EGR222 Mechatronics Lab #12 Supplement

Overview:

This document describes an alternate method of doing the software for EE222 Mechatronics Laboratory Exercise #12, Digital Motor Control, where the microcontroller is used with feedback to control the speed of the DC permanent magnet motor. The software approach in the basic exercise is to use a timing "busy wait" loop to determing the "high" time of a pulse modulated waveform that controls the motor speed. There are problems with that approach. Importantly, it does not allow the microcontroller to do anything else; doing the timing waits consumes all of its computational resources. If additional tasks were to be added, such as updating a display or communicating results, those tasks would cause the timing of pulses to the motor to change, resulting in irregularities in the motor movement. The algorithm also did not keep a consistent PWM frequency, since it only changed the "high" time of the motor waveform. So, as duty cycle changes, so does the frequency. If the duty cycle gets very large, then the interval between samples can get large, and control over the motor can be lost.

The microcontroller has facilities for doing tasks that depend on timing in a much more efficient manner. These were first described in the document "EGR222 Mechatronics Lab #11 Supplement" that described how a timer could be used to better control a stepper motor. This supplement shows the use of additional features of that same built-in timer to execute the same DC motor control function as the basic lab exercise, but to do it better. The program is decomposed into two independent tasks:

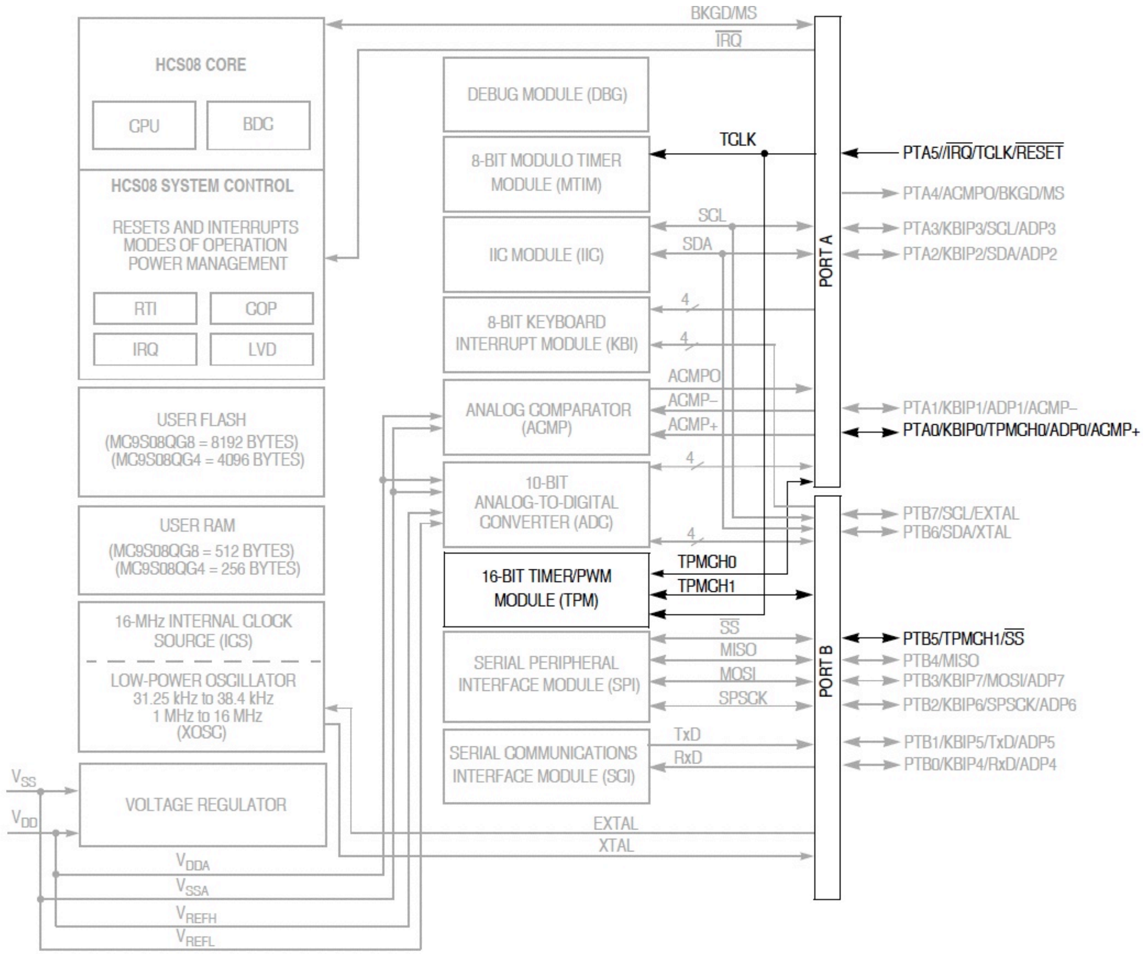
1. Run the stepper motor with a prescribed duty cycle. This is done entirely by the timer hardware without the program having to do anything, once it is configured and set to running.

2. Update the duty cycle at regular intervals (once per millisecond?) by sampling the motor speed, and comparing it to the desired speed indicated by the DIP switch connected to port B. This is essentially what the basic version of the lab exercise does, but now the timer does the actual motor control.

The timer:

The timer was described in the Lab #11 supplement with certain unused details omitted. It was used to trigger interrupts at regular intervals. It is assumed the reader is familiar with that exercise. Now, use of one of the timer's "channels" will be added. A "channel" allows an output signal to be sent out from the microcontroller that will automatically go from 1 to 0 whenever the timer count reaches 9and exceeeds) a given "channel value". So, if the counter is counting 0 to 999, and the channel value is 100, then the output of the channel will flip from 1 to 0 when the counter is 1/10 of the way through it's full range. That yields a 10% duty cycle. (The timer pins can also be used as inputs, to "capture" the time at which a pin changes value. We won't be doing that.)

An overview of the TPM timer in the context of the overall microcontroller is seen in the figure below, with the timer and its signals highlighted.



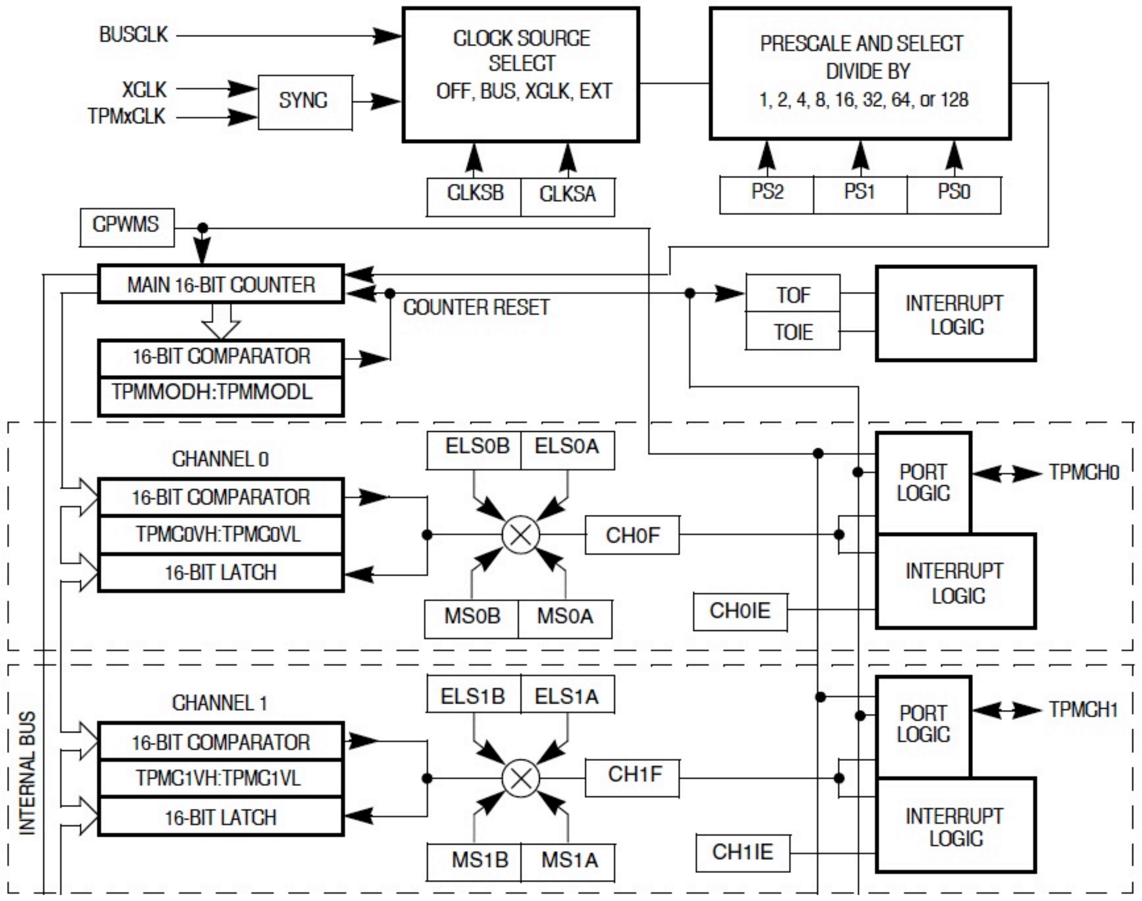
TPM Timer in the context of the HCS08QG8 Microcontroller (from HCS08QG8 manual)

This figure illustrates that the two TPM timer channels, Channels 0 and 1, come out on pins PTA0 and PTB5. Since we want to use all of the pins of Port B for the DIP switch inputs, we will have to use Channel 0. By default, that pin is connected to a potentiometer on the Demo board, so if you are using the microcontroller on the demo board you need to pull out the “shorting strap” jumper “RV1” that connects PTA0 to the potentiometer. (Put it on one of the pins of the connector but not the other so that it won’t complete the circuit, but won’t get lost either.)

You can also see in this figure that the TPM timer permits the use of an external clock “TCLK” that can be routed into the microcontroller on pin PTA5. That pin is normally the “Reset” pushbutton. So, if you want a “Reset,” you can’t have an external clock for the TPM timer.

This kind of situation is typical for microcontrollers. The number of pins is limited, since the price depends more on the package and number of pins than the silicon content inside. So, the microcontrollers give you as much functionality as they can, but with the limited number of pins you can’t use it all at the same time; each pin has several possible functions. If you want more functionality than the number of pins and peripherals provided can support, then shift to a larger and more capable microcontroller. There are literally hundreds or thousands to choose from. There are probably 100 or more HCS08 variants, and that’s just one family of microcontrollers.

The figure below shows a diagram of the TPM timer.



TPM timer functional diagram (from HCS08QG8 Manual)

The Lab #11 supplement illustrated the top part of this figure (in simplified form) but omitted the bottom part showing the channels. Briefly, the selected input clock (we will use the Bus clock) is divided by some power of 2 (we'll use 8) to determine the rate at which the TPM counter, TPMCNT, counts. (The TPM counter has a "top half" of 8 bits called TPMCNTH and a "bottom half" of 8 bits called TPMCNTL. In C we get them both together at the same time using "TPMCNT." The same is true for the other 16 bit registers for the TPM timer, as well as other 16 bit registers like ADCR for ADC results, when used to get the full 16 bits.) With a bus clock of 8 MHz and a prescaler value of binary 011 (for PS2, PS1, and PS0) for divide by 8, we get a count every microsecond.

The TPM timer is controlled by a "Control and Status" register called TPMSC. CLKSB and CLKSA are bits 4 and 3 of TPMSC. PS2, PS1, and PS0 are bits 2, 1, and 0 of TPMSC. We also want to set bit 6, "TOIE" which is the time out interrupt enable. Then we need to put the (decimal) value 999 into TPMMOD so that the timer will count 0 to 999. When it gets to 999, it will set a flag (bit 7 of TPMSC) and trigger the interrupt.

The lower part of the diagram shows the channels. The QG8 TPM module has only two channels, but we are going to use only one, channel 0 (that comes out on PTA0). Notice that the TPM count is passed to all of the channels, where it is compared to a "channel value". There are 5 bits that configure what the channel does. They are in a channel status and control register, TPMC0SC for channel 0. The control bits MS0B and MS0A need to be set to “10” for “edge aligned pulse width modulation”, which is what we want to do. Making ELS0B and ELS0A = 10 configures the output for “high-true pulses – clear output on compare”. That’s what we want: the pin will start out at 1 when the count is 0, then when the count gets up to the channel value the output will flip to 0. That makes the duty cycle proportional to te channel value. There are other lots of other configurations. For example, making ELS0B and ELS0A 01 instead causes the output to start at 0 and flip to 1 on compare instead. To configure the channel the way we want, we put the binary bit pattern 00101000 into TPMC0DS. Since we put 999 into the counter modulo register (the maximum count) putting 100 into the channel value will give a 10% duty cycle.

So, to configure the timer, we need the following statements:

```
TPMSC=0b01001011; /* turn on timer int, busclk/8 = 1MHz*/
TPMMOD=999;     /* Make the interrupt frequency 1KHz */
TPMC0SC=0b00011000; /* Channel 0 PWM, clear on compare */
TPMC0V=dutyCycle;       /* start out at 10% duty cycle */
```

We should do this before enabling interrupts. We don’t want the timer to start until after we have finished setting it up, because if interrupts are enabled it will immediately start after the first statement while the modulo value is still 0 (default out of reset) giving us an interrupt every microsecond. We don’t want that!

Once interrupts get turned on, this timer will start pumping out pulses to the motor at 1 KHz (if the bus clock is 8MHz). The duty cycle will initially be 10%. If nothing else happens, that will continue indefinitely whatever the main program does. (If the watchdog timer goes off, the processor will reset and we’ll automatically start back up at a 10% duty cycle.) We are not quite through with the timer, though. We enabled its interrupt, so we need to provide an interrupt service routine. That’s where we’ll do the rest of our business.

The interrupt service routine:

The interrupt service routine is similar to what was done in the lab 11 supplement, but now we will get an interrupt every 1 milisecond. What we will do, instead of advancing a stepper motor, is check motor speed and bump the duty cycle up or down a bit (by 1, for .1%) depending on whether the motor is going too fast or too slow. Just as with the stepper, the first thing we need to do is reset the interrupt flag so it won’t interrupt us again until the timer has completed another cycle. Getting data from the A/D converter works just like it did before: we put the channel we want (3 for PTA3) into ADCSC, then wait for the “COCO” flag to go to 1 indicating conversion complete. We compare the ADC result ADCR to the desired speed, PTBD, and increase or decereas the duty cycle accordingly. Then we put the modified duty cycle value into the channel value, and return from the interrupt to the main program. The program does not have to actually run the motor waveform; the timer channel does that for us. The main program doesn’t have to do anything! All it does is feed the dog. Or, anything else you might want the microcontroller to be doing in its spare time, like calculating prime numbers.

The complete program:

The program below includes a few features left out of the basic Lab 12 version. The A/D converter is “configured” to use a long sampling time, so that we perhaps smooth out some of the noise from the generator brushes. (Maybe this will allow us to use a amaller capacitor). Configuration options also include going to 10 bit mode for higher resolution. (If we did that, we’d need to multiply the DIP switch value by 4, using short variables.) Whether that would help give more stability I don’t know. It’s been left in 8 bit mode. Another change is to set APCTL3 to 1, which disconnects PTA3 from the parallel port, which by default is actually still connected.

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

short dutyCycle=100; /* initially use 10% (1000 would be 100%) */

void main(void) {
/* include your code here */
  TPMSC=0b01001011; /* turn on timer, interrupt, bus clk / 8 = 1MHz*/
  TPMMOD=999;     /* Make the interrupt frequency 1KHz */
  TPMC0SC=0b00011000; /* Channel 0 does PWM, clear output on compare */
  TPMC0V=dutyCycle;       /* start out at 10% duty cycle */
  PTBPE=0b11111111; /* enable pull-ups on port B (inputs)*/
  ADCCFG=0b00010000;/* 8 bit A/D operation with "long" sampling time */
  APCTL1_ADPC3=1;   /* port A pin 3 is an A/D input */
  PTADS_PTADS0=1;   /* high drive strength for Port A pin 0 output*/
  EnableInterrupts;

  for(;;) {
    __RESET_WATCHDOG(); /* feeds the dog */
    /* The main loop does nothing!  You can have it do other stuff */
  } /* loop forever */
  /* please make sure that you never leave main */
}

void interrupt 7 tpm_isr(void){
      char dummy;
      dummy=TPMSC;
      TPMSC_TOF=0; /* resets the tim-out flag */
      /* sample current speed using the A/D converter */
      ADCSC1=3; /* initiate collecting data for channel 3*/
      while(ADCSC1_COCO==0); /* wait for Conversion complete */
      /* Adjust the duty cycle up or down as needed */
      if(ADCR>PTBD){
            dutyCycle--; /* it's too fast; reduce duty Cycle */
            if(dutyCycle<0)dutyCycle=0;
            TPMC0V=dutyCycle;
      }
      else if(ADCR<PTBD){
            dutyCycle++; /* it's too slow; increase duty cycle */
            if(dutyCycle>1000)dutyCycle=1000;
            TPMC0V=dutyCycle;
      }
}
```

Lab #12 program using the TPM Timer

Status:

Currently this lab exercise supplement is in an “untested” status. The program does compile. It needs to be tested with a motor-generator circuit. The only difference between the circuit for this program and the normal circuit is that pin PTA0 instead of PTA2 needs to be used to run the motor, since it’s the one connected to the timer channel.