timer 0
                RET

L2:             MOV    DPH, R4        ; read the output
                                    ; address and put it in
                                    ; the DPTR

                MOV    DPL, R5
                PUSH   ACC            ; save A from corruption
                MOV    A, P1         ; get the input data
                MOVX   @DPTR, A      ; write it out
                POP    ACC           ; restore A
                CLR    TR0          ; stop timer 0
                SETB   C            ; set the return bit
                RET
                END

```

The above code does something we have not discussed yet - it returns a value. In this case, the function returns a bit value based on the timeout. If the hardware timed out, the operation is considered to be a failure, and the value 0 is returned. If it did not, the input byte is written out to the specified address and the value 1 is returned.

When returning values from a function, C51 uses internal memory by convention, much in the way that function parameters were passed. The compiler will always use the current register bank to return data from a function to the caller. The C51 convention for returning data to a caller is specified in Table 0-3.

Return Type	Register(s) Used
bit	carry flag
(unsigned) char	R7
(unsigned) int	R6...R7
(unsigned) long	R4...R7
float	R4...R7
pointer	R1...R3

Table 0-3

Functions which return these types can be written to use the return locations as local variables until they must be set with the return value at the time of the function exit. Thus, if you have a function that must return a long value, you can feel free to use R4 through R7 for any sort of temporary usage that you may see a need for in your function. In this way, you do not have to declare segments in memory to hold your locals, and memory usage is that much more optimized. Your functions should not assume that they can use any of the default registers that are not used to pass parameters in or out of your function however.

Inline Assembly Code

Occasionally, you will have situations in which you want or need to code part of a function in assembler but you don't want to code the entire function in assembler or have to call a new smaller function coded in assembler. Typical situations are hardware control operations which require very specific operations or some dead time that can't be controlled using C. When you run into these situations, you can use the 'asm' pragma to insert the assembly code you want into the C code you are running through the compiler. Take the following function as an example:

Listing 0-10

```
#include <reg51.h>

extern unsigned char code newval[256];

void func1(unsigned char param) {
    unsigned char temp;

    temp=newval[param];
    temp*=2;
    temp/=3;

    #pragma asm
        MOV     P1, R7           ; write the value of temp out
        NOP                     ; allow for hardware delay
        NOP
        NOP
        MOV     P1, #0          ; clear P1
    #pragma endasm
}
```

The code inside the 'asm' / 'endasm' pragma pair will be copied into the output .SRC file when the compiler is invoked with the 'src' option on its command line. If you do not specify the 'src' option, the compiler will simply ignore the text inside the 'asm' / 'endasm' pair. It is important to note that the compiler will not assemble your code and put it in the object file that it emits, you will have to take the .SRC file and run it through the assembler to get the final .OBJ file. Running the above code through the compiler yields the following .SRC file.

Listing 0-11

```
; ASMEXAM.SRC generated from: ASMEXAM.C
```

```
$NOMOD51
```

```
NAME      ASMEXAM
```

P0	DATA	080H
P1	DATA	090H
P2	DATA	0A0H
P3	DATA	0B0H
T0	BIT	0B0H.4
AC	BIT	0D0H.6
T1	BIT	0B0H.5
EA	BIT	0A8H.7
IE	DATA	0A8H
RD	BIT	0B0H.7
ES	BIT	0A8H.4
IP	DATA	0B8H
RI	BIT	098H.0
INT0	BIT	0B0H.2
CY	BIT	0D0H.7
TI	BIT	098H.1
INT1	BIT	0B0H.3
PS	BIT	0B8H.4
SP	DATA	081H
OV	BIT	0D0H.2
WR	BIT	0B0H.6
SBUF	DATA	099H
PCON	DATA	087H
SCON	DATA	098H
TMOD	DATA	089H
TCON	DATA	088H
IE0	BIT	088H.1
IE1	BIT	088H.3
B	DATA	0F0H
ACC	DATA	0E0H
ET0	BIT	0A8H.1
ET1	BIT	0A8H.3
TF0	BIT	088H.5
TF1	BIT	088H.7
RB8	BIT	098H.2
TH0	DATA	08CH
EX0	BIT	0A8H.0

```

IT0    BIT    088H.0
TH1    DATA  08DH
TB8    BIT    098H.3
EX1    BIT    0A8H.2
IT1    BIT    088H.2
P      BIT    0D0H.0
SM0    BIT    098H.7
TL0    DATA  08AH
SM1    BIT    098H.6
TL1    DATA  08BH
SM2    BIT    098H.5
PT0    BIT    0B8H.1
PT1    BIT    0B8H.3
RS0    BIT    0D0H.3
TR0    BIT    088H.4
RS1    BIT    0D0H.4
TR1    BIT    088H.6
PX0    BIT    0B8H.0
PX1    BIT    0B8H.2
DPH    DATA  083H
DPL    DATA  082H
REN    BIT    098H.4
RXD    BIT    0B0H.0
TXD    BIT    0B0H.1
F0     BIT    0D0H.5
PSW    DATA  0D0H

?PR?_func1?ASMEXAM  SEGMENT CODE
EXTRN  CODE (newval)
PUBLIC _func1
;
; #include <reg51.h>
;
; extern unsigned char code newval[256];
;
; void func1(unsigned char param) {

    RSEG  ?PR?_func1?ASMEXAM
    USING 0

_func1:
;---- Variable 'param?00' assigned to Register 'R7' ----
                ; SOURCE LINE # 6

;   unsigned char temp;
;
;   temp=newval[param];

```

```
                                ; SOURCE LINE # 9
    MOV     A,R7
    MOV     DPTR,#newval
    MOVC    A,@A+DPTR
    MOV     R7,A
;---- Variable 'temp?01' assigned to Register 'R7' ----
;   temp*=2;
                                ; SOURCE LINE # 10
    ADD     A,ACC
    MOV     R7,A
;   temp/=3;
                                ; SOURCE LINE # 11
    MOV     B,#03H
    DIV     AB
    MOV     R7,A
;
; #pragma asm
    MOV     P1, R7              ; write the value of temp out
    NOP                                ; allow for hardware delay
    NOP
    NOP
    MOV     P1, #0              ; clear P1
; #pragma endasm
; }
                                ; SOURCE LINE # 20
    RET
; END OF _func1

    END
```

As you can see, the text in the 'asm' section of the function was literally copied into the file. This file now must be assembled and then linked with the other object files to produce the final executable.

Improving the Assembly Generated by the Compiler

Many software designers who come from the "old school" of embedded systems design believe that the assembler code they generate is far superior to the code that any compiler emits and therefore feel that their project is much better off if they avoid development in higher order languages and stick to their trustworthy assembler. For these engineers, the efficiency they believe they are gaining by using assembler far outweighs the reasons to switch to C which we discussed earlier. It is my belief that if these engineers wrote their code both in C and assembler code and compared the output of the compiler to their own assembler code they would be very surprised. There is no doubt that a good assembly coder will be able to outperform a compiler for minimal algorithms and other small pieces of code, however, the speed and efficiency at which a project can be developed in a higher order language far outstrips the speed of assembler code development.

For those of you who are wavering on the fence between C and assembler, let me offer you an option. The Keil C compiler features a compile time switch which forces the emission of an assembler code file which can be run through the A51 assembler and then linked with the other modules. This path is equivalent to using the object file generated directly by the compiler. The advantage to this option is that

you can edit the assembler file from the compiler and perform any and all of the code tweaks you want to improve code size and efficiency and then use this modified assembler file to link in with your project.

For the most part, you will not want to alter the assembler produced by the compiler, since it is well optimized. However, there are cases that can be improved. One of the above examples showed you how to force a lookup table to a certain location in the CODE memory. This was done so that when the table must be indexed into, only the low order byte of the DPTR has to be calculated. Let's consider the following table access from the clock project of the previous chapter.

Listing 0-12

```
char code bcdmap[60][2]={
    "00", "01", "02", "03", "04", "05", "06", "07", "08", "09",
    "10", "11", "12", "13", "14", "15", "16", "17", "18", "19",
    "20", "21", "22", "23", "24", "25", "26", "27", "28", "29",
    "30", "31", "32", "33", "34", "35", "36", "37", "38", "39",
    "40", "41", "42", "43", "44", "45", "46", "47", "48", "49",
    "50", "51", "52", "53", "54", "55", "56", "57", "58", "59"
};

void disp_time(void) {
    static char time_str[32]="TIME OF DAY IS: XX:XX:XX    ";
    unsigned char i;
    time_str[T_HOURLT]=bcdmap[timeholder.hour][0];
    time_str[T_HOUR]=bcdmap[timeholder.hour][1];
    time_str[T_MINT]=bcdmap[timeholder.min][0];
    time_str[T_MIN]=bcdmap[timeholder.min][1];
    time_str[T_SECT]=bcdmap[timeholder.sec][0];
    time_str[T_SEC]=bcdmap[timeholder.sec][1];
    putchar(0xFF);
    for (i=0; i<32; i++) {
        putchar(time_str[i]);
    }
    disp_update=0;                // clear the display update flag
}
```

As you can see, 'bcdmap' consists of only 240 bytes and therefore can be accessed using a single byte as an offset from the base address. In the clock project, the address of 'bcdmap' was not fixed at a 256 boundary and thus the compiler could not make any assumptions about accessing the table and had to deal with the entire base address plus the index into the table. The assembler code for the table access as written by the compiler is shown in

Listing 0-13.

Listing 0-13

```
;   time_str[T_HOURT]=bcdmap[timeholder.hour][0];
                                ; SOURCE LINE # 214
MOV     A,timeholder
ADD     A,ACC
ADD     A,#LOW bcdmap
MOV     DPL,A
CLR     A
ADDC    A,#HIGH bcdmap
MOV     DPH,A
CLR     A
MOVC    A,@A+DPTR
MOV     time_str?42+010H,A
```

This same piece of code is repeated for all six accesses into the lookup table. As you can see, the compiler emits code to add the offset into the base address of 'bcdmap' and builds up the new address of the desired byte in the DPTR register pair. A simple way to improve this code is to force the address of 'bcdmap' to a fixed location on a 256 byte boundary in the CODE segment. The access into the table will then need only concern itself with the lower 8 bits of the address into the table. This change is done by generating a simple assembler code file as shown in Listing 0-14 and linking it with the object file obtained from the existing C code. The initialization of the table must be removed from the C file since it is done here. The C file will now contain an extern declaration of bcdmap. The definition is filled in by the assembler file.

Listing 0-14

```
                CSEG      AT 0400H
bcdmap:         DB  '0' , '0'
                DB  '0' , '1'
                DB  '0' , '2'
                ...
                DB  '5' , '7'
                DB  '5' , '8'
                DB  '5' , '9'

                END
```

The assembler code emitted by the compiler can then be improved to utilize the new method for accessing the bcdmap lookup table. The new code is shown in Listing 0-15.

Listing 0-15

```
;   time_str[T_HOURL]=bcdmap[timeholder.hour][0];  
                                     ; SOURCE LINE # 214  
MOV     A,timeholder  
ADD     A,ACC  
MOV     DPL,A  
MOV     DPH,#HIGH bcdmap  
MOVC    A,@A+DPTR  
MOV     time_str?42+010H,A
```

The method of table access used by the compiler took 11 processor cycles and 17 bytes of CODE space. In comparison, the new method takes 8 processor cycles and 12 bytes of CODE space. There is a fair amount of improvement here given the fact that only ten lines of assembler code have been touched. If your goal in this design is to optimize speed, then the job has been done on this section of code. However, if the goal is to optimize CODE space usage then the code which is repeated for all six 'bcdmap' accesses can be written into a single function which is called six times. This greatly reduces the amount of CODE space used by the 'disp_time' function. The function as coded in assembler is shown in Listing 0-16 along with the new code for source line number 214.

Listing 0-16

```
getbcd:      ADD     A,ACC  
             MOV     DPL,A  
             MOV     DPH,#HIGH bcdmap  
             MOVC    A,@A+DPTR  
             RET  
  
;   time_str[T_HOURL]=bcdmap[timeholder.hour][0];  
                                     ; SOURCE LINE # 214  
MOV     A,timeholder  
LCALL   getbcd  
MOV     time_str?42+010H,A
```

The 'getbcd' function ends up in the CODE segment for the 'disp_time' function since it is the only function calling it. In this way, only the 'disp_time' has any knowledge of the 'getbcd' function.

In addition to performing optimization tricks that a compiler will ignore, such as the table access above, you can alter the compiler's output to eliminate unnecessary calls to low level service functions inserted by the C51 system. Once again recall the clock project from Chapter Four. That project contained a section of code which updated the current time of day and copied it to a buffer location for display. The structure which held the time of day had the following type definition.

```
typedef struct {                               // define a type to hold  
    unsigned char hour, min, sec;              // the time of day  
} timestruct;
```

As you can see the number of bytes in a structure of this type is only three. Consider this section of code emitted by the compiler to copy the data in one of these structures into another.

Listing 0-17

```
;    timeholder=curtime;
; SOURCE LINE # 327
MOV    R0,#LOW timeholder
MOV    R4,#HIGH timeholder
MOV    R5,#04H
MOV    R3,#04H
MOV    R2,#HIGH curtime
MOV    R1,#LOW curtime
MOV    R6,#00H
MOV    R7,#03H
LCALL  ?C_COPY
```

This code takes 16 cycles and 11 bytes just to get to the call to copy one structure into another. The call to C_COPY takes another 70 processor cycles to complete its work. Since the amount of bytes to be copied is so small, the most logical optimization to perform here is to copy the structures by hand, byte by byte. Doing this yields the following code which is a substantial improvement over the above code.

Listing 0-18

```
;    timeholder=curtime;
; SOURCE LINE # 327
MOV    timeholder, curtime
MOV    timeholder+1, curtime+1
MOV    timeholder+2, curtime+2
```

This code fragment does the same amount of work without calling any "helper" functions. It consumes six processor cycles and six bytes of CODE space.

Editing the assembler code emitted by the compiler may not always give such favorable results as the ones documented here, but it bears repeating that it is a way to ensure that you get the speed and ease of development in C, but the tight and efficient code generated by an expert assembly coder.

Simulating Multiple Interrupt Levels

Many times throughout my experience with the 8051 I have found myself wishing that the processor supported more than 2 levels of priority. It is frequently the case that an embedded system will have one interrupt such as a power off interrupt that must always be serviced, no matter what else the system is doing. This interrupt invariably ends up being set to high priority. Therefore, every other interrupt ends up sharing the low priority level even though they should also have some level of prioritization. The Intel 8051 data book gives an example of a simple method by which assembler can be used to implement a system trick which will simulate a third level of interrupt priority. The Intel scheme requires that the first two levels of interrupt priority be done via the normal interrupt structure of the 8051. Those interrupts which are to have the highest level of priority are assigned to priority one and later reenabled during the ISRs of other priority one interrupts. A sample segment of code to implement this is shown in Listing 0-19.

Listing 0-19

```
        PUSH    IE          ; save the current IE value
        MOV     IE, #LVL2INTS ; enable only priority 2 ints
        CALL    DUMMY_LBL    ; call a bogus RETI

        ...                ; ISR code goes here

        POP     IE          ; restore IE
        RET

DUMMY_LBL: RETI             ; the false exit interrupt
```

The theory behind this code is simple. It saves the existing state of the interrupt system by pushing IE onto the stack. Only desired priority level 2 interrupts are allowed by changing the value assigned to IE. A false return from interrupt is then called to allow the hardware to generate more priority level one interrupts. Of course the only interrupts that are now allowed are those that have been given "level two" status.

This implementation allows you to extend the capabilities of the 8051 without any changes to the hardware system such as addition of PICs (Programmable Interrupt Controllers) and such. The additional software has a minimal effect on the capability of the ISR to respond to the interrupt. The overhead involved is 10 processor cycles per interrupt. Most systems will be able to withstand the added latency. The approach of simulating a third interrupt priority level in software is easily extended to allow each interrupt to be given its own level of priority if so desired. All that is involved is to place this code at the beginning of each ISR and ensure that the mask written to IE in the ISR allows only the desired interrupts to occur and strips out the remaining interrupts.

Suppose that you have a system that for one reason or another has to have each of the possible interrupt sources have its own level of priority. Thus, your system will need five different levels of interrupt priority! Assume that the interrupt sources are assigned to the priority levels as shown in Table 0-4.

Level	Source
0 (lowest)	Timer 0
1	External Interrupt 1
2	Serial Interrupts
3	Timer 1
4 (highest)	External Interrupt 0

Table 0-4

Implementation of this scheme will follow along the lines of the above example. In this case, you must carefully choose the mask values that you use for IE in each ISR to allow only higher priority interrupts. In other words, the value used in the serial interrupt ISR should only allow timer one and external one interrupts - it can not allow any other sources to be serviced. One simplifying fact in this scheme is that you do not have to deal with the added code for the timer zero interrupt since it is the lowest priority, and the external interrupt 0 since it is the highest priority.

The initialization code of this project must set the timer 0 interrupt to priority level zero in the IP register. All the remaining interrupt sources must be set to priority level one. For interrupt level one through interrupt level three you must establish the mask to be set into the IE register. These are shown in Listing 0-20.

Listing 0-20

```
EX1_MASK      EQU      99H      ; allow serial, timer 1 and ext 0 intrs
SER_MASK      EQU      89H      ; allow timer 1 and ext 0 intrs
T1_MASK EQU    81H              ; allow ext interrupt 0
```

At this point, the priority level simulation code can be included at the beginning of each ISR in the system. The following listing shows the assembly language prologue of each of the ISRs. The timer 0 interrupt and the external interrupt 0 ISRs are written entirely in C since they do not require any sort of trickery to execute properly with the rest of the system.

Listing 0-21

```
?PR?EXT1?LOWLVL      SEGMENT CODE

EXT1:                PUSH    IE                ; save the current IE value
                     MOV     IE, #EX1_MASK    ; enable only priority 2 ints
                     CALL    DUMMY_EX1        ; call a bogus RETI
                     LCALL   ex1_isr          ; ISR function in C
                     POP     IE                ; restore IE
                     RET
DUMMY_EX1:           RETI                      ; the false exit interrupt


?PR?SINTR?LOWLVL     SEGMENT CODE

SINTR:               PUSH    IE                ; save the current IE value
                     MOV     IE, #SER_MASK    ; enable only priority 2 ints
                     CALL    DUMMY_SER        ; call a bogus RETI
                     LCALL   ser_isr          ; ISR function in C
                     POP     IE                ; restore IE
                     RET
DUMMY_SER:           RETI                      ; the false exit interrupt


?PR?TMR1?LOWLVL      SEGMENT CODE

TMR1:                PUSH    IE                ; save the current IE value
                     MOV     IE, #T1_MASK    ; enable only priority 2 ints
                     CALL    DUMMY_T1        ; call a bogus RETI
                     LCALL   tmr1_isr        ; ISR function in C
                     POP     IE                ; restore IE
                     RET
```

```
DUMMY_T1:      RETI                      ; the false exit interrupt
```

With a small amount of assembler code, the system now has functionality far beyond that which the hardware provides. The best part of this design is that the majority of the system can still be coded in C. As always, the code for any ISR can be done in assembler and inserted in the above skeleton in place of the long call to the C routine.

Low Level Timing Issues

Many times you will have portions of assembler code which must perform tasks that require specific timing. If the timing required is sufficiently tight, these routines have to be coded in assembler. Quite frequently, such timing will have to be accurate to one or two processor cycles. In cases like this the simplest way to ensure your assembler code is correctly timed is to maintain a processor cycle count in the comment section. In this way, the code is easier to design and the count necessary is documented along with the code. When the code changes, the effects of the change on the timing are immediately seen and can be taken into account for.

As an example, suppose that you had to clock out a train of bits serially on pin T1. Another system is monitoring the output and sampling it at a rate of 100kHz. Each bit value must be prefaced by a high-low transition of 2μs and then the bit value must be presented for 3μs. The remainder of the time T1 should be held low. The timing diagram for this is shown in Figure 0-1.

The software executes on a system using a clock rate of 12MHz and thus has an instruction cycle time of 1μs. It would be difficult to guarantee such timing constraints using a C function, so the routine must be written in assembler. As its parameter it will receive the byte to ring out (MSB first) on pin T1 when called. The function is shown in Listing 0-22 below.

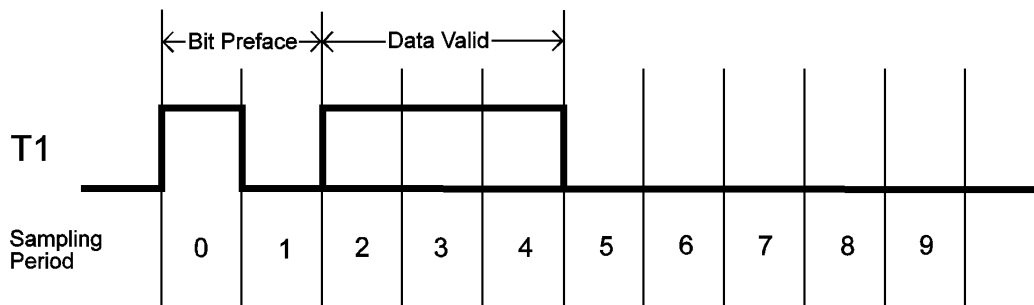


Figure 0-1 - Signal Timing

Listing 0-22

```
; this function has the following C declaration:
;      void sendbyte(unsigned char);

?PR?_sendbyte?SYS_I0  SEGMENT CODE
?DT?_sendbyte?SYS_I0  SEGMENT DATA OVERLAYABLE

        PUBLIC  _sendbyte
        PUBLIC  ?_sendbyte?BYTE

        RSEG    ?DT?_sendbyte?SYS_I0
?_sendbyte?BYTE:
BITCNT: DS      1
```

```
RSEG    ?PR?_sendbyte?SYS_IO
_sendbyte:  PUSH    ACC            ; save the accumulator
            MOV     BITCNT, #8     ; set to ring out 8 bits
            MOV     A, R7          ; get the output pattern

            RLC     A              ; get the first bit in C

LOOPSTRT:  JC      SETHIGH ; 2, 9  check the output value

            SETB    T1             ; 1, 0  set the preface
            CLR     T1             ; 1, 1  clear the preface
            RLC     A              ; 1, 2  get ready for the next bit
            NOP     ; 1, 4  burn extra time
            NOP     ; 1, 5
            NOP     ; 1, 6
            DJNZ    BITCNT, LOOPSTRT; 2, 7  check if we're done

SETHIGH:   SETB    T1             ; 1, 0  set the preface
            CLR     T1             ; 1, 1  clear the preface
            SETB    T1             ; 1, 2  set the data bit
            RLC     A              ; 1, 3  get ready for the next bit
            NOP     ; 1, 4
            CLR     T1             ; 1, 5  clear the output
            DJNZ    BITCNT, LOOPSTRT; 2, 7  check if we're done

            POP     ACC            ; restore the accumulator
            RET

            END
```

As you can see, the timing comment for each line consists of the number of cycles taken to execute that line of code followed by the total number of cycles used. Since this system is based on a ten cycle loop (because each bit time is ten cycles), the cycle count ends up being modulo 10 (i.e.: it counts from 0 to 9). It does not matter which instruction you choose to be the reference (count == 0) instruction as long as you are consistent in using that point in the timing everywhere. In this case I have chosen my reference instruction to be that point in the code where the preface bit is pulled high. You will notice that the count for both of these places in the code is set to 0. It is critical that if you have two paths through a timed loop such as this one that you make sure that both paths take an equal amount of time to execute - thus, one of the paths above contains more NOPs than does the other path to balance the number of cycles.

The above example code works great as long as there is no change in the oscillator frequency of the system it executes on or in any of the hardware which must monitor this system. For the most part it is safe to assume that the oscillator frequency will not be changing. However, let's assume that you are not producing the equipment which monitors the output of T1. Instead, some other company of "geniuses" has been contracted to this. They seem to be having some trouble getting their system to work at the 100kHz rate chosen. Due to design flaws "beyond their control" they correctly see the preface bit, but can not get their system to sample the signal in time to read the data bit that follows it. Their request is

that you hold the data bit longer so that they may have a chance to sample it. In this case, you know that the other engineers will probably want to change the hold time more than once. To avoid having to retime the assembler code each time they call with a new hold duration, you should rewrite the function to allow the caller specify the number of processor cycles the data bit should be held.

While this will make future changes a lot easier to do, it will certainly complicate matters for the assembler function. What this means is that the function which is called will have to have some way to kill a varying amount of time to allow for the varying hold time of the data bit. The function has now been changed to use a delay loop to kill the required amount of cycles between clocking the data bit out and freeing the signal.

Listing 0-23

```
?PR?_sendbyte?SYS_IO  SEGMENT CODE
?DT?_sendbyte?SYS_IO  SEGMENT DATA OVERLAYABLE
?BI?_sendbyte?SYS_IO  SEGMENT BIT OVERLAYABLE

                PUBLIC  _sendbyte
                PUBLIC  ?_sendbyte?BYTE
                PUBLIC  ?_sendbyte?BIT

                RSEG    ?DT?_sendbyte?SYS_IO
?_sendbyte?BYTE:
BITCNT:        DS      1
DELVAL:        DS      1

                RSEG    ?BI?_sendbyte?SYS_IO
?_sendbyte?BIT:
ODD:           DBIT    1

                RSEG    ?PR?_sendbyte?SYS_IO
_sendbyte:     PUSH    ACC                ; save the accumulator
                MOV     BITCNT, #8        ; set to ring out 8 bits

                CLR     C
                MOV     A, R5              ; get the number of cycles
                                          ; to delay
                CLR     ODD                ; assume that delay is even
                JNB     ACC.0, P_EVEN
                SETB    ODD                ; delay turned out to be even
                DEC     ACC                ; for odd numbers of delay,
                                          ; remove one cycle to account
                                          ; for the extra NOP
P_EVEN: SUBB     A, #4                    ; subtract out the overhead
                                          ; of the delay loop
                RR      A                  ; divide by 2 to get the number
                                          ; of extra DJNZs to execute
                MOV     DELVAL, A
```

```
MOV    R5, A
JNB    ODD, SEND_EVEN

SEND_ODD: MOV    A, R7          ; get the output pattern
          RLC     A             ; get the first bit in C

LOOP_ODD: JC     SETHIGH_0      ; 2, 9  check the output value

          SETB    T1           ; 1, 0  set the preface
          CLR     T1           ; 1, 1  clear the preface
          RLC     A             ; 1, 2  get ready for the next bit

          NOP                     ; 1, 3
          NOP                     ; 1, 4
          MOV     R5, DELVAL     ; 2, 6
          DJNZ    R5, $          ; 2, 8
          NOP                     ; 1, 9
          DJNZ    BITCNT, LOOP_ODD ; 2, 11 check if we're done

SETHIGH_0: SETB    T1           ; 1, 0  set the preface
          CLR     T1           ; 1, 1  clear the preface
          SETB    T1           ; 1, 2  set the data bit
          RLC     A             ; 1, 3  get ready for the next bit
          NOP                     ; 1, 4
          MOV     R5, DELVAL     ; 2, 6
          DJNZ    R5, $          ; 2, 8
          CLR     T1           ; 1, 9  clear the output
          DJNZ    BITCNT, LOOP_ODD ; 2, 11 check if we're done

          POP     ACC           ; restore the accumulator
          RET

SEND_EVEN: MOV    A, R7          ; get the output pattern
          RLC     A             ; get the first bit in C

LOOP_EVEN: JC     SETHIGH_E      ; 2, 9  check the output value

          SETB    T1           ; 1, 0  set the preface
          CLR     T1           ; 1, 1  clear the preface
          RLC     A             ; 1, 2  get ready for the next bit
          MOV     R5, DELVAL     ; 2, 4
          DJNZ    R5, $          ; 2, 6
          NOP                     ; 1, 7
```

```

NOP                                ; 1, 8
DJNZ  BITCNT, LOOP_EVEN ; 2, 10  check if we're done

SETHIGH_E:  SETB  T1                ; 1, 0  set the preface
             CLR   T1                ; 1, 1  clear the preface
             SETB  T1                ; 1, 2  set the data bit
             RLC   A                 ; 1, 3  get ready for the next bit
             MOV   R5, DELVAL        ; 2, 5
             DJNZ  R5, $             ; 2, 7
             CLR   T1                ; 1, 8  clear the output
             DJNZ  BITCNT, LOOP_EVEN ; 2, 10  check if we're done

             POP   ACC                ; restore the accumulator
             RET

END
```

The function first records whether the required delay will be odd or even. It then must determine the amount of iterations of a DJNZ loop it needs to execute to kill the requested delay. To do this, the minimum overhead of the delay loop (4 cycles for even delays and 5 cycles for odd delays) is subtracted out of the total delay. The remaining number is then divided by two to get the number of DJNZ instructions to run. The divide by two is done because the DJNZ instruction takes two cycles each. Since the minimum delay in the signal is now six cycles, the calling function must take this into account.

You now have a function which will meet the ever changing requirements of the other project. The code is easier to update since the program only has to be changed in the C code and not retimed in the assembler code. If this were really an advanced project, the delay would be received from the part of the project that monitors the output signal on pin T1. When changing the delay value, you must bear in mind that increasing this number will lower the amount of data that can be transferred by this system per unit time.

Conclusion

In general you can make the number of times you must code in assembler far and few between, but just because you are using C does not mean that you do not have the option of using assembler with it when required. The whole point of this chapter has been to demonstrate to you that assembler still has its definite place in systems development. The high level language is there for you to use as a tool to quickly and reliably develop your product. When you need to fine tune it or squeeze out that extra little bit, look to the tools and concepts discussed here to help you along.

- System Debugging

Introduction

Contrary to what many engineers would like to believe, there is not a set way to debug an embedded system. The complexity added by hardware interfaces and time constraints makes embedded systems far more complex than their PC or mainframe application counterparts. These systems can easily be run in a debugger or software probe type of environment and can easily be single stepped at the user's own pace. On the contrary, embedded systems often have to be run on the target hardware at full speed to complete their debugging. This means that the closest you can get to a debugger is an ICE. You will not be able to single step your way through the time critical portions of your code because the single stepping will mess up all the timing of the code. Systems which depend on a periodic tick for their timing or use a watchdog timer to reset the system are even more difficult to debug.

Because of the complexities in embedded systems, many debugging approaches exist. This chapter will explore some of these approaches and should be used not as a handbook to system debugging but as a starting point to give you ideas upon which to build when designing and debugging your own systems.

Designing the System to Aid Debugging

The success of your integration and debugging efforts can often be enhanced by designing in the appropriate features during the design phase. It is useful to have a spare serial port to dump debug information out of. Short of this, it is also useful to have a set of I/O pins that can be changed to reflect program state or variable state during different points of the program. The trouble with such approaches is that it often adds hardware to the system that would not normally be there. The argument for it however, is that there will be room to expand the capabilities of the system later if need be. In cases where neither of these options has been available to me, I have either dumped debug information to a display panel or recorded it in memory and downloaded it after the program had completed whatever test was in progress.

Regardless of the method you choose to debug your system, it will pay off big dividends to allow for some I/O capability as spare or for debugging use. During initial phases of hardware design you should ensure that a wire wrap board is built with these extended I/O capabilities and do as much of the integration and final testing of the system as you can at that point in the project. Obviously, you will have to perform more integration and testing once PCBs are made and, at this point your debug ports may be gone, but at least you will have eliminated most of the major problems in the system. Once you get to the point of debugging with a PCB you will find yourself using such tools as an oscilloscope and a logic analyzer to debug your system. You should not become attached to using the ICE. While the ICE is the most convenient way to debug your system, there is not always one available. I have worked in situations where there was no ICE at the entire company. At other times, there was only one ICE that everyone fought over. Rather than wait in a queue for the in circuit emulator to come free from another project, I learned to debug my systems without the aid of such luxurious tools.

Debugging without an ICE quickly makes you adept at using a digital storage scope. While this skill alone will not help you debug and understand your system, it will get you on the way. If you have a fairly good idea of what your system should be doing to and with the hardware at certain points, you can use a scope to determine where in the code the processor is at any given point or where in the code the processor went into la-la land or executed your logic error. Once you can determine these points, you can use techniques like inserting debugging statements to dump data to a display, serial port or I/O pin to help pinpoint the problem. At this point you can also fall back on the trusty simulator.

Using a Debug Port

One of the most basic approaches to getting inside your executing embedded system without an ICE is to dump data from a debug port. Typically, such data would include system events, debug statements indicating that the program had reached a certain point in the program, values of variables, etc. The debug port is typically a serial port that is either dedicated to debug information or is multiplexed in function between debug data and normal interface data for the system. The trouble with an 8051 based system is that there is typically just one serial port available for use which means that the port will have to be multiplexed. If you have the luxury of using one of the derivatives with two serial ports, take advantage of it and you will not have to worry about any of your debug data affecting the normal data flowing through the serial port.

A serial debug port begins to run into trouble when you wish to dump data to a PC using 10 bit frames, but your system normally uses the serial port in another mode or at a non-standard data rate. In this case, you will find that a debug port may be too obtrusive to the system when compared with the insight that it provides. Additionally, the extra overhead of dumping data out of a serial port may affect the timing of your system internally and cause bugs that would not normally be there or, even worse, mask bugs that would normally be there.

Debug ports are best suited in systems which do not place the highest premium on processing time and which have a spare serial port. However, this should be obvious to you from this discussion. The real time clock example from Chapter 4 would be a prime candidate for this type of debugging namely because of its spare serial port and bounty of spare processing time. If you were to use a debug port type of system with this example, the code would be interrupt driven and output debug data from a ring buffer. An example of a driver to do just that is shown in Listing 0-1.

Listing 0-1

```
#include <reg51.h>
#include <intrins.h>

#ifndef NULL
#define NULL ((void *) 0L)
#endif

#define DB_MAXSIZE      0x20

unsigned char db_head, db_tail, db_buffer[DB_MAXSIZE];

/*****
Function:      debug_init
Description:   Sets the serial port up for debug use and resets
               the ring buffer pointers to 0.
Parameters:   None.
Returns:      Nothing.
*****/
void debug_init(void) {
    SCON=0x90;                // use serial mode 2 and dump to
                             // another 8051 at high speed

    db_head=db_tail=0;        // set the head and tail to the
                             // base of the ring buffer
}
```

```
    ES=1;                                // allow the serial interrupt
}

/*****
Function:      debug_insert
Description:   Copies the contents of the memory block pointed to
               by the first argument into the ring buffer.
Parameters:   base - pointer. Indicates the point in memory from
               which to copy data into the ring buffer.
               size - unsigned char. Indicates the number of
               bytes to be copied from the memory block.
Returns:      Nothing.
*****/
void debug_insert(unsigned char data *base, unsigned char size) {
    bit sendit=0;                        // flag to indicate if a serial
                                        // transmission must be
                                        // initiated

    unsigned char i=0;
    if (!size || base==NULL) { return; } // check for NULL buffer
    if (db_tail==db_head) {              // if these two are equal before
                                        // insertion, the ring buffer is
                                        // empty

        sendit=1;
    }
    while (db_tail!=db_head && i<size) { // copy bytes while the
                                        // buffer has space and the
                                        // block has data

        db_buffer[db_tail]=base[i];     // copy the current byte
        i++;
        db_tail++;                      // move the pointer
        if (db_tail==DB_MAXSIZE) {      // check for pointer rollover
            db_tail=0;
        }
    }
    if (sendit) {                       // if a byte needs to be sent,
        SBUF=db_buffer[db_head];         // do it
    }
}

/*****
Function:      debug_output
Description:   ISR for the serial port. Increments the ring
               buffer head pointer and sends out the next byte if
*****/
```

```
the pointer has not caught the tail pointer.
Parameters:    None.
Returns:       Nothing.
*****/
void debug_output(void) interrupt 4 {
    RI=0;                      // clear RI no matter what
    if (_testbit_(TI)) {       // check and clear TI
        db_head++;             // advance the head pointer
        if (db_head==DB_MAXSIZE) { // watch out for rollover
            db_head=0;
        }
        if (db_head!=db_tail) { // check for more data in the
                                // buffer
            SBUF=db_buffer[db_head]; // send the next byte
        }
    }
}
```

You can see that data blocks are inserted into the ring buffer by calling a manager function which takes a pointer to a memory block holding outgoing data and a count which indicates the number of bytes that are in this block. The data is then copied into the ring buffer and the tail pointer of the buffer is updated accordingly. You can vary the size of the ring buffer depending upon the amount of RAM you have to work with and the amount of debug data you are dumping from the program. The RTC program from Chapter 4 does not have an external RAM and thus must implement the ring buffer in internal RAM. This severely limits the amount of data that can be queued up at one time. Fortunately, the program is relatively small and does not execute at blinding rates. Therefore, the ring buffer should be of sufficient size for this application.

Using Monitor-51

If you have some flexibility in the design of the development board for your system, you might want to request that it be designed such that you can perform both read and writes to the CODE memory space. This means that the system will have to tricked into believing that it now has only one memory space instead of two. In other words, the system will be Von Neumann wired to allow this. The advantage to this is that you can download your code to the memory using a simple driver program on the 8051. This will get you out of the painful cycle of making a code change, compiling it, burning it on a EPROM, testing it and repeating.

If your system can be set up to write to the CODE space, you should consider using the Monitor-51 program that comes with the Keil C51 package. This program will allow you to run your code on the target and perform many of the debugging functions that an ICE would let you do. This package requires that you load a communications and control module into the CODE space with your software. This module will communicate via your 8051's serial port to a PC. On the PC you run another Keil program called MON51.EXE. This program acts as an interface between you and your target system. Its character based approach to the interface is not the prettiest thing in the world, but it effectively gets the job done. If you are looking for a poor man's emulator, look no farther than the Keil Monitor package.

The Monitor program will let you view various memory segments and alter their contents. You can check the values in the SFRs, disassemble your code, and add new code using an in-line assembler. You can also set execution break points, and run the program in real time to these points. Once the system is halted, you can single step through your assembler code instruction by instruction. You should note that many of these features require that your system be able to read and write to the CODE memory segment.

Monitor can be configured to run on systems which do not have an SRAM in the CODE memory space, but you will lose the capability to set break points, single step the program and change the contents of the CODE memory space. Monitor-51 will need to take over control of your serial port and one of your timers (unless you have an 80515 or 80517 controller). However, these are probably the most serious of its requirements. Otherwise, you will have to give up 2816 bytes of the CODE space and 256 bytes of the XDATA space, but neither of these amounts is high at all.

You will have to adjust the Monitor-51 install.a51 file to suit your systems needs. It comes set up to use the serial port at 9600 baud if you have an 11.059MHz system. Additionally, it will want to push all of your interrupt vector above address 8000H. If this is not what you want, there is a constant at the beginning of install.a51 that you can change. All of this, and the operation of the control interface on the PC is well documented in the Keil Software manuals, and thus will not be repeated here. If your company can not afford an emulator, I strongly suggest that you look into using the Monitor package to aid you in testing and debugging your system.

Using I/O Ports for Debug

If you are not fortunate enough to be able to use the serial port as a debug port, another popular method to gain insight into an embedded system is to use discrete I/O lines. These can be anything from the port pins of the 8051 (for example port 1) to a 74373 data latch mapped in the external memory space. At the very least you should be able to find a few spare pins to use to indicate the internal execution point of your system. Obviously, the best thing for you as a system integrator is to have eight pins that you can use to show one byte at a time. It is nice to be able to connect these pins to LEDs and such so you can get a visual indication of the state of the lines, but you can also use an oscilloscope, if you are stuck using the more painful route.

Most of the systems I have had to integrate have been done using output pins to indicate the state of something. Oftentimes, one pin is used to indicate that the system is actually executing. To do this the pin is toggled at some set frequency, and all you have to do is check the pin for a given period of time to know that the system is still running and is fairly stable. Other pins are used to indicate that the program has passed some point in the code or is in a given state waiting for input for example. You can also latch the value of a register to these output pins and then stop the program waiting for an outward indication that it is okay to proceed. The main point is that there is no set way to use the output pins to debug the system. You will have to determine the usage for each pin at each stage of your debugging process. One thing that usually happens is that you end up debugging the system piece by piece. What I mean to say by this is that you will only be able to see enough information to debug one problem at a time. This is contrasted by a serial debug port which will allow you dump a fair amount of data and gather information on many portions of the program.

If you are lucky enough to have access to a logic analyzer you can run the outputs you are using to debug the system into the analyzer and have it record what it sees. In this way you will be able to dump more information to the port than you could if you had to monitor the entire system using an oscilloscope. If you end up using just a scope and your wits to debug the system, take heart because it can be done. In fact, most the projects I have worked on have not had any tools other than a scope to use in the debugging cycle. This just makes those projects with an ICE all the more pleasurable!

Using the ICE

This section can not and will not provide a discussion of how to operate the various in circuit emulators available for the 8051. There are too many varieties and too many updates to the user interface coming out for a book like this to handle. Furthermore, any operational issues regarding the ICE you have or are evaluating should be covered by operational manuals and technical support personnel. This section will, however, give you a few pointers on the use of the ICE.

The first and most obvious point is to prepare your code for debugging in the ICE by recompiling all source files (including assembler files) using the 'debug' option. For C files which are using structures or arrays that you will wish to access, alter or examine by index compile them with the 'objectextend' switch. This will force the system to place all the necessary debug information in the object files produces and ultimately in the executable file produced. This may sound like an intuitively obvious point, but you would be surprised to find out how many new comers dump their code into the ICE without debug information and are shocked when their C code does not appear but is instead replaced by assembly code lacking any symbols at all.

When installing the ICE into your system you should pay careful attention to the settings of the ICE pod. Many of these pods allow you run the ICE off of your system's clock or a built in clock. If you have code that is at all dependent upon the frequency of the oscillator, it is necessary that you make the ICE use your system's oscillator otherwise you will find that your code does not exhibit the behavior that you expected at all. Any sort of timer generated system ticks will occur at the wrong rate, and if you use the serial you will find that your receiving system gets nothing but garbage because of baud rate errors.

Similarly, you should force the pod to run off of your system's power and reset signals. This way, any external watchdog timers or glitches in the system hardware will affect the ICE and you will get warning of these occurrences so they can be handled properly. Additionally, using the hardware's power supply to run the pod will allow you to test that much more of the hardware.

When debugging systems that have a watchdog or periodic system tick (whether from a timer or external source) keep in mind that single stepping the system or employing breakpoints will not stop these timers. They will keep running while you are examining code or data and once the program is single stepped again, their interrupt will be fired. Oftentimes this provides a source of great inconvenience. Usually it is helpful to disable watchdog timers until the final stages of testing. As for system ticks, a good practice is to disable their interrupt as soon as code execution is halted. This way, you won't have to run through your tick ISR every time you want to single step code in another section of the program.

If you are evaluating ICE's for purchase, I strongly suggest that you buy one with a trace buffer. Many products offer trace buffers from 16K all the way up to 128K frames. These buffers store the instruction executed, the value of the instruction pointer, the port pins, and can also store the value of other input/output pins that you connect to the pod. The trace buffer will more than pay for itself the first time you use it to find the source of a system crash. Using a trace buffer, this can usually be done in a fraction of the time taken by other methods. The ability of many emulators to record the values of data lines that you specify or connect to the pod will give you some of the capability of a logic analyzer without the extreme cost of one.

Conclusion

If you can get regular use of an ICE, it will prove to be a very powerful and valuable tool in your system testing and integration efforts. However, it is important that you not become completely dependent upon the ICE for debugging the system. Eventually, you will face the situation where there is no ICE or the ICE is of little use in debugging the system. Such is often the case in systems which must control hardware components like EEPROMs or custom hardware interfaces and the signal timing is critical. In these cases you will have to be able to use a scope or some other tool to instrument your system.

This brief chapter should have provided you some insight into the other tools available for your use in debugging your projects. It is important to reiterate that the methods for debugging a real time system, or any system for that matter, are not learned from a book. They are learned by experience. Books can only point you in certain directions which may or may not be useful to you. The purpose of this chapter has been to provide you with some tools and concepts that you can apply in the different situations you face.

- Interrupt System Issues

Introduction

This chapter will discuss several system design issues that you should be aware of when designing a real time system. Many of these issues will deal with software design only, but others will have a bearing on hardware design. The main hardware focus in this chapter will be in the area of system interrupts. That is, it will discuss the main differences between using an interrupt to trigger some action and using a polling method to trigger some action. Software design issues will focus on program structure and foundation. With no further ado, let's get into it.

Interrupt Driven Systems vs. Polled Systems

During the design of an embedded system which does a large amount of processing based on input signals you will find yourself trying to decide if a given signal should be polled or if it should trigger a system interrupt. The answer to this question is fairly straightforward. A good approach involves examining two aspects of the system. First, consider how fast the system must respond to a state transition of the input signal. If the maximum response time is very low, then the signal must be allowed to generate an interrupt instead of being polled. No matter how well your system polls an input, its average response time will always be slower than an interrupt driven system. Second, consider how quickly you expect the input signal to change state. If it is a signal that will toggle at a frequency that approaches 1/10th of the instruction cycle frequency, then you should look to have the 8051's interrupt system monitor the signal because of the tight polling loop that is otherwise required.

Obviously, there will be exceptions to the above two cases. Most notably are systems where there are many input sources each of which seems to require an interrupt. In these cases you must either work out a scheme to share the interrupts amongst the input sources or relegate some of them to being polled. In other words, there must be some sort of ranking by importance established for the input signals. For example, if you have an engine management system which must monitor the sensor signals coming from the engine as well as respond to status requests from a master CPU you would most certainly rate the sensor inputs above the status request. Missing a status request would probably not be catastrophic whereas missing a change in input state from a critical sensor may prove to be very costly to the engine. There is no way for anyone to tell you how to determine the importance of your input signals, only you and your design team can decide such things. These decisions must be based on analysis from a systems standpoint, not just a software or hardware point of view.

Once you have established the ranking amongst your system inputs, you must determine the points at which you will bring these inputs into the processor. When you are dealing with polled inputs, the ones which require quick access should obviously be brought directly to the processor via one of its port pins (usually port one). This will allow for minimum access times to the signal since the port pins can be read in one instruction cycle. Signals which do not require such speedy access can be interfaced to the processor via an input latch through the system bus. From earlier discussions, you should remember that this will take two processor cycles once you have the DPTR set to the correct address. When deciding upon the placement of these signals, you may also want to consider the frequency of access to them. If you have a signal that you will be polling 10000 times a second and one that you will be polling once every other second, it will not make sense to interface the first signal to the system via the bus while the second input is attached to one of the port pins due to the total amount of time that will be spent monitoring each signal. In general, you must sample your signal at a rate that is twice your expected frequency if you expect to respond to any changes in signal state.

The method you use to poll your input signals varies based upon the signal and the actions associated with its state changes. If you have inputs which relate to a human interface then it will suffice to poll them at a rate of 10Hz. A human is a relatively slow I/O device, and you will find that when they are using things like switches and buttons to interact with your system they do not notice the difference between polling at 10Hz or polling at some faster rate. A user interface signal can thus be polled in a tick

interrupt or response section to a tick in the main loop. Examples of such approaches were discussed in Chapter 4.

When you are dealing with signals that change state at a very fast rate or which are of high importance you have two approaches. You can still use a software tick routine to poll the signal, however, you will need to cause the tick interrupt to occur at a much higher frequency. Otherwise you can make the processor constantly sample the signal when it is not performing any other tasks. The drawback to both of these approaches is that they both use up more of the available processing power of the system.

Increasing the tick rate does this because the system must process more interrupts. Constantly polling the signal does this because any spare processing time is now tied up in monitoring inputs manually. If you are designing a system which must run off of battery power and places a premium on battery life, neither of these approaches are very attractive. In these cases, you must look at your inputs and rank them in order of importance. The higher speed, more critical inputs should be interfaced to the processor via the external interrupts, the remainder of the signals can be polled.

The main purpose behind this is to allow you to establish an internal priority scheme between the interrupts and the actions triggered by the polled inputs. I typically put my most important interrupt signals on the INT0 and INT1 pins in order of significance. There is no specific reason for this other than simple convention. If you need more than these two distinct external interrupt sources, there are ways to expand the number of external interrupt using other port pins of the 8051 or sharing the two external interrupts that exist. These topics will be discussed later in this chapter. For now, you should remember that your signals which will require immediate reaction by your system should be given the ability to fire an interrupt in the processor. You have seen in previous chapters that the 8051's interrupt latency is very low for ISRs written in C or assembly. Similarly the inputs which will not be in an active state for very long are better served by interfacing them to the processor via an interrupt. This will eliminate the need for a sampling routine which is triggered periodically at a rate dictated by the nature of the signal.

Level vs. Edge Triggered Interrupts

There are two interrupt triggering methods supported for external interrupts on the 8051 - level and edge triggered interrupts. There are trade offs between these two types of interrupts and certain signal types will dictate the interrupt triggering type that should be used.

Level Triggered Interrupts

Level triggered interrupts are the easiest to understand. The processor merely samples the input signal every instruction cycle and if it sees a logic level 0 on the input, it fires an interrupt in the system. Thus, as long as the given input is low, the processor will keep signaling that there is an interrupt. Conversely, if you have a signal which goes low for one cycle and then returns to a logic level one the interrupt will not be cleared. The 8051 will hold the interrupt signal until the routine can be vectored to and is completed as signaled by the execution of a RETI instruction. If, however, your input signal goes low and fires an interrupt, but does not return high after your ISR for that interrupt is complete, the processor will immediately request another interrupt for the same source. As long as the signal is low, the interrupt signal will be held active by the processor. This may not be desired in situations in which the input signal is not freed by the hardware device requesting the interrupt until some time after you service the interrupt. If this is the case, you will find that your system executes the ISR for that input many more times than is required. In such a case, it makes sense to avoid level triggered interrupts and change over to a system which uses edge triggered interrupts.

Edge Triggered Interrupts

An edge triggered interrupt is caused by a high to low transition in the input signal. As was the case with level triggered interrupts, the input signal is sampled once every instruction cycle. However, in the edge triggered method, an interrupt is caused by a sample in which the signal is a high immediately followed by a sample in which the signal is a low. Once this transition is detected, an interrupt will be signaled by the processor. As was stated above, such a method is useful for devices which will not pull the signal back to a logic level one immediately upon being serviced. This is because only the falling edge of the signal will cause the interrupt, not the logic level zero state of the input signal. This means that

eventually your input signal must return to a logic level one if it is to ever cause another interrupt in the system.

When designing your interrupt structure you need to keep the above discussion in mind. You will find that edge triggering works well for devices that do not make an explicit interrupt request and therefore do not typically require software service to clear their interrupt signal. The most common example of this is a system tick. Such signals are usually generated by an RTC or multivibrator circuit. These devices often provide an input signal with a fixed duty cycle (such as 50%) meaning that for a certain portion of their cycle they are held high and then they are held low for the remaining portion of their cycle. The problem with using a level triggered interrupt is that the entire portion of the cycle that the signal is low will cause a big burst of interrupt activity which, if it does not mess up the timing and function of the software, will at the least prove to be a big waste of system resources.

Other systems that fall into this same category are signal decoders. As an example, consider a system which converts serial data to parallel by sampling an input signal and running the sample through a decoder circuit. This circuit generates an interrupt each time the signal meets certain criterion. The trouble is that the signal meets the criterion for a fixed duration and this time period can be sufficiently long to cause more than one interrupt if the system is set to level triggering.

Level triggered interrupts are most useful in situations where a device may be requesting interrupts very frequently. For example, if you have a device which periodically has high bursts of service requests, it may want to trigger another interrupt before you can service the previous request, and thus will not pull the request line back high. In an edge triggered setup, you will never receive another interrupt from this device because the request line will always be held low and thus the processor will not see any more high to low transitions in the state of the signal. A level triggered interrupt will not have this problem.

Level triggered interrupts will also come in handy in situations where multiple devices are requesting service from the processor via the same input (i.e.: the interrupt request line is being shared). In this case, your code may service one interrupt request and while this is occurring, but before the input line was pulled back high, another device requests service by holding the line low. In an edge triggered system no more interrupts will occur because the line is held low and will not transition again. It is easy to see that this situation and the one discussed above are corrected by using level triggered interrupts. Since the input line is held in the low state, another interrupt will be caused as soon as the ISR finishes its first run. This will force the software to provide whatever service is being requested via that interrupt signal. This process will repeat until the software has serviced all the devices.

Invariably you will have systems which have more interrupt signals than there are interrupt pins. These situations will occur regardless of the tricks you use to extend the number of external interrupts. In these cases you will have to utilize some method which will allow you to share the interrupt pin amongst more than one possible interrupt source. The way in which you do this will depend upon the number of sources that must share the interrupt, the speed in which you must determine which source fired the interrupt and the number of parts you can add to your system.

Sharing Interrupts

When you must multiplex an interrupt line among several inputs there are at least three approaches which can be taken, each of which incrementally raises the number of hardware components that must be added to the system.

Assume that you have two input signals from devices which indicate that they require service by pulling a signal low and holding it low until the service request is fulfilled. The architecture of the system you are designing dictates that these two signals share the INT1 pin of the 8051. In terms of parts count, the best way to do this is to AND the two signals together and attach the output of the AND gate to INT1. Both signals can then be routed to spare input port pins of the controller to allow the processor to determine which source is causing the interrupt. In the following example, you will see that I have chosen to use P1.0 and P1.1 as the input pins.

In this case, the hardware is assuming that the device requesting service will pull its request line low and not release it until the service is no longer required (i.e.: until the service is performed or the request times out). Because the second device can request service while the first slave has its request line asserted, the software should either set INT1 to be level triggered or ensure that the ISR for INT1 checks both service request lines at P1.0 and P1.1 before exiting. Ideally, both of these actions would be performed. If you choose to make INT1 edge triggered, most of the time the system will function normally. You will run into trouble in situations when one slave asserts its service request line and before it is cleared, the second slave asserts its request line. In this case, the service interrupt will never occur again if the software does not check both service lines before exiting the ISR because only the first device will be serviced.

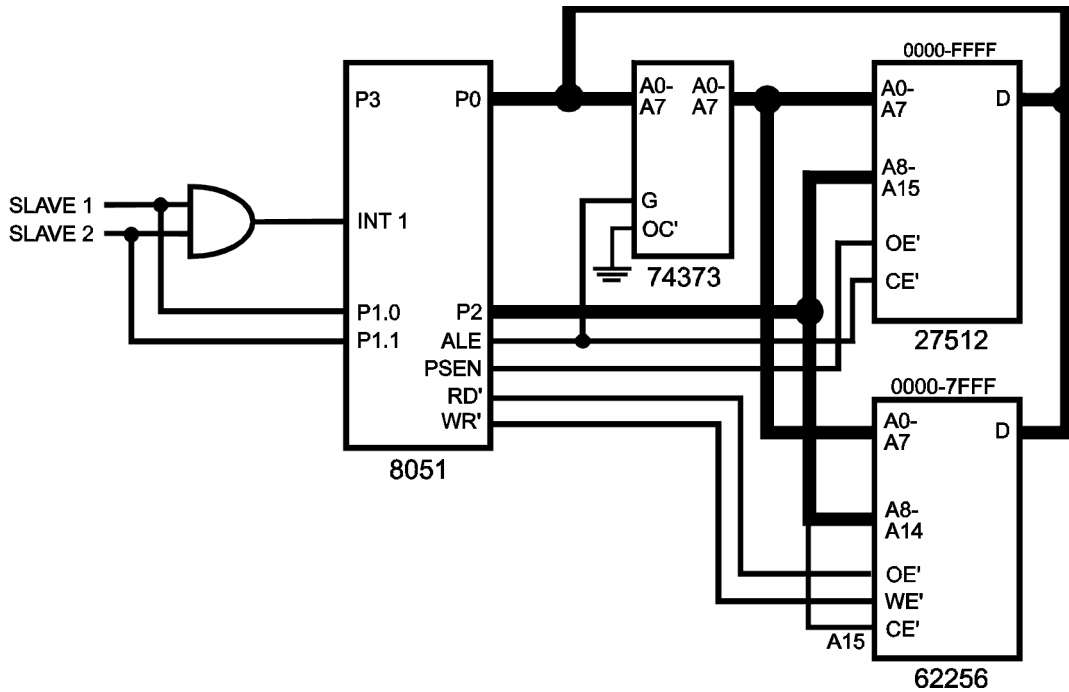


Figure 0-1 - Interrupt 1 Shared

For cases like the above, set the interrupt mode to level triggered, if the constraints of the input signals will allow it. The ISR for the interrupt architecture above is shown in

Listing 0-1. Note how the ISR can be structured to give priority to one of the inputs over the other simply by changing the order in which the inputs are checked. Additionally, the ISR checks the lower "priority" input for a service request once it has completed the initial service. If the higher "priority" input is asserted during this final check, it will still be caught by the system since the interrupt mechanism for INT1 was chosen to be level triggered. Thus, as long as a service request exists, the controller will fire Interrupt One.

Listing 0-1

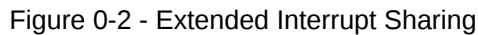
```
sbit SLAVE1 = P1^0;           // name the input signals
sbit SLAVE2 = P1^1;

void int1_isr(void) interrupt 2 {
    if (!SLAVE1) {             // check slave one first
        slave1_service();
    }
    if (!SLAVE2) {             // now check slave two
        slave2_service();
    }
}
```

This routine could be changed to not exit the ISR while one of the inputs was asserted by simply adding a 'do...while' loop. However, this may result in the system infinitely being caught in this ISR thus preventing other system activities. The hardware system as a whole must be designed to keep the slaves requesting service reasonably well behaved so that they do not take up all of the controller's time with their requests for service if the system must perform other functions.

The above system of interrupt sharing can be extended to include as many interrupt sources you want on a given pin. To do this, all you have to do is AND the new sources with the rest of the inputs and interface the new input to the controller at a another port pin. Eventually, you will run out of pins. When you hit this point, you will want to put these inputs in a data latch and trigger the interrupt in the manner it was triggered before. This makes for a more complex system in terms of parts count and software operation to run it. However, the increase in complexity from the preceding method to the new one is not very great. The main difference between the two methods is that the inputs now must be read from the bus via a data latch.

The advantage of the latch is that it usually allows you to add a few more hardware parts that will let you use signals which are not constantly asserted to cause an interrupt. The signal causing the interrupt is marked by the hardware and it is this mark that is read by the software from the data latch. An example of such a system is shown above. The interrupt triggering scheme used before the data latch is left to your own design and will change depending on the type of inputs you are trying to combine into one interrupt.



Listing 0-2

Page 125

```
sbit signal4 = intmask^4;
sbit signal5 = intmask^5;
sbit signal6 = intmask^6;
sbit signal7 = intmask^7;

void int1_isr(void) interrupt 2 {
    intmask=XBYTE[INTREG];           // read the latch to determine
                                     // the cause of this interrupt

    if (signal0) {                   // check all the causes in a
                                     // non-specific order

        signal0_isr();
    }

    ...

    if (signal7) {
        signal7_isr();
    }

    reset_int();                     // perform any reset functions
                                     // required by the interrupt
                                     // logic
}
```

The hardware in the interrupt logic box will depend on the requirements of your system. For example, you may have some signals which are asserted and held until service is performed as was the case in the above example. However, other signals may only pulse to the active state to indicate that the corresponding subsystem is requesting service. In this case the logic needs to latch the falling edge of the signal and generate an interrupt request based on this. The software will have to forcibly clear the interrupt signal within the interrupt logic circuit once the required service has been performed. Additionally, the interrupt logic will keep the IRQ line asserted until all of the interrupt requests have been serviced and the corresponding signals are cleared. Thus, INT1 should once again be set to level triggered to ensure that all interrupts are responded to.

The implementation of the interrupt logic in the above example becomes very much like some of the commercially available interrupt controllers. The main difference is that the above logic array would most likely be designed so that the interrupt triggering method of each specific input signal is not changeable. That is, if you later decided that signal number 3 had to use edge triggering instead of level triggering, this would require a hardware change. Use of a prepackaged interrupt system or design of a good one in house would include the capability to change the type of triggering associated with each signal much like the 8051 itself allows you reconfigure the trigger type of INT0 and INT1.

An interrupt controller allows each input to be individually enabled or disabled by hardware mechanism. Thus, if you wish to shut off a given interrupt, you do not have to block it in software or tie up outputs and add gates to block it in hardware. The capability already exists in the interrupt controller. A system using such a part will have to interface to it via the system bus, so one of the drawbacks you will want to consider when designing a system to use an interrupt controller is that it will take slightly longer to determine the cause of an interrupt and thus response latency will be added to your system. Therefore, highly time critical inputs should still be interfaced directly to the processor if at all possible.

Expanding the Number of External Interrupts

Even though Intel decided some time ago that the number of external interrupts on the 8051 would not exceed two, there are certain tricks you can employ to expand this number to five. There are two simple tricks to do this: the first is to turn the timer/counters into external interrupts and the second is to trick the serial port into being another interrupt. Obviously, if you need to use any of these peripherals for their normal function then you will not be able to use them as extra external interrupt sources. Thus, if you needed one timer and the serial port, you will have to live with using only the other timer as your extra external interrupt source. Because of this, you should keep the above discussion about sharing interrupts in mind when you design your system.

The simplest way to expand the number of external interrupts in your system is to use the timer/counters in count mode and hook the signal to be monitored to the appropriate pin (T0 or T1). In the software design, you must be willing to give up use of the associated timer, and run it in counter mode. To allow the signal to cause an interrupt on its next high to low transition, you set the counter to eight bit auto reload mode. The reload value must be set to FFH. The next time the counter sees a falling edge in the signal, it will increment and overflow to 100H. This will cause the counter to request an interrupt and voila! There you are, one more external interrupt. The code to set all of this up is very simple and can be seen in Listing 0-3.

Listing 0-3

```
#include <reg51.h>

void main(void) {

    ...

    TMOD=0x66;                // use both counters in 8 bit
                              // mode

    TH1=0xFF;                  // set the reload value
    TH0=0xFF;
    TL0=0xFF;
    TL1=0xFF;
    TCON=0x50;                // start both counters
    IE=0x9F;                  // enable the interrupts
    ...

}

/*****
The timer 0 interrupt service routine is written to assume that
the interrupt will clear itself once the software has performed
its service.
*****/

void timer0_int(void) interrupt 1 {

    ...                        // simply service interrupt, hardware will
```

```
        // reload the timer for us

    }

/*****
The timer 1 interrupt service routine is written to assume that
the interrupt is not serviced until the T1 line is pulled back
high.
*****/
void timer1_int(void) interrupt 3 {

    while (!T1) {                // ensure that the interrupt clears
        ...                     // perform functions to clear
                                // interrupt
    }

}
```

Before you get too excited about all of this you should recognize that there are certain limitations to this approach. Number one on the list is the fact that your extra interrupt is an edge triggered device only. So if you are looking for another level triggered interrupt, keep looking or be prepared to sit in an ISR sampling the T0 or T1 line until it is pulled back high. The second point is that you will lose one cycle between the time the edge is detected and the time the interrupt is fired by the counter. This is because the value in the counter is not incremented until the cycle after the cycle in which the falling edge was detected. So if you need extra fast response to your signal's falling edge, you will want to keep this fact in mind when you do your timing budget on interrupt response.

If you have an 8052 device or other 8051 family member which has more timers that use an external pin, you can use this same approach to increase the number of edge triggered external interrupts in your system. It bears repeating however, that any timer you use for this purpose will not be available for other purposes unless your software multiplexes the function of the peripheral.

Using the serial port for another external interrupt is not as straightforward as using the timers for the same purpose. The RXD pin of the processor will become the input signal, and must exhibit a high to low transition, just as the other external interrupts require. You must alter SCON to place the UART in mode 2, which is the 9 bit UART mode with a baud rate derived from the frequency of the oscillator (see Chapter Two). Once the high to low transition in the input signal is detected, the UART will generate an interrupt eight serial bit times later. Once the interrupt occurs, the RI bit must be cleared by the software to clear the interrupt from the system. The following code shows how to set up the UART and how to structure your ISR.

Listing 0-4

```
#include <reg51.h>

void main(void) {

    ...

    SCON=0x90;                // mode 2 with reception enabled
    IE=0x9F;                  // enable the interrupts
    ...

}

void serial_int(void) interrupt 4 {
    if (!_testbit_(RI)) {

        ...                    // perform service here

    }

}
```

As with the timer system, there are drawbacks to using the serial port as another interrupt. First, the interrupt can only be edge triggered. We have discussed this situation already. Secondly, the input signal must be held low for at least 5/8 of a bit time if the low level is to be recognized. This time period is dictated by the UART since it is sampling the line looking for a start bit, and must believe that the input signal going low is a start bit. Third, the interrupt will not be fired until the "eighth" bit time which is when the UART would normally request an interrupt. Additionally, the signal should not remain low for more than nine bit times for this scheme to work. Since this scheme depends on tricking the UART into believing that it is seeing a valid incoming serial byte at the RXD pin, the timing limitations of the UART are imposed on the signal. These limitations will depend on the oscillator frequency of your system since the UART will be deriving its baud rate from that frequency. Different timing constraints could be established by changing the mode of the UART and using one of the internal timers. However, the required timing and the interrupt latency will only lengthen, not shorten.

What to Put in Your ISRs

Many new designers of embedded systems get quite confused when it comes to the issue of structuring their ISRs. The main problem that people seem to run into is how to decide what functions should be carried out in the ISR and what functions should be left to the main level of the application. The answer to this is straightforward, but the implementation of it is not. In general, only the bare minimum processing should be done in the ISR itself. You should always strive to keep the amount of work done in your ISRs to a minimum. Attaining this goal will provide a couple of benefits. First, you will have a system that makes itself available to respond to system interrupts a larger percentage of the time. In systems where missing an interrupt or responding slowly to an interrupt is disastrous, the extra time will be of immeasurable value. Second, it will keep your ISRs simple in structure, with less things to go wrong in them. The more you try to pack into your ISRs the more likely they will be to step all over each others' functionality and data structures. Minimizing your ISRs means that you have more critical code sections in your software, but these can all be compacted into one area of the code - the main execution loop. The design of the ISRs in your system can be most critical to the success or failure of the product. You should give careful thought to the timing of the interrupts with respect to each other and the amount of time each ISR will take. Any ISRs which will operate on the same data structures in the software should be given careful attention. A couple of examples will help punctuate some of the points made above.

Suppose you have a system which receives a stream of serial data via the internal UART and must find the correct messages in this stream and respond to them. This system should have an ISR associated with the serial interrupt that mainly reads the data from SBUF and puts it in a circular queue. The main level of the application will then be responsible for checking this queue for data, popping the data and parsing it into a message. When a complete message is received, it is handled accordingly. In such a situation it would also be acceptable to parse the incoming serial stream into valid messages and push these messages into a queue for the main loop to handle. The main point of the example is that the ISR should not be responsible for responding to the incoming messages because this will be a time consuming process. For the most part, you can execute responses from the bottom interrupt level.

There are some cases in which the actions associated with the interrupt can not be separated from the ISR because of time constraints or relationships of the actions to other interrupt actions. For example, if you had a system with a peripheral that requested an interrupt when it had clocked in valid data and began to present this data to your processor at the rate of one data unit per 20 microseconds, you would more than likely be forced to remain in the ISR and receive all the data because if you exited the ISR after one data unit and some other interrupt were fired before the next data unit fired an interrupt you would probably lose the next incoming data unit. Such a situation can often be solved by using the priority scheme of the 8051, but in more complex systems, this option does not exist.

Another example of code you would want to leave in an ISR are functions which modify shared data structures. Take for example a system which has several interrupts. Two of these interrupts are of the same priority and operate on the same data structure. One of the interrupts is fired when an A/D sampling unit completes a conversion. This occurs once every 10ms. The system must record these conversions and send them out serially in real time. The other interrupt is a system tick which is checking the shared data structure (a queue) for the presence of new conversions. When a new conversion is available, its job is to package it in a serial message and initiate a serial transfer. You can easily see that it is desirable for these two ISRs to not try to access the queue simultaneously. We have all chased down bugs in systems where critical data corruption occurs. In this case, it makes a lot of sense for the input ISR to read in the data and complete the operation of queuing it before the ISR is exited. Likewise, the system tick should complete the operation of dequeuing the data, building the message, and initiating the serial transfer before it exists.

Conclusion

This chapter has focused on ways to enhance the interrupt system of the 8051. Using these techniques in conjunction with earlier discussions, such as simulating extra interrupt priority levels in hardware will give you better flexibility and capability than the designers of the 8051 imagined. You should note that when you design the interrupt portion of your system, you must give careful attention to the matching of input signals to interrupt lines/sources on the 8051. Additionally, the same care should be applied to the design of the software interrupt system. This means that the selection of priority levels, and structure/functions of the ISRs must be well thought out. For best results, the hardware and software portions of the interrupt system should be designed in conjunction with each other. If they are not designed in parallel, you should at least be willing to go through a certain amount of iterations to refine the design. Above all, keep in mind that the interrupt system is the most critical part of any real time embedded system.

- The Serial Port

Introduction

The main communication port of a typical 8051 system is not some form of parallel transfer or shared memory, it is the serial port that is built into the 8051. As we saw in Chapter Two, the on board UART is very flexible and can be set up to communicate with other systems at high speeds. This chapter will present software designs to transfer data back and forth between systems effectively.

Slow Speed Serial I/O - Interfacing to a PC

In many embedded applications with the 8051, the processor reports back events and other data to some master system, often times a PC. Usually, the master also issues commands over the serial link between the two systems. Frequently, a voltage standard (and often times a handshaking standard) such as RS-232 is used to allow the 8051 based system to communicate with a wide range of common systems such as PCs. If the length of the communication link is not too great, then the 8051 does not require an RS-232 line driver to support RS-232 type communications. A simple circuit will allow the processor to receive the input bytes and transmit data back out to a PC. Most PC systems do not require that the transmissions to them fully adhere to the RS-232 voltage standards. As you can see in the schematic shown below, this makes for a very simple circuit when interfacing to a PC. The 12 volt output of the PC in to the 8051's serial in line is handled by passing it through a voltage divider which limits the maximum voltage that can be seen to 5 volts.

Once this simple serial interface is designed, you need some code to go along with it to control all data coming in and out of the UART. The simplest approach for handling incoming data is to allow the first byte to cause a serial interrupt and then poll for the remaining bytes in the input stream, assuming that your message protocol allows you to predict the number of bytes in a message. Output data is handled in a similar manner, except that there is no "first interrupt." When it is time to start transmitting, the software forces a byte into SBUF and polls the SCON register to see when the next byte can be sent. When all the bytes have been sent, this cycle is terminated.

The above software design will work for systems which have little else to do in real time, except for deal with the serial port. Using such an approach will also simplify the software and allow you to get communications up and running that much quicker. However, for more complex systems the "polling" approach will not suffice. A better design is as follows. For incoming data, each input byte fires a serial interrupt. The ISR for the UART is responsible for reading the contents of SBUF and deciding if it is part of a valid message. An FSA (Finite State Automaton) is used to parse the incoming byte stream into a valid message. When a valid message is received, it is pushed into a queue and acted upon by the main loop processing level. Output data is handled in a reverse, yet similar method. A message is built and loaded into a queue. The first byte of this message is sent, causing a transmit interrupt from the UART. The ISR for the UART then reads one byte from the output queue and writes it to SBUF as long as there are bytes in the queue.



Imagine that the clock project from Chapter Four is incorporated into a monitoring system in which it is polled serially by a PC which also has the task of polling several other monitoring devices over a shared serial link. A simple system diagram is shown in Figure 0-2. The functions of the other devices (1 through n) in the system do not matter - for now, we are only concerned with the clock.

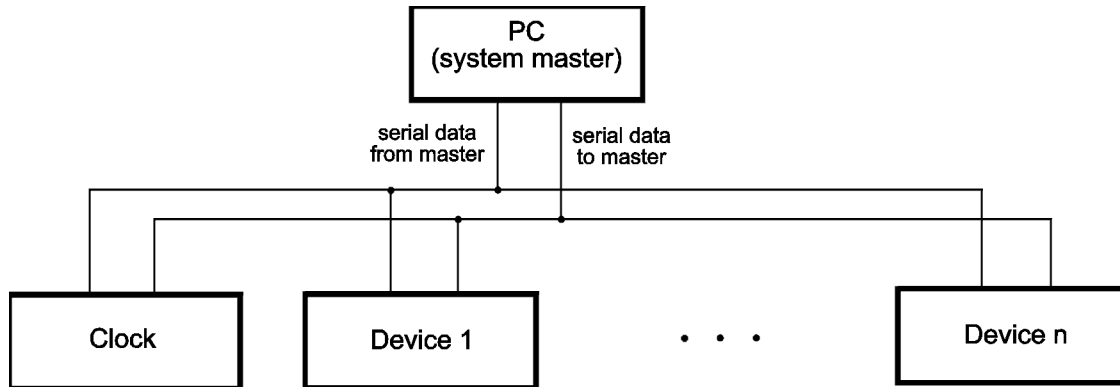


Figure 0-2 - Master / Slave Serial Network

For its new application in this monitoring system the clock has been slightly altered. The most significant change allows for serial communications with the PC via standard RS-232 voltage level signals. The clock now has the design shown in Figure 0-3.

The PC in this design executes a program which has the capability to request the current time from the clock, set the current time to be used in the clock, reset the time to all zeroes, and place a 32 character text message on the display. Remember that the serial input to the clock will not always be directed to the clock itself and the devices on the serial link must have some ways of distinguishing their messages from the rest of the data in the incoming stream. Therefore, the messages must have a structure which allows the receiver to do this. Each message will begin with a sync byte and will contain the address of the unit being addressed. The clock's address will be 43H. The message will also contain the command being issued, the size of the optional data block, the data itself, and a single byte checksum. The format of a typical message is shown below.

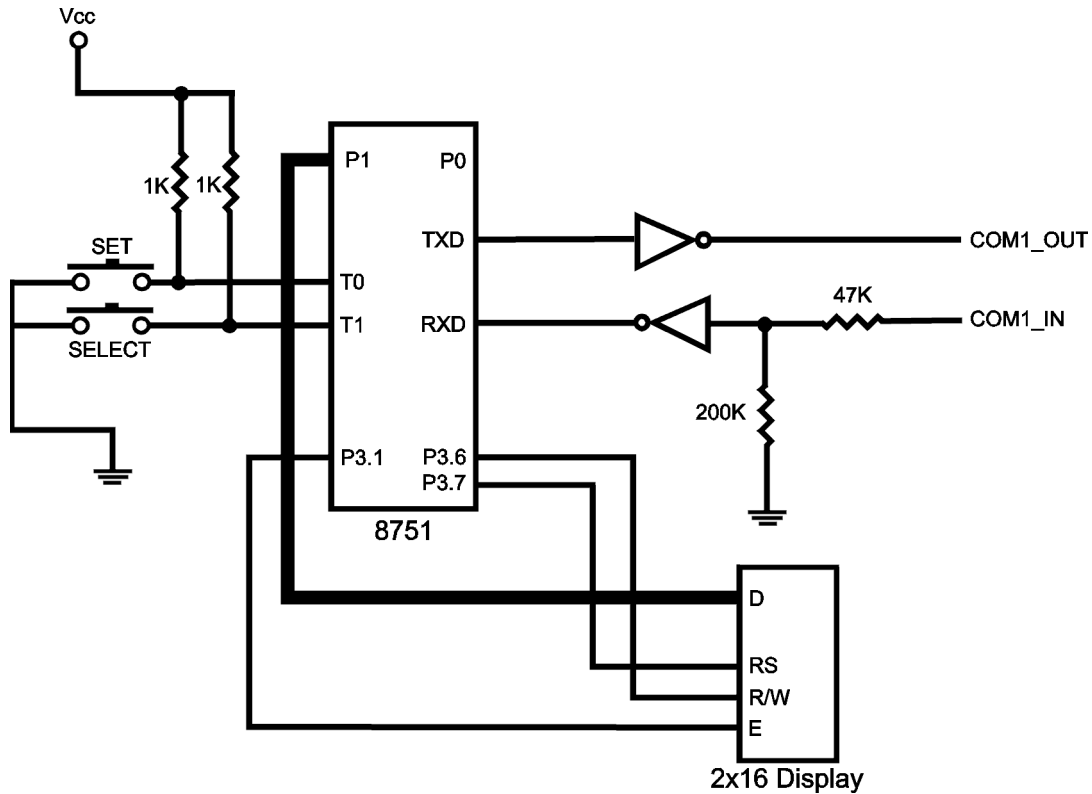


Figure 0-3 - Clock Project as a Serial Slave

Field	Description
Synchronization Byte	Must be set to 33H
Unit Address	Address of destination unit. The clock's address == 43H, the PC's address == 01H.
Command	Indicates the message being sent.
Data Size	Indicates number of bytes in following data block (0 to 255 bytes). If no extra data is required with this message, this byte will be 00H.
Data Block	Extra data required with this message. The number of bytes must equal the value of the Data Size field.
Checksum	Single byte addition of all previous bytes in the message (including the Synchronization Byte) ignoring rollover.

Table 0-1

For all command messages from the PC, the responding unit must send an acknowledge back. The clock will also have to be responsible for the four messages listed above (time request, time set, time reset, text display). The format of these messages follows.

CHAPTER 8 - THE SERIAL PORT

Message: Reset time to zero. Command 01H. This command resets the time to 00:00:00.	
Field	Value
Synchronization Byte	33H
Unit Address	43H
Command	01H
Data Size	00H
Checksum	77H

Table 0-2

Message: Time Set. Command 02H. This command sets the time to the values provided.	
Field	Value
Synchronization Byte	33H
Unit Address	43H
Command	02H
Data Size	03H
Hours	Indicates current hours, must be 0 to 23.
Minutes	Indicates current minutes, must be 0 to 59.
Seconds	Indicates current seconds, must be 0 to 59.
Checksum	7BH + Hours + Minutes + Seconds

Table 0-3

Message: Time Request. Command 03H. This command requests the current time. Direction is from the PC to the clock.	
Field	Value
Synchronization Byte	33H
Unit Address	43H
Command	03H
Data Size	00H
Checksum	79H

Table 0-4

Message: Time Request. Command 03H. This command reports current time. Direction is from the clock to the PC.	
Field	Value
Synchronization Byte	33H
Unit Address	43H
Command	03H
Data Size	03H
Hours	Indicates current hours, must be 0 to 23.
Minutes	Indicates current minutes, must be 0 to 59.
Seconds	Indicates current seconds, must be 0 to 59.
Checksum	7CH + Hours + Minutes + Seconds

Table 0-5

Message: Text Display. Command 04H. This command specifies a 32 character text message to display on the LCD panel.	
Field	Value
Synchronization Byte	33H
Unit Address	43H
Command	04H
Data Size	20H
Text	32 ASCII characters for display. Must be space padded.
Checksum	9AH + Σ (Text field bytes)

Table 0-6

Message: Acknowledge. Command FFH. This command is sent following receipt of any valid command which does not require a return of data to the PC.	
Field	Value
Synchronization Byte	33H
Unit Address	43H
Command	FFH
Data Size	01H
Return	Command number of the message being acknowledged
Checksum	76H + Return

Table 0-7

Now that you understand the incoming byte stream and the responsibilities of the clock processor a little better, the ISR for incoming serial bytes can be designed. The first thing to keep in mind is that the ISR will be responsible for parsing the byte stream. This means that a simple finite state automaton can be used for this task. For those of you who don't know, an FSA is essentially a software machine which will move from one state to another based on input that it receives. The FSA will have an initial state in which it is looking for the sync byte, and several intermediate states which correlate to the next expected part of the message. The final state will read the checksum byte and verify it. If at any point, a byte is received which does not conform to the structure of a valid message, the FSA returns to the initial state and begins looking for another sync byte.

The theory behind this type of serial receiver is very simple to implement. If you are writing the ISR for the incoming data in C the most common thing to do is declare a variable which will keep track of the current state of the system. This is done by giving each possible state a number and storing this number in the variable. When an input byte fires an interrupt, the C routine does a 'switch' on the state variable to determine the next action of the FSA. Since the clock has only a small number of states on an eight bit machine, the state variable will be declared as an 'unsigned char'. If you were writing the system in assembler, the 'switch' statement could be replaced by a more efficient jump table arrangement. You will find that the gains in size and speed to be had by coding the ISR in assembler are not significant when compared to the output of the C51 compiler unless you are dealing with a high speed serial system.

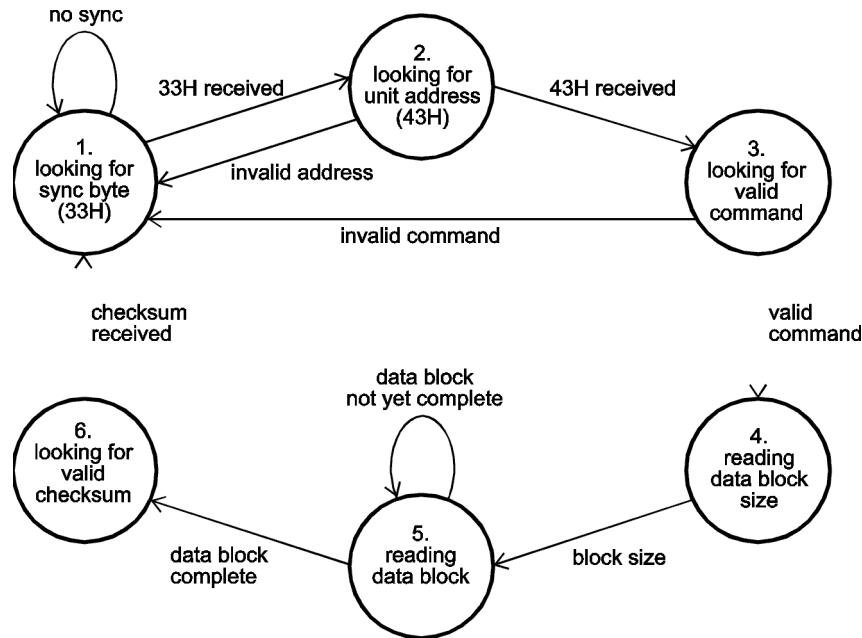


Figure 0-4 - Receive FSA

In this case the clock is only transmitting data at 9600 baud which takes roughly 1.042 ms per byte. The crystal frequency of the clock will be changed to 11.059 MHz to allow the system to easily generate 9600 baud. This will change the time of one instruction cycle to 1.085 microseconds. Dividing this into the byte time reveals that there will be 960 instruction cycles between back to back serial input interrupts, which should be more than enough time to maintain the input FSA. The initial code structure for the serial ISR is shown in Listing 0-1.

Listing 0-1

```

// define constants for the FSA states
#define FSA_INIT          0
#define FSA_ADDRESS      1
#define FSA_COMMAND      2
#define FSA_DATASIZE     3
#define FSA_DATA         4
#define FSA_CHKSUM       5

// define constants for message parsing
#define SYNC              0x33
#define CLOCK_ADDR       0x43

// define constants for the input commands
#define CMD_RESET         0x01
#define CMD_TIMESYNC     0x02
#define CMD_TIMEREQ      0x03
#define CMD_DISPLAY      0x04
#define CMD_ACK          0xFF
    
```

```
#define RECV_TIMEOUT 10    /* define the interbyte timeout */

unsigned char
    recv_state=FSA_INIT,  // indicates the current
                          // serial state
    recv_timer=0,         // counts down the interbyte
                          // timeout
    recv_chksum,          // holds the current checksum
                          // value of the incoming
                          // message
    recv_size,            // index into the receive
                          // buffer for incoming data
                          // block
    recv_buf[35];         // holds the incoming message

unsigned char code valid_cmd[256]={ // This array determines if
                                   // the current command byte is
                                   // valid.  If the corresponding
                                   // entry is 1 the command
                                   // ID is valid.

    0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 00 - 0F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 10 - 1F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 20 - 2F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 30 - 3F

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 40 - 4F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 50 - 5F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 60 - 6F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 70 - 7F

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 80 - 8F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 90 - 9F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // A0 - AF
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // B0 - BF

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // C0 - CF
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // D0 - DF
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // E0 - EF
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // F0 - FF
};
```

```

/*****
Function:      serial_int
Description:   Runs the serial port FSAs.
Parameters:   none.
Returns:      nothing.
Side Effects: none.
*****/

void serial_int(void) interrupt 4 {
    unsigned char data c;
    if (_testbit_(TI)) {
        // handle transmit duties
    }
    if (_testbit_(RI)) {
        c=SBUF;
        switch (recv_state) {
        case FSA_INIT:                // looking for sync byte
            if (c==SYNC) {            // check for sync byte
                recv_state=FSA_ADDRESS; // move to next state
                recv_timer=RECV_TIMEOUT; // set interbyte timeout
                recv_chksum=SYNC;       // set initial checksum value
            }
            break;
        case FSA_ADDRESS:             // looking for unit ID
            if (c==CLOCK_ADDR) {       // make sure the message is
                                        // for the clock
                recv_state=FSA_COMMAND; // set next FSA state
                recv_timer=RECV_TIMEOUT; // set interbyte timeout
                recv_chksum+=c;          // maintain checksum
            } else {                  // message is not for the
                                        // clock
                recv_state=FSA_INIT;     // return to initial state
                recv_timer=0;            // clear interbyte timeout
            }
            break;
        case FSA_COMMAND:             // looking for cmd id
            if (!valid_cmd[c]) {        // make sure command is good
                recv_state=FSA_INIT;    // reset FSA
                recv_timer=0;
            } else {
                recv_state=FSA_DATASIZE; // move to next FSA state
                recv_chksum+=c;          // update checksum
                recv_buf[0]=c;           // save command ID
                recv_timer=RECV_TIMEOUT; // set interbyte timeout
            }
        }
    }
}

```

```
break;    case FSA_DATASIZE:           // looking for length byte
recv_chksum+=c;           // update checksum
recv_buf[1]=c;           // save data block size
if (c) {                 // see if there is a data
                        // block
    recv_ctr=2;           // set indexing byte
    recv_state=FSA_DATA;  // move to next FSA state
} else {
    recv_state=FSA_CHKSUM; // message done, get checksum
}
recv_timer=RCV_TIMEOUT;
break;

case FSA_DATA:           // reading in data block
    recv_chksum+=c;       // update checksum
    recv_buf[recv_ctr]=c; // save data byte
    recv_ctr++;           // update count of data block
    if ((recv_ctr-2)==recv_buf[1]) { // see if receive data
                                // block counter minus the
                                // offset into the receive
                                // buffer == the data block
                                // size
        recv_state=FSA_CHECKSUM; // done receiving data block
    }
    recv_timer=RCV_TIMEOUT; // set interbyte timeout
    break;

case FSA_CHECKSUM:       // reading in checksum
    if (recv_chksum==c) { // verify checksum
        c=1;              // use c to indicate if an
                            // ack message should be built

        switch (recv_buf[1]) { // act upon the command
            case CMD_RESET:    // reset the clock to 0
                break;
            case CMD_TIMESYNC: // set the clock
                break;
            case CMD_TIMEREQ:  // report system time
                break;
            case CMD_DISPLAY:  // display ASCII message
                break;
        }
    }
    if (c) {
        // build ack message
    }
}
d
```

```
efault:
    recv_timer=0;                // reset the FSA
    recv_state=FSA_INIT;
    break;
}
}
}
```

The code that runs the FSA directly represents the model that was shown in Figure 0-4Figure . The ISR shown here is not fully fleshed out and is intended to show you the way the receive code will be structured. Obviously there will be more design issues such as the way to transmit out data and the execution of the commands received from the serial input stream.

Transmitting data back to the PC is a very simple matter and is handled by the serial ISR once an initial byte transmission kicks off the whole thing. The clock project will make the simple minded assumption that the PC will only issue one valid command at a time to it, and thus will not attempt to maintain a queue of outgoing messages. This assumption is made to simplify the example so that we don't get bogged down worrying about a lot of data queues and critical code section issues in this chapter. When a section of code needs to transmit out a message, it simply builds the message into a transmit buffer, and sets a variable which indicates the number of bytes in the buffer. The first byte must then be written to SBUF and the checksum byte set. Writing the first byte is like priming a pump: it causes the first transmit interrupt. Once the first interrupt is fired, the serial interrupt routine can complete the business of sending out the data by itself. The code structure as designed for the serial interrupt is shown in Listing 0-2.

Listing 0-2

```
// define constants for message parsing
#define SYNC          0x33
#define CLOCK_ADDR    0x43

unsigned char
    trans_buf[7],          // holds the outgoing message
    trans_ctr,             // index into trans_buf
    trans_size,            // holds the total number of
                           // bytes to send
    trans_chksum;          // computes the output checksum

/*****
Function:      serial_int
Description:   Runs the serial port FSAs.
Parameters:   none.
Returns:      nothing.
Side Effects: none.
*****/

void serial_int(void) interrupt 4 {
    unsigned char data c;
    if (_testbit_(TI)) {    // check for transmit
                           // interrupt
```

```
trans_ctr++;                // move the output index up
if (trans_ctr<trans_size) {  // see if all data is output
    if (trans_ctr==(trans_size-1)) { // send checksum as last
                                    // byte
        SBUF=trans_chksum;
    } else {
        SBUF=trans_buf[trans_ctr]; // send current byte
        trans_chksum+=trans_buf[trans_ctr]; // update checksum
    }
}
}
if (_testbit_(RI)) {
    c=SBUF;
    switch (recv_state) {

// run the receive FSA

case FSA_CHECKSUM:          // reading in checksum
    if (recv_chksum==c) {    // verify checksum
        c=1;                // use c to indicate if an
                                // ack message should be built

        switch (recv_buf[1]) { // act upon the command
            case CMD_RESET:    // reset the clock to 0
                break;
            case CMD_TIMESYNC: // set the clock
                break;
            case CMD_TIMEREQ:  // report system time
                c=0;
                break;
            case CMD_DISPLAY:  // display ASCII message
                break;
        }
    if (c) {                // build ack if necessary
        trans_buf[0]=SYNC;  // set up header
        trans_buf[1]=CLOCK_ADDR;
        trans_buf[2]=CMD_ACK;
        trans_buf[3]=1;
        trans_buf[4]=recv_buf[1]; // put in command being ack'ed
        trans_ctr=0;              // set buffer pointer to first
                                    // byte

        trans_size=6;            // six bytes total
        SBUF=SYNC;              // send first byte
        trans_chksum=SYNC;      // initialize checksum
    }
}
```

```
    }  
    default:  
        recv_timer=0;  
        recv_state=FSA_INIT;  
        break;  
    }  
}  
}
```

As you can see, the ISR code for the output data is very simple to write and takes very little CODE space as well as little memory space. As was the case with the receive logic, the checksum for the message is built concurrently with the transmission of the message itself.

The ISR as designed will function without a hitch within the confines of the clock project. Taking care of the command actions within the serial interrupt routine will allow the system to avoid many critical problems in design. Most notable among these is that the serial ISR has the capability to modify the time structure, as does the system tick interrupt. The clock software can be designed such that the serial interrupt and the timer interrupt have the same priority level. Once this is set up, either ISR can modify the time structure at will without fear that the other is doing so at the same time. This is not the case when the messages are handled from the main loop. However, the execution of the command actions does not really belong in the ISR, because it forces the system to spend too much time in the interrupt routine and possibly miss other interrupts. For example, it is reasonable to assume that if the PC decided to send a long series of display commands back to back, that the clock could eventually miss a system tick or two because of the extended amount of time it spent in the serial ISR writing data to the LCD panel. In a more complicated system, the incoming messages should be placed in a queue and executed by the main loop. In the clock project, let's continue to assume that the PC will not send another valid message to the clock until it has acknowledged the current message. This will allow the queue to be simply a buffer for the current message.

The new design of the ISR will be very much like the other, except that when a complete and valid message is received, it will be pushed into a secondary buffer and a buffer valid flag will be set. The code to execute commands which previously resided in the ISR will be moved to a new function which is called when the main loop detects that there is something in the secondary buffer. The new ISR code is shown in its entirety below in Listing 0-3.

Listing 0-3

```
// define constants for the FSA states  
#define FSA_INIT          0  
#define FSA_ADDRESS      1  
#define FSA_COMMAND      2  
#define FSA_DATASIZE     3  
#define FSA_DATA         4  
#define FSA_CHKSUM       5  
  
// define constants for message parsing  
#define SYNC              0x33  
#define CLOCK_ADDR       0x43  
  
// define constants for the input commands  
#define CMD_RESET        0x01
```

```
#define CMD_TIMESYNC      0x02
#define CMD_TIMEREQ      0x03
#define CMD_DISPLAY      0x04
#define CMD_ACK          0xFF

#define RECV_TIMEOUT  10    /* define the interbyte timeout */

unsigned char
    recv_state=FSA_INIT,    // indicates the current
                           // serial state
    recv_timer=0,           // counts down the interbyte
                           // timeout
    recv_chksum,            // holds the current checksum
                           // value of the incoming message
    recv_size,              // index into the receive
                           // buffer for incoming data
                           // block
    recv_buf[35];           // holds the incoming message

unsigned char
    trans_buf[7],           // holds the outgoing message
    trans_ctr,              // index into trans_buf
    trans_size,             // holds the total number of
                           // bytes to send
    trans_chksum;           // computes the output
                           // checksum

unsigned char code valid_cmd[256]={ // This array determines if
                                   // the current command byte is
                                   // valid. If the corresponding
                                   // entry is 1 the command ID
                                   // is valid.
    0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 00 - 0F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 10 - 1F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 20 - 2F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 30 - 3F

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 40 - 4F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 50 - 5F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 60 - 6F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 70 - 7F

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 80 - 8F
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // 90 - 9F
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // A0 - AF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // B0 - BF

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // C0 - CF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // D0 - DF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // E0 - EF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // F0 - FF
};

/*****
Function:      serial_int
Description:   Runs the serial port FSAs.
Parameters:   none.
Returns:      nothing.
Side Effects:  none.
*****/

void serial_int(void) interrupt 4 {
    unsigned char data c;
    if (_testbit_(TI)) {           // check for transmit
                                   // interrupt

        trans_ctr++;              // move the output index up
        if (trans_ctr<trans_size) { // see if all data is output
            if (trans_ctr==(trans_size-1)) { // send checksum as last
                                           // byte

                SBUF=trans_chksum;
            } else {
                SBUF=trans_buf[trans_ctr]; // send current byte
                trans_chksum+=trans_buf[trans_ctr]; // update checksum
            }
        }
    }
    if (_testbit_(RI)) {
        c=SBUF;
        switch (recv_state) {
        case FSA_INIT:             // looking for sync byte
            if (c==SYNC) {         // check for sync byte
                recv_state=FSA_ADDRESS; // move to next state
                recv_timer=RECV_TIMEOUT; // set interbyte timeout
                recv_chksum=SYNC;     // set initial checksum value
            }
            break;
        case FSA_ADDRESS:          // looking for unit ID
            if (c==CLOCK_ADDR) {    // make sure the message is
```

```

                                // for the clock
    recv_state=FSA_COMMAND;      // set next FSA state
    recv_timer=RECV_TIMEOUT;     // set interbyte timeout
    recv_chksum+=c;              // maintain checksum
} else {                        // message is not for the
                                // clock
    recv_state=FSA_INIT;        // return to initial state
    recv_timer=0;               // clear interbyte timeout
}
break;
case FSA_COMMAND:               // looking for cmd id
    if (!valid_cmd[c]) {        // make sure command is good
        recv_state=FSA_INIT;    // reset FSA
        recv_timer=0;
    } else {
        recv_state=FSA_DATASIZE; // move to next FSA state
        recv_chksum+=c;         // update checksum
        recv_buf[0]=c;          // save command ID
        recv_timer=RECV_TIMEOUT; // set interbyte timeout
    }
    break;
case FSA_DATASIZE:              // looking for length byte
    recv_chksum+=c;             // update checksum
    recv_buf[1]=c;              // save data block size
    if (c) {                    // see if there is a data
                                // block
        recv_ctr=2;             // set indexing byte
        recv_state=FSA_DATA;    // move to next FSA state
    } else {
        recv_state=FSA_CHKSUM;  // message done, get checksum
    }
    recv_timer=RECV_TIMEOUT;
    break;
case FSA_DATA:                  // reading in data block
    recv_chksum+=c;             // update checksum
    recv_buf[recv_ctr]=c;       // save data byte
    recv_ctr++;                 // update count of data block
    if ((recv_ctr-2)==recv_buf[1]) { // see if receive data
                                // block counter
                                // minus the offset into the
                                // receive buffer == the data
                                // block size
        recv_state=FSA_CHECKSUM; // done receiving data block
    }
}
```

```
    recv_timer=RECV_TIMEOUT;        // set interbyte timeout
    break;
case FSA_CHECKSUM:                  // reading in checksum
    if (recv_chksum==c) {           // verify checksum
                                    // save the message, if good
        memcpy(msg_buf, recv_buf, recv_buf[1]+2);
        msg_buf_valid=1;
    }
default:                            // reset the FSA after
                                    // checksum byte or in case of
                                    // erroneous state

    recv_timer=0;
    recv_state=FSA_INIT;
    break;
}
}
```

The ISR now is only responsible for knowing how to parse incoming serial messages and how to transmit data out of a buffer. The smarts relating to actions associated with incoming serial messages are now contained in a function which is called by the main loop when it detects that there is data in the secondary message buffer. The code which executes the command in the message and the new main loop are shown below.

Listing 0-4

```
/******
Function:      execute_cmd
Description:   Executes the actions associated with a serial
               command.
Parameters:    none.
Returns:       nothing.
Side Effects:  none.
*****/
void execute_cmd(void) {
    bit need_ack=1;
    switch (recv_buf[1]) {           // act upon the command
    case CMD_RESET:                  // reset the clock to 0
        EA=0;                       // stop interrupts while the
                                    // time structure is used

        curtime.sec=curtime.min=curtime.hour=0;
        timeholder=curtime;
        EA=1;                       // allow interrupts now
        break;
    case CMD_TIMESYNC:               // set the clock
        EA=0;                       // stop interrupts while the
```

```
                                // time structure is used
    curtime.hour=recv_buf[3];
    curtime.min=recv_buf[4];
    curtime.sec=recv_buf[5];
    timeholder=curtime;
    EA=1;                        // allow interrupts now
    break;
case CMD_TIMEREQ:                // report system time
    trans_buf[0]=SYNC;           // set up header
    trans_buf[1]=CLOCK_ADDR;
    trans_buf[2]=CMD_TIMEREQ;    // sending current time
    trans_buf[3]=3;
    EA=0;                        // stop interrupts while the
                                // time structure is used
    trans_buf[4]=curtime.hour;   // put current time in the data
                                // block
    trans_buf[5]=curtime.min;
    trans_buf[6]=curtime.sec;
    EA=1;                        // allow interrupts now
    trans_ctr=0;                 // set buffer pointer to first
                                // byte
    trans_size=8;                // eight bytes total
    need_ack=0;
    break;
case CMD_DISPLAY:                // display ASCII message
    recv_buf[34]=0;              // set a null terminator at the
                                // end of the string
    printf("\xFF%s", &recv_buf[2]); // display the string
    display_time=100;            // set timeout to 5 seconds
    break;
}
if (need_ack) {                  // build ack if necessary
    trans_buf[0]=SYNC;           // set up header
    trans_buf[1]=CLOCK_ADDR;
    trans_buf[2]=CMD_ACK;
    trans_buf[3]=1;
    trans_buf[4]=recv_buf[1];    // put in command being ack'ed
    trans_ctr=0;                 // set buffer pointer to first
                                // byte
    trans_size=6;                // six bytes total
}
SBUF=SYNC;                      // send first byte
trans_chksum=SYNC;              // initialize checksum
}
```

```

/*****
Function:      main
Description:   This is the entry point of the program. This
               function initializes the 8051, enables the correct
               interrupt source and enters idle mode. The idle
               mode loop checks after each interrupt to see if
               the LCD panel must be updated.
Parameters:    None.
Returns:       Nothing.
*****/

void main(void) {
    disp_init();                // set up display
    TMOD=0x21;                  // set timer 0 in 16 bit mode
                                // timer 1 is a baud rate
                                // generator
    TCON=0x55;                  // start both timers
                                // both external ints
                                // are edge triggered
    TH1=0xFD;                   // set timer 1 reload for 9600
                                // baud
    SCON=0x50;                  // serial port mode one
    IE=0x92;                    // enable the timer 0 interrupt
    for (;;) {
        if (_testbit_(msg_buf_valid)) { // check for a new serial
                                        // message
            execute_cmd();              // execute the command
        }
        if (disp_update) {
            disp_time();                 // display new time;
        }
        PCON=0x01;                     // enter idle mode
    }
}

```

This method of dealing with the messages in the main loop is simple to implement and becomes important to do in systems which perform a lot of interrupt driven input and output. Since the interface to the PC is relatively slow, there is plenty of time in between bytes to perform other tasks such as responding to the system tick. Undoubtedly, a real world system will have more interrupts happening than does the clock...this is just a simplified example to help illustrate the principles of serial I/O on the 8051. The next section will discuss handling serial input and output in a high speed application.

High Speed Serial I/O

The discussions earlier in this chapter centered around a system which used the clock project as a timer that was periodically polled. The serial messages came from a PC at 9600 baud and were not always intended for the clock itself since there were other devices on the serial line. Let's suppose that the designers of this system decided that the PC was too busy running a user interface and crunching data

and thus it was communicating to all the devices too slowly. To correct these issues a simple system is designed which is based on the 8051. The job of this circuit will be to replace the PC as the master of the serial link and perform all the polling and computation that the PC software was doing. This new device will be called the System Monitor. The system diagram is now slightly more complex, but as far as the clock is concerned it has remained unchanged.

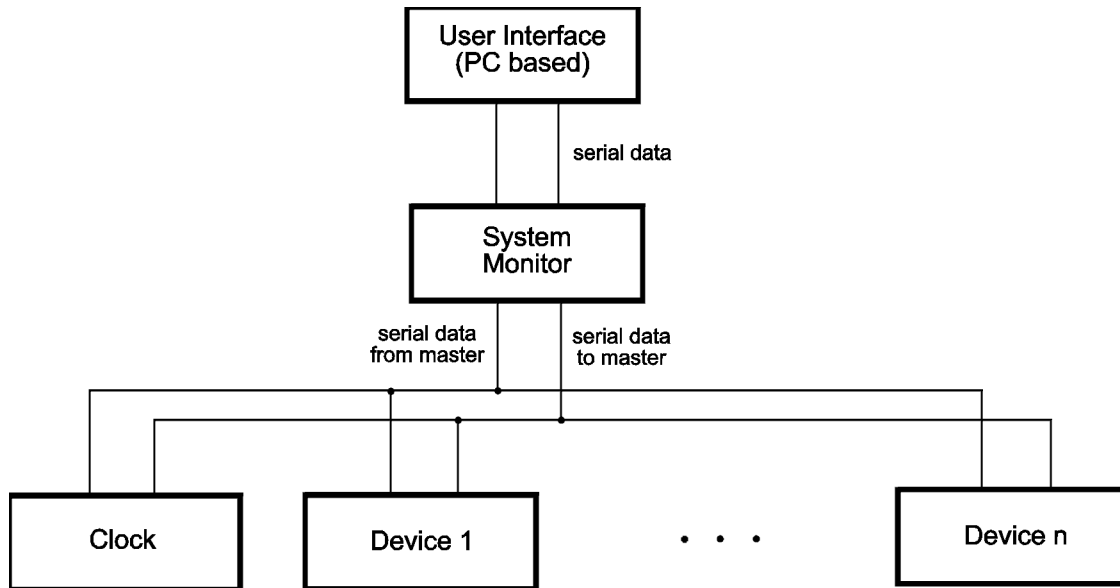


Figure 0-5 - New Serial Network

The polling will still be performed via a serial link, however, the baud rate will now be much higher. Since all the devices in this system are using an 8051 running at 11.059MHz, the serial ports can be hooked up directly to each other. This means that the voltage divider and the inverters that were used when interfacing to a PC are no longer needed. The new schematic for the circuit is shown in Figure 0-6.

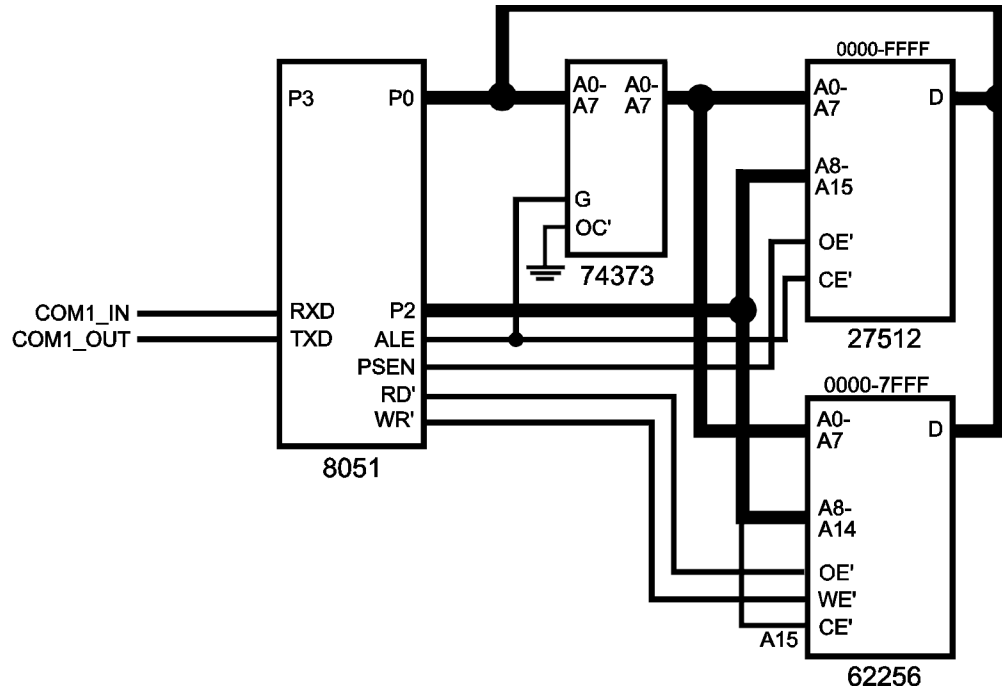


Figure 0-6 - 8051 as a High Speed Slave

Since it is now connected to other 8051's, the UART will be run in mode 2 which will yield a baud rate of 345593.75 baud. The new time in between bytes will be 31.829 μ s (or just less than 30 instruction cycles) instead of the 1.04 ms realized at 9600 baud. Due to the lack of time between bytes, the serial interface format will be changed slightly to accommodate the various devices which must send and receive messages. The UART on the 8051 now must be initialized to mode 2. This mode uses one start bit, eight data bits, a stick bit, and a stop bit. The stick bit will be used to interrupt the processor when the beginning of a serial message is being received. Thus, any device sending out a message should still adhere to the message format, but must add the following conditions. First, the stick bit must be set to one for the sync byte of every message. Second, the stick bit must be cleared for all other bytes in a message. All devices receiving these messages will now set the UART to fire an interrupt only when a byte with the stick bit set is received.

To accommodate the dramatic increase in speed of the serial link, the serial receive routine in the clock must be rewritten so that it will be able to keep up with the incoming stream of bytes. This means that once the ISR is called by a byte with the stick bit set, the remainder of the message will be polled for in the serial routine. The completed message will then be pushed into a message queue, just the same as before.

The code for 'main' for the clock function must be changed to adapt to the new UART mode. Note that timer one is no longer needed for a baud rate generator and is now free for whatever uses come up for it. In the case of the clock, it will be used as an interbyte time-out in the ISR during reception to ensure that the clock does not infinitely wait for a serial byte that is not coming.

Listing 0-5

```
// define constants for message parsing
#define SYNC          0x33
#define CLOCK_ADDR    0x43
```

```
// define constants for the input commands
#define CMD_RESET          0x01
#define CMD_TIMESYNC      0x02
#define CMD_TIMEREQ       0x03
#define CMD_DISPLAY       0x04
#define CMD_ACK           0xFF

// set interbyte timeout to 128 instruction cycles
#define TO_VAL            0x80

unsigned char data recv_chksum,      // holds the current checksum
                                     // value of the incoming
                                     // message
                                     recv_buf[35];      // holds the incoming message

unsigned char  trans_buf[7],         // holds the outgoing message
               trans_ctr,            // index into trans_buf
               trans_size,           // holds the total number of
                                     // bytes to send
               trans_chksum;         // computes the output
                                     // checksum

unsigned char code valid_cmd[256]={  // This array determines if
                                     // the current command byte is
                                     // valid.  If the corresponding
                                     // entry is 1 the command ID is
                                     // valid.

    0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // 00 - 0F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // 10 - 1F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // 20 - 2F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // 30 - 3F

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // 40 - 4F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // 50 - 5F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // 60 - 6F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // 70 - 7F

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // 80 - 8F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // 90 - 9F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // A0 - AF
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // B0 - BF

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  // C0 - CF
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // D0 - DF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // E0 - EF
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, // F0 - FF
};

/*****
Function:      ser_xmit
Description:   Handles a serial transmit interrupt.
Parameters:   none.
Returns:      nothing.
Side Effects:  none.
*****/

void ser_xmit(void) {
    trans_ctr++;                // move the output index up
    if (trans_ctr<trans_size) { // see if all data is output
        if (trans_ctr==(trans_size-1)) { // send checksum as last byte
            SBUF=trans_chksum;
        } else {
            SBUF=trans_buf[trans_ctr]; // send current byte
            trans_chksum+=trans_buf[trans_ctr]; // update checksum
        }
    }
}

/*****
Function:      push_msg
Description:   Puts the current message in the serial message
               queue.
Parameters:   none.
Returns:      nothing.
Side Effects:  none.
*****/

void push_msg(void) {
    memcpy(msg_buf, recv_buf, recv_buf[1]+2);
    msg_buf_valid=1;            // indicate that buffer is good
    recv_chksum=SYNC+CLOCK_ADDR; // set checksum to value of
                                // first two message bytes.
                                // This makes the assembly
                                // code a little faster and
                                // easier
}

/*****
Function:      execute_cmd
```

Description: Executes the actions associated with a serial command.

Parameters: none.

Returns: nothing.

Side Effects: none.

```
*****/
void execute_cmd(void) {
    bit need_ack=1;
    switch (recv_buf[1]) {          // act upon the command
    case CMD_RESET:                // reset the clock to 0
        EA=0;                    // stop interrupts while the
                                // time structure is used

        curtime.sec=curtime.min=curtime.hour=0;
        timeholder=curtime;
        EA=1;                    // allow interrupts now
        break;
    case CMD_TIMESYNC:            // set the clock
        EA=0;                    // stop interrupts while the
                                // time structure is used

        curtime.hour=recv_buf[3];
        curtime.min=recv_buf[4];
        curtime.sec=recv_buf[5];
        timeholder=curtime;
        EA=1;                    // allow interrupts now
        break;
    case CMD_TIMEREQ:            // report system time
        trans_buf[0]=SYNC;        // set up header
        trans_buf[1]=CLOCK_ADDR;
        trans_buf[2]=CMD_TIMEREQ; // sending current time
        trans_buf[3]=3;
        EA=0;                    // stop interrupts while the
                                // time structure is used

        trans_buf[4]=curtime.hour; // put current time in the
                                // data block

        trans_buf[5]=curtime.min;
        trans_buf[6]=curtime.sec;
        EA=1;                    // allow interrupts now
        trans_ctr=0;              // set buffer pointer to first
                                // byte

        trans_size=8;             // eight bytes total
        need_ack=0;
        break;
    case CMD_DISPLAY:            // display ASCII message
        recv_buf[34]=0;          // set a null terminator at
```

```

                                                                    // the end of the string
    printf("\xFF%s", &recv_buf[2]); // display the string
    display_time=100;              // set timeout to 5 seconds
    break;
}
if (need_ack) {                // build ack if necessary
    trans_buf[0]=SYNC;          // set up header
    trans_buf[1]=CLOCK_ADDR;
    trans_buf[2]=CMD_ACK;
    trans_buf[3]=1;
    trans_buf[4]=recv_buf[1];    // put in command being ack'ed
    trans_ctr=0;                 // set buffer pointer to first
                                // byte
    trans_size=6;                // six bytes total
}
SBUF=SYNC;                     // send first byte
trans_chksm=SYNC;              // initialize checksum
}

/*****
Function:      main
Description:   This is the entry point of the program. This
               function initializes the 8051, enables the correct
               interrupt source and enters idle mode. The idle
               mode loop checks after each interrupt to see if
               the LCD panel must be updated.
Parameters:    None.
Returns:       Nothing.
*****/
void main(void) {
    disp_init();                // set up display
    TH1=TO_VAL;                 // set reload for timer 1 to
                                // the interbyte timeout
                                // period
    TMOD=0x21;                  // set timer 1 to 8 bit mode
                                // and timer 0 to 16 bit mode
    TCON=0x15;                  // start timer 0. both
                                // external ints are edge
    SCON=0xB0;                  // set the UART to mode 2 and
                                // require that the stick bit
                                // is high for an interrupt
    IE=0x92;                    // enable the serial and TF0
                                // interrupts
    for (;;) {
```

```

if (_testbit(msg_buf_valid)) { // check for a new serial message
    execute_cmd();           // execute the command
}
if (disp_update) {
    disp_time();             // display new time;
}
PCON=0x01;                  // enter idle mode
}
}

```

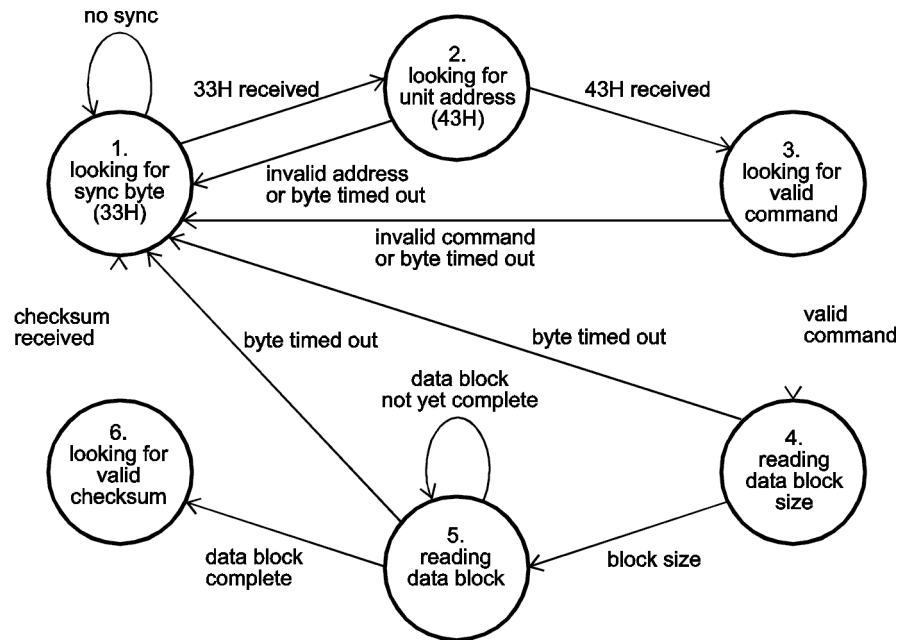


Figure 0-7 - New Receive FSA

The changes to 'main' are relatively simple and straightforward. Additionally, the code for handling serial transmit interrupts has been given its own function which will be called from the new serial ISR. Another function called 'push_message' has been added to help the serial ISR. It will simply copy the current message into the serial message queue for handling later. While these changes are all simple to implement, the changes to the serial interrupt service routine will not be so simple. The serial ISR now must be coded in assembler. When TI is the cause of the interrupt, the new C routine ('ser_xmit') will be called to handle transmission of the next byte. However, the portion of the interrupt routine which services receive interrupts will remain in assembler because of the short amount of processing time available between bytes.

Following is a listing of the new ISR, as coded in assembler. Note again that timer one is now used as an interbyte time-out counter while the ISR polls for the next incoming byte. If this time-out expires, the current message is terminated and the FSA returns to its original state. The new FSA is shown in Figure 0-7. The only change to it is the interbyte time-out that prevents the software from infinitely waiting for a serial byte that may never arrive.

The source code for the ISR simply implements the new FSA. The code is now written in assembly so that it can guarantee that no byte will be missed due to excessive code delays. In mode two of the UART, time is short in between bytes (slightly less than 30 instruction cycles). This means that if you waste more than 30 cycles before retrieving a byte from SBUF that you may have lost a byte. Thirty

cycles is not a lot of time, and thus each instruction cycle is at a premium. The code for the new assembly ISR in Listing 0-6 was written with this in mind.

Listing 0-6

```
EXTRN      DATA      (recv_chksum) ; used to calculate the
                                           ; checksum of the incoming message
EXTRN      DATA      (recv_buf)    ; holds incoming message
EXTRN      CODE        (ser_xmit)    ; handles serial transmit interrupts
EXTRN      CODE        (valid_cmd)   ; table which decides if a
                                           ; command byte is valid
EXTRN      CODE        (push_msg)    ; puts a complete message in
                                           ; the message queue

SYNC       EQU        33H           ; constant indicating the sync
                                           ; byte
CLK_ADDR   EQU        43H           ; constant indicating this
                                           ; unit's address

CSEG       AT          23H
           ORG          23H
           LJMP        SER_INTR     ; install ser_intr in the vector

PUBLIC     SER_INTR
?PR?SER_INTR?SER_INTR SEGMENT CODE
           RSEG        ?PR?SER_INTR?SER_INTR

;*****
;
; Function:      readnext
; Description:   Reads in one byte from the serial port and
;               returns. Sets the carry flag if a timeout occurs.
; Parameters:    none.
; Returns:       the byte read in the location pointed to by R0
; Side Effects:  none.
;*****
readnext:    CLR        C            ; clear the return flag
             MOV        TL1, TH1     ; use T1 as a timeout counter
             SETB       TR1          ; start T1
RN_WAIT:     JBC        RI, RN_EXIT   ; if RI is set, a byte has rung in
             JBC        TF1, RN_TO    ; see if the byte has timed out
RN_TO:       SETB       C            ; interbyte time exceeded,
                                           ; return error
             CLR        TR1          ; stop the timer
             RET
```

```
RN_EXIT:    MOV    A, SBUF        ; write the byte to R0's point
            CLR    TR1            ; stop the timer
            RET

;*****
; Function:    SER_INTR
; Description: This is the ISR for the 8051's UART. It is called
;              automatically by the hardware for both transmit
;              and receive interrupts.
; Parameters:  none.
; Returns:    nothing.
; Side Effects: none.
;*****
SER_INTR:    JBC    RI, RECV_INT   ; check for receive interrupt
CHK_XMIT:    JBC    TI, XMIT_INT   ; finally check for transmit
                                           ; interrupt
            RETI

XMIT_INT:    LCALL   ser_xmit      ; transmit interrupt - call
                                           ; handler for it
            RETI

                                           ; the following code block is
                                           ; jumped into depending on how
                                           ; many variables have been
                                           ; pushed on the stack
CHK_XMIT3:    POP    DPL
            POP    DPH
CHK_XMIT2:    POP    00H
CHK_XMIT1:    POP    ACC
            SETB   SM2            ; ensure that the stick bit
                                           ; requirement is restored
            JMP    CHK_XMIT       ; after restoring stack, check
                                           ; for transmit interrupt

RECV_INT:    PUSH   ACC            ; will be using ACC - save it
            MOV    A, SBUF        ; save the incoming byte
            CJNE   A, #SYNC, CHK_XMIT1 ; if it isn't a sync
                                           ; byte get out

            CLR    SM2            ; remove the stick bit
                                           ; requirement now

            PUSH   00H            ; save R0 before use
            MOV    R0, #recv_buf  ; get the base address of
```

CHAPTER 8 - THE SERIAL PORT

```
                                ; recv_buf in R0
CALL    readnext                ; read in the next byte
JC      CHK_XMIT2                ; timeout - exit
CJNE    A, #CLK_ADDR, CHK_XMIT2 ; this byte must be the
                                ; clock's address

CALL    readnext                ; get the next byte
JC      CHK_XMIT3                ; timeout - exit

PUSH    DPH                      ; save the DPTR
PUSH    DPL

MOV      DPTR, #valid_cmd ; prepare to validate the
                                ; command byte
MOVC     A, @A+DPTR            ; use the command byte as an
                                ; offset into the validation
                                ; table
JZ       CHK_XMIT3              ; if the value in the table is
                                ; 1, this is a valid command
MOV      @R0, A                ; save the command byte
ADD      A, recv_chksum ; update the checksum
MOV      recv_chksum, A
INC      R0                      ; move the buffer pointer

CALL    readnext                ; get the size byte
JC      CHK_XMIT3                ; timeout - exit

MOV      @R0, A                ; save the size byte
ADD      A, recv_chksum ; update the checksum
MOV      recv_chksum, A
MOV      A, @R0                ; if the size byte is 0, go on
                                ; to the checksum
JZ       RECV_CHK
MOV      DPL, A                ; save the size byte as a counter

RECV_DATA: INC    R0              ; move the buffer pointer
CALL    readnext                ; get the next data block byte
JC      CHK_XMIT3                ; timeout - exit
ADD      A, recv_chksum ; update the checksum
MOV      recv_chksum, A
DJNZ     DPL, RECV_DATA ; see if there are any more
                                ; bytes to receive

RECV_CHK: CALL    readnext        ; read in the checksum byte
JC      CHK_XMIT3                ; timeout - exit
```

```
        CJNE    A, recv_chksum, CHK_XMIT3 ; validate it

; push_msg should set the chksum var to the unit address plus the
; sync byte

        LCALL   push_msg          ; good message, push it in the
                                   ; receive queue

        JMP     CHK_XMIT3

        END
```

The above code listing is small. This is good because of two things. First, there is not much time between bytes to be messing around, so the fewer cycles wasted, the better. Second, the code remains simple and easy to maintain. You will note that the logic to poll and receive a byte, plus the time-out logic is all embedded in one function - 'readnext'. This function simply waits for the next byte and returns with the carry set if the time-out period elapses before the next byte rings in. The calling function expects to receive the next byte in the accumulator. It only has to check the value of the carry bit to determine if the contents of the accumulator are good or not. In the case of this ISR, when the 'readnext' function returns the error code the current message parsing is aborted. Using the accumulator to hold the return value allows the calling function to perform a lot of checks on the value of the incoming byte without having to perform any extra move operations. Additionally, when the byte must be stored somewhere, it usually saves one instruction cycle by having to move the byte from the accumulator instead of some other location.

Essentially, the serial scheme as implemented is a master - slave system in which the clock project functions as a slave. Managing access to the shared serial line is simple for the slaves, since all the control resides in the master device. Even the control in the master device is simple because it knows that no device will talk unless it is told to do so. Thus, there are no collisions. The next chapter will discuss ways to avoid collisions in serial systems where there is not a master.

Conclusion

This chapter presented methods for simple serial connections between your 8051 project and other devices such as a PC and other 8051's. The PC interface presented was simple, but got the job done. If you wanted a more complete RS-232 driver there are many commercially available UARTs from companies such as National Semiconductor that you can incorporate into your design. The parts will then be external UARTs that must be interfaced to the 8051 via its bus. You can also use an RS-232 line driver chip which will convert the 8051's zero to five volt serial signals to the correct RS-232 voltage levels. This approach allows you to use the UART on board the 8051 for serial communications. The next chapter will discuss the use of the UART in the context of network schemes more complex than was done in this chapter. This will be of interest to anyone who has a project that must do a large amount of communications.

- Networking with the 8051

Multiplexing the Serial Port

Suppose that the master in Chapter Eight's serial system must still convey all the slave's data to a monitoring station. Because of some design issues, such as a more complex user interface, the monitoring station is a PC. This PC will periodically send data and commands to the serial master (which will be called the System Monitor) and expect that the System Monitor will respond back with the necessary data. The link between the PC and the System Monitor will be identical to the link between the System Monitor and the slaves, except for two things. First, the PC will initiate all serial conversations in the same way that the System Monitor initiated all conversations with the slave devices hooked to its serial line. Second, the PC will use standard RS-232 format serial data. This means that the System Monitor must expect the PC to send 10 bit frames at 9600 baud to it at any time. For the System Monitor, this poses something of a challenge. It must always be capable of responding to serial interrupts from the PC in addition to performing its polling duties with its slave devices.

There are a couple of design approaches that can be taken with the System Monitor. The first approach is to add an RS-232 type UART to the core hardware which consists of an 8051. This external UART will fire an interrupt to the 8051 whenever a byte has rung in or out of the device. The second approach is to "multiplex" the serial port that already exists on the 8051. This means that it will be used to service both the PC messages and the shared serial line with the slaves. Since this device is being designed by people who believe that cheaper is better and that lower parts counts means cheaper, they opt to go with the second design. After all, it eliminates the need for an external UART and a 12 volt supply to go with it to handle the RS-232 stuff.

Thus, the software for the System Monitor has been greatly complicated. However, the task is "do-able" and once again software saves the day! The software solution will not be the overall best since there will be times that the serial port on the System Monitor is communicating with the slave devices and will be unavailable to receive messages from the PC. To correct for this, the PC will have to be able to recognize situations in which the System Monitor has failed to receive a message. This will be handled by requiring the sender to retry the same message. Similarly, the serial port control logic in the System Monitor must be written to assume that the PC has the highest priority on the serial channel. What this means is that whenever the UART is not being used to communicate with slave devices it is set up to receive serial input from the PC. In addition to changing the mode and the baud rate of the UART to communicate with both the PC and the slaves, the hardware design of the System Monitor will allow the input to the UART to be gated so that one of the serial "channels" can be rejected while the other one is in use. The hardware design of the System Monitor is shown in Figure 0-1.

Listing 0-1

```

/*****
Function:      ser_9600
Description:   Dispatches UART interrupts to the correct routine.
               Used only when the UART is at 9600 baud.
Parameters:    none.
Returns:       nothing.
Side Effects:  none.
*****/

void ser_9600(void) {
    if (_testbit_(TI)) {                // call the transmit handler
        ser_xmit();
    }
    if (_testbit_(RI)) {                // call the receive handler
        ser_rcv();
    }
}

/*****
Function:      ser_rcv
Description:   Handles a serial receive interrupt when the system
               is run at 9600 baud.
Parameters:    none.
Returns:       nothing.
Side Effects:  none.
*****/

void ser_rcv(void) {
    unsigned char c, temp;
    c=SBUF;
    switch (rcv_state) {
    case FSA_INIT:                      // looking for sync byte
        if (c==SYNC) {                 // check for sync byte
            rcv_state=FSA_ADDRESS;      // move to next state
            rcv_timeout=RCV_TIMEOUT;    // set interbyte timeout
            rcv_chksm=SYNC;             // set initial checksum value
        }
        break;
    case FSA_ADDRESS:                  // looking for unit ID
        if (c==SM_ADDR) {              // make sure the message is
                                        // for the clock
            rcv_state=FSA_COMMAND;      // set next FSA state
            rcv_timeout=RCV_TIMEOUT;    // set interbyte timeout
            rcv_chksm+=c;               // maintain checksum
        } else {                      // message is not for the clock

```

```
    recv_state=FSA_INIT;           // return to initial state
    recv_timeout=0;                // clear interbyte timeout
}
break;
case FSA_COMMAND:                 // looking for cmd id
    if (!valid_cmd[c]) {          // make sure command is good
        recv_state=FSA_INIT;      // reset FSA
        recv_timeout=0;
    } else {
        recv_state=FSA_DATASIZE;  // move to next FSA state
        recv_chksum+=c;           // update checksum
        recv_buf[0]=c;           // save command ID
        recv_timeout=RECV_TIMEOUT; // set interbyte timeout
    }
    break;
case FSA_DATASIZE:               // looking for length byte
    recv_chksum+=c;              // update checksum
    recv_buf[1]=c;               // save data block size
    if (c) {                     // see if there is a data block
        recv_ctr=2;              // set indexing byte
        recv_state=FSA_DATA;      // move to next FSA state
    } else {
        recv_state=FSA_CHKSUM;    // message done, get checksum
    }
    recv_timeout=RECV_TIMEOUT;
    break;
case FSA_DATA:                   // reading in data block
    recv_chksum+=c;              // update checksum
    recv_buf[recv_ctr]=c;        // save data byte
    recv_ctr++;                  // update count of data block
    if ((recv_ctr-2)==recv_buf[1]) { // see if receive data block
        // counter minus the offset into
        // the receive buffer == the
        // data block size
        recv_state=FSA_CHKSUM;    // done receiving data block
    }
    recv_timeout=RECV_TIMEOUT;    // set interbyte timeout
    break;
case FSA_CHKSUM:                 // reading in checksum
    if (recv_chksum==c) {        // verify checksum
        push_msg();
    }
default:
    recv_timeout=0;
```

```

    recv_state=FSA_INIT;
    break;
}
}
/*****
Function:      ser_xmit
Description:   Handles a serial transmit interrupt.
Parameters:   none.
Returns:      nothing.
Side Effects:  none.
*****/
void ser_xmit(void) {
    trans_ctr++;                // move the output index up
                                // see if all data is output
    if (trans_ctr < ser_queue[XMIT_QUEUE]
        .entry[ser_queue[XMIT_QUEUE].head].size) {
        // send checksum as last byte
        if (trans_ctr == (ser_queue[XMIT_QUEUE]
            .entry[ser_queue[XMIT_QUEUE].head]
                .size-1)) {
            SBUF=trans_chksum;
        } else {
            // send current byte
            SBUF=ser_queue[XMIT_QUEUE]
                .entry[ser_queue[XMIT_QUEUE].head].buf[trans_ctr];
            // update checksum
            trans_chksum+=ser_queue[XMIT_QUEUE]
                .entry[ser_queue[XMIT_QUEUE].head]
                    .buf[trans_ctr];
        }
    } else {
        // the current message is done
        // sending
        // if no reply is required...
        if (!ser_queue[XMIT_QUEUE]
            .entry[ser_queue[XMIT_QUEUE].head].retries) {
            if (queue_pop(XMIT_QUEUE)) { // get out the next message
                check_stat();           // see if the pending message
                                        // can go
            }
        } else {
            xmit_timeout=XMIT_TIMEOUT; // set reply timeout counter
        }
    }
}
}

```

The source listing for the assembly code routine to handle the high speed serial I/O is shown below.

Listing 0-2

```
; FILE NAME: SERINTR.A51
EXTRN      DATA    (uart_mode)      ; holds the value to write into SCON
EXTRN      BIT      (baud_9600)      ; set when the baud rate is 9600
EXTRN      DATA    (recv_chksum)    ; used to calculate the
                                      ; checksum of the incoming message
EXTRN      DATA    (recv_buf)       ; holds incoming message
EXTRN      CODE     (ser_xmit)       ; handles serial transmit interrupts
EXTRN      CODE     (valid_cmd)      ; table which decides if a command
                                      ; byte is valid
EXTRN      CODE     (push_msg)       ; puts a complete message in the
                                      ; message queue
EXTRN      CODE     (ser_9600)       ; handler for slow serial comm

SYNC       EQU      33H              ; constant indicating the sync byte
SM_ADDR    EQU      40H              ; constant indicating this unit's
                                      ; address

CSEG       AT        23H
           ORG        23H
           LJMP      SER_INTR        ; install ser_intr in the vector

           PUBLIC    SER_INTR
?PR?SER_INTR?SER_INTR SEGMENT CODE
           RSEG      ?PR?SER_INTR?SER_INTR

;*****
;
; Function:      readnext
; Description:   Reads in one byte from the serial port and
;               returns. Sets the carry flag if a timeout occurs.
; Parameters:    none.
; Returns:       the byte read in the location pointed to by R0
; Side Effects:  none.
;*****
;
readnext:    CLR      C                ; clear the return flag
             MOV      TL1, TH1        ; use T1 as a timeout counter
             SETB     TR1             ; start T1
RN_WAIT:    JBC      RI, RN_EXIT      ; if RI is set, a byte has rung in
             JBC      TF1, RN_TO      ; see if the byte has timed out
RN_TO:      SETB     C                ; interbyte time exceeded, return
                                      ; error
             CLR      TR1             ; stop the timer
```

```

RET
RN_EXIT:    MOV    A, SBUF        ; write the byte to R0's point
           CLR    TR1            ; stop the timer
           RET

;*****
;
; Function:    SER_INTR
; Description: This is the ISR for the 8051's UART. It is called
;              automatically by the hardware for both transmit
;              and receive interrupts.
; Parameters:  none.
; Returns:    nothing.
; Side Effects: none.
;*****
SER_INTR:    JNB    baud_9600, FAST_ISR ; make sure the right
                                           ; handler gets called

           LCALL   ser_9600
           RETI

FAST_ISR:    JBC    RI, RECV_INT  ; check for receive interrupt
CHK_XMIT:    JBC    TI, XMIT_INT  ; finally, check for transmit
                                           ; interrupt

           RETI

XMIT_INT:    LCALL   ser_xmit      ; transmit interrupt - call handler
                                           ; for it

           RETI

                                           ; the following code block is jumped
                                           ; into depending on how many
                                           ; variables have been pushed

CHK_XMIT3:    POP    DPL
           POP    DPH
CHK_XMIT2:    POP    00H
CHK_XMIT1:    POP    ACC
           MOV     SCON, uart_mode ; ensure that the stick bit
                                           ; requirement is restored
           JMP     CHK_XMIT        ; after restoring stack, check for
                                           ; transmit interrupt

RCV_INT:    PUSH    ACC            ; will be using the accumulator -
                                           ; save it
           MOV     A, SBUF        ; save the incoming byte
           CJNE    A, #SYNC, CHK_XMIT1 ; if it isn't a sync byte
                                           ; get out

```

```
PUSH    00H          ; save R0 before use
MOV     R0, #recv_buf ; get the base address of recv_buf
                        ; in R0
CALL    readnext     ; read in the next byte
CJNE    A, #SM_ADDR, CHK_XMIT2 ; this byte must be the
                        ; clock's address

CLR     SM2          ; remove the stick bit requirement
CALL    readnext     ; get the next byte

PUSH    DPH          ; save the DPTR
PUSH    DPL
MOV     DPTR, #valid_cmd ; prepare to validate the
                        ; command byte
MOVC    A, @A+DPTR    ; use the command byte as an offset
                        ; into the validation table
JZ      CHK_XMIT3     ; if the value in the table is 0,
                        ; this is a valid command
MOV     @R0, A        ; save the command byte
ADD     A, recv_chksum ; update the checksum
MOV     recv_chksum, A
INC     R0            ; move the buffer pointer

CALL    readnext     ; get the size byte
MOV     @R0, A        ; save the size byte
ADD     A, recv_chksum ; update the checksum
MOV     recv_chksum, A
MOV     A, @R0        ; if the size byte is 0, go on to
                        ; the checksum
JZ      RECV_CHK
MOV     DPL, A        ; save the size byte as a counter

RECV_DATA: INC R0      ; move the buffer pointer
CALL    readnext     ; get the next data block byte
ADD     A, recv_chksum ; update the checksum
MOV     recv_chksum, A
DJNZ    DPL, RECV_DATA ; see if there are any more bytes
                        ; to receive

RECV_CHK: CALL readnext ; read in the checksum byte
CJNE    A, recv_chksum, CHK_XMIT3 ; validate it

; push_msg should set the chksum var to the unit address plus the
```

```
; sync byte
        LCALL    push_msg      ; good message, push it in the queue
        JMP      CHK_XMIT3

        END
```

The System Monitor software will use the basic tick routine from the clock project. In the System Monitor, the tick will merely count down the time until the System Monitor must next poll the slave devices. Once this time has expired, the tick will initiate a transfer to the first slave if the serial port is available for use at the higher baud rate. If the serial port is not available, the messages will be pushed into a message queue where messages wait to have the serial port given to them.

Listing 0-3

```
/******
Function:      system_tick
Description:   This is the ISR for timer 0 overflows. It
               maintains the timer and reloads it with the
               correct value for a 50ms tick. The time is counted
               in this routine for any functions that require a
               timeout.
Parameters:    None.
Returns:       Nothing.
*****/
void system_tick(void) interrupt 1 {
    TR0=0;                      // temporarily stop timer 0
    TH0=RELOAD_HIGH;           // set the reload value
    TL0=RELOAD_LOW;
    TR0=1;                      // restart the timer

    if (poll_time) {           // see if it's time for another
                                // slave polling sequence

        poll_time--;
        if (!poll_time) {
            poll_time=POLL_RATE;
            start_poll();       // push the set of polling
                                // messages
        }
    }

    if (xmit_timeout) {        // see if the message at the
                                // head of the transmit queue
                                // has not received its response

        xmit_timeout--;
        if (!xmit_timeout) {   // it has...check for message
                                // retries

```

```
ser_queue[XMIT_QUEUE]
    .entry[ser_queue[XMIT_QUEUE].head].retries--;
if (ser_queue[XMIT_QUEUE]
    .entry[ser_queue[XMIT_QUEUE].head].retries) {
    // another retry - start the
    // message transmission
    SCON=uart_mode=ser_queue[XMIT_QUEUE]
        .entry[ser_queue[XMIT_QUEUE].head]
        .uart_mode;
    ser_queue[XMIT_QUEUE]
        .entry[ser_queue[XMIT_QUEUE].head]
        .status=STAT_SENDING;
    SBUF=ser_queue[XMIT_QUEUE]
        .entry[ser_queue[XMIT_QUEUE].head].buf[0];
    trans_ctr=0;                // set buffer pointer to first
                                // byte set size indicator
    trans_size=ser_queue[XMIT_QUEUE]
        .entry[ser_queue[XMIT_QUEUE].head].size;
                                // set message checksum
    trans_chksum=ser_queue[XMIT_QUEUE]
        .entry[ser_queue[XMIT_QUEUE].head]
        .buf[0];
} else {                        // no more retries...this
                                // message is done
    if (queue_pop(XMIT_QUEUE)) { // get rid of it and see if
                                // there are any pending
                                // messages make sure the new
                                // message can use the serial
                                // port now. it can if the
                                // current UART mode is the same
                                // as its UART mode or
                                // if the receive FSA is idle
        check_stat();           // start the next message
                                // transmitting
    } else {                    // ensure that the UART is set
                                // to receive from the PC
        TH1=T09600_VAL;         // set reload for timer 1 to
                                // the interbyte timeout period

        TR1=0;
        TL1=TH1;                // force reload the timer so
                                // that the UART immediately has
                                // the right baud rate

        TF1=0;
        TR1=1;
```

```
        baud_9600=1;
        P1=0x09;           // allow PC I/O
    }
}
}
}

if (recv_timeout) {       // check the receive interbyte
                           // timeout

    recv_timeout--;
    if (!recv_timeout) {   // if it has expired...
        recv_state=FSA_INIT; // reset the FSA
        check_stat();
    }
}
}

/*****
Function:      start_poll
Description:    This function loads the transmit queue with the
                required polling messages for the system slaves.
Parameters:    None.
Returns:       Nothing.
*****/
void start_poll(void) {
    unsigned char i, temp;

                                // push one message for each slave
    for (i=0; i<NUM_SLAVES; i++) {
        temp=queue_push(XMIT_QUEUE); // get a queue entry
        if (temp!=0xFF) {           // make sure its valid
                                // copy the message to the entry
            memcpy(ser_queue[XMIT_QUEUE].entry[temp].buf,
                slave_buf[i],
                slave_buf[i][3]+4);

                                // set the message size byte
            ser_queue[XMIT_QUEUE].entry[temp].size=slave_buf[i][3]+5;
                                // set the message retries to 3
            ser_queue[XMIT_QUEUE].entry[temp].retries=3;
                                // set the pending message UART
                                // mode
            ser_queue[XMIT_QUEUE].entry[temp].uart_mode=BAUD_300K;
                                // set the message status
            ser_queue[XMIT_QUEUE].entry[temp].status=STAT_WAITING;
        }
    }
}
```

```
}  
    check_stat();                // see if the next message can  
                                // begin use of the UART  
}
```

You will note that timer one is used in both serial modes. In the 9600 baud mode it serves as the baud rate generator for the UART. Interbyte time-outs are provided by the system tick being generated by timer zero. In the 300K baud mode, timer one serves as the interbyte time-out mechanism. The multiplexing of timer one means that when the UART mode is changed, the software must be careful to also change the reload value of the timer. One important thing when changing the timer and UART to operate at 9600 baud is that the low byte of timer one should be force fed the reload value in TH1 immediately after the mode is changed. Otherwise, the first timer one overflow will not occur at the right time and you will not immediately get 9600 baud. This is a minor detail but if overlooked can hose up the first byte of an outgoing message.

The serial port will be treated as a system resource which each outgoing message must wait its turn for. The outgoing queue functions as the staging area where the messages await their turn to use the serial port. The message at the head of the queue will be given the use of the serial port only under certain conditions: a message may use the serial port whenever it is idle, i.e.: if both the transmit and the receive FSAs are in the idle state. A message may also use the serial port if the transmit FSA is idle, the serial port is set to the correct baud rate for the message and an incoming message is being received.

The message at the head of the transmit queue is checked every time a receive message is pushed into the queue (since the receive FSA is then idle), every time a receive message is popped (since it may be a response for a message in the transmit queue thus clearing it), and whenever the message time-out timer expires. This ensures that the UART is passed from message to message in a timely manner. The function performing this task is shown in Listing 0-4.

Listing 0-4

```
/******  
Function:      check_stat  
Description:   Checks the transmit queue to see if the head  
               message is waiting for the serial port and if it  
               can be started. If it can begin transmission, it  
               is assigned use of the UART and transmission is  
               begun.  
Parameters:   none.  
Returns:      nothing.  
Side Effects: none.  
*****/  
void check_stat(void) {  
    if (ser_queue[XMIT_QUEUE].head!=UNUSED) { // if there is a  
                                                // message waiting...  
                                                // check its status  
  
        if (ser_queue[XMIT_QUEUE]  
            .entry[ser_queue[XMIT_QUEUE].head]  
            .status == STAT_WAITING) {  
                                                // the next message is waiting,  
                                                // so check the UART availability
```

```
if (recv_state==FSA_INIT ||
    (uart_mode == ser_queue[XMIT_QUEUE]
      .entry[ser_queue[XMIT_QUEUE].head]
      .uart_mode)) {
    // start the message
    SCON=uart_mode=ser_queue[XMIT_QUEUE]
      .entry[ser_queue[XMIT_QUEUE].head]
      .uart_mode;
    if (uart_mode==BAUD_300K) { // make sure T1 has the right
        // reload value

        TH1=T0300K_VAL;
        baud_9600=0;
        P1=0x06; // allow slave I/O
    } else {
        TH1=T09600_VAL; // set reload for timer 1 to
        // the interbyte timeout period

        TR1=0;
        TL1=TH1; // force reload the timer so
        // that the UART immediately has
        // the right baud rate

        TF1=0;
        TR1=1;
        baud_9600=1;
        P1=0x09; // allow PC I/O
    }
    ser_queue[XMIT_QUEUE]
      .entry[ser_queue[XMIT_QUEUE].head]
      .status=STAT_SENDING;
    SBUF=ser_queue[XMIT_QUEUE]
      .entry[ser_queue[XMIT_QUEUE].head].buf[0];
    trans_ctr=0; // set buffer pointer to first
    // byte
    trans_size=ser_queue[XMIT_QUEUE]
      .entry[ser_queue[XMIT_QUEUE].head].size;
    trans_chksum=ser_queue[XMIT_QUEUE]
      .entry[ser_queue[XMIT_QUEUE].head].buf[0];
}
}
}
```

All incoming messages are pushed into a receive queue upon successful receipt. The main loop then checks the receive queue to see if there are any pending receive messages. If there are, a function is called to execute the actions associated with the message at the head of the queue. For the most part commands from the PC will cause the Serial Monitor to take some action, such as passing data to a

slave, and then send an acknowledge message to the PC. These response messages will be built in the receive executive and pushed into the transmit queue. If the message is an acknowledge or data report from either the PC or a slave, the executive checks to see if the message satisfies the message at the head of the serial transmit queue. If it does, the transmit queue head is popped and the next message begins transmission assuming the necessary conditions are met. The basic structure of this function is shown in Listing 0-5.

Listing 0-5

```

/*****
Function:      push_msg
Description:   Puts the current message in the serial message
               queue, ensures that the baud rate is 9600, and
               that T1 is back to auto reload mode.
Parameters:   none.
Returns:      nothing.
Side Effects: none.
*****/
void push_msg() {
    unsigned char temp;
    temp=queue_push(RECV_QUEUE);    // get a queue entry
    if (temp!=0xFF) {               // make sure its valid
                                    // copy the data into the buffer
        memcpy(ser_queue[RECV_QUEUE].entry[temp].buf, recv_buf,
               recv_buf[1]+2);

                                    // set the message size
        ser_queue[RECV_QUEUE].entry[temp].size=recv_buf[1]+2;
                                    // set the message status
        ser_queue[RECV_QUEUE].entry[temp].status=STAT_IDLE;
                                    // ensure that retries is null
        ser_queue[RECV_QUEUE].entry[temp].retries=0;
                                    // record current UART mode
        ser_queue[RECV_QUEUE].entry[temp].uart_mode=uart_mode;
    }
    recv_checksum=SYNC+SM_ADDR;    // set checksum to value of
                                    // first two message bytes.
                                    // This makes the assembly code
                                    // a little faster and easier
    recv_state=FSA_INIT;           // set the FSA to idle mode
    check_stat();                  // see if the next outbound
                                    // message can go
}

/*****
Function:      ser_exec
Description:   This function handles all incoming messages.
*****/
```

```
Parameters:    None.
Returns:       Nothing.
*****/
void ser_exec() {
#ifdef USEEXEC
    do {
        switch (ser_queue[RECV_QUEUE].entry[head].buf[1]) {
            ...
        }
    } while (ser_queue(RECV_QUEUE));    // keep going until all
                                        // messages are handled
#endif
    check_stat();                      // see if the head of the
                                        // transmit queue should be
                                        // started
}
```

Queue Implementation

Because the Serial Monitor project must be able to process the incoming and outgoing messages quickly, the message queues are built around arrays of “entries.” Each entry has storage for one message, a byte indicating the size of the message, the number of times to re-send the message if it is not acknowledged, the mode of the UART required and its associated state. The state variable is used to determine when the message can have use of the UART and when it must release use of the UART in the transmit mode. In the receive mode, the status byte means little. The array of these message entries is fixed in size and the normal head and tail pointers associated with a queue are implemented as indices into this array. This allows the compiler to implement everything by allocating one byte to the head and tail and avoiding all pointer arithmetic.

Listing 0-6

```
typedef struct {                // define a queue entry
    unsigned char buf[MSG_SIZE]; // message data
    unsigned char    size,       // message size
                   retries,     // number of retries
                   uart_mode,   // SCON mask
                   status;      // current message status
} entry_type;

typedef struct {                // define a queue
    unsigned char    head,      // head of the queue
                   tail;       // tail of the queue
    entry_type entry[QUEUE_SIZE]; // array of messages for queue
                                // use
} queue_type;

extern queue_type ser_queue[2]; // need a transmit and receive
                                // queue
```

Normally, a data structure such as a queue would be implemented by dynamically allocating memory from a heap whenever a new entry is needed and freeing the allocated memory when the data is popped from the queue. Keil does provide dynamic memory allocation routines with the C-51 package, but they should not be used for high performance systems if it can at all be avoided.

Because the queues are nothing but fixed size arrays, the push and pop operations become simple to implement. In these queues all pointers are implemented as indices into the array. Messages are popped from the head of the queue and are pushed into the tail of the queue. Some simple checks prevent overflow of the queue.

Listing 0-7

```
/******
Function:    queue_push
Description: Assigns the next queue entry in the circular queue
             specified to the caller.
Parameters:  queue - unsigned char. Must indicate the queue to
             use.
Returns:     The index of the allocated entry or 0xFF if there
             are no available queue entries.
Side Effects: none.
*****/
unsigned char queue_push(unsigned char queue) {
    unsigned char temp;
    if (ser_queue[queue].head==UNUSED) { // if the queue is empty...
                                                // allocate entry 0
        ser_queue[queue].head=ser_queue[queue].tail=0;
        return 0;
    }
```

```
}
temp=ser_queue[queue].tail;          // save the tail value
                                     // increment the tail
ser_queue[queue].tail=(ser_queue[queue].tail+1) % QUEUE_SIZE;
                                     // ensure that the tail does not
                                     // lap the head
if (ser_queue[queue].head == ser_queue[queue].tail) {
    ser_queue[queue].tail=temp;      // there are no available entries
    return 0xFF;
}
return ser_queue[queue].tail;        // return the allocated entry
}

/*****
Function:      queue_pop
Description:   Pops the entry of the circular queue specified.
Parameters:    queue - unsigned char. Must indicate the queue to
               use.
Returns:       0 if the queue is now empty, 1 if the queue still
               holds data.
Side Effects:  none.
*****/
bit queue_pop(unsigned char queue) {
                                     // increment the head
    ser_queue[queue].head=(ser_queue[queue].head+1) % QUEUE_SIZE;
                                     // if the head has caught the
                                     // tail, the queue is empty
    if (((ser_queue[queue].head-ser_queue[queue].tail)==1) ||
        (!ser_queue[queue].head &&
         (ser_queue[queue].tail==QUEUE_SIZE-1))) {
        ser_queue[queue].head=ser_queue[queue].tail=UNUSED;
        return 0;
    }
    return 1;
}
```

You will note that the code which responds to incoming messages is left up to your own design. Rather than make up several meaningless messages, I have left it to you, the reader, to fill in the details of your own project. You must note that each incoming message should be compared to the message at the head of the transmit queue (if that message is waiting for a response). If the current receive message satisfies the transmit queue head, then the transmit queue should be popped so the next message can take possession of the UART.

Messages in the receive queue will not always be in response to the message at the head of the transmit queue; instead they will be commands from the PC. In this case, the slaves may need to be sent some data and the PC will definitely require a response. These messages should be built into transmit queue entries and queued up for transmission.

TDMA Control Using the On-Board Timers

Many communications systems can not use a method as simple minded as the polling scheme used by the System Monitor and its associated slaves. The trouble with a polling scheme is that more and more time will be wasted by polling overhead as more devices are added to the network. Eventually, the time between polls for each unit will become unacceptably long and the amount of data that must be passed from the system master to the slaves becomes overwhelming. When slaves must talk amongst themselves, all the data must pass through the master. This section will present a simple solution to these problems and take our serial network design in a new direction.

Imagine that you still have the same network of devices as in the previous section. The main difference now is that they can all talk and listen to each other. Instead of only hearing what the System Monitor says and sending data only to the system Monitor, all messages that pass across the network can be monitored by all devices. This will affect the design of the slave devices in two significant ways: first, their capability to communicate will be greatly enhanced since one slave can now directly communicate with the other slave without having to use the System Monitor as a go-between; second - the number of serial interrupts on each device will be greatly increased because of the higher amount of potential traffic on the COM lines. The network topography for connecting the slave devices is shown in Figure 0-2.

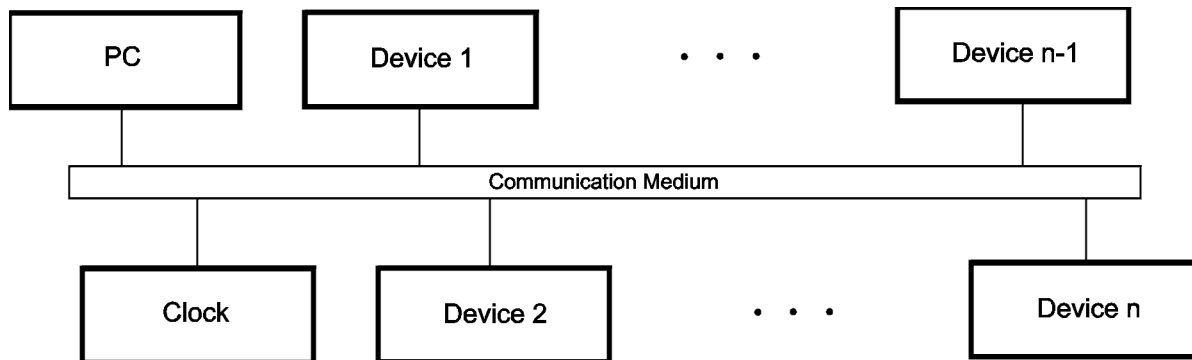


Figure 0-2 - Network Topology

When comparing this to the previous network topology, you will notice that the system Monitor is gone, and the PC is again in the network. The PC will now be responsible for pulling in all the data it needs from the network of 8051's. The baud rate of this network will be 9600 for the convenience of the PC. In the new network scheme, a device on the network may speak to any other device any time its "turn" has come up.

Turns are assigned in sequential fashion. Thus, node one has the first slot, node number two the second, and so on. After each node has had a turn, the first node gets another turn. This sequence continues on infinitely. Thus, during its turn a node owns the network and can talk to whoever it pleases at will. When its turn (or time slot) is up, it must relinquish control of the network. This is the basic concept behind a Time Division - Multiple Access (TDMA) network.

In this design, each device is assigned a slot number, told the total number of slots, and is coded to assume that a slot is a fixed amount of time (this information is given to simplify this example). The size of a slot is usually selected according to the type and size of messages you will be sending out over the network. In this case let's arbitrarily designate a slot as 50ms. Given the knowledge of its slot number, the total number of slots and slot size, it is a simple matter for the software to keep track of each slot and count when the correct slot to talk in is coming up. This section will present a design for implementing this simple TDMA network using an 8051 family member. We will make the simple minded assumption that all devices are started at the same time and thus have synchronized themselves at power on of the

system. In the real world this is not practical, and there are many ways to synchronize the system which depend on system design.

The basic hardware design for a node on the TDMA network is very simple. In this example, a bank of eight DIP switches is connected to port one. The lower nibble of port one will determine a unit's slot number, the upper nibble of port one will determine the number of devices in the network. As stated above, the communications layer of each node will be coded assuming that the TDMA slot width is 50ms.

A "conversation" between two devices on the network is very simple. Assume that slave one wanted to request some data from slave two. In its time slot, slave one would send the data request message to slave two. Slave two will receive and parse the message from slave one. Once it is decoded and determined to be good, slave two pushes it in its receive queue for handling. The message is handled by slave two's executive code which generates the necessary response. This response is pushed into the transmit queue on slave two, and it is sent the next time slave two's slot comes up on the network. Meanwhile, slave one is maintaining a message time-out counter similar to the one used in the System Monitor. This counter allows slave one to decide that slave two is not going to respond to his message for some reason and retry sending the message to slave two.

When a network node's slot comes up on the network, it should look to send as many messages as possible. For example, if there are five messages pending in the transmit queue of the slave, it should not send only one message, but as many as will fit in the 50ms slot. If this means that if three of the messages can be sent, then so be it. The other two messages will remain queued up and will be sent during the node's next TDMA slot.

The structure of the communications layer of a network node is straightforward and will be derived from the code written for the System Monitor. The main loop which performs the system initialization is very similar.

Listing 0-8

```
/******  
Function:      main  
Description:   This is the entry point of the program. This  
               function initializes the 8051, enables the correct  
               interrupt source and enters idle mode.  
Parameters:    None.  
Returns:       Nothing.  
*****/  
void main(void) {  
    slotnum=P1 & 0x0F;           // get this node's slot number  
    slottot=P1 / 16;             // get the total number of nodes  
    TH1=T09600_VAL;             // set reload for timer 1 to  
                                // the interbyte timeout period  
    TH0=RELHI_50MS;             // set the rel value of Timer 0  
    TL0=RELO_50MS;  
    TMOD=0x21;                  // timer 0 = 16 bit  
                                // timer 1 = 8 bit auto reload  
    TCON=0x55;                  // start timers. both external  
                                // ints are edge  
    SCON=BAUD_9600;             // UART mode 2  
    IE=0x92;                    // enable the timer 0 interrupt
```

```

// and the serial interrupt

init_queue();           // set all queues to empty

for (;;) {
    if (tick_flag) {    // check for system tick
        system_tick();
    }

    if (rcv_queue.head!=UNUSED) { // if there is something in the
        // receive queue
        ser_exec();      // handle it
    }
    PCON=0x01;          // enter idle mode
}
}

```

You will note that the 'system_tick' function is called from the main loop now whenever a flag has been set true by the timer 0 ISR. The tick functions are no longer performed by the timer 0 ISR because the slave cannot afford to miss a timer 0 interrupt due to the fact that it is now responsible for counting the system slots and must be very accurate to prevent drift in slot boundaries. The timer 0 routine is now written in assembler and an instruction count is maintained to ensure that every cycle between the 50ms interrupt is accounted for. This is the same technique that was used in the Clock project in Chapter Four.

Listing 0-9

```

EXTRN      BIT      (tick_flag)    ; set to indicate to main loop that
                                   ; a tick has occurred
EXTRN      CODE     (start_xmitt)  ; this node's slot is here...
EXTRN      XDATA    (curslot)      ; keeps track of the current slot

; timer 0 reload value for 50ms based on an 11.059MHz clock. Note
; that the code delay of 9 cycles has been taken out of the reload
; value.
REL_HI EQU    04CH
REL_LOW EQU   007H

SEG        AT      0BH
           ORG      0BH
           LJMP     T0_INTR        ; install T0_intr in the vector

           PUBLIC   T0_INTR
?PR?T0_INTR?T0INT    SEGMENT CODE
           RSEG     ?PR?T0_INTR?T0INT

; *****
;
; Function:      T0_INTR

```

```
; Description:   This is the ISR for the system tick generated by
;               Timer 0. It reloads the timer with the value for
;               50ms minus the code overhead and checks to see if
;               this node's slot has come up.
; Parameters:   none.
; Returns:      nothing.
; Side Effects: none.
;*****
T0_INTR:        CLR     TR0             ; 1, 3  reset timer 0
                MOV     TH0, #REL_HI    ; 2, 5
                MOV     TL0, #REL_LOW   ; 2, 7
                CLR     TF0             ; 1, 8
                SETB    TR0             ; 1, 9
                SETB    tickflag        ; tell main that a tick has occurred
                LCALL   check_slot      ; see if our slot has come up

                PUSH    ACC
                PUSH    B
                PUSH    DPH
                PUSH    DPL

                MOV     DPTR, #curslot ; read the current slot count
                                   ; into the accumulator
                MOVX    A, @DPTR
                INC     A               ; increment the current slot
                MOV     B, A
                MOV     A, P1          ; read in the total number of
                                   ; slots
                SWAP    A
                ANL     A, #00FH
                XCH     A, B
                CLR     C
                SUBB    A, B           ; see if the current slot number
                                   ; is >= the total number of slots
                JC      L1
                CLR     A              ; it is, clear curslot
                MOVX    @DPTR, A
L1:             MOVX    A, @DPTR      ; read in current slot number
                MOV     B, A
                MOV     A, P1          ; read in this node's slot number
                ANL     A, #00FH
                CLR     C
                SUBB    A, B           ; see if curslot==slotnum
```

```
                JNZ     L2             ; it is not
                LCALL   start_xmit     ; curslot==slotnum, start xmission

L2:             POP     DPL
                POP     DPH
                POP     B
                POP     ACC
                RETI

                END
```

The standard tick work for the system timer has been moved to a new C function.

Listing 0-10

```
/******
Function:      system_tick
Description:   This is the ISR for timer 0 overflows. It
               maintains the timer and reloads it with the
               correct value for a 50ms tick. The time is counted
               in this routine for any functions that require a
               timeout.
Parameters:    None.
Returns:       Nothing.
******/
void system_tick(void) {
    unsigned char i;

    tick_flag=0;                // clear the tick flag

    for (i=0; i<MAX_MSG; i++) {
        if (xmit_timeout[i][0]) {        // see if the msg timed out
            xmit_timeout[i][0]--;
            if (!xmit_timeout[i][0]) {    // if so, check retries
                check_msg(xmit_timeout[i][1]);
            }
        }
    }

    if (recv_timeout) {            // check the receive interbyte
                                    // timeout
        recv_timeout--;
        if (!recv_timeout) {        // if it has expired...
            recv_state=FSA_INIT;
            check_stat();
        }
    }
}
```

```
}  
}
```

The 'system_tick' function has many of the same functions it had in the System Monitor project but more closely resembles the tick function in the Clock project in the way that it is invoked from the main loop. The main task of this 'system_tick' function is to maintain message timers to allow the node to recover from network communications errors.

You will note that the timer 0 ISR keeps track of system slots and when it determines that the node's slot has come up calls a function to start transmission of any pending messages. The function which does this is responsible for building a buffer for transmission from as many of the queued messages as it can. The serial interrupt routine for transmit interrupts then sends the data out of this buffer. When replies are received it is the responsibility of the serial messages executive code to determine if a transmit message has been satisfied and can be removed from the queue. Those messages which have not received replies are handled in the time-out code called by the 'system_tick' function. The serial port ISR code from the System Monitor remains unchanged except for the code to transmit data. In this function, the structure is the same, the only difference is that the data is sent out of a new buffer instead of directly out of the transmit queue's head.

Listing 0-11

```
/******  
Function:      start_xmit  
Description:   Fills the trans_buf with as many pending messages  
               as possible. The first message is then started.  
Parameters:    none.  
Returns:       nothing.  
Side Effects:  none.  
*****/  
void start_xmit(void) {  
    unsigned char    maxbytes=45,    // total number of bytes to be  
                                   // sent in one slot time  
    msgnum=0,        // total number of messages put  
                                   // in trans_buf  
    i;  
  
    if (tq_head == UNUSED) {        // don't waste time if the queue  
                                   // is empty  
        return;  
    }  
    while (maxbytes) {               // while there is space in the  
                                   // buffer  
        if (maxbytes>=tq[temp].size) { // make sure the next message fits  
                                   // copy it in and build a checksum  
            for (i=0, chksum=0, trans_size=0;  
                 i<tq[temp].size;  
                 i++, trans_size++) {  
                trans_buf[trans_size]=tq[temp].buf[i];  
                chksum+=tq[temp].buf[i];  
            }  
        }  
        maxbytes-=tq[temp].size;  
        temp++;  
    }  
}
```

```
    }
    trans_buf[trans_size]=chksum; // save the checksum
    xmit_timeout[msgnum][0]=MSG_TIMEOUT; // save the timeout info
    xmit_timeout[msgnum][1]=tq[temp].retries;
    msgnum++; // increment the number of
              // messages in the buffer
    maxbytes-=tq[temp].size+1; // reduce amount remaining by
                              // amount used

    temp=tq[temp].next;
} else {
    maxbytes=0; // get out of this loop
}
}
}

/*****
Function:      ser_xmit
Description:   Handles a serial transmit interrupt.
Parameters:   none.
Returns:      nothing.
Side Effects:  none.
*****/
void ser_xmit(void) {
    trans_ctr++; // move the output index up
                // see if all data is output

    if (trans_ctr < trans_size) {
        // send checksum as last byte

        if (trans_ctr==trans_size-1) {
            SBUF=trans_chksum;
        } else {
            // send current byte

            SBUF=trans_buf[trans_ctr];

            // update checksum

            trans_chksum+=trans_buf[trans_ctr];
        }
    }
}
}
```

The transmit queue must now have a different structure than the receive queue since the head of the queue will not necessarily have been responded to. Thus, if the first three messages are sent out during the node's slot and only the second and third messages are responded to, the queue must still hold the first message, and the second and third messages should be removed. The current queue structure does not allow for this and the new queue must be slightly more complicated.

The transmit queue will still use a fixed amount of possible entries, but instead of assuming that entry one always follows entry zero in the queue order, there must be some sort of link in each entry to

establish the relationship between one entry and the entry that follows it since they entries may not always be in order. The array of entries will now hold two linked lists - a used list and an unused list. When a new entry is required it is unlinked from the free list, filled in, and linked into the used list. Likewise, when an entry is no longer needed it is unlinked from the used list and inserted back into the free list. The source code for the new transmit queue is shown below in Listing 0-12.

Listing 0-12

```

/*****
Function:      tq_init
Description:   Sets up the free and used lists for the transmit
               queue.
Parameters:    none.
Returns:       Nothing.
Side Effects:  none.
*****/

void tq_init(void) {
    tq_head=tq_tail=UNUSED;           // set head and tail pointers to
                                     // empty
    tq_free=0;                        // initialize free list
    for (i=0; i<QUEUE_SIZE; i++) {
        tq[i].next=i+1;
    }
    tq[QUEUE_SIZE-1].next=UNUSED;     // make sure last free list
                                     // entry is grounded
}

/*****
Function:      tq_push
Description:   Assigns the next free queue entry in the transmit
               queue to the caller.
Parameters:    none.
Returns:       The index of the allocated entry or 0xFF if there
               are no available queue entries.
Side Effects:  none.
*****/

unsigned char tq_push(void) {
    unsigned char temp;
    if (tq_free==UNUSED) {            // if there are no free entries...
        return UNUSED;                // tell the caller
    }
    temp=tq_free;                     // get the first free entry
    tq_free=tq[tq_free].next;         // point the head of the free
                                     // list at the next free entry
    tq[temp].next=UNUSED;              // this entry will be the last
                                     // entry in the used list
}

```

```
    tq[tq_tail].next=temp;           // point the current used tail
                                     // to the new entry

    tq_tail=temp;
    return temp;                     // return the index of the new
                                     // entry
}

/*****
Function:      tq_pop
Description:   Pops the specified entry of the transmit queue.
Parameters:    entry - unsigned char. Must indicate the entry to
               pop.
Returns:       0 if the queue is now empty, 1 if the queue still
               holds data.
Side Effects:  none.
*****/
bit tq_pop(unsigned char entry) {
    unsigned char temp, trail;
    if (tq_head==UNUSED || entry>(QUEUE_SIZE-1)) { // don't bother
                                                    // if the queue is empty or the
                                                    // entry is invalid

        return (tq_head==UNUSED) ? 0 : 1;
    }

    if (entry==tq_head) {                       // special handling for when the
                                                    // head is popped

        temp=tq_head;
        tq_head=tq[tq_head].next;               // move the head pointer
        tq[temp].next=tq_free;                   // put the old head in the free
                                                    // list

        tq_free=temp;
    } else {
        temp=trail=tq_head;                     // set up tracking pointers
        while (temp!=entry && temp!=UNUSED) { // look until the list
                                                    // is exhausted
                                                    // or the entry is found

            trail=temp;
            temp=tq[temp].next;
        }
        if (temp!=UNUSED) {                     // if the list was not
                                                    // exhausted...

            tq[trail].next=tq[temp].next; // link around the entry
            tq[temp].next=tq_free;         // put the entry in the free list
            tq_free=temp;
        }
    }
}
```

```
        if (temp==tq_tail) {
            tq_tail=trail;
        }
    }
}
return (tq_head==UNUSED) ? 0 : 1;
}
```

Keeping the Slots Synchronized

As you can see, the code for managing the network is relatively simple. The slotted approach to dividing access to the network assures each node if getting a fair and equal shot at transmitting data on the network. As was mentioned before, however, this simple design does not allow for synchronizing the nodes when they are not all reset at exactly the same time. The easiest way to ensure that the nodes are synchronized between each other is to give them some sort of signal that they can all synchronize from.

Thus, a new approach is designed for the network. In this case, the PC will generate a low going pulse whenever the beginning of the slot cycle is reached. Thus, the falling edge of this signal will indicate to all nodes that slot zero has just begun. The code in each node now must include another ISR to handle this incoming signal. In this case, assume that it is connected to the INT0 pin of the 8051. The responsibility of this ISR is simply to reload and start timer 0 which will no longer be started by the initialization code in 'main'. Timer 0 will then handle all the rest of the timing that is required and stop itself when all the slots have gone by. The slot sequence will begin again when the PC generates another falling edge on the synchronization signal.

Listing 0-13

```
/******
Function:      start_tdma
Description:   This ISR responds to the signal from the network
               master to being counting slots.  Timer 0 is started
               to perform this work.
Parameters:    None.
Returns:       Nothing.
*****/
void start_tdma(void) interrupt 0 {
    TH0=RELHI_50MS;           // set timer 0 to the right value
    TL0=RELO_50MS;
    TF0=0;
    TR0=1;
    curslot=0xFF;             // ensure that the first timer 0
                               // interrupt causes slot 0 to
                               // start
}
```

The beauty of this approach is twofold. First, as was mentioned above, it is easy for all the nodes to stay synchronized to each other since they are all referencing the same signal. Secondly, it gives the PC the capability to be the master of the communications network. If you alter the code in timer 0 to stop counting slots as soon as the node's slot is up, that node will never transmit again until the beginning of slot zero is indicated again by the PC. This allows the PC to stop access to the network by all nodes and perform any sort of overhead function that it has to. For example, if it must send a long series of

commands to the other nodes but does not want to wait for several slot periods to do it, the PC can stop the network and send out all the data that it has to in one big burst, then restart the normal network functions.

Another minor change that can be made to the network protocol is to allow the PC the capability to send messages to the node which are responded to immediately. In other words, the PC sends a command to the node which instead of building a response and queuing it for its next slot builds the response and begins immediate transmission of it. This in effect makes the network a hybrid of a TDMA system and a polling system. If the master of the network is intelligently coded, this type of scheme will maximize information transfer from the slave nodes to the PC.

CSMA With the 8051

The above TDMA network design allows for great flexibility and efficiency in data transfer under the condition that most nodes on the network will need to use most of their slot a large amount of the time. When this condition is met, the TDMA network achieves near full utilization and is a good and simple solution for network communication.

However, many systems do not meet the above condition. Consider the case where a given node has very little to say except for brief periods. In this case, the slot assigned to that node will remain unused for most of the time while there may be other nodes on the network that need to send more data than is allowable for every one of their slots. Obviously in such a case the TDMA approach is not appropriate.

One answer to this problem is to allow a network node to speak on the communication medium any time it needs to. The trouble with this solution is that there will inevitably be times when two devices decide that they want to talk at the same time. The result will be that neither device will be able to transmit its data reliably since the data from the other node will be colliding with it on the network. To solve this problem, the nodes need some way of detecting when the network is in use as well as detecting collisions with their own transmissions. A network which handles the sort of scheme addressed above is called a Carrier Sense - Multiple Access (CSMA) network. The focus of this section is a set of low level routines that will allow the 8051 to participate in a CSMA network without the collision problems.

To interface the 8051 to a CSMA network, the network node hardware must be such that it allows the on board UART of the 8051 to receive data being transmitted by all nodes including itself. Secondly, the processor type will be changed from an 8051 to an 8052 to give the system another on board timer to play with. The approach to this new type of network access will be simple. First, the system tick function now performed by timer 0 in a TDMA node will be moved, unchanged, to timer 2 in the new node design. Secondly, timer 0 will be used to keep track of a byte time-out period from receipt of the last serial byte. Each time RI fires an interrupt, timer 0 will be loaded with this time-out value and started. At the same time, a flag which indicates that the network is busy will be set. Once the timer overflows and timer 0's interrupt routine is executed, the timer will be stopped and the network busy flag will be cleared. The code which starts transmission of a serial message will have to check this flag to see if the network is in use before it starts sending data out of the serial port.

The timer 0 routine again will serve as the heart of the design for the implementation of the CSMA network. This timer is reloaded every time a byte is received from the network. Additionally, this timer will be used to execute the random back off delay required when a collision between two (or more) nodes on the network is detected. To allow it to perform both functions, a flag will be set to true whenever the random hold off value has been loaded into timer 0. When timer 0 overflows under this condition it will be responsible for restarting transmission of the message in the transmit buffer (if the network is free) and will have to set the message time-out to the correct value since the message has been restarted. The code for the timer 0 interrupt is shown in Listing 0-14.

Listing 0-14

```

/*****
Function:      network_timer
Description:   This interrupt is fired by timer 0 when either a
               network hold off period has expired or the
               interbyte timeout has expired.

Parameters:   none.
Returns:      nothing.
Side Effects: none.
*****/

void network_timer(void) interrupt 1 {
    TR0=0;                      // stop the timer
    if (delay_wait) {           // if we're waiting because of a
                                // network collision...
        delay_wait=0;          // clear that flag
        trans_restart();        // restart the transmission
    }
    network_busy=0;             // the network is no longer busy
    check_status();             // see if a message should be
                                // started
}

/*****
Function:      trans_restart
Description:   This function begins transmission of the message
               held in trans_buf. It assumes that the message is
               good and that the retries and size variables are
               already set.

Parameters:   none.
Returns:      nothing.
Side Effects: none.
*****/

void trans_restart(void) {
    SBUF=trans_buf[0];          // write out the first byte
    last_out=trans_buf[0];      // save it for collision
                                // detection
    trans_ctr=0;                // set buffer pointer to first byte
    trans_chksum=trans_buf[0];  // set the checksum
}

```

Each byte that is written to SBUF will be stored in a temporary location and compared with the value of SBUF when a serial receive interrupt is activated. If the value of SBUF and the temporary do not agree, then the software assumes that there has been some sort of collision during its message transfer. The node will then stop transmitting data for a random period of time and once this time period is over will

again attempt to send the pending message. The code for receive and transmit interrupts on the serial port and the hold off code is shown in Listing 0-15.

Listing 0-15

```

/*****
Function:      ser_xmit
Description:   Handles a serial transmit interrupt.
Parameters:   none.
Returns:      nothing.
Side Effects: none.
*****/

void ser_xmit(void) {
    trans_ctr++;                // move the output index up
                                // see if all data is output

    if (trans_ctr < trans_size) {
                                // send checksum as last byte

        if (trans_ctr==trans_size-1) {
            SBUF=trans_chksum;
            last_out=trans_chksum;
        } else {
                                // send current byte

            SBUF=trans_buf[trans_ctr];
            last_out=trans_buf[trans_ctr];
                                // update checksum

            trans_chksum+=trans_buf[trans_ctr];
        }
    }
}

/*****
Function:      ser_rcv
Description:   Handles a serial receive interrupt when the system
              run at 9600 baud.
Parameters:   none.
Returns:      nothing.
Side Effects: none.
*****/

void ser_rcv(void) {
    unsigned char c, temp;
    c=SBUF;

    if (TH0 > NET_DELAY_HI) {    // don't allow the byte timeout
                                // to shorten any holdoff time

        TR0=0;                  // set the timeout delay in timer 0
    }
}

```

```
    TH0=NET_DELAY_HI;
    TL0=NET_DELAY_LO;
    TR0=1;
}

if (transmitting) {
    // if a message is being sent by
    // this node...

    if (c!=last_out) {
        // the current byte in should be
        // the same as the last byte
        // written to SBUF

        trans_hold();
        // it's not - there has been a
        // network error

    }
} else {
    switch (recv_state) {
        ...
        // parse incoming message
    }
}
}

/*****
Function:      trans_hold
Description:   This function sets a random hold off time in the
               range of 2.00 ms to 11.76 ms.
Parameters:    none.
Returns:       nothing.
Side Effects:  none.
*****/

void trans_hold(void) {
    unsigned int holdoff;
    trans_chksum=trans_ctr=0;
    holdoff=(unsigned int) rand();
    holdoff/=3;
    holdoff+=TWO_MS;
    holdoff=(0xFFFF-holdoff)+1;
    TR0=0;
    TL0=(unsigned char) (holdoff & 0x00FF);
    TH0=(unsigned char) (holdoff / 256);
    delay_wait=1;
}
```

```
                                // collision
    TR0=1;
}
```

You will note that the code to handle transmit interrupts is essentially the same as it has been all along. The biggest difference here is that the 'last_out' variable must be set with the value that was last written to SBUF. Remember, that this value is expected to come into the serial port as the next complete byte. If it does not, there has been some sort of network error.

The remainder of the code for the CSMA version of our network node looks very much like the code in the System Monitor. It is assumed that each time a message is placed on the network, it will either be a command or request to another node, in which case a response is expected, or it will be a reply to some other node, in which case no response will be expected. Thus, the same queue structure and serial command executive code can be reused from the System Monitor with slight modifications. The changes to the code are very simple and are dependent upon your own implementation.

Conclusion

This chapter has presented several networking methods for use with the 8051 family of microcontrollers. These, however, are not the only types of networks that can be implemented with the 8051. This text is not intended to be a lesson on network theories for you, and if you need more information on network design and analysis, I recommend that you seek out other texts to help you with this.

- Controlling the Compiler and Linker

Porting Your Existing C Code to Keil C

Whether you have existing code that you are porting from another processor to the 8051, or 8051 based code that you are changing over for use with the Keil development tools, you have made a wise choice to use Keil tools on the 8051. As has been discussed throughout this book, the 8051 family of microcontrollers and the Keil package of tools are a powerful combination that will help you accomplish almost any task. Porting your existing C code to Keil C is an easy task since the C51 compiler fully implements the ANSI standard for the C language. So as long as you have not written code that uses language extensions that are not ANSI compliant, you will have no trouble compiling your code with the C51 compiler.

When you do begin your code changeover, there are several things you will want to keep in mind at each stage of the process. When porting code to the 8051, begin by examining the code and determining what sort of changes to variable declarations and code structure should be made to help it run better on the 8051. Following this, examine the design of the software and ensure that it will correctly translate into the 8051's architecture.

If your project was previously hosted on another microcontroller, you should pay close attention to the tips that were given for optimizing your code in Chapter Three. These tips all apply here. The main thing to keep in mind is that you are running your code on an eight bit machine and you should thus try to keep all variables and other data elements within the eight bit boundary. The gains to be had by accomplishing this cannot be stressed enough. Any variables that are simply performing the function of a flag (i.e.: they only have two values) should be declared as bit variables. Along this same vein, if you have a variable in which you frequently access a single bit at a time, consider declaring it as bdata.

Another thing to keep a close eye on when first changing porting your code to the 8051 is the usage of pointers. This again was discussed in Chapter Three, but it bears repeating. You can save a fair amount of code size and execution time if you can limit your pointers to a certain memory space and specify this to the C51 compiler using the C language extensions it provides. This will allow the compiler to write much better code for your source that uses these pointers.

Once you have made an initial pass at optimizing your code for the eight bit world, you need to examine the structure of the software and decide which parts of the code will be your interrupt routines and which will be called from 'main'. Once you establish the functions that will respond to processor interrupts you should run the code through the compiler and linker and let the linker generate warnings to you for functions that are called from multiple interrupt calling trees. These warnings will immediately lead you to those functions which are potential critical code sections. These are sections which may be called more than once at a time either via recursion or the interrupt mechanism. The C51 compiler does not automatically generate code to handle recursion or multiple calls to functions from separate calling trees because of the 8051's architecture. If you were dealing with a processor that had full stack capabilities like an 80x86 processor, then a calling frame could be built and pushed onto the stack for each function invocation. However, you are not dealing with an 80x86, and you should know by now that the internal stack space of the 8051 is not sufficiently large to support calling frames for more than a few functions. For functions that must be called recursively, the C51 compiler gives you the capability of declaring them as 'reentrant'. In such cases, the C51 compiler will simulate a stack in the default memory area. This is a time and memory consuming process and thus the 'reentrant' key word should be used sparingly.

Not all linker warnings are cause for alarm. There will be times when the linker warns you that a function is being called from multiple interrupt paths when, in fact, the call is not physically possible. For example, you may have a function which is called by both the ISRs for timer 0 and external interrupt 1, but these interrupts have been set to the same priority. Given that they are the same priority and are the only calling trees that can invoke this function, it is very safe to assume that there is not a problem because an interrupt of a given priority cannot interrupt an ISR of like priority. One way to get rid of the linker warning is to delete the reference from one of the calling trees. This frees you from having the compiler generate that unwanted reentrant stack. This issue will be discussed in more detail later in this chapter.

Once you have adapted your code to an eight bit machine and established your interrupt functions and reentrant functions, you will want to consider the manner in which you access external memory. Many C programmers set a pointer at whatever physical address they wish to access and then perform all operations on that location using the pointer. This method will still work in C51, but you can make your code look a little neater by using the CBYTE, CWORD, XBYTE, XWORD, DBYTE, DWORD, PBYTE, PWORD macros provided in `absacc.h`. These macros allow you to treat external memory as one big array (of char, int, and long) thus making your code more readable. Additionally, if your hardware architecture changes to something odd, and accessing a device is no longer a simple matter of a MOVX, you can rewrite the macro to implement your hardware's new memory accessing scheme.

If you are porting code from another compiler package such as Archimedes or Avocet, you must keep in mind the above discussion and take care to change over their keywords to the correct Keil ones. Since the other packages do not support all the features that the Keil compiler supports (such as bdata variables, reentrant functions, and specification of function register bank) you will want to examine your code to ensure that you take advantage of all the things that Keil supports. On one project that was ported over from Archimedes C to Keil C51, we avoided all of the quirky bugs that Archimedes had, gained extra space in our CODE and XDATA segments, and had to change certain parts of the code to slow it down because the extra speed gained had shown that the host hardware was a little slower than originally thought! The moral of the story is that if you use the C51 package well, there are great gains in efficiency to be made.

Porting Your Assembly Code to Keil Assembly

There really are not too many issues for you to be concerned with when changing an assembly project from one assembler to the Keil assembler. The main thing to remind you of is to change the names of your segments to be compatible with the Keil naming conventions. This will make things easier for you in the future and will allow you to interface your code with Keil C easier. Of course, if your project is a combination of C and assembly then you will want to make the changes to the segment names. Refer to Chapter Three for some guidance on interfacing your C code and your assembly code.

I have never encountered many problems in reassembling code with the Keil assembler. In fact, the only thing I have ever HAD to change in code that I was reassembling was to delete a line which defined the PCON register when changing programs from the old Avocet assembler. The reason was that the Avocet assembler was so old, it was done before the power saving modes were added to the 8051 family and thus it had to have an explicit definition of the address of the PCON SFR.

Use of the ‘using’ Keyword

You will recall that the Intel 8051 family of controllers has four register banks with each bank having eight registers. These thirty two bytes reside in the bottom of the DATA memory area. Each register bank is referred to by its number (zero through three). By default, the 8051 sets itself up to use register bank zero by clearing the RS0 and RS1 bits in the PSW SFR, however, the software is able to change the default register bank of the controller at any time to any one of the four register banks. Part of the discussion in Chapter Three revolved around the register banks and their usage in interrupt functions. That chapter showed you the assembly code emitted by the compiler for an interrupt function compiled without any extra keywords and compared it to an interrupt function which specified register bank zero as the default bank. The difference was that the registers in the second version of the function were not pushed. This section will discuss how to put this fact to use.

Chapter Three pointed out that there are thirty two processor cycles to be saved every interrupt invocation by assigning a specific register bank to an interrupt routine. To take advantage of this, the people at Keil recommend that you assign a separate register bank to each “interrupt level” in your program. For example, the main loop and the initialization code would have no specific assignment and thus would be compiled to use register bank zero. Each interrupt routine which responded to a priority zero interrupt would be coded to use register bank one, and each interrupt routine which responded to a priority one interrupt would be coded to use register bank two. Any functions that are called by the ISRs must either use the same register bank as the caller or must be compiled using a compiler directive (‘NOAREGS’) which ensures immunity from the current register bank. The following piece of code illustrates this basic design approach for selecting register banks for the ISRs.

Listing 0-1

```
void main(void) {
    IP=0x11;                // serial intr and ext intr 0
                            // have high priority

    IE=0x97;                // enable serial, external 1,
                            // timer 0 and external 0
                            // interrupts

    init_system();
    ...

    for (;;) {
        PCON=0x81;          // enter idle mode
    }
}

void serial_intr(void) interrupt 4 using 2 {
                            // serial interrupt has high
                            // priority and thus will use
                            // register bank 2

    if (_testbit_(RI)) {
        recv_fsa();
    }
}
```

```
    if (_testbit_(TI)) {
        xmit_fsa();
    }
}

void recv_fsa(void) using 2 {           // recv_fsa must use the same
                                        // register bank as serial_intr
                                        // since serial_intr calls it
    ...
}

void xmit_fsa(void) using 2 {           // xmit_fsa must use the same
                                        // register bank as serial_intr
                                        // since serial_intr calls it
    ...
}

void intr_0(void) interrupt 0 using 2 {
                                        // high priority interrupt - use
                                        // register bank 2

    handle_io();
    ...
}

void handle_io(void) using 2 {          // called by an ISR using RB2
                                        // must use RB2 also
    ...
}

void timer_0(void) interrupt 1 using 1 {
                                        // low priority interrupt - use
                                        // register bank 1
    ...
}

void intr_1(void) interrupt 2 using 1 {
                                        // low priority interrupt - use
                                        // register bank 1
    ...
}
```

Note that the register bank is specified for each ISR as well as the functions called by the ISR. Any functions called by the main routine will not need a register bank definition since C51 will automatically

assume that they are to use register bank zero. The calling tree for this simple project is shown below. The separate paths do not cross.

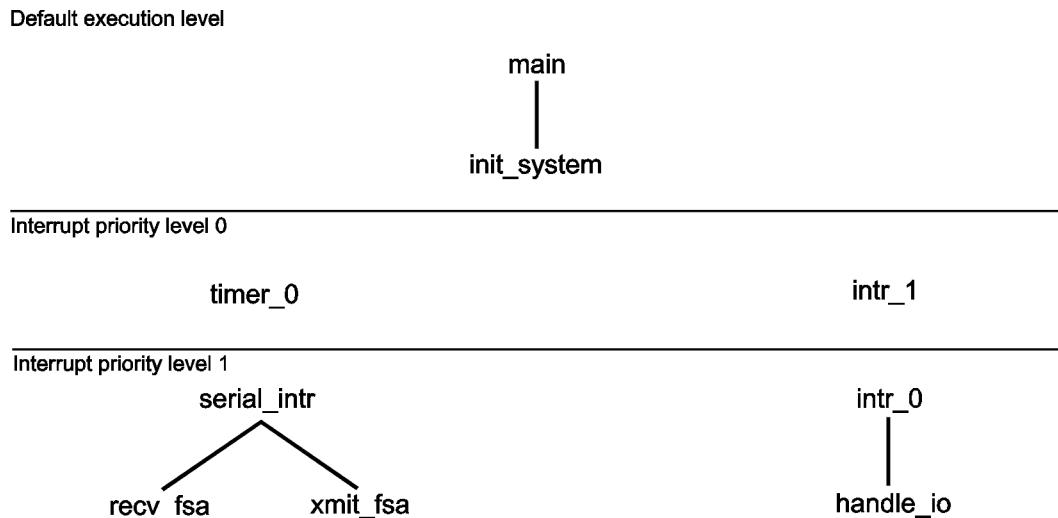


Figure 0-1 - Simple Calling Tree

Most real time systems do not have a calling tree as simple as the one shown above. Usually there are at least a couple of utility type functions that are called by more than one interrupt routine in addition to the main loop. Take, for example, the following code listing. This program is the same as the one above, except now it has a new function called display that is executed by both of the possible interrupt levels and the main loop level.

Listing 0-2

```
void main(void) {
    IP=0x11;                // serial intr and ext intr 0
                           // have high priority
    IE=0x97;                // enable serial, external 1,
                           // timer 0 and external 0
                           // interrupts

    init_system();
    ...

    display();              // write a message to a display
                           // panel

    for (;;) {
        PCON=0x81;         // enter idle mode
    }
}

void serial_intr(void) interrupt 4 using 2 {
```

```

// serial interrupt has high
// priority and thus will use
// register bank 2

if (_testbit_(RI)) {
    recv_fsa();
}
if (_testbit_(TI)) {
    xmit_fsa();
}
}

void recv_fsa(void) using 2 {    // recv_fsa must use the same
                                // register bank as serial_intr
                                // since serial_intr calls it
    ...

    display();                  // write FSA status to the
                                // display panel
}

void xmit_fsa(void) using 2 {    // xmit_fsa must use the same
                                // register bank as serial_intr
                                // since serial_intr calls it
    ...
}

void intr_0(void) interrupt 0 using 2 {
                                // high priority interrupt - use
                                // register bank 2

    handle_io();
    ...
}

void handle_io(void) using 2 {   // called by an ISR using RB2
                                // must use RB2 also
    ...
}

void timer_0(void) interrupt 1 using 1 {
                                // low priority interrupt - use
                                // register bank 1
    ...
}
```

```
display();                                // write a timeout message to
                                         // the display panel
}void intr_1(void) interrupt 2 using 1 {
                                         // low priority interrupt - use
                                         // register bank 1

    ...
}

void display(void) {
    ...
}
```

Now there is a function, 'display', which is called by every execution level of the 8051. This means that it is possible for the 'display' function to be interrupted by an interrupt which will in turn call the 'display' function. Remember that each ISR has specified its own register bank and thus is not saving any of the data in the current register bank. By default, the compiler will code the 'display' function to use absolute register addressing for register bank zero. This means that instead of generating R0..R7 type references to the registers, C51 will instead generate absolute addresses. In the case of the 'display' function as specified, the compiler will generate 00..07 for the register addresses since it assumes register bank zero to be the active register bank.

This will cause a problem for any code that was using register bank 0 when the display function is called from an interrupt routine because the data in the registers will be corrupted. If the display function were only called by one interrupt level, the default register bank could be specified for the function and the problem would be solved. However, this will not work in this case since the function is called by more than one interrupt level as you can see in the calling diagram shown below.

The calls to 'display' from multiple levels in the code will also cause the linker to generate several warnings which we will deal with later. For right now, the important thing is to tell the compiler how to deal with 'display' so that register corruption does not occur when the code executes. The solution is to make 'display' compile so that it uses the current register bank when it performs register access rather than assuming that register bank zero is active. This is done by using the Keil pragma 'NOAREGS' around the display function. The code for display will be emitted to use R0..R7 type references for the registers rather than absolute addresses. The 'display' function will remain unchanged but will have the 'NOAREGS' pragma before it to make it insensitive to register bank changes and the 'AREGS' pragma after it to allow any remaining functions in the file to compile using the C51 defaults.

```
#pragma NOAREGS
void display(void) {
    ...
}
#pragma AREGS
```

Default execution level

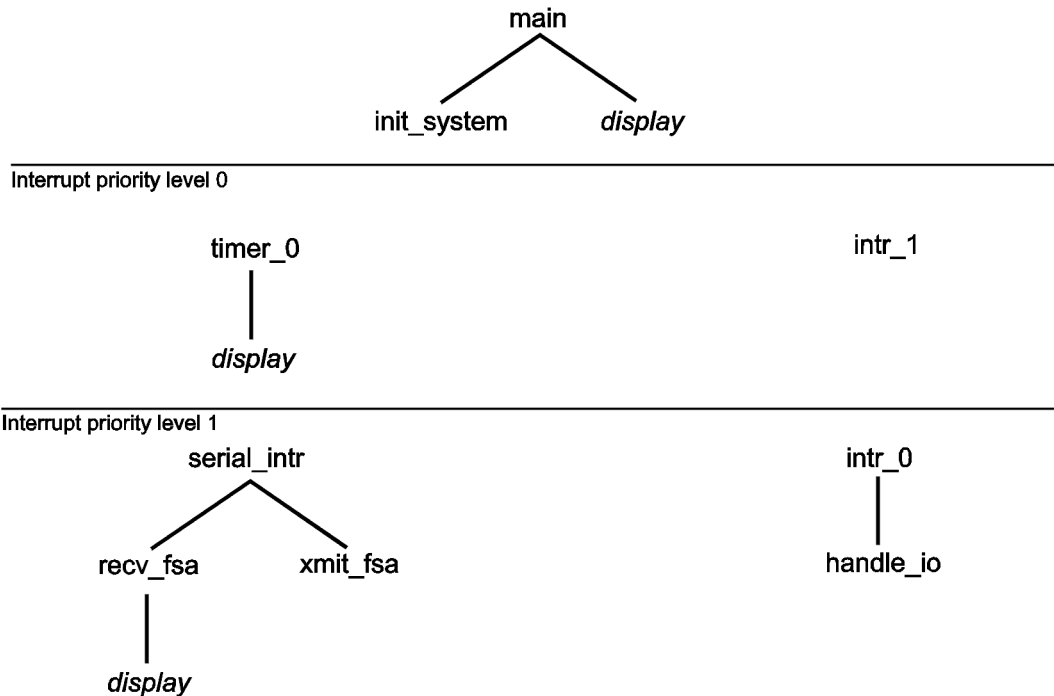


Figure 0-2 - Calling Tree with Calls from Multiple Interrupt Levels

Another problem with the 'display' function still exists. Assume that 'display' uses several local variables to perform its work. By default, the C51 compiler allocates a block of memory for storage of the local variables in a "compiled stack." This storage may be in the default memory segment or may be optimized into a register depending upon the results of the compiler's optimizations. Nevertheless, each invocation of the display function will use exactly the same memory locations for the local variables no matter where in the calling tree it is invoked from. This is because there is no function stack in the 8051 in the sense of a stack in an 80x86 or 680x0 type processor as was discussed before. Normally, this is not a problem, but when a function must be called recursively or in a reentrant manner then there will inevitably be corruption of the data in the local variables.

Consider the case where the 'timer_0' ISR calls 'display'. During the execution of 'display', a serial interrupt occurs and because it is an incoming byte, 'rcv_fsa' is eventually called by the serial ISR. 'rcv_fsa' is in a state which requires it to call 'display'. The execution of 'display' completes and the values of all of the local variables are changed. Any locals that were optimized into registers will be okay because of the register bank switching, but any locals which could not be so optimized have been corrupted by 'rcv_fsa's call to 'display'. Eventually, the serial ISR is completed and control of the 8051 is returned to the 'timer_0' ISR which was in the middle of a call to 'display'. Any local variables that were allocated in the default memory segment for the 'display' function have now been corrupted and we can not predict the results of the 'display' function.

To correct this problem, Keil C51 allows you to optionally generate code which will simulate a stack for functions such as 'display' by declaring them as reentrant. Once this keyword is applied to the 'display' function, it will be compiled to allocate all of its locals off of a function invocation stack which is kept in the default memory segment. Thus, each invocation of 'display' will have its own set of addresses for its local variables. After correcting the reentrancy problem, the 'display' function will now look like this.

```
#pragma NOAREGS
void display(void) reentrant {
    ...
}
#pragma AREGS
```

You should be aware that declaring functions to be reentrant adds a considerable amount of overhead to them, and thus should be used sparingly. Additionally, the amount of space taken by the reentrancy stack can only be predicted by computing the worst case number of reentrant function invocations that can be active at a time. The C51 package leaves it up to you to ensure that you have enough memory in the default memory space to handle the reentrant stack. The stack is designed to start at the top of the default memory segment (for example, 0FFFFH in the XDATA segment) and grow downwards towards your variables (which begin allocation at the bottom of the memory segment). Once you have compiled and linked your application you should carefully examine the '.M51' file that is generated by the linker to assure that there is sufficient space for the reentrant stack.

Controlling The Linker's Overlay Process

You will have many applications in which you have functions that are called from more than one calling tree, but because of C51's lack of a real stack, are not possible. This type of situation was discussed above. Consider the case in which the 'display' function is used as shown in Listing 0-3.

Listing 0-3

```
void main(void) {
    IP=0x00;                      // all ints have the same priority

    init_system();

    ...

    display(0);

    IE=0x8A;                      // enable timer 0, and external
                                // 0 interrupts

    for (;;) {
        PCON=0x81;               // enter idle mode
    }
}

void timer_0(void) interrupt 1 using 1 {
                                // low priority interrupt - use
                                // register bank 1
```

```
...
display(1);
}

void intr_1(void) interrupt 2 using 1 {
    // low priority interrupt - use
    // register bank 1
    ...

    display(2);
}

void display(unsigned char x) {
    ...
}
```

According to the linker, this code contains a conflict in its calling trees because the 'display' function can be called from the main calling tree as well as the timer 0 ISR and the external interrupt 1 ISR. Because of the conflict, the linker will emit the following warnings.

```
*** WARNING 15: MULTIPLE CALL TO SEGMENT
    SEGMENT: ?PR?_DISPLAY?INTEXAM
    CALLER1: ?PR?TIMER_0?INTEXAM
    CALLER2: ?PR?INTR_1?INTEXAM

*** WARNING 15: MULTIPLE CALL TO SEGMENT
    SEGMENT: ?PR?_DISPLAY?INTEXAM
    CALLER1: ?PR?INTR_1?INTEXAM
    CALLER2: ?C_C51STARTUP
```

The linker is warning you that a call to 'display' may be interrupted by an ISR which can also call 'display', thus causing corruption of 'display's local variables. The first warning tells us that a call to 'display' exists in both the timer 0 calling tree and the interrupt 1 calling tree; the second warning tells us that there is also a conflict between the interrupt 1 calling tree and the main calling tree. Note also that the same conflict exists between the calling tree for main and for timer 0. Careful examination of the code structure shows that the "reentrant" call to 'display' is not a problem.

The timer 0 and the external interrupt 1 interrupts have been assigned the same priority in the software and thus could not cause any sort of conflicting calls to 'display'. Thus, there is no software design concern arising from the first warning. The second warning is telling us that the 'main' routine's call to 'display' may get interrupted by interrupt one's call to 'display'. Again, this is not a problem since the interrupts aren't even enabled when the main routine calls 'display', and thus the call to 'display' could not get interrupted by anything. Both warnings have proven to be unfounded. This is not to say that the linker did not function properly. In fact the linker has done its job in giving these warnings. Its job (in addition to normal linker functions) is to look for these possible reentrant situations and warn you about them when they have not been handled by declaring the function to be reentrant. It is not the linker's job

to perform any sort of analysis on the code and determine which interrupts can occur and when. This is the job of the engineer.

Now that it has been determined that the warnings are not meaningful to the code, what should be done about them? The first inclination is to simply ignore them, but this will affect the way in which the linker resolves references between modules and assigns addresses to relocatable objects. Thus, while the linker may still output a fully functional executable, it will not take fullest advantage of the memory spaces it has to work with because the linker will not be able to correctly perform overlay analysis. In fact, there will be situations in which the program emitted by the linker will not function properly. The moral is that the linker warnings should not be ignored.

In this case, there are two ways in which the warnings can be eliminated. The first way is to tell the linker not to perform any sort of overlay analysis. This will lead to code that uses an unnecessarily large amount of DATA space, but is the easiest to implement. The second way is to help the linker's overlay analysis by forcing it to ignore certain references from the calling trees to the 'display' function. Once you have told it to delete all but one tree's reference, the code will link without any warnings and overlay analysis will be properly performed. The second approach is clearly the more desirable, while you may elect to take the first approach if you are short of time and long on memory.

The above code was linked by invoking L51 with the following command line.

```
L51 example.obj
```

The first method of eliminating the multiple call warnings stated that you should simply tell the linker not to perform overlay analysis. Implementing this requires that you clear the "Enable variable overlaying" checkbox in the linker options dialog box in the workbench.

The second method is somewhat more complex but the rewards for its implementation are higher. Method number two requires that you delete calling references from two of the three functions which call 'display'. This will be done using the overlay option on the command line. Remember that the 'display' function is called by 'main', 'timer_0', and 'intr_1', and you must choose two of these functions to delete the call to 'display' from. In such a situation, try to leave the call to the function in question in the calling tree which will be calling the function the most. In this case, let's assume that interrupt 1 occurs rarely and that timer 0 is the system tick and thus occurs more frequently. Given this situation, you would delete the call to 'display' from 'main' and from 'intr_1', leaving it in the calling tree of 'timer_0'. The new L51 command line looks like this. The "overlay" section of the command line should be entered in the "Additional" tab of the linker configuration dialog box in the Workbench.

```
L51 example.obj overlay(main ~ _display, intr_1 ~ _display)
```

Most of your projects will initially have a fairly good sized list of multiple call warnings when you first run them through the linker. Proper attention to the above discussion of deleting references from the calling trees and correctly using the reentrant keyword will eliminate the warnings. Remember that you will probably not eliminate all linker warnings in a single pass - it is an iterative process and thus will take several steps, but make sure that you do eliminate all linker warnings.

Using 64K (and more) of RAM

If you are developing complex systems with the 8051, you may eventually run into a situation in which you have device that needs 64K bytes of RAM, but also must perform memory mapped I/O functions. In applications such as this you are forced to overlap the RAM addresses with the addresses of the I/O devices. The RAM is then enabled and disabled using either one of the port 1 pins or a pin from a data latch. An example of such a design is shown in Figure 0-3.

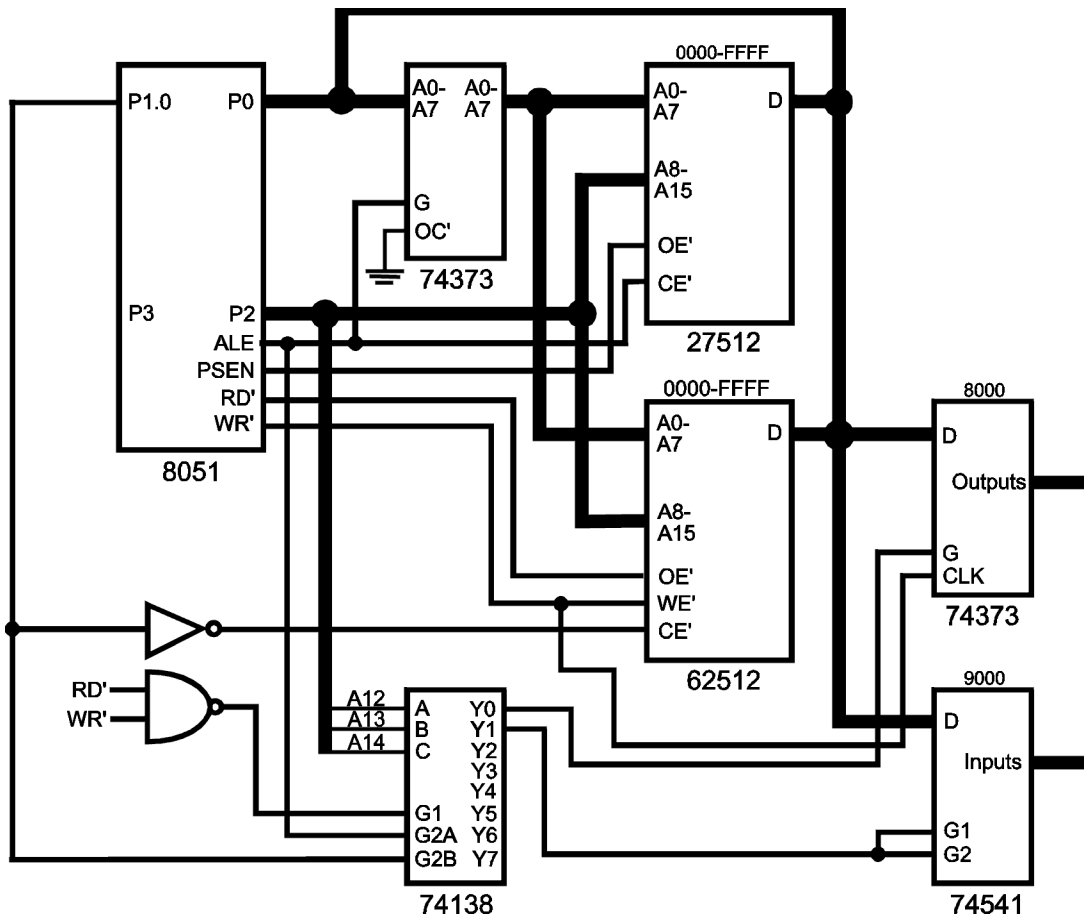


Figure 0-3 - Overlapping RAM and I/O

The software accesses the RAM by holding the P1.0 pin high and then performing the desired access. The logic level one was chosen to enable the RAM so that it would automatically be enabled by the 8051 when the processor is reset. When the software wants to access an I/O device via its bus, it simply pulls P1.0 low to disable the RAM and enable the address decoder and then performs the device access.

During normal program operation, the RAM is left enabled so the software does not have to spend time concerning itself with the state of the enable signal. When the software must access an external I/O device via the bus a special routine is called which disables the RAM and performs the necessary access passing data in and out of the internal RAM. Once this process is complete, the RAM is re enabled and software execution continues as normal. The necessary routines to perform the system I/O are shown in Listing 0-4. Note that the parameters to this function will be passed via internal RAM and the return values will also be passed via internal RAM. Thus, there is no need for the SRAM to be enabled during the actual access of the peripheral.

Listing 0-4

```
#include <reg51.h>
#include <absacc.h>

sbit SRAM_ON = P1^0;

/*****
Function:      output
Description:   Writes a specified value to a specified XDATA
               address.
Parameters:   address - unsigned int.  Indicates address to write
               data to.
               value - unsigned char.  Holds the data to write
               out.
Returns:      nothing.
Side Effects: The external RAM is disabled and thus an interrupt
               routine should not be invoked during this time.  To
               help compensate for this, the interrupt system is
               temporarily halted.
*****/
void output(unsigned int address, unsigned char value) {
    EA=0;                // block all interrupts
    SRAM_ON=0;           // disable the RAM
    XBYTE[address]=value; // write value to the address
    SRAM_ON=1;           // enable the RAM
    EA=1;                // allow interrupts again
}

/*****
Function:      input
Description:   Reads the value at the specified XDATA address.
Parameters:   address - unsigned int.  Indicates address to read
               from.
Returns:      The value read at the specified address.
Side Effects: The external RAM is disabled and thus an interrupt
               routine should not be invoked during this time.  To
               help compensate for this, the interrupt system is
               temporarily halted.
*****/
```

```
unsigned char input(unsigned int address) {  
    unsigned char data value;  
    EA=0;                                // block all interrupts  
    SRAM_ON=0;                            // disable the RAM  
    value=XBYTE[address];                 // read in the data  
    SRAM_ON=1;                            // enable the RAM  
    EA=1;                                // allow interrupts again  
    return value;  
}
```

The concept of enabling and disabling the RAM can be extended to allow you to access more than 64K of system RAM. If you're thinking that 64K of RAM is more than enough for an 8051 based system, you would be correct most of the time. However, systems that do a lot of instrumentation or have large amounts of data stored in tables may exceed 64K of RAM usage. In these cases, it is useful to run the extra address lines from the RAM to either port 1 or a 74HC373 and select the active page of the RAM via software. In such applications, page 0 is left enabled most the time, and the other pages are accessed much in the same way as the I/O devices were accessed in the system described above. In this design, RAM page 0 will be used for program variables, parameter passing, compiled stacks, etc. The other RAM pages will hold such things as system event tables, lookup tables, and other data that will not typically have a high amount of accesses performed on it. This approach treats the other pages of the RAM as if they were a mass storage device. A sample circuit for this system is shown in Figure 0-4. The only real change from the previous schematic is that the P1.1 and P1.2 lines are now used to control the upper two address lines of a 256Kb RAM.

The input and output devices connected to the bus of this processor are accessed in the same manner as they were in the previous example. The new thing about this circuit is the way in which the pages of the SRAM are accessed. Let me start out by saying again that the default state of this system is to have the SRAM enabled by pulling P1.0 high and to have RAM page 0 selected by pulling P1.1 and P1.2 low. If you like you can add an inverter to each of the two extra address select lines so that the processor selects RAM page 0 automatically upon power up. Otherwise, if you stay with this design you will have to ensure that you add a line of code to the startup.a51 file to clear P1.1 and P1.2 if you have XDATA variables which are initialized at compile time.

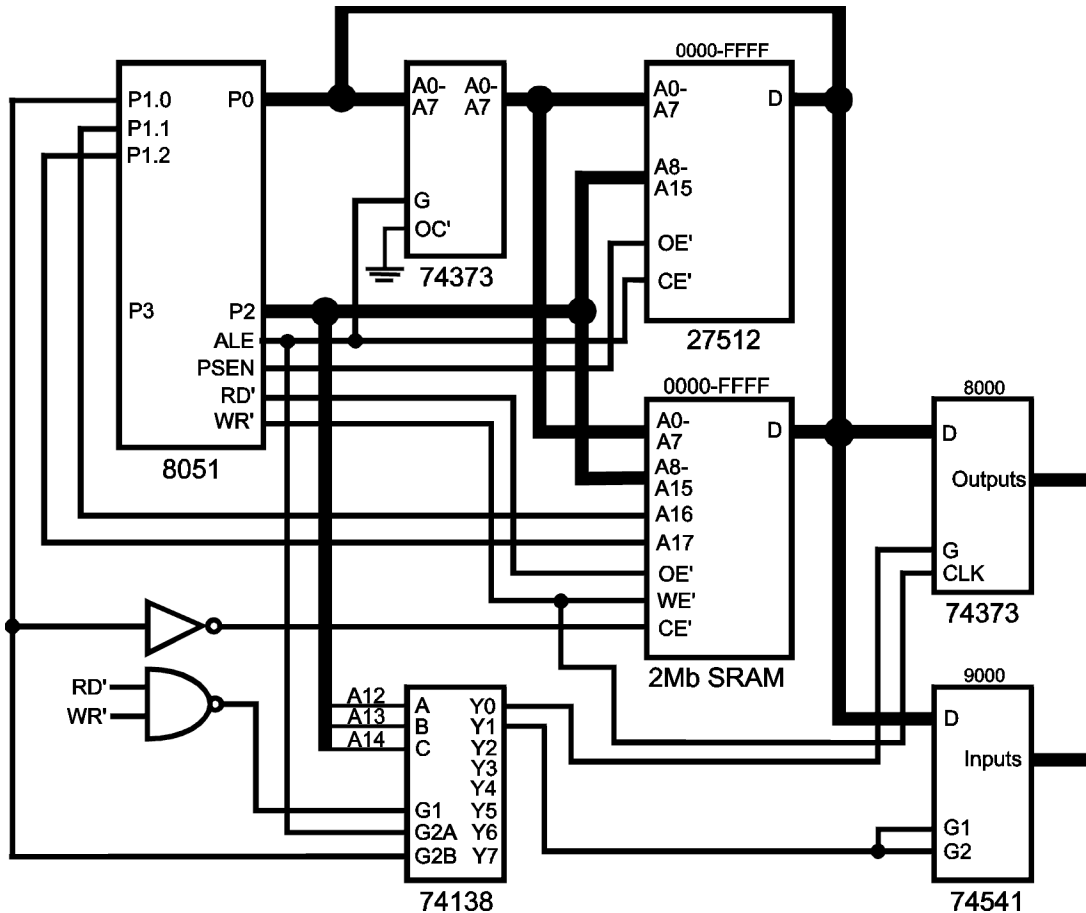


Figure 0-4 - Paging the RAM

Writing data to and reading data from the other pages of RAM is done in block fashion. This means that the RAM page access function will take a RAM page, and the number of bytes to read or write as arguments and do all data transfers using a buffer in internal RAM. This prevents the routine from having to constantly switch from the page being accessed to RAM page 0 to write a byte of data. Thus, the execution time of accesses are smaller, but only at the cost of a chunk of internal data and a limit on the maximum number of bytes that may be transferred at a time. The source code in Listing 0-5 shows the necessary functions and definitions required to implement this RAM paging scheme.

Listing 0-5

```
#include <reg51.h>
#include <absacc.h>

sbit SRAM_ON = P1^0;

unsigned char data xfer_buf[32];
```

```

/*****
Function:      page_out
Description:   Writes a specified value to a specified XDATA
               address.
Parameters:   address - unsigned int.  Indicates address to write
               data to.
               page - unsigned char.  Indicates the RAM page to
               use.
               num - unsigned char.  Indicates the number of bytes
               to write.
Returns:      nothing.
Side Effects: The external RAM is disabled and thus an interrupt
               routine should not be invoked during this time.  To
               help compensate for this, the interrupt system is
               temporarily halted.
*****/
void page_out(unsigned int address, unsigned char page,
               unsigned char num) {
    unsigned char data i;
    unsigned int data mem_ptr;          // declared so that no XDATA
                                       // access will be generated by
                                       // the compiler to move a
                                       // pointer through the
                                       // referenced RAM page

    mem_ptr=address;
    num&=0x1F;                          // limit number of bytes to 32
    page&=0x03;                         // limit page select to 0..3
    page<=1;
    page|=0x01;                         // make sure RAM stays selected
    EA=0;                               // block all interrupts
    P1=page;                            // select new page
    for (i=0; i<num; i++) {
        XBYTE[mem_ptr]=xfer_buf[i];    // write value to the address
        mem_ptr++;
    }
    P1=1;                               // select page 0
    EA=1;                               // allow interrupts again
}

/*****
Function:      page_in
Description:   Reads the specified number of bytes from the
               address on the RAM page specified.
*****/

```

```
Parameters:    address - unsigned int.  Indicates address to read
               from.
               page - unsigned char.  Indicates the RAM page to
               use.
               num - unsigned char.  Indicates the number of bytes
               to read.
Returns:       nothing.
Side Effects:  The external RAM is disabled and thus an interrupt
               routine should not be invoked during this time.  To
               help compensate for this, the interrupt system is
               temporarily halted.
*****/
void page_input(unsigned int address, unsigned char page,
               unsigned char num) {
    unsigned char data i;
    unsigned int data mem_ptr;          // declared so that no XDATA
                                       // access will be generated by
                                       // the compiler to move a
                                       // pointer through the
                                       // referenced RAM page

    mem_ptr=address;
    num&=0x1F;                          // limit number of bytes to 32
    page&=0x03;                          // limit page select to 0..3
    page<=1;
    page|=0x01;                          // make sure RAM stays selected
    EA=0;                                // block all interrupts
    P1=page;                             // select new page
    for (i=0; i<num; i++) {
        xfer_buf[i]=XBYTE[mem_ptr];     // read next address
        mem_ptr++;
    }
    P1=1;                                // select page 0
    EA=1;                                // allow interrupts again
}
```

Note that the way the compiler generates code forced me to use the local variable 'mem_ptr'. The original design of the 'for' loop called for incrementing address each iteration. However, the compiler then chooses to allocate an integer in XDATA to use to hold the ever changing value of address. Declaring a temporary variable (in this case 'mem_ptr') to hold this value eliminates the XDATA accesses in the compiler's code.

The RAM page functions shown above work for most situations. However, some programmers may want to have a set of memory manipulation functions which look much like those provided with the C51 library, such as the 'memcpy' function. For some amount of work, you can develop a set of functions which closely resemble the library functions, but accept the generic memory pointers which now can reference a RAM page. Recall that a generic pointer in Keil C consists of three bytes: a two byte address and a one byte selector which determines the memory space referenced by the pointer. The selector has a range of values from one to five depending upon the memory space being referenced. Thus, the leading nibble of the selector byte is unused and will now be used to indicate the RAM page being referenced by the pointer when the selector indicates that the referenced memory segment is the XDATA area. This simple modification to the function interface will allow the new RAM page library functions to look almost exactly like their standard counter parts. This section will present sample code for the 'memcpy' function, and leave the other functions to you to implement. The C declaration to the 'page_memcpy' function is shown below.

```
void * page_memcpy(void *dest, void *source, int num);
```

Because the function will end up switching between memory pages quite frequently, it will be coded in assembler. The code listing, as coded for the large memory model, is shown below.

Listing 0-6

```
?PR?PAGE_MEMCPY?PAGE_IO      SEGMENT CODE
?XD?PAGE_MEMCPY?PAGE_IO      SEGMENT XDATA  OVERLAYABLE

PUBLIC      _page_memcpy, ?_page_memcpy?BYTE

RSEG      ?XD?PAGE_MEMCPY?PAGE_IO
?_page_memcpy?BYTE:
dest:      DS      3
src:       DS      3
num:       DS      2

;*****
;
; Function:      _page_memcpy
; Description:   Copies the number of bytes specified from the
;               source pointer to the destination pointer. Allows
;               xdata pointers to specify the RAM page using the
;               leading nibble of the selector byte.
; Parameters:    dest - generic pointer with optional RAM page
;               specifier. This parameter is passed in R1..R3. It
;               indicates the starting address to copy data to.
;               src - generic pointer with optional RAM page
;               specifier. Indicates the starting address to copy
;               data from.
;               num - unsigned integer. Indicates the number of
;               bytes to copy.
; Returns:       the dest address
; Side Effects:  none.
```

```

,*****
RSEG      ?PR?PAGE_MEMCPY?PAGE_IO
_page_memcpy: PUSH    07          ; save used regs
              PUSH    06
              PUSH    02
              PUSH    01
              PUSH    00
              PUSH    ACC
              PUSH    B
              MOV     DPTR, #?_page_memcpy?BYTE+6
              MOVX    A, @DPTR      ; read in num
              MOV     06, A
              INC     DPTR
              MOVX    A, @DPTR
              MOV     07, A
              ORL     A, 06

              JZ      GTF0          ; if (!num) { return }

              MOV     DPTR, #?_page_memcpy?BYTE
              MOV     A, 03          ; load dest pointer
              MOVX    @DPTR, A
              INC     DPTR
              MOV     A, 02
              MOVX    @DPTR, A
              INC     DPTR
              MOV     A, 01
              MOVX    @DPTR, A

L1:         LCALL    GETSRC          ; get next source byte
              LCALL    PUTDEST       ; write next byte to dest

              MOV     A, 07          ; num--
              CLR     C
              SUBB    A, #1
              MOV     07, A
              MOV     A, 06
              SUBB    A, #0
              MOV     06, A
              ORL     A, 07
              JZ      GTF0          ; if (!num) { return }
              JMP     L1

GTF0:      POP     B                ; restore all regs

```

```

        POP     ACC
        POP     00
        POP     01
        POP     02
        POP     06
        POP     07
        RET

;*****
;
; Function:      GETSRC
; Description:    Reads the byte referenced by the pointer in src,
;                increments src and returns the data read.
; Parameters:    none.
; Returns:       the data read in A.
; Side Effects:  none.
;*****
GETSRC:    MOV     DPTR, #?_page_memcpy?BYTE+3
           MOVX    A, @DPTR      ; get source selector
           MOV     B, A          ; save it
           DEC     A             ; scale selector to 0..4
           ANL     A, #00FH      ; get rid of RAM page
           MOV     DPTR, #SEL_TABLE1
           RL       A
           JMP     @A+DPTR       ; switch on selector
SEL_TABLE1: AJMP    SEL_IDATA1    ; idata
           AJMP    SEL_XDATA1    ; xdata
           AJMP    SEL_PDATA1    ; pdata
           AJMP    SEL_DATA1     ; data or bdata
           AJMP    SEL_CODE1     ; code

SEL_PDATA1: MOV     00, #00      ; for pdata, leading byte
           ; of address must be 00
           MOV     DPTR, #?_page_memcpy?BYTE+5
           JMP     L2
SEL_XDATA1: MOV     DPTR, #?_page_memcpy?BYTE+4
           MOVX    A, @DPTR      ; read in address
           MOV     00, A
           INC     DPTR
L2:        MOVX    A, @DPTR
           MOV     DPH, 00        ; set DPTR to XDATA address
           MOV     DPL, A
           MOV     A, B           ; get correct RAM page
           ANL     A, #0F0H

```

```
        SWAP    A
        RL      A
        ORL     A, #01H
        MOV     P1, A          ; select RAM page
        MOVX    A, @DPTR      ; read in byte
        MOV     01, A         ; save it
        MOV     P1, #01H      ; restore RAM page
        INC     DPTR          ; advance source address
        MOV     00, DPL
        MOV     A, DPH
                                ; save new source address
        MOV     DPTR, #?_page_memcpy?BYTE+4
        MOVX    @DPTR, A
        INC     DPTR
        MOV     A, 00
        MOVX    @DPTR, A
        MOV     A, 01          ; return the byte in A
        RET

SEL_CODE1:  MOV     DPTR, #?_page_memcpy?BYTE+4
        MOVX    A, @DPTR      ; get current source address
        MOV     00, A
        INC     DPTR
        MOVX    A, @DPTR
        MOV     DPH, 00        ; set DPTR with current
                                ; address
        MOV     DPL, A
        CLR     A
        MOVC    A, @A+DPTR    ; read in byte
        MOV     01, A
        INC     DPTR          ; advance source pointer
        MOV     00, DPL
        MOV     A, DPH
        MOV     DPTR, #?_page_memcpy?BYTE+4
        MOVX    @DPTR, A      ; save source pointer
        INC     DPTR
        MOV     A, 00
        MOVX    @DPTR, A
        MOV     A, 01          ; return the byte in A
        RET

SEL_IDATA1:
SEL_DATA1:  MOV     DPTR, #?_page_memcpy?BYTE+5
        MOVX    A, @DPTR      ; get the one byte address
```

```

MOV    00, A
MOV    A, @R0          ; access data and idata
                        ; indirectly
INC    R0              ; advance source pointer
XCH    A, 00
MOVX   @DPTR, A        ; save source pointer
XCH    A, 00          ; return the byte in A
RET

;*****
;
; Function:      PUTDEST
; Description:   Writes the byte in A to the address referenced
;               by the pointer in dest.  dest is then incremented.
; Parameters:    none.
; Returns:       nothing.
; Side Effects:  none.
;*****
;
PUTDEST:  MOV    02, A          ; save output byte
          MOV    DPTR, #?_page_memcpy?BYTE
          MOVX   A, @DPTR      ; get dest selector
          MOV    B, A          ; save selector
          DEC    A             ; scale selector to 0..4
          ANL    A, #00FH
          MOV    DPTR, #SEL_TABLE2
          RL     A
          JMP    @A+DPTR       ; switch on selector
SEL_TABLE2: AJMP  SEL_IDATA2    ; idata
            AJMP  SEL_XDATA2    ; xdata
            AJMP  SEL_PDATA2    ; pdata
            AJMP  SEL_DATA2     ; data or bdata
            AJMP  SEL_CODE2     ; code

SEL_PDATA2: MOV    00, #00      ; for pdata leading byte of
                        ; address must be 0
            MOV    DPTR, #?_page_memcpy?BYTE+2
            JMP    L4
SEL_XDATA2: MOV    DPTR, #?_page_memcpy?BYTE+1
            MOVX   A, @DPTR     ; read in address
            MOV    00, A
            INC    DPTR
L4:        MOVX   A, @DPTR
            MOV    DPH, 00      ; set DPTR to XDATA address
            MOV    DPL, A
            MOV    A, B        ; get correct RAM page

```

```
        ANL    A, #0F0H
        SWAP   A
        RL     A
        ORL    A, #01H
        MOV    P1, A          ; select RAM page
        MOV    A, 02
        MOVX   @DPTR, A      ; write out byte
        MOV    01, A          ; save it
        MOV    P1, #01H      ; restore RAM page
        INC    DPTR           ; advance dest address
        MOV    00, DPL
        MOV    A, DPH
                                ; save new dest address
        MOV    DPTR, #?_page_memcpy?BYTE+1
        MOVX   @DPTR, A
        INC    DPTR
        MOV    A, 00
        MOVX   @DPTR, A
        RET

SEL_CODE2:    RET            ; can't write to code memory

SEL_IDATA2:
SEL_DATA2:    MOV    DPTR, #?_page_memcpy?BYTE+2
               MOVX   A, @DPTR      ; get the one byte address
               MOV    00, A
               MOV    @R0, 02       ; indirectly access internal
                                   ; RAM
               INC    R0            ; advance the dest pointer
               XCH    A, 00
               MOVX   @DPTR, A      ; save dest pointer
               RET

        END
```

Changing the memory model of the above function is a simple matter of changing the code which accesses the arguments. Additionally, the other memory operator functions can be easily created using this function as a template.

Using More than 64K of CODE Space

In 8051 applications which have survived several product cycles or which are very complex, the amount of software often increases greatly due to added features and enhancements to existing features. As these improvements are made, the size of the executable code continually increases. When the CODE space already consumes 64K this becomes a very big problem. You are faced with a critical decision with several options.

The first option is that you can refuse to implement the enhancement on the grounds that there is no more CODE space for it. This will not go over well with upper management and/or marketing. The second option is to delete other features from the application to fit in this new one. However, marketing rarely wants to take a step backwards in features to add a new one. Thirdly, you can try to optimize your code to squeeze out that extra amount of space to fit in the feature, but you will find this task increasingly difficult to do particularly if you have been generating efficient code. This option is only a short term option if it is plausible at all. Your fourth option is to redesign the system and go to a new controller which can support a larger CODE space. This means new hardware, new software, and new development tools - a really bad idea to present to management. Finally, you can change your hardware slightly and increase the size of the EPROM in the system from 64K to something larger. At this point you're probably wondering how that helps you since the 8051 can only address 64K of CODE. The answer is that Keil delivers the capability to bank switch the EPROM with their development package - this ability gives you up to 1MB of CODE space!

Using the banked linker (BL51) from Keil, the CODE space can be increased by utilizing a method similar to the one discussed above for paging the RAM. In this design, the EPROM is broken into many CODE pages. The size of each page and the method in which the pages are switched is dependent upon your application. However, in all designs, there must be a common area which is available to the microcontroller at all times. This common area contains the interrupt vector, the interrupt functions (which may call code in any EPROM page), the C51 library functions, the code to switch EPROM pages, and constants which are used by more than one CODE page. The common area can be implemented by mapping a small EPROM to address 0 and having larger EPROMs mapped above it, or by duplicating the common area in the bottom of all pages of the CODE space. Each approach is equally viable, however, I prefer to copy the common area into all pages of the CODE space since this allows for maximum flexibility in the common area's size and the usage of the rest of the page.

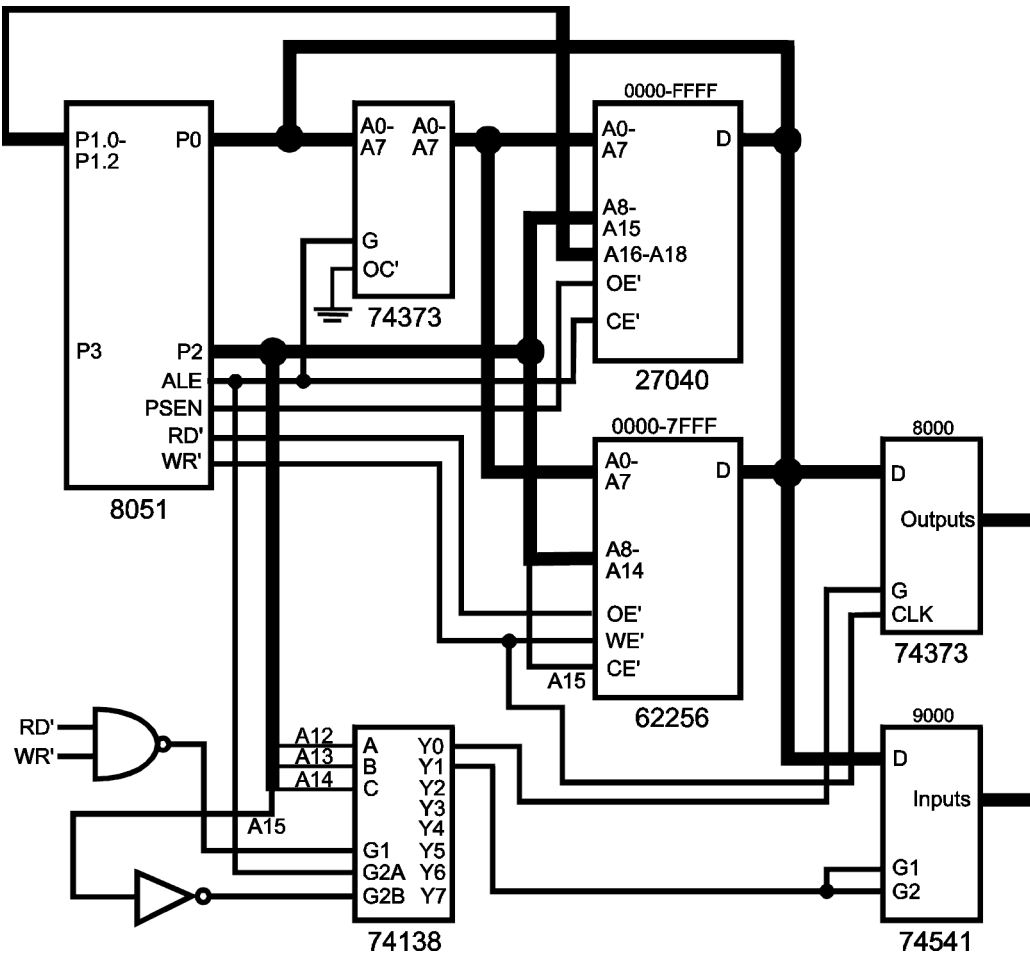


Figure 0-5 - Paging the EPROM

For example, if you need 20K for the common area, you still have 44K left over on each page for your own use. However, if you have chosen to use a single EPROM for the common area, it must be a 32K part, which leaves 12K of it unused, and only 32K maximum available on each CODE page.

The paging is performed by connecting address/enable lines on the EPROMs to either port 1 lines or pins from a data latch which is mapped into the XDATA segment, much as in the RAM paging scheme. A sample circuit is shown in Figure 0-5. Note that this circuit uses a 512K byte EPROM instead of eight 64K EPROMs. The only additional hardware required to add the extra CODE space is the larger EPROM which is simply a matter of changing EPROM sockets and having your circuit board laid out for the new address lines.

Utilizing the increased CODE memory is very simple. You simply write and compile your code as normal, however you will want to keep related functions together in a file so that they load into the same EPROM page when the linker resolves addresses. This will help minimize the number of bank switches that are necessary at run time. The more bank switches you can avoid, the more processing time you will have to perform other tasks. Additionally, you should limit the use of any constants or variables that are kept in the CODE space to a single file. This will help keep the size of the common area down and allow you to have more space per CODE page.

The best way to analyze your program to make the above decisions on function grouping and the like, is to first build your calling trees for each interrupt. In this way you can see what functions should be in the same CODE bank to avoid unnecessary bank switches. Then build another list in which you note

functions access which variables and constants stored in the CODE memory segment. Any constants which are accessed by a group of functions should be put in a file with that group if it is possible so that the functions and the constants will be written to the same CODE page. As you decide which functions and constants to group together you will find that you have to make trade offs between the size of the common area in each bank and the number of bank switches that will occur at run time. If you feel you have plenty of memory available, go for decreasing the number of bank switches by optimizing the grouping of functions. If you are still short of memory in spite of bank switching, then attempt to minimize widespread constant and code based variable accesses.

Once you have finished your code and arranged the functions and modules, you will need to modify the L51_BANK.A51 file. This configures the code which performs the bank switching by informing it as to the number of banks, and the method to use to switch the banks. You can read more about this file in the Keil manuals. Only made two changes in the configuration section need to be made to L51_BANK.A51. These changes are shown in Listing 0-7.

Listing 0-7

```
$NOCOND DEBUGPUBLICS
;-----
; This file is part of the BL51 Banked Linker/Locater package
; Copyright (c) KEIL ELEKTRONIK and Keil Software GmbH 1989-1994
; Version 1.2
;-----
;***** Configuration Section *****

change one - ensure that the proper number of code banks is set

?B_NBANKS      EQU      8          ; Define max.  Number of Banks          *
;                                                     *
?B_MODE        EQU      0          ; 0 for Bank-Switching via 8051 Port      *
;                                                     ; 1 for Bank-Switching via XDATA Port  *
;                                                     *
IF ?B_MODE = 0;                                     *
;-----*
; if ?BANK?MODE is 0 define the following values      *
; For Bank-Switching via 8051 Port define Port Address / Bits *
?B_PORT        EQU      P1          ; default is P1                        *

change two - set the bank switching LSB

?B_FIRSTBIT    EQU      0          ; default is Bit 3                      *
;-----*
ENDIF;
```

The object files generated from your source are then linked with the object file generated from compiling your customized version of l51_bank.a51. The linking process, however, is not performed with L51, but instead it is performed with BL51 which is Keil's enhanced linker with the capability to handle multiple CODE banks as well as the real time operating system that is available from Keil.

BL51 accepts all the same commands as L51 so you will be able to control it using your existing link file (if you have one). You will have to add some extra commands to the BL51 command file to specify how you want it to arrange your modules and segments amongst the CODE banks. This is where you discover how important it is to intelligently arrange your functions.

When you run BL51, you will have to provide some more data to allow it to correctly resolve addresses for all relocatable segments in your project. The first of the BL51 directives you will need to use is the 'BANKAREA' directive. 'BANKAREA' tells BL51 where you have physically mapped the CODE pages. In the case of the example circuit in this section, each CODE page is mapped to 0000H - FFFFH. You will also need to tell BL51 which modules and functions to load into the common area and into each bank. You can specifically place functions, segments, and entire modules in the common area by using the 'COMMON' directive which is defined for you in the BL51 manual. You also control the placement of functions, segments, and entire modules into CODE banks by using the 'BANKx' directive where x is a number from 0 to 15 corresponding to the bank you are working with. In the case of the circuit in this section, the following BL51 invocation would be appropriate.

```
BL51    COMMON{C_ROOT.OBJ}, &
        BANK0{BANK0.OBJ}, &
        BANK1{BANK1.OBJ}, &
        BANK2{BANK2.OBJ}, &
        BANK3{BANK3A.OBJ,BANK3B.OBJ}, &
        BANK4{BANK4.OBJ}, &
        BANK5{?PR?MYFUNC?MISC, BANK5.OBJ}, &
        BANK6{TABLES.OBJ,BANK6.OBJ}, &
        BANK7{BANK7.OBJ,MISC.OBJ} &
        BANKAREA(0000H,0FFFFH)
```

Note that you can include more than one module in a bank and that you can specify that only certain segments from a module be included in a bank by the linker. This is also true for the common area. One final point is that the linker will assign addresses to the segments in each bank in the order that they are listed. So, if for some reason you want a function to be at the lower addresses of a CODE bank, place its name first in the 'BANKx' directive for that bank.

Conclusion

This chapter has discussed ways for you to improve your programs by utilizing the features afforded to you with the Keil 8051 development package. By properly assigning register banks to your ISRs and controlling the linker's overlay process you can create very high performance code for the 8051. It has also discussed memory expansion techniques for both the RAM and the ROM of your system. I hope that you have found these discussions helpful and will be able to put the information here to full use in the near future. The next chapter will discuss a design technique that some believe is the wave of the future. However, it is here now and you can implement it on the 8051.

- Bumpin' Fuzzies with the 8051

Introduction

It seems that the world of software development is subject to waves and fads just like normal society is. These days, the concepts of object oriented design and code portability are vogue just like grunge music and environmentalism. In the area of embedded systems, particularly control systems, the latest fad is fuzzy logic. Now that American engineers have finally given up laughing at the name "fuzzy logic" they are beginning to examine the concepts behind it and are finding out that fuzzy logic can be a very valuable tool for solving certain problems. However, there are people out there who would have you believe that fuzzy logic is the "Silver Bullet" of embedded systems. They will tell you that fuzzy logic will solve all your embedded systems problems if you'll just buy and use their fuzzy logic development package. Well, you should be aware that fuzzy logic can be an excellent design approach for many embedded systems, but that there are many more embedded systems which will not benefit from fuzzy logic. This chapter will show you that you don't need to buy alot of research material to learn about fuzzy logic and an expensive development environment and code generator to run fuzzy logic on the 8051; instead there is a simple and efficient approach to embedded fuzzy on the 8051. However, before you can implement it, you need to know what the heck fuzzy logic is.

What is Fuzzy Logic?

In the world of crisp logic (logic as we generally know it), something is either true or false - it cannot be both at the same time. For example, the statement that the number five is less than the number ten is always true. This form of logic models some situations (such as linear problems) very well, and has to be massaged for other situations in which the solution space is curve. The good thing about crisp logic is that it works very well on a binary machine such as a computer since something will either be true (1) or false (0). The bad thing about crisp logic is that it does not work so well for situations which have gray areas or gradients of truth.

In the real world, we know that most things will have some degree of truth and some degree of falsehood to them. In fuzzy logic, the concept that something can be partially true and partially false at the same time is fundamental. Typically, the way fuzzy logic expresses this is by specifying a degree of membership for a data point in a given set. A value of one indicates that the data point is fully within the given set. A value of zero indicates that it is not within the set at all. Between one and zero there are an infinite amount of degrees of membership (such as .25, .5, .75, etc.).

Type of Day	Degree of Membership
Cold	0.00
Chilly	0.00
Mild	0.00
Warm	0.25
Hot	1.00

Table 0-1

For example, if it is 90 degrees outside you may say that this temperature corresponds to various descriptions of the type of day as follows.

In this case, each type of day listed can be thought of as a set in which the data point (90 degrees) has some degree of membership. In fuzzy logic systems, the degree of membership (denoted as μ) is determined by a membership function which you must specify for each fuzzy set you wish to define. A membership function is strictly a mapping from the input value to μ . The point on the y axis at which the input value intersects the plot of the membership function indicates the value of μ for that data point. Consider the following of membership functions (Cold, Chilly, Mild, Warm, and Hot) for the type of day discussed above.

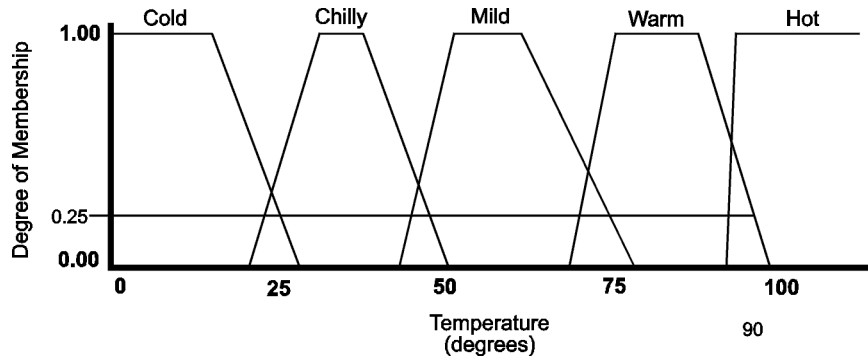


Figure 0-1 - Sample Membership Functions for Temperature

You can see that the value 90 degrees intersects two membership functions - Warm and Hot. Thus, 90 degrees has a non zero μ value for the Warm and Hot fuzzy sets. The remainder of the membership functions are not intersected and thus 90 degrees has a μ of 0 for those fuzzy sets. The value of y where 90 degrees intersects the Warm membership function is 0.25 and this becomes the value of μ for this set. Similarly, Hot is intersected at a point at which y is 1.00, thus 90 degrees lies completely within the Hot set.

The shape of the membership function can be anything that you desire, however, trapezoidal is the most common choice since other shapes can be easily derived from a trapezoidal representation. Some of the possible fuzzy membership functions are shown below.

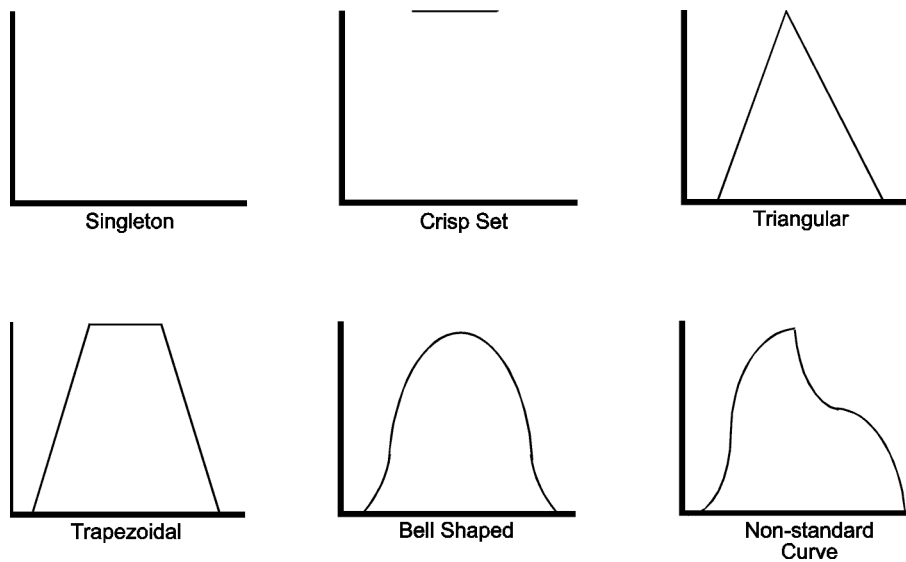


Figure 0-2 - Sample Membership Function Types

The first two curve types (singleton and crisp) allow fuzzy logic to include basic crisp logic since a data point is either a member of these sets ($\mu = 1.00$) or it is not ($\mu = 0.00$). The remaining curves all have various degrees of membership depending upon the data point selected. Note also that by modifying the inflection points of the trapezoidal curve you can obtain a singleton, crisp set, or triangular set. Thus, you can use one type of curve modeling (trapezoidal) to represent four types of fuzzy sets (trapezoidal,

triangular, crisp, and singleton). The other two types of membership functions are harder to represent because of their curves, but they can be represented.

Membership in a set is the basis for fuzzy analysis. A fuzzy system is composed of rules which make statements about an input's relationship to a fuzzy set and an associated action. For example, a fuzzy system which controls an exhaust fan's speed based on temperature might have rule which says "if the temperature is Hot then the fan speed is High." A fuzzy system will examine this and determine to what degree the temperature is hot. This degree of membership will yield a truth value for the rule which is then compared to truth values for the other rules in the system. The inter-rule comparison yields a decision on the value of fan speed.

In general, a fuzzy logic rule consists of an "if" part (the antecedent) and a "then" part (the consequence). A rule may have more than one antecedent in the "if" part and more than one consequence in the "then" part. Antecedents and consequences can be combined using logic operators such as AND, OR, and NOT. There are other fuzzy operators, but these three are the most commonly used. The mathematical implementation of the three common fuzzy operators is illustrated in Table 0-2. Note that each is a simple mathematical operation.

A fuzzy logic system consists of a set of rules which are composed using the above operators. As was said before, each rule must have a series of antecedents and consequences. The number of each can be anywhere from one to n where n is determined by your system limitations. This set of rules is typically called a rule base.

In addition, you may choose to implement a fuzzy system which assigns a weight to the rule. In most fuzzy systems the weight of each rule is set to one to indicate that each rule is just as important as the next. However, you may end up with a system in which you believe a certain rule carries more importance than the rest. In this case you can assign a weight to this rule which is greater than the weight given to all the other rules. This can be done by giving the important rule a weight of one and making the rest of the rules have a lesser weight or by leaving the other rules' weight at one and raising the weight of the more important rule. If sticking with convention is important to you, you will designate your most important rule as weight one, and lower the weights of all the remaining rules, since fuzzy logic usually deals in values from 0 to 1.

Your rule base size will depend upon the problem you are solving. Most fuzzy logic systems have a small rule base (15 rules or so); more complex systems have more rules, but usually the number of rules even for very large systems is less than 60. Keep in mind that the more rules you have, the longer your fuzzy system will take to make decisions. Usually you will not have to implement every possible rule in a system to get it to function the way you want. There is a smaller subset of rules which will properly govern the operation of the system, however, more rules will help to make the system a little more stable. One of the nice things about fuzzy logic systems is that they are extremely tolerant of input signals going bad. This property has to do with the way a rule base covers the control surface.

Operator	Implementation
$\mu_{(a \text{ AND } b)}$	$\min(\mu_a, \mu_b)$
$\mu_{(a \text{ OR } b)}$	$\max(\mu_a, \mu_b)$
$\mu_{(\text{NOT } a)}$	$1 - \mu_a$

Table 0-2

The Structure of a Fuzzy System

A fuzzy logic system requires three stages of operation to be implemented: input preprocessing, fuzzy inference, and defuzzification. The relationship of these stages is shown in Figure 0-3.

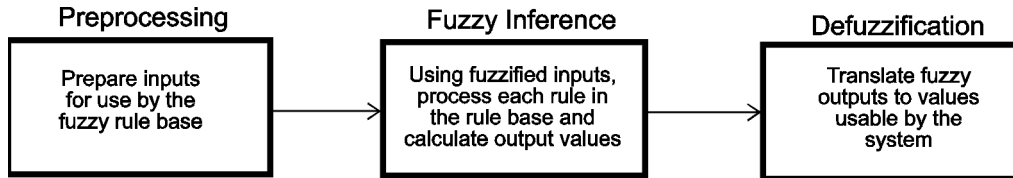


Figure 0-3 - Fuzzy System Structure

The preprocessing stage is used to prepare the inputs for evaluation with the fuzzy rules. This usually entails scaling the input to the range of values expected by your set membership functions. It may also involve computing some inputs. For example, if you have a fuzzy system for a car that uses acceleration as one of its inputs, you may have to compute this input by using velocity samples over a given time period (assuming that you don't have an accelerometer).

The fuzzy inference stage evaluates each fuzzy rule and computes its effect on the outputs specified in the consequences of the rules. The rules are evaluated using the methods described above for implementing the fuzzy operators. The value of μ for the antecedents becomes the degree to which each of the consequences is true. When a rule has a value of μ which is greater than zero, it is said to have "fired."

Each consequence of a rule refers to an output and a fuzzy set. The inference stage stores the maximum value of μ for each fuzzy output in each of its possible sets. Using the above example of "if the temperature is Hot then fan speed is High" and using 90 degrees as the input for temperature, the value of μ for the antecedents will be 1.00 since we already established that 90 degrees lied completely within the Hot fuzzy set. Thus, the degree of truth for the consequence "fan speed is High" is 1.00 since this was the value of μ for the antecedents. If the current degree of truth for fan speed in the set High is 0.00, it will now become 1.00. However, just because the fuzzy rule base has decided that fan speed has a degree of membership in the set "High" of 1.00 this does not mean that the fan speed will be set to High. This will depend on fan speed's degree of membership in its other output fuzzy sets.

The defuzzification stage takes the output membership values for the various sets and uses a mathematical method to compute the final values for the system outputs. There are a few common methods for performing defuzzification. The most simplistic is the maximizer method. This method dictates that the highest value of μ for a given output becomes the action associated with the output. For example, if the output fan speed has been given these degrees of truth:

```
 $\mu_{low} = 0.00$   
 $\mu_{medium} = 0.57$   
 $\mu_{high} = 1.00$ 
```

Then the fan speed will be set to High since this set has the highest degree of membership associated with it. This method is very simple to implement but it misses the subtleties that give fuzzy logic its power: the ability of a data point to be in more than one set at the same time.

The most common method of defuzzification is the center of gravity method. In this method, each output membership function is clipped by its degree of truth and the center of the area under the resulting curve is computed. This result is then used as the fuzzy output. Consider the following output membership functions for fan speed. The value computed for the center of gravity will be the output of the defuzzification process. As in the previous method of defuzzification, this value is applied to the system

as necessary. The center of gravity method probably yields the best results, but it is computationally intensive to determine the center of gravity of the shaded areas.

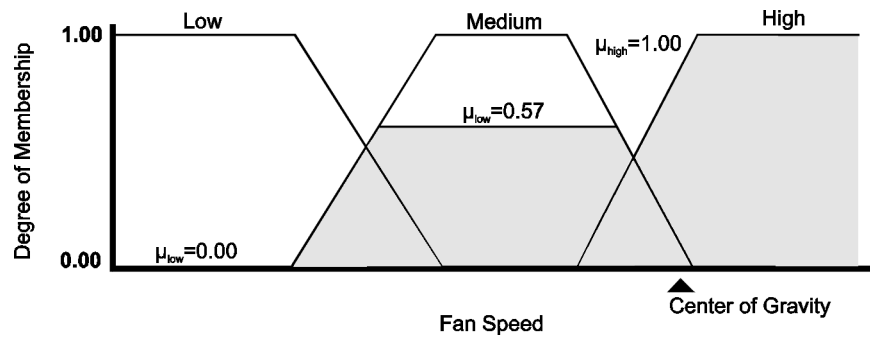


Figure 0-4 - Defuzzification

A simplification of the center of gravity method is to specify the output sets as singletons. This allows the defuzzification method to be reduced to a single loop with a multiply and divide operation. In general the formula for this method is stated as follows:

$$\frac{\sum_{i=0}^n (V_i * U_i)}{\sum_{i=0}^n U_i}$$

where n is the number of sets for the output, V_i is the value of the singleton defining the output set and U_i is the degree of truth assigned to the current output set. This function is easier to implement in software than the center of gravity equations, and yields very similar results.

Where to Use Fuzzy Logic

At this point you are probably wondering which applications will be suitable for fuzzy logic and which ones will not. A general rule of thumb for application of fuzzy logic is that if you already have a precise mathematical model of your system which has a relatively good order of efficiency when coded in conventional logic it should not be implemented in fuzzy logic. Applications in which fuzzy logic will usually excel are problems in which you do not have a precise idea of what is happening but you have a sort of “seat of the pants” idea of how to control the system. This is frequently the case in complex non-linear applications. Usually, you can consult some expert on the system and determine a set of rules by which the system operates. Fuzzy logic will be able to process these rules to obtain reasonable outputs even when one or more of the inputs fails.

One of the strong points of fuzzy logic is that you can express the solution to the problem you are solving in linguistic terms. This makes the solution more accessible to humans, since they deal in linguistics, not numerics. Additionally, the behavior of a fuzzy logic system is easily altered by changing input rules and membership functions. One drawback to a fuzzy logic implementation is that you will have to verify that your fuzzy logic solution handles all points of the desired control surface by performing simulation. In other words, you will not be able to mathematically prove that your solution works for all inputs as you may be able to do for conventional control systems.

Getting Fuzzy

Now that we’ve gone over some of the basics of fuzzy logic, let’s run through a small design example through which you can learn the approach to designing a fuzzy logic system. The way to start a fuzzy design is to understand the system to be modeled in simple linguistic terms. This means that you should gain an understanding of what inputs and outputs you have as well as the classifications they can fall into.

Assume that you have to design a system which controls a self powered vehicle that is programmed with a point at which it must stop. To allow it to do this it will use the distance from its location to the point and current velocity as its inputs. As an output, it will specify an amount of braking to use to slow the vehicle. The vehicle has a drive system which can be overcome by a light amount of braking when the it is at very low speeds or stopped.

Given the above description of the system, the inputs have been identified as the distance to the stopping point and the current vehicle velocity. The output of the system has been identified as the level of braking to be used. The next task is to define fuzzy sets for the inputs and the output. At first, you do this in very basic terms without concerning yourself with mathematical values at all.

You come up with the linguistic terms based on the intuitive knowledge you have of the system. Such knowledge may be derived from your own expertise or through investigation into the system either by studying it or by interviewing with an expert. For example, if the vehicle in the example had formerly been operated by a driver who held that job for twenty years, you could speak with that person (who you are putting out of a job) and find the parameters he used to operate the vehicle. Since there is no expert in this case, let’s just wing it, basing the information on the common experience of driving a car.

The first input to consider is the distance to the stopping point. This system will not begin activation until the stopping point is close enough to be concerned with. In other words, if you were driving a car and you saw that one half of a mile away was a stop sign, you would not concern yourself with slowing for the stop sign until a few hundred feet before it. It is the position at which you would begin to slow for the stop sign that the fuzzy system kicks in. Given this information, the distance to the stopping point (hereafter given the linguistic label DISTANCE) can be broken into the following categories: FAR, NEAR, CLOSE, VCLOSE (for very close). For right now, don’t concern yourself with the membership functions of these sets other than the fact that the VCLOSE set will include the notion that the stopping point has been reached and that the FAR set will include distances beyond which the fuzzy stopping system was activated.

The second input to the system is the velocity of the vehicle (hereafter given the linguistic label VELOCITY). This input is also broken into categories (as was DISTANCE) but the categories are not the same. VELOCITY has been assigned the following categories: VFAST (not the 28.8k baud standard!), FAST, MEDIUM, SLOW, VSLOW. Similar to the first input, you should note that the VSLOW set includes the notion that the vehicle has stopped.

The output of the system was defined as the amount of braking to be used to slow the vehicle. This output will be given the label BRAKE. As was the case with the two inputs, BRAKE is divided into the following categories: NONE, LIGHT, MEDIUM, HARD, VHARD. The mathematical meaning of these sets will be defined later.

Now that the inputs and outputs have been defined, the next step is to make an initial cut at the rules which will be needed in the system. Some fuzzy designers state that the membership functions for the fuzzy sets should be defined before the rules, but this is just a matter of preference. My reasoning for defining an initial cut at the rule base first is to solidify general understanding of the system before mathematical definitions of the system are made. In this way, the rough draft of the rule base can be used to prove the design concept before it has gone too far.

The simplest way to define your rule base is to generate a matrix of your inputs and then fill in the matrix with the output type you want to occur. Thus, instead of writing out a bunch of rules of the form "if x and y then z" you can fill in a matrix. This makes the system easier to visualize. Keep in mind that the matrix is really only appropriate when you are using the AND operator. In our example, there will not be a need to perform any operation other than AND. Table 0-3 shows the rule matrix for the VELOCITY and DISTANCE inputs before it has been filled in.

		DISTANCE			
		FAR	NEAR	CLOSE	VCLOSE
VELOCITY	VFAST				
	FAST				
	MEDIUM				
	SLOW				
	VSLOW				

Table 0-3

Once you have structured your matrix as shown above, you simply treat it like a truth table and fill it in with the linguistics. The upper left square in the matrix should contain the consequence of the antecedent "if VELOCITY is VFAST and DISTANCE is FAR." Since you are filling the table in for the first time, don't worry about being exact, just get a rough idea of what's going on. The filled in table is shown in Table 0-4.

		DISTANCE			
		FAR	NEAR	CLOSE	VCLOSE
VELOCITY	VFAST	MEDIUM	HARD	VHARD	VHARD
	FAST	MEDIUM	HARD	VHARD	VHARD
	MEDIUM	LIGHT	MEDIUM	HARD	HARD
	SLOW	NONE	NONE	LIGHT	MEDIUM
	VSLOW	NONE	NONE	NONE	LIGHT

Table 0-4

As was said before, the matrix is just a starting point for your rule base. It may be that you need to implement all the rules in the matrix, but usually, a fuzzy logic system functions with a subset of the rules in the matrix. Unfortunately, there is no theorem for eliminating unneeded rules, the only common method is trial and error. In some cases, however, a set of rules can be simplified into one rule. For example, if the rule “if VELOCITY is VSLOW and DISTANCE is VCLOSE then BRAKE is LIGHT” had a consequence of “BRAKE is NONE” instead, the bottom row of the matrix could be replaced by the rule “if VELOCITY is VSLOW then BRAKE is NONE.”

Now that the initial system rules have been established, the membership functions for each fuzzy set specified can be established. To perform this function, you need to know the possible range of values for each input you are dealing with. For example, to establish the fuzzy set membership functions for VELOCITY, you would need to know that the VELOCITY can range from 0 MPH to 25 MPH and base your fuzzy sets upon this. For this example, let's use the above range for velocity and define the fuzzy membership functions.

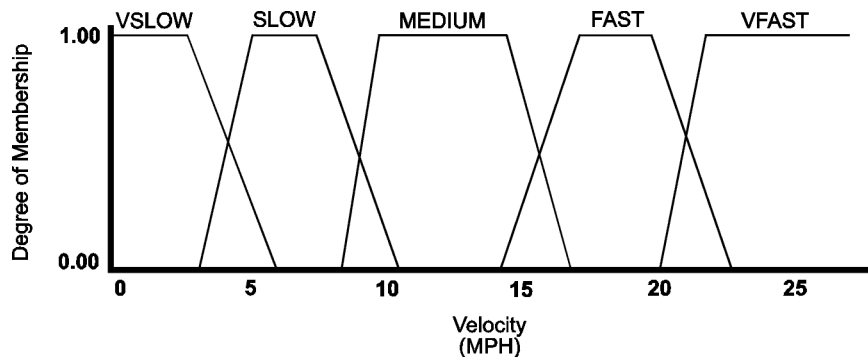


Figure 0-5 - Membership Functions for Velocity

The membership functions drawn above are according to my definition of the term in the context of the problem to be solved. Someone else may have slightly differing membership functions. You should note that there are no specific requirements as to how the membership functions interrelate. Thus, it is not a rule that you have to include every point on the X axis in a fuzzy set. It is also not a rule that a given point on the X axis could not be completely in two sets at the same time, or have a nonzero value of μ for three or more sets. Remember, the definition of the membership functions is dependent upon the context of the problem you are solving, not some predefined theoretical rules. The membership functions for both the DISTANCE input and the BRAKE output have been defined using the range indicated in the plots.

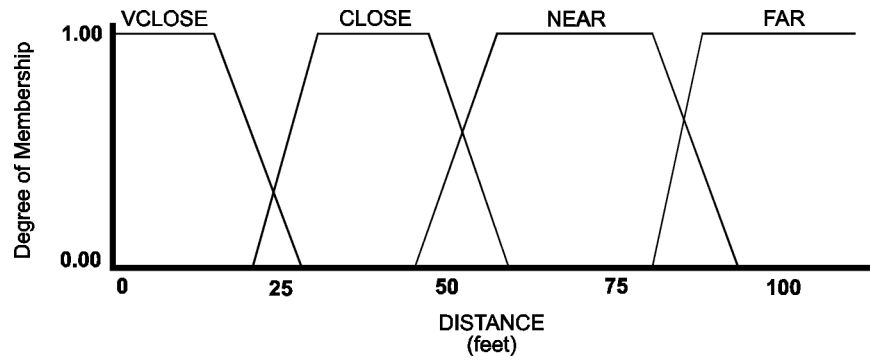


Figure 0-6 - Membership Functions for Distance

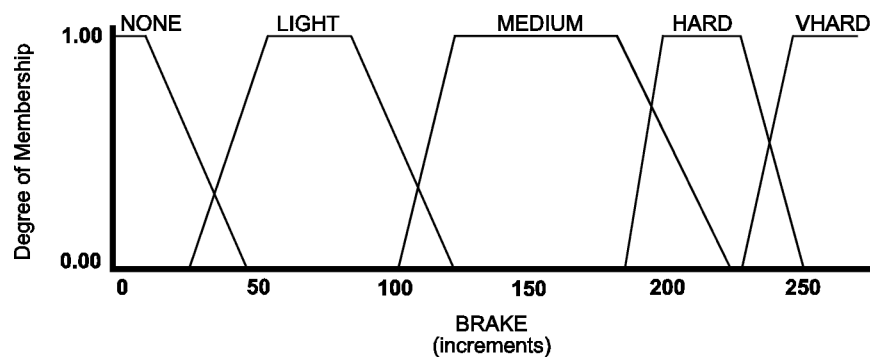


Figure 0-7 - Membership Functions for Brake Pressure

Now that the rules and the membership functions have been established, all the pieces of the fuzzy logic system design are in place. All that is needed now is the software implementation of a fuzzy logic engine to run the rules and give us outputs.

Starting the Engine

Implementing fuzzy logic for an eight bit controller is actually a straightforward job. Once you have an engine written that runs in a reasonable amount of time, the only work you will have to do to set up a new fuzzy application will be to define the rules and membership functions. This can easily be done by hand or by using one of the many commercially available tools. Personally, I do the work by hand, but that does not mean that manually is the best way to go.

When designing the fuzzy logic engine for an eight bit system, the first thing to consider is how you will represent the rules in the system. Since the 8051 does best with dealing with eight bits at a time, you can make certain limitations to the fuzzy logic to allow the antecedents and consequences to be represented in eight bit chunks. Thus, the fuzzy logic system will not currently support rule weights or use of parenthesis to force a certain order of operation in a evaluation of the antecedents. These types of changes are easily made and are left up to your own design. The fuzzy logic implementation discussed here does support both the AND and OR operators as well as up to eight inputs and eight outputs each of which can have up to eight fuzzy sets associated with them.

The fuzzy rule base is kept in an array that is stored in the CODE space. Each element of the array is one byte and holds a clause of a fuzzy rule. This byte indicates the input or output referenced, the fuzzy set referenced, whether the clause is an antecedent or a consequence as well as the fuzzy operator to use with the clause if it is an antecedent. This information is packed into a single byte to save space on the EPROM at the expense of some processing time during execution; however, if you are long on storage space, you could just as easily put all of this data into a structure which would take more storage, but would allow you to get to the information about the clause quicker. Table 0-5 shows the arrangement of the data in a clause byte.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0 = antecedent 1 = consequence	membership function for input or output used			0 = AND 1 = OR	input or output number to use		

Table 0-5

When bit 7 of a clause is cleared, the fuzzy engine assumes that the clause is an antecedent and thus, the value in bit 2 to bit 0 must indicate an input. The value in bit 6 to bit 4 refers to the fuzzy set to use with that input. Bit 3 is also examined to determine if the value of μ for this clause should be ANDed or ORed with the μ value computed thus far for the rule.

When bit 7 of the clause is set, the fuzzy engine assumes that the clause is a consequence and that the value in bit 2 to bit 0 refers to an output. When bit 7 is set, the value of bit 3 has no bearing on the resulting analysis.

Given the internal representation of a clause, rules are very simple to form. Continuing our example of the moving vehicle, let's assign input number zero to velocity and input number one to distance. Since there is only one output (brake pressure), it will be assigned number zero. The membership functions are also assigned indices in order from 0 to n. In the case of this example, the highest value of n is five since there are five membership functions for the velocity input. Once these assignments have been made, you can create a section of code which defines constants for each of the possible clauses you may want to use. For example, a clause which says "VELOCITY is VFAST" would be translated into a constant named "VEL_VFAST." Do this for every possible combination of input or output and membership label. That way, if you later decide you want to use a clause you haven't used before, there is already a constant defined for it. In the case of the vehicle example, the constants defined are shown in Listing 0-1.

Listing 0-1

```
// define constants for the velocity input
#define VEL_VSLOW      0x00
#define VEL_SLOW       0x10
#define VEL_MEDIUM     0x20
#define VEL_FAST       0x30
#define VEL_VFAST      0x40
// define constants for the distance input
#define DIST_VCLOSE     0x01
#define DIST_CLOSE     0x11
#define DIST_NEAR      0x21
#define DIST_FAR        0x31
// define constants for the brake output
#define BRAKE_NONE      0x80
#define BRAKE_LIGHT     0x90
#define BRAKE_MEDIUM    0xA0
#define BRAKE_HARD      0xB0
#define BRAKE_VHARD     0xC0
```

The above constants allow rules to take a very simple format in the rule base. For the most part simple systems such as the vehicle system will have rules of the form “if input x1 is label y1 and input x2 is label y2 then output x3 is label y3.” You can, however, have any combination of AND and OR you want just by arranging the clauses in the right order to build the rule you want. Thus, if you wanted to say “if velocity is fast or velocity is slow and distance is far then brake is none” it would be represented with the following line of constants:

```
VEL_FAST, VEL_SLOW | 0x08, DIST_FAR, BRAKE_NONE
```

Rules are built out of clause strings like the one above and are packed back to back into an array stored in the CODE memory space. In the vehicle example, I have coded every possible rule and put it in the rule base array as shown in Listing 0-2. You would not necessarily have to represent every rule in the rule base, the fuzzy system will probably function just as well without them all, but I have done it to make the example a little easier to follow.

Listing 0-2

```
unsigned char code rules[RULE_TOT]={    // fuzzy system rules
// if...      and...      then...
    VEL_VSLOW, DIST_VCLOSE, BRAKE_LIGHT,
    VEL_VSLOW, DIST_CLOSE,  BRAKE_NONE,
    VEL_VSLOW, DIST_NEAR,   BRAKE_NONE,
    VEL_VSLOW, DIST_FAR,    BRAKE_NONE,
    VEL_SLOW,  DIST_VCLOSE, BRAKE_MEDIUM,
    VEL_SLOW,  DIST_CLOSE,  BRAKE_LIGHT,
    VEL_SLOW,  DIST_NEAR,   BRAKE_NONE,
    VEL_SLOW,  DIST_FAR,    BRAKE_NONE,
    VEL_MEDIUM, DIST_VCLOSE, BRAKE_HARD,
    VEL_MEDIUM, DIST_CLOSE,  BRAKE_HARD,
    VEL_MEDIUM, DIST_NEAR,   BRAKE_MEDIUM,
    VEL_MEDIUM, DIST_FAR,    BRAKE_LIGHT,
    VEL_FAST,   DIST_VCLOSE, BRAKE_VHARD,
    VEL_FAST,   DIST_CLOSE,  BRAKE_VHARD,
    VEL_FAST,   DIST_NEAR,   BRAKE_HARD,
    VEL_FAST,   DIST_FAR,    BRAKE_MEDIUM,
    VEL_VFAST,  DIST_VCLOSE, BRAKE_VHARD,
    VEL_VFAST,  DIST_CLOSE,  BRAKE_VHARD,
    VEL_VFAST,  DIST_NEAR,   BRAKE_HARD,
    VEL_VFAST,  DIST_FAR,    BRAKE_MEDIUM
};
```

Now that the implementation of the rules is complete, the next thing that needs to be defined is the implementation of the fuzzy membership functions. The fuzzy engine for the 8051 assumes that your membership functions will have a shape that is either trapezoidal or can be derived from a trapezoid. Thus, you can also have a crisp set, triangular set or a singleton. In fact, outputs will be made singletons to simplify the defuzzification process.

Storing the input membership functions as trapezoids will allow the software to represent your membership area by using just four bytes per function. This is done by looking at the trapezoid as a clipped triangle. The software then represents the larger triangle by storing the inflection points of the triangle and the slope of the two vectors that define the triangle. An illustration of this is shown in Figure 0-8.

Inflection point 1 and inflection point 3 are stored as part of the membership function as are the values of slope 1 and slope 2. Using these four values the software can determine the y value along the correct vector and thus obtain a value for μ . μ is represented as an unsigned char with the value FFH indicating complete membership and 00H indicating lack of membership. The computation of μ for a given membership function is done with integers in case the y value exceeds FFH or goes negative. In either of these cases, μ is clipped back into the legal range (00H to FFH). The values for the membership functions are stored in a three dimensional array which allows for up to eight different inputs each having eight different membership functions. This table is shown in Listing 0-3.

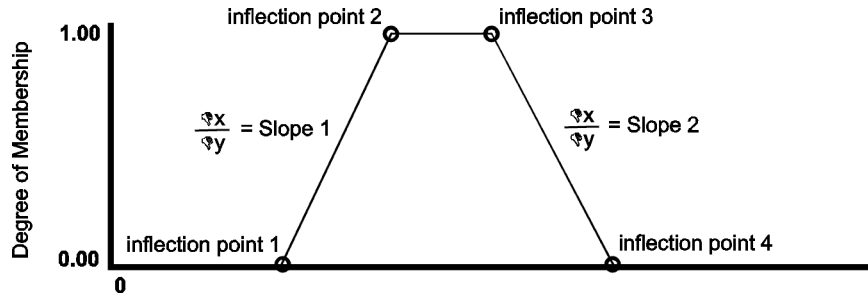


Figure 0-8 - Trapezoid Inflection Points

Listing 0-3

```
unsigned char code input_memf[INPUT_TOT][MF_TOT][4]={
    // input membership functions in point slope form. The first
    // dimension is the input number, the second dimension is
    // the label number and the third dimension refers to the
    // point/slope data

    // membership functions for velocity
    {
        { 0x00, 0x00, 0x1E, 0x09 },      // VSLOW
        { 0x1E, 0x0D, 0x50, 0x09 },      // SLOW
        { 0x50, 0x0D, 0x96, 0x0D },      // MEDIUM
        { 0x8C, 0x06, 0xC8, 0x09 },      // FAST
        { 0xC8, 0x0D, 0xFF, 0x00 }       // VFAST
    },

    // membership functions for distance
    {
        { 0x00, 0x00, 0x2B, 0x0A },      // VCLOSE
        { 0x33, 0x08, 0x80, 0x0A },      // CLOSE
        { 0x6E, 0x07, 0xC7, 0x08 },      // NEAR
        { 0xC7, 0x0A, 0xFF, 0x00 }       // FAR
    }
};
```

The hex values in this table are derived from the membership functions you design. For example, once you have created membership functions that make sense to you and have drawn them out using the actual range of values you expect, simply convert the range into a new range from 0 to FFH. Then convert the values of the four inflection points by scaling the values from the original scale you used to the new scale of 00H to FFH. Once you have this amount of information, you can enter it into the simple program shown below which will take as input the four inflection points and return as output the fuzzy representation of the membership function (point 1, slope 1, point 3, slope 2). The listing for this program is shown in Listing 0-4.

Listing 0-4

```
#include <stdio.h>

void main(void) {
    unsigned char val[4], output[4], ans, flag;
    do {
        printf("\n\nenter 4 hex points: ");
        scanf(" %x %x %x %x", &val[0], &val[1], &val[2], &val[3]);
        output[0]=val[0];
        output[2]=val[2];
        if (val[1]-val[0]) {
            output[1]=(0xFF+((val[1]-val[0])/2))/(val[1]-val[0]);
        } else {
            output[1]=0;
        }
        if (val[3]-val[2]) {
            output[3]=(0xFF+((val[3]-val[2])/2))/(val[3]-val[2]);
        } else {
            output[3]=0x00;
        }
        printf("\nThe point-slope values are: %02X %02X %02X\n\n",
            output[0], output[1], output[2], output[3]);
        do {
            flag=1;
            printf("run another set of numbers? ");
            while (!kbhit());
            ans=getch();
            if (ans!='y' && ans!='Y' && ans!='n' && ans!='N') {
                flag=0;
                printf("\nhuh?\n");
            }
        } while (!flag);
    } while (ans=='y' || ans=='Y');
    printf("\nlater, hosehead!\n");
}
```

This simple little tool will allow you to complete development of your membership functions for the system inputs. Once you have established the input membership functions, you need to come up with singleton values for the output membership functions. You can do this by selecting the point in the middle of the mass of the output membership function. The singletons are used for the outputs instead of membership functions because they greatly simplify the math required in the defuzzification phase and the results obtained from using singletons are almost the same as the results obtained from using full membership functions. Figure 0-9 shows the output membership functions for the automated vehicle and the lines at which I chose to draw the singletons.

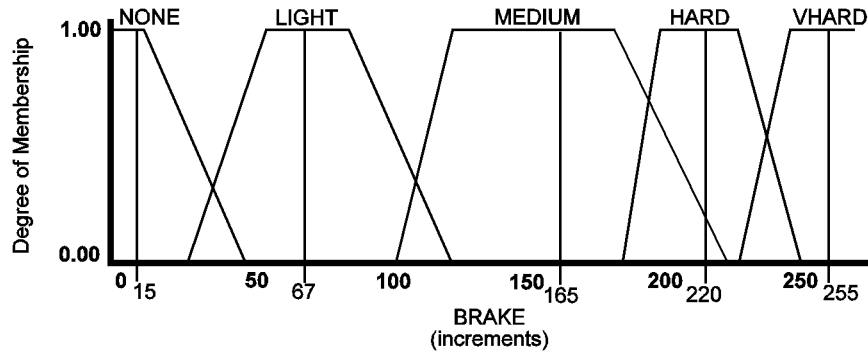


Figure 0-9 - Output Membership Function Singletons

The singleton values for the outputs are then stored in a table which is kept on the EPROM. The listing for this table is shown below.

Listing 0-5

```
unsigned char code output_memf[OUTPUT_TOT][MF_TOT]={
    // output membership singletons
    // The first dimension is the output number, the second is
    // the label number

    { 15, 67, 165, 220, 255, 0, 0, 0 } // braking force singletons:
                                     // NONE, LIGHT, MEDIUM, HARD,
                                     // VHARD
};
```

The fuzzy logic engine performs evaluation of the rule base by looping through the rule base array one element at a time. While it is analyzing antecedents it keeps track of the current value of μ for the rule in a variable called 'if_val'. Once the antecedents have been completed and the consequences are examined, the fuzzy engine evaluates the consequences by checking 'if_val' against the current μ values for the output and label specified in the clause. If the magnitude of 'if_val' is greater than the previous μ for the output, 'if_val' becomes the new μ . Once the consequences have all been analyzed and a new rule begins, the value of 'if_val' is reset.

The source code for the fuzzy engine function is shown in Listing 0-6. Note that the current clause under evaluation is stored in the bit addressable segment and that a couple of other bit variables have been assigned to point into this value. This arrangement speeds up access to the information in the clause.

Listing 0-6

```

/*****
Function:      fuzzy_engine
Description:   Executes the rules in the fuzzy rule base.
Parameters:   none.
Returns:      nothing.
Side Effects:  none.
*****/
unsigned char bdata clause_val;      // fast access storage for the
                                     // current clause
sbit operator = clause_val^3;        // this bit indicates fuzzy
                                     // operator to use
sbit clause_type = clause_val^7;     // this bit indicates if the
                                     // clause is an antecedent
                                     // or a consequence

void fuzzy_engine(void) {
    bit then;                        // set true when consequences
                                     // are being analyzed
    unsigned char    if_val,         // holds the mu value for
                                     // antecedents in the current rule
                    clause,          // indicates the current
                                     // clause in the rule base
                    mu,              // holds the mu value of the
                                     // current clause
                    inp_num,         // indicates the input used by
                                     // the antecedent
                    label;           // indicates the membership
                                     // function used by the
                                     // antecedent

    then=0;                          // assume that the first
                                     // clause is an antecedent
    if_val=MU_MAX;                   // max out mu for the first rule
    for (clause=0; clause<RULE_TOT; clause++) { // loop through
                                     // all the rules
        clause_val=rules[clause];    // reads the current clause
                                     // into bdata
        if (!clause_type) {          // if the current clause is an
                                     // antecedent...
            if (then) {              // if a then part was being run...
                then=0;              // change back to if
                if_val=MU_MAX;       // and reset mu for the rule
            }
        }
    }
}

```

```
inp_num=clause_val & IO_NUM;    // get the referenced input number
label=(clause_val & LABEL_NUM) / 16; // get the referenced
                                   // membership function

mu=compute_memval(inp_num, label); // get value of mu for
                                   // this clause
if (operator) {                    // if this is an OR
                                   // operation...
    if (mu > if_val) {              // implement the MAX function
        if_val=mu;
    }
} else {                           // if this is an AND operation
    if (mu < if_val) {              // implement the MIN function
        if_val=mu;
    }
}
} else {                           // the current clause is a
                                   // consequence
    then=1;                        // ensure that the engine
                                   // knows it is running
                                   // consequences

    // if the current rule's mu is higher than the referenced
    // output value then store if_val as the new fuzzy output
    // value
    if (outputs[clause_val & IO_NUM]
        [(clause_val & LABEL_NUM) / 16] < if_val) {
        outputs[clause_val & IO_NUM]
            [(clause_val & LABEL_NUM) / 16]=if_val;
    }
}
}
defuzzify();                        // compute the fuzzy outputs
                                   // using the COG method and
                                   // defuzzify the outputs
}
```

It is the responsibility of the code calling this function to ensure that the system inputs have been fuzzified (i.e.: converted to a scale from 00H to FFH to match the membership functions).

Evaluation of each clause takes place in a function called 'compute_memval'. The job of this function is to determine the value of μ for the clause given the input to use and the fuzzy set within that input to evaluate. Keeping this code in a single function allows it to be easily replaced with other code should the implementation of the membership functions need to change.

Listing 0-7

```

/*****
Function:      compute_memval
Description:   Calculates mu for a given antecedent assuming that
               the membership functions are stored in point slope
               format.
Parameters:   inp_num - unsigned char.  The input number to use.
               label - unsigned char.  The membership function for
               that input to use.
Returns:      unsigned char.  The computed value for mu.
Side Effects:  none.
*****/
unsigned char compute_memval(unsigned char inp_num,
                             unsigned char label) {

    int data temp;
    if (input[inp_num] < input_memf[inp_num][label][0]) {
                                                // if the input is not
                                                // under the curve, mu is 0

        return 0;
    } else {
        if (input[inp_num] < input_memf[inp_num][label][2]) {
                                                // the input falls under
                                                // the first vector

            temp=input[inp_num];                // use point-slope math to
                                                // compute mu

            temp-=input_memf[inp_num][label][0];
            if (!input_memf[inp_num][label][1]) {
                temp=MU_MAX;
            } else {
                temp*=input_memf[inp_num][label][1];
            }
            if (temp < 0x100) {                  // if mu did not exceed 1
                return temp;                    // return the computed value
            } else {
                return MU_MAX;                  // make sure that mu stays in range
            }
        } else {                                // the input falls under the
                                                // second vector

            temp=input[inp_num];                // use point-slope math to
                                                // compute mu

            temp-=input_memf[inp_num][label][2];
            temp*=input_memf[inp_num][label][3];
            temp=MU_MAX-temp;
            if (temp < 0) {                      // make sure that mu is not

```

```
                                // less than 0
    return 0;
} else {
    return temp;                // mu was positive - return
                                // its value
}
}
}
return 0;
}
```

On completion of the iterations through the rule base, the fuzzy engine calls the defuzzify function to convert the μ values held in memory to COG outputs which will be usable by the system. This function implements the mathematical calculation which was discussed before. It is the defuzzification portion of the fuzzy engine which takes the largest amount of time because of the math involved. The code for the defuzzify function is shown in

Listing 0-8.

Listing 0-8

```

/*****
Function:      defuzzify
Description:   Computes the center of gravity of the fuzzy
               outputs and calls a function to convert the fuzzy
               COGs to outputs usable by the system.
Parameters:    none.
Returns:       nothing.
Side Effects:  The outputs[][] array is cleared.
*****/
void defuzzify(void) {
    unsigned long numerator, denominator;
    unsigned char i, j;

    for (i=0; i<OUTPUT_TOT; i++) {        // for all outputs...
        numerator=0;                       // reset the summation values
        denominator=0;
        for (j=0; j<MF_TOT; j++) {        // compute the summation values
            numerator+=(outputs[i][j]*output_memf[i][j]);
            denominator+=outputs[i][j];
            outputs[i][j]=0;               // clear the output as its used
        }
        if (denominator) {                 // make sure that a divide by
                                           // 0 does not occur
            fuzzy_out[i]=numerator/denominator; // finalize the COG
        } else {
            fuzzy_out[i]=DEFAULT_VALUE;    // no rules fired for this
                                           // output - set a default value
        }
    }
    normalize();                           // change fuzzy output to
                                           // normal output
}

```

Tuning the Engine

The fuzzy logic engine as described above consumes a relatively low amount of memory. I compiled the engine without regard for any optimization of memory space in the input membership function and output membership function arrays and discovered that it only used 3 bytes of internal RAM, 80 bytes of external RAM, and 1290 bytes of CODE memory (380 bytes of which were used by arrays). This is a relatively small amount of memory when you consider the power that a fuzzy logic engine can bring to a system. The performance of the system was also quite reasonable. I ran the system with sample inputs and obtained the following results for processor usage.

While not oppressively slow, the above performance numbers are probably not as good as they could be. At this point, there is a trade off to be made. Using the vehicle example as the case study, if you needed more EPROM space, you could optimize the allocation of the membership function tables to give yourself a couple hundred more bytes of storage and live with the performance

limitations of the system. In fact, for many embedded systems, an average time of less than 33000 processor cycles may be more than fast enough. However, if you find that this is too slow for your system, you can speed up the fuzzy engine at the expense of some system memory.

The quickest way to improve the system is to change the implementation of the fuzzy membership functions. The point-slope storage format is great for applications that don't have a lot of space, but it also costs a fair amount of processing time. If you are willing to give up some EPROM space you can store the membership value for each of the 256 points along the input range and avoid computation of μ at run time. In the case of the vehicle example it will take 2304 bytes of EPROM space to represent the fuzzy membership functions. In addition to this change, it makes sense to limit the table dimensions of the input and output arrays to reflect the problem being solved. Thus, the arrays in the vehicle fuzzy logic system have been limited to one output, two inputs and five membership functions. This change will allow the defuzzify function to execute in less time since there will be less passes through the output loop. Additionally, the time to execute μ will be greatly reduced because the 'compute_memval' function has been replaced by the following line of code.

```
// get value of mu for this clause
mu=input_memf[inp_num][label][input[inp_num]];
```

Once the changes described above have been made, the new fuzzy logic system has the following amazing stats.

By simply adapting the general fuzzy engine to the vehicle example the performance of the system has improved by well over four times. The execution time of 7500 cycles is now far more acceptable for any system than was the execution time of 33000 cycles. The new system uses one byte of internal RAM, 8 bytes of external RAM, and 3010 bytes of EPROM of which 2625 bytes are tables for the membership functions.

Fuzzy Velocity (hex)	Fuzzy Distance (hex)	Processor Cycles to Run Fuzzy Engine
00	00	29576
12	35	31115
57	29	31714
80	43	33167
D0	D0	37112
FF	00	33283
Average processor cycles		32661

Table 0-6

Fuzzy Velocity (hex)	Fuzzy Distance (hex)	Processor Cycles to Run Fuzzy Engine
00	00	7506
12	35	7549
57	29	7579
80	43	7549
D0	D0	7568
FF	00	7561
Average processor cycles		7552

Table 0-7

This engine can be fine tuned to deliver even more performance. Since the vehicle example does not need to use the OR operator, this section of the code can be eliminated. By definition of the AND operator, once an antecedent has been given a value of 0.00 for μ the remainder of the rule can be skipped. This change has been made to the code and has resulted in the following numbers. The code now uses one byte of internal RAM, 8 bytes of external RAM, and 3031 bytes of EPROM of which 2625 are consumed by tables. The biggest change comes in the performance of the system.

The fuzzy logic engine as it exists for the vehicle system is shown in Listing 0-9. The entire engine is kept in one file and should have a header file defined for it to allow easy integration with the rest of a system.

Fuzzy Velocity (hex)	Fuzzy Distance (hex)	Processor Cycles to Run Fuzzy Engine
00	00	5684
12	35	5750
57	29	6076
80	43	5750
D0	D0	6118
FF	00	5739
Average processor cycles		5853

Listing 0-9

Table 0-8

```
#define OUTPUT_TOT      1
#define MF_TOT          5
#define INPUT_TOT       2

#define MU_MAX          0xFF

#define IO_NUM          0x07
#define LABEL_NUM       0x70
#define DEFAULT_VALUE   0x80

unsigned char           outputs[MF_TOT], // fuzzy output mu values
                      fuzzy_out;        // fuzzy engine outputs

unsigned char           input[INPUT_TOT] = { // fuzzified inputs
    0, 0
};

unsigned char code input_memf[INPUT_TOT][MF_TOT][256]={
// input membership functions
{
    { // velocity: VLOW
        0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
        0xFF, 0xFF,
        0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
        0xFF, 0xF6,
        0xED, 0xE4, 0xDB, 0xD2, 0xC9, 0xC0, 0xB7, 0xAE, 0xA5, 0x9C, 0x93, 0x8A, 0x81, 0x78, 0x6F, 0x66,
        0x5D, 0x54, 0x4B, 0x42, 0x39, 0x30, 0x27, 0x1E, 0x15, 0x0C, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00,
    }
}
```


[illegible]

THE FINAL WORD ON THE 8051

```
    0x78, 0x7E, 0x84, 0x8A, 0x90, 0x96, 0x9C, 0xA2, 0xA8, 0xAE, 0xB4, 0xBA, 0xC0, 0xC6,
    0xCC, 0xD2,
    0xD8, 0xDE, 0xE4, 0xEA, 0xF0, 0xF6, 0xFC, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF6, 0xED, 0xE4, 0xDB, 0xD2,
    0xC9, 0xC0,
    0xB7, 0xAE, 0xA5, 0x9C, 0x93, 0x8A, 0x81, 0x78, 0x6F, 0x66, 0x5D, 0x54, 0x4B, 0x42,
    0x39, 0x30,
    0x27, 0x1E, 0x15, 0x0C, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00,
    },
    { // velocity: VFAST
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0D, 0x1A, 0x27, 0x34, 0x41,
        0x4E, 0x5B,
        0x68, 0x75, 0x82, 0x8F, 0x9C, 0xA9, 0xB6, 0xC3, 0xD0, 0xDD, 0xEA, 0xF7, 0xFF, 0xFF,
        0xFF, 0xFF,
        0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
        0xFF, 0xFF,
        0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
        0xFF, 0xFF,
    }
},
{
    { // distance: VCLOSE
        0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
        0xFF, 0xFF,
        0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
        0xFF, 0xFF,
    }
}
```

[illegible]

THE FINAL WORD ON THE 8051

```
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00
},
{ // distance: NEAR
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x07,
    0x0E, 0x15, 0x1C, 0x23, 0x2A, 0x31, 0x38, 0x3F, 0x46, 0x4D, 0x54, 0x5B, 0x62, 0x69,
0x70, 0x77,
    0x7E, 0x85, 0x8C, 0x93, 0x9A, 0xA1, 0xA8, 0xAF, 0xB6, 0xBD, 0xC4, 0xCB, 0xD2, 0xD9,
0xE0, 0xE7,
    0xEE, 0xF5, 0xFC, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF7, 0xEF, 0xE7, 0xDF, 0xD7, 0xCF,
0xC7, 0xBF,
    0xB7, 0xAF, 0xA7, 0x9F, 0x97, 0x8F, 0x87, 0x7F, 0x77, 0x6F, 0x67, 0x5F, 0x57, 0x4F,
0x47, 0x3F,
    0x37, 0x2F, 0x27, 0x1F, 0x17, 0x0F, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00
},
{ // distance: FAR
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00
```

```

    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0A, 0x14, 0x1E, 0x28, 0x32, 0x3C,
    0x46, 0x50,
    0x5A, 0x64, 0x6E, 0x78, 0x82, 0x8C, 0x96, 0xA0, 0xAA, 0xB4, 0xBE, 0xC8, 0xD2, 0xDC,
    0xE6, 0xF0,
    0xFA, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF
}
}
};

unsigned char code output_memf[MF_TOT]={
15, 67, 165, 220, 255          // braking force singletons:
                                // NONE, LIGHT, MEDIUM, HARD,
                                // VHARD
};

//*****
// The rule base is defined below. Each clause can be an
// antecedent or a consequence depending upon the value of bit 7.
// If bit 7==0 then the clause is an antecedent; if bit 7==1 then
// the clause is a consequence. Bits 6 through 4 identify the
// label number referred to by the rule. Bit 3 indicates the
// operator to use if the rule is an antecedent. If bit 3==0, the
// AND operator should be used. If bit 3==1, the OR operator
// should be used. Bits 2 through 0 identify the input or output
// referred to by the clause.
//*****
// define constants for the velocity input
#define VEL_VSLOW      0x00
#define VEL_SLOW       0x10
#define VEL_MEDIUM     0x20
#define VEL_FAST       0x30

```

```
#define VEL_VFAST      0x40
// define constants for the distance input
#define DIST_VCLOSE    0x01
#define DIST_CLOSE     0x11
#define DIST_NEAR      0x21
#define DIST_FAR        0x31
// define constants for the brake output
#define BRAKE_NONE     0x80
#define BRAKE_LIGHT    0x81
#define BRAKE_MEDIUM   0x82
#define BRAKE_HARD     0x83
#define BRAKE_VHARD    0x84

#define RULE_TOT       60

unsigned char code rules[RULE_TOT]={    // fuzzy system rules
// if...      and...      then...
  VEL_VSLOW, DIST_VCLOSE, BRAKE_LIGHT,
  VEL_VSLOW, DIST_CLOSE,  BRAKE_NONE,
  VEL_VSLOW, DIST_NEAR,   BRAKE_NONE,
  VEL_VSLOW, DIST_FAR,    BRAKE_NONE,
  VEL_SLOW,  DIST_VCLOSE, BRAKE_MEDIUM,
  VEL_SLOW,  DIST_CLOSE,  BRAKE_LIGHT,
  VEL_SLOW,  DIST_NEAR,   BRAKE_NONE,
  VEL_SLOW,  DIST_FAR,    BRAKE_NONE,
  VEL_MEDIUM, DIST_VCLOSE, BRAKE_HARD,
  VEL_MEDIUM, DIST_CLOSE,  BRAKE_HARD,
  VEL_MEDIUM, DIST_NEAR,   BRAKE_MEDIUM,
  VEL_MEDIUM, DIST_FAR,    BRAKE_LIGHT,
  VEL_FAST,  DIST_VCLOSE, BRAKE_VHARD,
  VEL_FAST,  DIST_CLOSE,  BRAKE_VHARD,
  VEL_FAST,  DIST_NEAR,   BRAKE_HARD,
  VEL_FAST,  DIST_FAR,    BRAKE_MEDIUM,
  VEL_VFAST, DIST_VCLOSE, BRAKE_VHARD,
  VEL_VFAST, DIST_CLOSE,  BRAKE_VHARD,
  VEL_VFAST, DIST_NEAR,   BRAKE_HARD,
  VEL_VFAST, DIST_FAR,    BRAKE_MEDIUM
};

/*****
Function:      defuzzify
Description:   Computes the center of gravity of the fuzzy
               outputs and calls a function to convert the fuzzy
               COGs to outputs usable by the system.
*****/
```

```

Parameters:    none.
Returns:       nothing.
Side Effects:  The outputs[] array is cleared.
*****/
void defuzzify() {
    unsigned long numerator, denominator;
    unsigned char j;

    numerator=0;                // reset the summation values
    denominator=0;
    for (j=0; j<MF_TOT; j++) {  // compute the summation values
        numerator+=(outputs[j]*output_memf[j]);
        denominator+=outputs[j];
        outputs[j]=0;          // clear the output as its used
    }
    if (denominator) {          // make sure that a divide by
                                // 0 does not occur
        fuzzy_out=numerator/denominator; // finalize the COG
    } else {
        fuzzy_out=DEFAULT_VALUE; // no rules fired for this
                                // output - set a default value
    }
    normalize();                // change fuzzy output to
                                // normal output
}

/*****
Function:      fuzzy_engine
Description:   Executes the rules in the fuzzy rule base.
Parameters:    none.
Returns:       nothing.
Side Effects:  none.
*****/
unsigned char bdata clause_val; // fast access storage for the
                                // current clause
sbit operator = clause_val^3;   // this bit indicates fuzzy
                                // operator to use
sbit clause_type = clause_val^7; // this bit indicates if the
                                // clause is an antecedent
                                // or a consequence

void fuzzy_engine() {
    bit then;                    // set true when consequences
                                // are being analyzed

```

```
unsigned char    if_val,          // holds the mu value for
                                     // antecedents in the current rule
                                     clause,          // indicates the current
                                     // clause in the rule base
                                     mu,              // holds the mu value of the
                                     // current clause
                                     inp_num,         // indicates the input used by
                                     // the antecedent
                                     label;           // indicates the membership
                                                     // function used by the
                                                     // antecedent

then=0;          // assume that the first
                 // clause is an antecedent
if_val=MU_MAX;   // max out mu for the first rule
for (clause=0; clause<RULE_TOT; clause++) { // loop through all
                                     // the rules
    clause_val=rules[clause];        // reads the current clause
                                     // into bdata
    if (!clause_type) {              // if the current clause is an
                                     // antecedent...
        if (then) {                  // if a then part was being run...
            then=0;                  // change back to if
            if_val=MU_MAX;           // and reset mu for the rule
        }
        inp_num=clause_val & IO_NUM; // get the referenced input number
        label=(clause_val & LABEL_NUM) / 16; // get the referenced
                                     // membership function

        mu=input_memf[inp_num][label][input[inp_num]]; // get value
                                                         // of mu for this clause

        if (!mu) {                      // this rule will not fire
            do {                          // skip the antecedents
                clause++;
            } while (clause<RULE_TOT && !(rules[clause]&0x80));
                                     // skip the consequences
            while (clause+1<RULE_TOT && (rules[clause+1]&0x80)) {
                clause++;
            }
            if_val=MU_MAX;                // set up for the next rule
        } else {
            if (mu < if_val) {            // implement the MIN function
                if_val=mu;
            }
        }
    }
}
```

```
    }
  }
} else {                                // the current clause is a
                                        // consequence

  then=1;                               // ensure that the engine
                                        // knows it is running
                                        // consequences

  // if the current rule's mu is higher than the referenced
  // output value then store if_val as the new fuzzy output
  // value
  if (outputs[clause_val & 0x07] < if_val) {
    outputs[clause_val & 0x07]=if_val;
  }
}
}
defuzzify();                            // compute the fuzzy outputs
                                        // using the COG method and
                                        // defuzzify the outputs
}
```

Conclusion

Fuzzy logic is a new way of looking at old problems. It will not be the solution to the world's problems, but it will help you solve many problems in a more efficient manner than you could before. You must keep in mind that you do not need any fancy tools to develop a fuzzy logic application. In fact, you now have all the tools you need. Once you get a few fuzzy logic projects under your belt, you may wish to invest in some of the tools that are out there to help you design membership functions and rule bases. Some of these tools will also help you simulate and test your fuzzy logic system on a PC. These tools are fine to use once you have some experience with fuzzy logic and are sure that it will apply to your systems, but don't get suckered into buying an expensive package to generate fuzzy logic applications for you until you have done a few applications on your own.

- Conclusion

This book has presented many concepts for you to use in your 8051 projects. Hopefully, you have gained a fair amount of knowledge and expertise with the 8051 from reading and using this book. If you have not already purchased a C compiler for the 8051 I urge you to do so. Using C with your 8051 projects will make your life and the lives of the people who must maintain your projects much easier.

I have covered many topics in this book from optimizing your C and assembly to networking the 8051 to using fuzzy logic. It is my hope that you will be able to use the information presented here to improve the products you develop.