EGR222 Mechatronics   Lab #11 Supplement

Overview:

This document describes an alternate method of doing the software for EE222 Mechatronics Laboratory Exercise #11, Stepper Motor Control. The software approach in the basic exercise is to use timing "busy wait" loops to do the delays between each step in controlling a stepper motor. The problem with that approach is that it does not allow the microcontroller to do anything else; doing the timing waits consumes all of its computational resources. Furthermore, if some additional tasks were to be added, such as updating a display to show the speed, those tasks would cause the timing of pulses to the motor to change, resulting in unwanted irregularities in the motor movement.

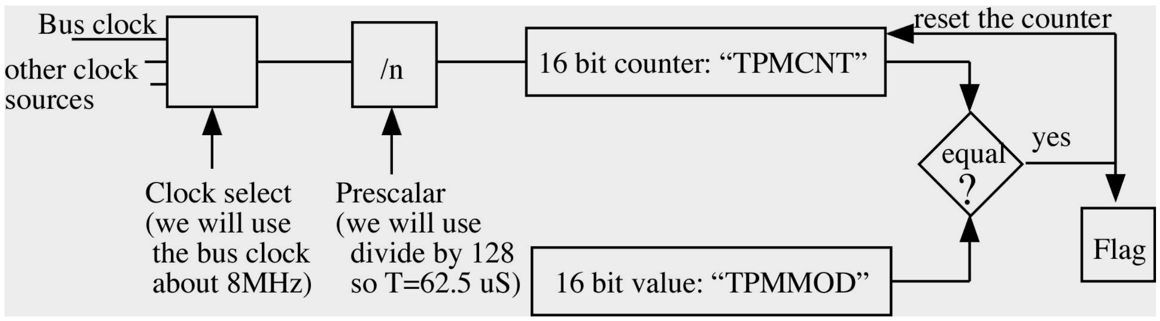
The microcontroller has facilities for doing tasks that depend on timing in a much more efficient manner. This supplement shows the use of a built-in timer, together with interrupt processing, to execute the same function as the basic lab exercise, but to do it better. Essentially, the program is decomposed into two independent tasks:

1. Read in the desired step period from the DIP switch. This task is done in the "background", in the sense that the main program loop will continually read the DIP switch and place the value in a global variable giving the step period.

2. Update to the outputs to the motor drive pins at regular intervals that depend on the specified period. This task is done by an "interrupt" driven by a hardware timer inside the microcontroller. It will interrupt regularly at times specified by the global variable mentioned above.

The timer:

Among the many "peripherals" inside the microcontroller, you have so far used only the "parallel ports" A and B. Another peripheral is called the "TPM Timer". The figure below illustrates how the timer works. The "bus clock" that regulates memory and peripheral interactions is the clock input we will select. It runs at about 8 MHz. (?) That clock will be slowed down to 16 KHz by dividing it by 128 in a "prescaler" unit. Every time the 16 KHz signal clocks up, the counter "TPMCNT" counts upward by 1. When it gets to the value TPMMOD, it resets, and raises a flag to indicate that the specified period (in 1/16 mSeconds) has elapsed. That sets a "flag" that can trigger an "interrupt".



TPM Timer Diagram

By making the TPMMOD register (at a particular memory location, hexadecimal 43 and 44, or 0x43, 0x44) different values, we can control how often the timer "times out" and sets the flag. For example, if TPMMOD was 15, then the counter would count 0-15, up one each 1/16 of a second. So, the "time out" would occur at intervals of 1 mSec. If the TPMMOD value was 31, then the timeout would occur every 2 mSec. So, to get an interval of a certain mumber of milliSeconds, we can just take the number of milliSeconds we want and multiply it by 16, and put that value into TPMMOD. The TPMCNT value is at addresses 0x41 and 0x42 if we wanted to look at it.

The timer is controlled by a "Status and Control Register" called TPMSC (at address hexadecimal 40, or 0x40). Different bits of this register control the clock selection, the prescaler, and whether we get an "interrupt." The flag is also in this register. In order to enable the timer and turn on the interrupt, we just need to put the bit pattern for what we want to do into this register. The executable C statement below does this:

```
TPMSC = 0b01001111;  /* Set the TPM timer for 16KHz, int enabled */
```

The first bit (set to 0) is the Timer flag; settingit to zero here doesn't really do anything. The next bit (set to 1) is the interrupt enable. The next bit (set to 0) controls "centered pulse width modulation" which we won't be doing. Finally, two bits set to 01 select the Bus clock, and the last three bits set to 111 specify divide by 128. (The details of this and other features of the HCS08QG8 device can be found in the device "data sheet", a 316 page document that can be downloaded from:
http://cache.nxp.com/files/microcontrollers/doc/data_sheet/MC9S08QG8.pdf?fpsp=1&WT_TYPE=Data%20Sheets&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation&fileExt=.pdf
The TPM timer material is in Chapter 16, with the status and control register TPMSC detailed on page 236.

If we were to now add a line of code:

```
TPMMOD=15;  /* set TPM timer for 1 mSec */
```

Now we would get time-out events at 1 mSec intervals. What will we do when that happens? Why, we will advance the stepper motor one step. But to do that, the "interrupt" we generate with the timer must be "caught" by an interrupt service routine.

The Interrupt Service Routine:

An "Interrupt Service Routine" (ISR) is a C "function" very much like "main ( ) or delay ( ) or any other function (or subroutine, if you want to call it that) you might choose to write. The difference is that it is not "called" by some other function. It is "called" by the hardware, in our case by the time out flag being set. The main program is "interrupted." The processor stops what it is doing, saves information about what it was doing so it can go back and resume that later, then it calls the ISR. So, an ISR can be called from anyplace; you don't have control over that (unless you disable interrupts, preventing the ISR from being called). Think of it as saying, "Come from wherever you are and do this." After the job is done, the processor uses the information stored to restore its status to whjat it was prior to the interrupt, and the previous task is resumed.

To write an ISR, you write a function, but use the key word "interrupt" and the interrupt number to indicate which interrupt the ISR responds to. In the case of the TPM timer, it is interrupt number 7. The ISR does not take an argument (data passed into it from the calling function) because there is no calling function. Similarly, it returns nothing because there is noplace to return it to. Hence, the "void" for both return value and calling argument.

```
Void interrupt 7 tpm_isr (void) {
    /* body of the ISR */
}
```

Inside the ISR, we need to first turn off the flag. If we don't, as soon as we return the interrupt will strike again becauser the flag is still set. For each of the many kinds of interrupts, the device manual describes how to resetr the flag. In this case, we reset it by first reading the status and control register TPMSC, then writing a "0" into the timeout flag. The variable "dummy" is a character (byte) variable defined for the ISR.

```
    char dummy;
    dummy = TPMSC;
    TPMSC_TOF=0;  /* reset the time out flag */
```

After turning off the flag, the next step is simply to push the data for the next step out to the stepper motor.

```
    PTAD=stepData[stepOut];
    stepOut++;
    if(stepOut>3)stepOut=0;
```

Somewhere earlier, our global variables for the sequence of steps needs to be defined, as well as a variable used to keep track of which one we are on:

```
static char stepData[4]={0x05,0x06,0x0a,0x09};
char stepOut=0;
```

Finally, the ISR needs to set the TPMMOD value for how long the next interval will be. Here that will be the period in Milliseconds that was read from Port B in the main routine:

```
    TPMMOD=periodMsec;
```

The main program:

The main program now will have different responsibilities. It does not need to advance the stepper motor; that is handled by the ISR. The main program simply needs to initialize the timer, then continuously read Port B to get the value for the stepper timing. In orrder to get timing in miliseconds that will be relatively slow, we will take the Port B value to be in units of 4 milliseconds, so a value of "1" would be 4 mSec, "2" would be 8 mSec, and 255 would be 1020 mSec (about a second), which is as slow as we will go. Thus, for the value 1 we would set the TPMMOD value to 4 * 16 since "1" indicates 4mSec and the counts occur at 16 KHz. So, we need to multiply the Port B value by 64. We can do that by shifting the bit pattern left by 6 bits. So this is what we need to do inside the main loop:

```
for(;;) {
    __RESET_WATCHDOG(); /* feeds the dog */
    /* read in period in msec from Port B */
    periodMsec=PTBD<<6;
} /* loop forever */
```

Earlier in main we need to initialize not only the timer but also the parallel ports, just as ion the busy-wait timer approach. We also need to provide the variable "periodMsec. Ultimately, here's what our whole program main.c looks like:

```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */

/* Global Variables */
short periodMsec=0;
static char stepData[4]={0x05,0x06,0x0a,0x09};
char stepOut=0;

void main(void) {
  EnableInterrupts;
  /* include your code here */
  TPMSC=0b01001111; /* turn on timer interrupt, bus clk / 128 */
  TPMMOD=65535;     /* initially as slow as possible */
  PTBPE=0b11111111; /* enable pull-ups on port B */
  PTADD=0x0f;       /* turn on 4 bits of port A to be outputs */
  PTADS=0x0f;       /* high drive strength */

  for(;;) {
     __RESET_WATCHDOG();    /* feeds the dog */
     /* read in period in msec from Port B */
     periodMsec=PTBD<<6;
  } /* loop forever */
  /* please make sure that you never leave main */
}
```

(continued on next page)

```
void interrupt 7 tpm_isr(void){
    char dummy;
    dummy=TPMSC;
    TPMSC_TOF=0; /* resets the time-out flag */
    PTAD=stepData[stepOut];
    stepOut++;
    if(stepOut>3)stepOut=0;
    TPMMOD=periodMsec;
}
```

Making it happen:

The hardware for this approach is exactly as for the basic lab 11 exercise: Attach your stepper motor circuit to the 4 bottom pins of Port A, and your DIP switch to the pins of Port B (each between the corresponding pin and ground). Download the timer program into your microcontroller, put it in your circuit, and watch it go. As before, very small values into Port B will likely cause your stepper to just vibrate because it can't go that fast. What isn't visible and obvious is the following advantages that this approach has:

1) The main routine doesn't have to be occupied continuously; it could be doing something else. Indeed, you could check port B in the ISR leaving the main loop to do nothing but feed the dog. The main loop could easily do something else useful, such as run anohter motor using different pins, or communicate with a PC, or, well, whatever you want. The ISR takes very little time.

2) The timing of the stepper motor is now more predicatable. The time between steps ius determined by the settings of a timer, not based on execution of a lot of instructions in a "busy wait" loop. (I'm not sure the default Bus Clock is 8 MHz. If it is something else the calculation of the value put into TPMMOD would need to change correspondingly.)

Looking Forward:

The TPM timer actually has many more features not described here. It has two "Channels" that can be connected to output pins (PTA0 and PTB5). By setting the timer period (with TPMMOD) and by setting the count times at which the channel outputs change (the channel values TPMC0V and TPMC1V) these two pins can output waveforms that could be used to run a stepper motor or anything else that needs one or two timing pulses to operate, and it will do this without even needing an interrupt at all. This is one way that Lab 12 (control of a DC motor) could be done better!