Contents

1 Interrupts ................................................................. 1
  1.1 Interrupt Initialization ........................................ 2
  1.2 Interrupt Triggers/Flags .................................... 2
  1.3 Interrupt Vectors ............................................... 2
  1.4 Interrupt Density ............................................. 3
  1.5 What can go wrong? ............................................ 3

2 Examples ................................................................. 4
  2.1 Serial Port Operation with Interrupts ..................... 4
  2.2 Timer Interrupt ................................................ 5

1 Interrupts

Typically a microcontroller will be used to monitor and act on several inputs and outputs, seemingly at the same time. Up until now we have polled the interrupt flags, i.e. waited in the main program for an event to occur. We cannot obviously do this efficiently for many event, and it restricts us from doing anything useful in the main program while we are polling (i.e. the main program is blocked on IO wait).

Interrupts are used in many computers and microcontrollers (even your home PC) to allow efficient non-blocking I/O. Typically an interrupt waits on some given interrupt flags (or logical combination), indicating that some I/O even has taken place. It then vectors

There are 6 (7 if you consider a system reset event) possible interrupts on the 8051 (and their corresponding IE flags):

1. EX0 - External Interrupt 0
2. ET0 - Timer 0 Overflow
3. EX1 - External Interrupt 1
4. ET1 - Timer 1 Overflow
5. ES - Serial Transmit/Receive Interrupt
6. ET2 - Timer 2 Overflow
(see Table 6-1 in the book for a great reference). Note that this is also the \textit{polling order} of the interrupt flags, i.e. if both Timer 0 and Timer 1 have overflowed, and both interrupts are the same priority, Timer 0 will be serviced first.

1.1 Interrupt Initialization

The register IE (interrupt enable) contains flag bits to enable/disable each interrupt (as shown above) and has a global interrupt enable/disable flag (EA). During the initialization of our program we can initialize the entire IE byte with a MOV, for example to enable only timer 1 interrupts we can use

\begin{verbatim}
MOV  IE, #10001000B
\end{verbatim}

However after initialization we should only use the individual flags (so as not to modify any of the other interrupt enable/disable flags). For example to enable timer 1 interrupts

\begin{verbatim}
SETB  ET1
SETB  EA
\end{verbatim}

1.2 Interrupt Triggers/Flags

You are probably recalling the first time you heard the word interrupt in this course, and it was probably when you were told about the “receive interrupt” RI and “transmit interrupt” TI. Now we know what interrupts are what is the relation? In the case where ES is set (serial interrupts are enabled) either of these flags (i.e. TI OR RI) trigger the serial interrupt, and hence the execution of the serial interrupt ISR. Similarly TF0, TF1, TF2 trigger the timer overflow interrupts, and IE0, IE1 in TCON trigger external interrupts.

It is important to note that unlike when we manually polled these flags from the main program, when an interrupt is enabled the associated flags are cleared as soon as the microcontroller “vectors” to the ISR (with the important exception of RI and TI). Hence we do not need to (and shouldn’t) manually clear the flags.

1.3 Interrupt Vectors

Each interrupt has an associated \textit{Interrupt Vector}, a code address that the 8051 “vectors” to on that interrupt. The code located at the interrupt vector that handles the interrupt is referred to as an \textit{Interrupt Service Routine} (ISR). On the 8051 an ISR looks very much like the typical subroutine except for two important differences:

- an ISR \textit{must} start at a fixed address (the interrupt vector) in code memory.

- an ISR returns with a RETI instruction, \textit{not} a RET instruction. The RETI instruction restores the PC to its location in the main program before the interrupt was serviced.

It is important to note that the space between interrupt vectors is very small (8 bytes to be exact). If an ISR is bigger it is necessary to jump to another area of code memory (practically always an LJMP). Hence typically the interrupt vectors will resemble a jump table.
1.4 Interrupt Density

The interrupt density gives a measure of how pervasive the ISRs are in a program, and is defined as

\[
\text{INTERRUPT DENSITY} = \sum \frac{\text{TIME FOR INTERRUPT}}{\text{TIME BETWEEN INTERRUPTS}}.
\]

1.5 What can go wrong?

You may be thinking that interrupts make 8051 programming much simpler, and indeed this is the case once you have mastered them. In reality they are the primary cause for problems in assignments, and are extremely difficult to debug. As in all cases, its much easier to use preventative measure to avoid the problem than debug the problem, so here are some common problems you should avoid!

Register Clobbering An ISR doesn’t push/pop a register it uses (directly or indirectly!) on to the stack, on return to the main program weird things happen. In general it is good practice to always push the PSW and accumulator.

\text{RETI} \neq \text{RET} a minor typo results in hours of misery, don’t end your ISR with a RET!

Very Large Interrupts An ISR should always be “short”. If you are writing substantial code in an ISR, you are doing something wrong.

Everything is an ISR! Don’t get too interrupt crazy, there are many cases where an ISR is not the best way to do something (or indeed the wrong way).

Interrupt Priorities Don’t use interrupt priorities! There is much more potential to do something wrong than any benefit, and in general there is no benefit in the course assignments.

Large ISR in Interrupt Vector this might overwrite the address of another interrupt vector, remember if your ISR is larger than 8 bytes (or when unsure) LJMP to another location.

Critical Sections If a section of the main program is time sensitive you must make it a critical section by surrounding it with a \text{CLR EA} and \text{SETB EA} to stop an ISR taking over in the section.
2 Examples

2.1 Serial Port Operation with Interrupts

; SERIAL INTERRUPT TO ECHO OUTPUT

CSEG AT 4023h

SER_VEC:    LJMP SER_INT

CSEG AT 5000h

START:     MOV SCON, #50h ; serial port 8 bit mode, enable REC, clear TI
           MOV IE, #10010000b ; turn on serial interrupts
LOOP:      SJMP LOOP ; loop forever (let the interrupts take over)

SER_INT:
           PUSH PSW ; save PSW and ACC
           PUSH ACC
           JB RI , RECEIVED ; if this interrupt was triggered by receiving a character, go to RECEIVED
           SENT: ; otherwise, it was triggered by transmitting, go to SENT
           CLR TI ; after a transmit, there’s nothing to do, so just clear the interrupt flag
           JMP SER_DONE ; and return
RECEIVED:  MOV A, SBUF ; move the character out of and then back into the serial buffer to echo
           MOV SBUF, A
           CLR RI ; clear the receive interrupt flag so we don’t loop forever
           SETB TI ; trigger a transmit interrupt so that the contents of SBUF will be echoed back to the serial port

SER_DONE:  POP ACC ; restore ACC and PSW
           POP PSW
           RETI

END

Note that because the serial interrupt can be triggered by RI or TI, we must check both of these flags in the serial ISR, and also clear the flags.
2.2 Timer Interrupt

```
ORG 0              ; reset entry point
LJMP MAIN          ; jump above interrupt vectors
ORG 000BH          ; Timer 0 interrupt vector

TIMER: CPL P1.0     ; toggle port bit
RETI               ; return from interrupt

ORG 0030H          ; Main program entry point
MAIN: MOV TMOD, #02H ; timer 0, mode 2
MOV TH0, #-50      ; 50 us delay
SETB TR0           ; start timer
MOV IE, #82H       ; enable timer 0 interrupt
SJMP $             ; do nothing
```

Compare this program with the ones we used before to generate a square wave! Even though our main program is doing nothing after interrupt initialization, the program is much smaller and simpler.