

Tutorial 2

Basic M16C Programming

1 Introduction

The goal for Tutorial 2 is to take a more in-depth look at the sample program from the previous tutorial, and make use of it as a foundation for writing your own interrupt driven programs. In this tutorial, we will cover two essential M16C programming topics:

- Using programmed I/O on the M16C; and
- Writing an Interrupt Service Routine (ISR)

After completing the tutorial, you should be able to:

- Configure I/O ports for input or output
- Read and write from I/O ports
- Write your own ISR
- Hook your ISR into the M16C Interrupt Vector table

Don't forget you will need the `m16c_tut2.zip` archive, which you can download off the ELEC3730 homepage next to the link were you downloaded this tutorial pdf from.

2 The Sample Program in Detail

The “sample2” program (S2) from the last tutorial is simple C program that cycles LED2 on the M16C development board through the digits 0-9 at the rate of one digit per second. In this tutorial, we want to take a closer look at the code for S2 and develop an understanding of some common M16C programming techniques.

The S2 program can be broken down into two parts:

- The main() function

main() is the entry-point of the program (all C programs have a main() function, where your program begins executing). In the case of S2, the main() function is responsible for initializing I/O ports p0 and p1, used for displaying digits on LED2, and configuring an M16C timer, timer A0, to generate an interrupt every 100ms.

- The TimerA0int() function

TimerA0int() is an interrupt service routine (ISR); a function that is called automatically by the M16C in response to an asynchronous event. In the case of S2, TimerA0int() is called periodically, every 100ms, in response to interrupts generated by timer A0. The TimerA0int() function is responsible for updating LED2.

We will now examine each of these functions in detail.

1 The main() Function

```
void main(void)
{
    initport();
    initTimerA0();
    setTimerA0int();
    startTimerA0();
    while(1); // endless loop
}
```

The first statement in main() calls the initport() function, responsible for initialising I/O ports on the M16C. The code for initport() is covered in later.

The next three statements initTimerA0(), setTimerA0int(), and startTimerA0() are responsible for initialising timer A0, setting the timer A0 interrupt priority, and starting the timer respectively. The procedure for configuring and starting timers is detailed later.

The final statement in main() is an infinite loop. An infinite loop is a technique we use to halt¹ a program when there is no more work to be done. All further processing, i.e. updating the digit displayed on LED2, is done in the TimerA0int() interrupt

¹ We use the term “halt” here, though technically this is not the case. While execution does not proceed beyond the infinite loop statement, the microprocessor has not truly halted; rather, it is busy-waiting, literally caught in the loop until the program is stopped.

service routine.

2

3 Programmed I/O on the M16C

The `initport()` function is our first encounter with programmed I/O on the M16C. The two I/O ports we are using in this function are **p0** and **p1**. In this example, **p1** is responsible for selecting LED1 or LED2, and **p0** is responsible for enabling/disabling the individual segments of the selected LED. The code for the `initport()` function is listed below.

```
void initport(void)
{
    pd0 = 0xFF;    // output mode
    pd1 = 0xFF;    // output mode
    pu00 = 0; // no pull up for P0_0 to P0_3
    pu01 = 0; // no pull up for P0_4 to P0_7
    pu02 = 0; // no pull up for P1_0 to P1_3
    p1 = 0xFD;    // enable LED2 and disable LED1
    p0 = 0xFF;    // turn off all segments
}
```

Many I/O ports on the M16C are bi-directional; that is, they may be configured as either input or output ports. In our case, we wish to *write* to **p0** and **p1**, so we must first configure them as *output* ports. This is accomplished by writing the value 0xFFh to the direction registers **pd0** and **pd1**. Conversely, had we wanted input ports, we'd have written the value 0x00h instead.

Keep in mind that it is possible to configure individual bits of an I/O port to be either input or output. As an example, suppose we want to configure **p0** as a split port such that the upper six bits are inputs and the lower two bits are outputs. Writing the value 0x03h (0000011b) to **pd0** would achieve the desired configuration.

Finally, two important properties of M16C I/O ports:

- Writing data to an input port has no effect; and
- Reading from an output port returns the last value written to the port.

4 Using Timers

The next useful programming technique we can learn from our example program is how to configure and use the M16C timers. Timers on the M16C are very flexible devices, capable of operating in one of the four following modes:

- Timer mode

- Event counter mode
- One-shot timer mode; or
- PWM (pulse-width modulation) mode

In this tutorial, we will learn how to configure a timer in Timer mode, and use it as a periodic interrupt generator (clock) to signal the microprocessor that a 100ms interval has passed. Detailed information on programming timers in the event counter, one-shot, or PWM modes can be found in the M16C Users Manual.

1. 1.Initialising the Timer

The `initTimerA0()` function is responsible for initialising Timer TA0.

```
void initTimerA0(void)
{
    ta0mr = 0x80;
    ta0 = 0xC350;
}
```

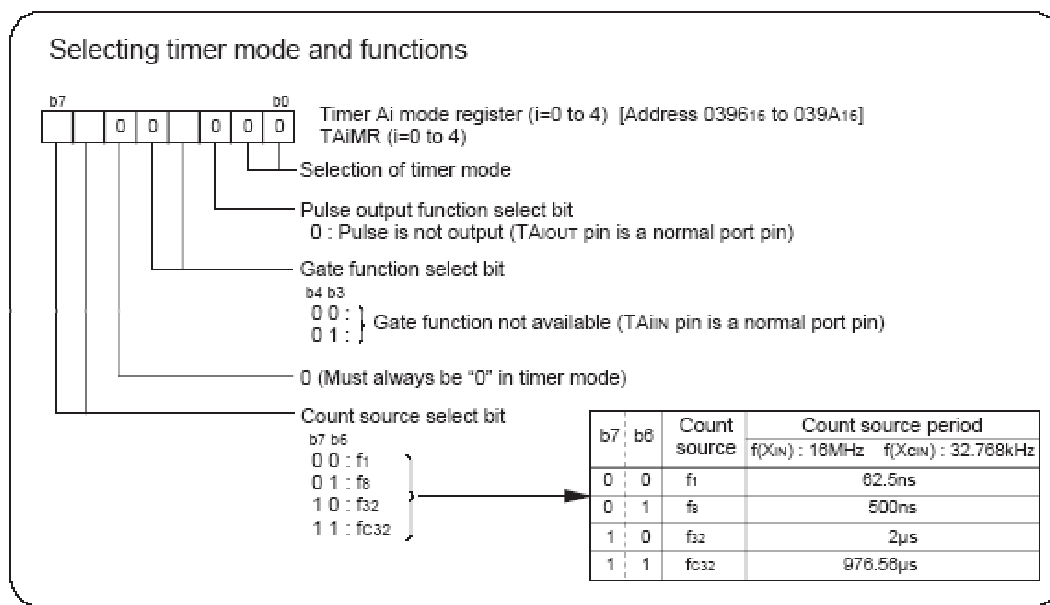


Figure - This diagram, taken from the M16C Users Manual, details the function of each bit-field within the mode register for a timer operating in 'timer' mode.

The first statement in the code snippet above configures timer A0 in Timer mode. This is accomplished by writing the value 0x80h to the mode register of timer A0. As you can see from , TA0 will be configured as follows: Select f₃₂ count source; gate function TA_{0in} disabled; no pulse output to TA_{0out}.

The field you should pay most attention to is the *count source frequency selection* field, bits 7-6 of TA0MR. Setting TA0MR to 0x80h selects count source f₃₂:

f₃₂ = system clock speed / 32

$$= 16\text{MHz} / 32$$

$$= 500\text{kHz}$$

Therefore, the value stored in the TA0 count register will be decremented every 2 μ s.

The next step is to program the TA0 reload register. The reload register contains the value the count register will be reset to each time the timer expires. Our example program initialises the reload register with the value 0xC350h, or 50000 decimal, for an interrupt period of 100ms:

$$T_{\text{TA0}} = 50000 \times 2\mu\text{s} = 50000 \times 2 \times 10^{-6}\text{s} = 0.1\text{s} = 100\text{ms}$$

2. Configuring the Interrupt Control Register

The setTimerA0int() function is responsible for programming the TA0 interrupt control register, TA0IC.

```
void setTimerA0int(void)
{
    ta0ic = 0x01;    // ---- XXXX
                    // ||||
                    // |||+-- Interupt priority level select bit
                    // ||+--- Interupt priority level select bit
                    // |+---- Interupt priority level select bit
                    // | 000: Level 0 (interrupt disabled)
                    // | 001: Level 1
                    // | 010: Level 2
                    // | 011: Level 3
                    // | 100: Level 4
                    // | 101: Level 5
                    // | 110: Level 6
                    // | 111: Level 7
                    // +----- Interupt request bit
                    // 0: Interrupt not requested
                    // 1: Interrupt requested
}
```

In this example, the TA0 interrupt priority has been set to level 1 (lowest priority).

Note that writing the interrupt control register will not overwrite or change the status of the interrupt request bit. The microprocessor internally sets, or resets, this bit to indicate that an interrupt is pending, or not; it is not possible to change the interrupt request bit from a program. Bits 7-3 of the byte written to the interrupt control register have no effect.

3. Starting the Timer

Finally, the timer must be started.

```
void startTimerA0(void)
{
    tabsr |= 0x01;    // 1: start timer A0 (count flag)
}
```

Timer TA0 is started by setting bit 0 of the Timer A/B Start Register (TABSR). Note that we used a logical OR to set bit 0 of TABSR so as not to interfere with the start flags of the other timers controlled by TABSR. Keep this in mind: simply writing the value 0x01 to TABSR may inadvertently disable other timers.

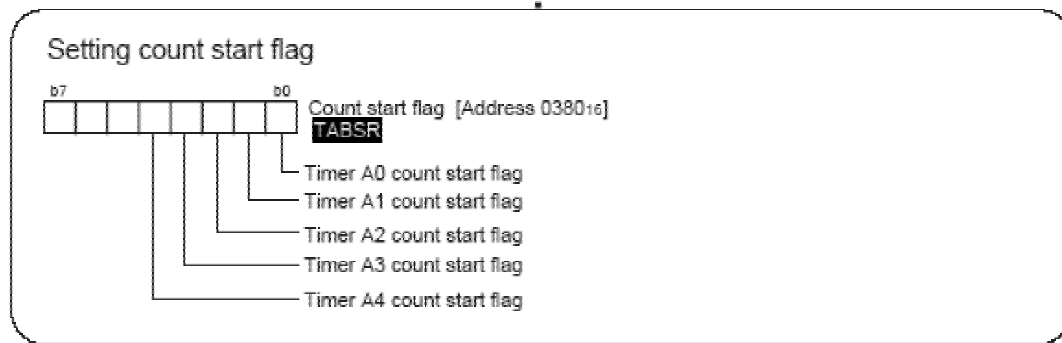


Figure - Setting the count start flag (taken from M16C Users Manual)

Alternatively, you can use the following, safer, method by accessing the TA0 start flag directly:

```
void startTimerA0(void)
{
    ta0s = 1;    // 1: start timer A0 (count flag)
}
```

5 The Interrupt Service Routine (ISR)

Interrupt service routines, or interrupt handlers, are functions specifically written to deal with asynchronous events generated by a peripheral device. A peripheral device generates an interrupt as a means of signaling the microprocessor, informing it that an event or condition has occurred that warrants attention.

In the example program, the peripheral device is timer A0 and the interrupt condition corresponds to expiry of the timer. This interrupt condition informs the microprocessor that one time period (100ms) has elapsed.

The code for TimerA0int() is listed below. We will take a look at some important features of this function in the following sections.

```
#pragma INTERRUPT TimerA0int
void TimerA0int(void)           // interrupt function
{
    static unsigned char one_hundred_msec_count = 0;
    static unsigned char digit = 0;

    _asm("fset I");           // enable interrupts
```

```

    one_hundred_msec_count++;
    if (one_hundred_msec_count == 10) // 1 sec has elapsed if true
    {
        one_hundred_msec_count = 0;
        digit++;
        if (digit == 10)
            digit = 0;
        p0 = leddigit[digit]; // display digits '0' - '9' in LED2
    }
}

```

4. **#pragma INTERRUPT**

The `#pragma INTERRUPT TimerA0int` statement informs the compiler that the `TimerA0int()` function is to be used as an interrupt service routine.

- A function registered as ISR cannot take arguments, nor can it return a value.

This makes sense if you think about how an interrupt service routine will be executed. Remember, an ISR is executed automatically by the microprocessor in response to an asynchronous event; it is **not** called from another function in your program. In short, an ISR *has no calling function*; therefore, it cannot receive arguments or return value to its caller.

Among other things, the `#pragma INTERRUPT` preprocessor directive will cause the compiler to issue a warning if you try to register a function with a signature that is not `void myFunctionName(void)`.

1. **The TimerA0int() function: Updating LED2**

The `TimerA0int()` function has two integer variables: `one_hundred_msec_count` to count the number of 100ms intervals, and `digit` to index the `digit[]` array.

The `one_hundred_msec_count` variable is incremented on each invocation of `TimerA0int()`. When `one_hundred_msec_count` reaches 10, i.e. one second has elapsed, the counter is reset, the array index `digit` is incremented, and LED2 is updated by writing data to port **p0**. The program cycles through the digits 0-9 by resetting the array index to zero whenever it exceeds 9.

2. **Inserting an ISR into the Interrupt Vector Table**

The final step in writing an interrupt service routine is to add your ISR to the interrupt vector table. The interrupt vector is a look-up table that the microprocessor uses to locate interrupt service routines. Each entry in the vector table is a pointer to a location in memory: the starting address of an interrupt service routine. The interrupt vector is structured such that the ISR for `int0` is pointed to by table entry 0, the ISR for `int1` is pointed to by entry 1, and so forth.

The interrupt vector is located in the file **sect30.inc**. The code listing below is a small snippet of the vector table from sect30.inc of our example program:

```
.lword  dummy_int           ; uart2 transmit(for user)(vector 15)
.lword  dummy_int           ; uart2 receive(for user)(vector 16)
.lword  dummy_int           ; uart0 transmit(for user)(vector 17)
.lword  dummy_int           ; uart0 receive(for user)(vector 18)

;
.lword  0FF900H             ; uart1 transmit - Monitor V2
.lword  dummy_int           ; uart1 transmit(for user)(vector 19)
.lword  0FF900H             ; uart1 receive - Monitor V2
;
.lword  dummy_int           ; uart1 receive(for user)(vector 20)

;
.glb  _TimerA0int
.lword  _TimerA0int         ; timer A0(for user)(vector 21)
;
.lword  dummy_int           ; timer A0(for user)(vector 21)
```

The `dummy_int` ISR is a special interrupt service routine that performs no action. It serves a dual purpose: firstly, it behaves as a placeholder, preventing the interrupt vector table from becoming misaligned; and secondly, it ensures the processor behaves predictably in the event of an unanticipated interrupt.

There are three interrupt vector table entries of interest in this code snippet: one for the UART1 transmit interrupt (int19), one for the UART1 receive interrupt (int20), and one for the TimerA0int() function in our sample program (int21).

The UART1 RX/TX interrupts (int19, int20) point to the absolute memory address 0x0FF900h, informing the microprocessor that the ISR for handling UART1 RX/TX interrupts is located at address 0x0FF900h in memory. The UART1 interrupt handler is part of the M16C monitor program and allows the M16C to communicate with the PC over the serial connection.

The timer A0 interrupt, int21, points to the address of our ISR function, `TimerA0int()`. Unlike the UART1 interrupt handlers, we don't know ahead of time the absolute address of the `TimerA0int()` code in memory², so we must reference `TimerA0int()` by name instead. This is done in two steps:

- Use a `.glb` directive to declare a reference to an external symbol. The symbol name must be the name of your ISR prefixed with an underscore.

```
.glb  _TimerA0int
```

- Insert your ISR into the vector table

```
.lword  _TimerA0int         ; timer A0(for user)(vector 21)
```

3. Writing Interrupt Service Routines: Summary

² The memory address of the `TimerA0int()` function is determined by the object linker when the program is compiled.

- Writing an ISR is just like writing any other function. The only difference between an ISR and a regular function is that an ISR cannot take arguments, nor can it return a value.
- You must register your ISR with the `#pragma INTERRUPT` directive to let the compiler know that the function will be used as an ISR.
- You must insert your ISR into the interrupt vector table in the appropriate position. The interrupt vector table is contained in the file “sect30.inc”.

3 **sfr62.h**

The `sfr62.h` header file contains definitions for the SFR memory area on the M16C. This area of memory contains the M16C control registers, such as interrupt control registers, and control registers for peripheral devices such as I/O ports, A/D and D/A converters, timers, and serial I/O. Including this header file in your project allows programs to access the M16C I/O ports by name rather than directly addressing memory.

As an example, the two lines of code below perform the exact same function; in both cases, the value of register TA0 is set to 0xABCD.

```
// without sfr62.h
*(unsigned_far short*)(0x0386) = 0xABCD;

// with sfr62.h
ta0 = 0xABCD;
```

4 Exercises

1. By default, the code does not display ‘0’ as the first digit; it begins counting at ‘1’. Modify the code so that it begins counting from zero.
2. The example program cycles forward through the digits 0-9 at a rate of one digit per second. Modify the code so that it cycles through the digits backwards, rather than forwards.
3. Modify the code so that it cycles through the digits at double the speed (2 digits per second).
4. Modify the code so that it cycles through the digits at *exactly* three times the speed. (hint: you will have to reprogram the timer to achieve this).

Thanks to Jacob Hart for substantial input in writing this tutorial.