

ELE538 Microprocessor Systems

Lab 4: Motor Control

&

Using the Hardware Timer

Peter Hiscocks
Department of Electrical and Computer Engineering
Ryerson Polytechnic University
phiscock@ee.ryerson.ca
August 26, 2002

Contents

1	Objectives	2
2	Overview	2
3	Motor Control Routines	2
3.1	General Purpose Output Register	2
3.2	Controlling Individual Bits	3
3.3	Motor Routines	3
4	The Hardware Timer	4
4.1	The 68HC11 Timer Mechanism	4
4.2	The Timer Overflow Flag	5
4.3	Introduction to Interrupts	5
4.4	Setting up the Timer Overflow Interrupt	6
4.5	Debugging Interrupts	7
4.6	Timer Overflow Counter, Summary	9
4.7	Alarm Time	9
4.8	Example: A Timed Delay	10
4.9	A Potential Bug: Overflow of the Overflow Counter	10
5	The Assignment	11
5.1	Motor Control	11
5.2	Timer Overflow	12
5.3	Timer Alarms	12
6	References	12

List of Figures

1	Timer Overflow Routines (Part 1)	7
2	Timer Overflow Routines (Part 2)	8

1 Objectives

In this lab, we will develop

- Software routines to control the *eebot* motors.
- A software clock using the 68HC11 interrupt-driven *hardware timer overflow flag*
- A *timed alarm* mechanism that will be useful in controlling the *eebot* robot in various manoeuvres such as turns.

These routines will be used as subroutines in the *Robot Roaming* exercise of Lab 5.

2 Overview

Robot *roaming* behaviour can be obtained with a very simple set of rules. Initially, the robot drives in a straight line¹. If it doesn't encounter any obstacles, after a certain interval it stops, executes a turn, and then runs again in a straight line.

If the robot encounters an obstacle, it executes a *back-and-turn* manoeuvre. It drives backward for a fixed interval and then turns. Then it resumes driving forward in a straight line again.

All of these manoeuvres require timed delays, so that is our next challenge. First, we will develop basic routines to control the *eebot* motors. Then, we will study the 68HC11 hardware timer that will be used to create the various time delays.

3 Motor Control Routines

Output registers are *mapped* into the address space of the 68HC11. This means that the 8 individual bits of some byte in memory are connected to 8 output pins somewhere in the hardware. If the software bit corresponding to a particular line is set to a logic *zero*, then the voltage on that line will be zero volts (or close to it). If the software bit corresponding to that line is set to a logic *one*, then the voltage on that line will be (approximately) 5 volts.

In the MPP Board system, there are two output registers that are controlled in this fashion.

3.1 General Purpose Output Register

The first of these registers and the easiest one to understand is located at address \$1100. This is external to the 68HC11, so the Pink Book does not apply to this device and we can pick our own name for it. We'll refer to this register as GPOUT (General Purpose OUTput), so programs will need an equate statement to relate GPOUT to \$1100.

¹Well, not *quite* straight. Both motors are powered equally and should ideally rotate at the same rate, driving the robot in a straight line. In practice, one of the motors is a bit faster than the other and the path is slightly curved.

In the robot system, two bits of GPOUT control the direction of the two robot drive motors. Three more bits determine which of the 6 optical sensors is read on channel PE1 of the A/D converter. The remaining 3 bits are not used.

3.2 Controlling Individual Bits

When individual control lines are mapped into the bits of a memory byte, as in this case, we need routines that can set and clear each of the individual bits. When you write to a location such as GPOUT, you potentially change all 8 bits at once, and this is often undesirable.

Individual bits in a byte may be set or cleared using the *Logical Operation* instructions: AND, OR, EOR and COM (often known as NOT). The logical operation is performed on the output location (GPOUT in this case) and a logical *mask*. The mask is another 8 bit byte that determines which bits are set and which are cleared. Each of the two *eebot* drive motors is controlled by a *speed* bit and a *direction* bit, per the following table (see *The eebot Technical Description* document):

Motor	Function	Pin	Notes
Starboard	Speed	PA5	1 for on, 0 for off
	Direction	GPOUT1	1 for fwd, 0 for rev
Port	Speed	PA4	1 for on, 0 for off
	Direction	GPOUT0	1 for fwd, 0 for rev

Consequently, there are two independent bits in the PORTA register (\$1000) that control the speed of the two motors. There are also two independent bits in the GPOUT register (\$1100) that control the direction of the two motors.

3.3 Motor Routines

You will need 4 subroutines to control motor power, as follows:

Routine Name	Notes
STARON	Starboard motor ON
STAROFF	Starboard motor OFF
PORTON	Port motor ON
PORTOFF	Port motor Off

You will need 4 subroutines to control motor direction, as follows:

Routine Name	Notes
STARFWD	Starboard motor FORWARD
STARREV	Starboard motor REVERSE
PORTFWD	Port motor FORWARD
PORTREV	Port motor REVERSE

Each of these routines will be very short, since they just flip bits in the shadow registers and update one of the output registers PORTA or GPOUT. However, when changing bits in PORTA or GPOUT, use the AND and OR mask instructions to ensure that only the desired bits are affected.

Write and assemble these 8 subroutines as one program. Assemble and download the program into memory, as per usual. Use the Buffalo monitor CALL instruction to execute the various subroutines and drive the *eebot*

motors. For example, with CALLs to different subroutines, you should be able to start the Starboard motor in forward direction and then start the Port motor in forward direction.

To energise the eebot, you will need to enable the LOGIC power switch and the MOTOR switch on the rear panel of the robot.

CAUTION: Make sure the *eebot* wheels are lifted off the ground when you run these routines or the bot will drive itself off the bench and be damaged.

4 The Hardware Timer

The simplest method of creating a delay is the *software loop*, which we developed in previous exercises. The software loop works well in simple applications, but for this program we need something more sophisticated.

The software loop has a major disadvantage: it requires continuous servicing. The computer must sit in the software loop, updating the loop counter, so it is not available for any other tasks. It would be much more convenient for the hardware to maintain a clock that the program could periodically consult to determine elapsed time.

Then when we want computer program to do something for a period of time, the computer algorithm would be something like this:

```
Read the current time Tc
Set the alarm time Ta to Tc+Td,
  where Td is the desired delay
Repeat
  .
  .
  (do something useful)
  .
  .
Until Tc > Ta
```

Providing the program checks the current time often enough, this will suffice to produce the desired delay.

The vague quantity *often enough* implies that the computer program has to check the time frequently compared to the required resolution of the delay time. Consequently, the block labelled *do something useful* cannot occupy the computer program for any significant length of time. In the program we are building, the delays do not need to be precise – we are worried about delays in the order of seconds and the timer resolution is in the order of milliseconds – so this is not a difficult requirement to satisfy.

For the Robot Roaming program, we will take this *semi-automatic* approach: the clock will be updated automatically, and the software will check it manually.

(There are other possibilities. For example, it is possible to have the 68HC11 automatically call a certain routine after a specified delay. We'll tackle that later.)

4.1 The 68HC11 Timer Mechanism

The timer subsystem of the 68HC11 is quite powerful and consequently moderately complex. For our purposes in this exercise, we will restrict ourselves to the *16 bit free running counter* and the *timer overflow flag*.

The basis of the entire timer subsystem is a 16 bit counter called TCNT (memory location \$100E). TCNT is driven by the system clock, so the basic clock tick is 1 μ second. The counter *free-runs*, that is, there is no way to stop it or reset it. The location \$100E is *read-only*².

(Use the Buffalo monitor to examine location \$100E with the MD command. Because the counter is running continuously, it will be different every time you examine it.)

We would like to create delays in the order of a few seconds or so. Since TCNT is a 16 bit counter driven by a 1 μ second clock, it will overflow every $2^{16} \times (1 \times 10^{-6}) = 0.065$ seconds. This is too short an interval to be useful. We need an additional counter stage to divide this down even further to a useable time interval.

It would have been convenient if Motorola had provided an additional 8 bit hardware counter stage. That would have provided a clock with a resolution of 0.065 seconds and a maximum count of $2^8 \times 0.065 = 16.6$ seconds, which is more useful for this application.

They didn't provide the hardware, but they did provide a *hook* that allows us to do the additional counting in software. Their reasoning ran along these lines:

"We will provide an *overflow detector* that allows the user to maintain additional counter stages in software. This is acceptable because whatever routine does this will only have to operate every 0.065 seconds, and will therefore not cause much overhead. On the other hand, the previous stages must be implemented in hardware, because they have to count much more frequently."

4.2 The Timer Overflow Flag

The device that Motorola provided is the one bit *timer overflow flag*, known as TOF. This bit is the most significant bit of register TFLG2 (memory location \$1025). It is *set* every time the free running counter overflows³.

If you read location \$1025, you will find that the high bit is effectively always set because it gets set every 0.065 seconds. Taking the reciprocal, this is 15 times per second. So even if you cleared the flag (we'll deal with that next), you would be too slow to see it change.

We can use the timer overflow flag to maintain an 8 bit counter in software, because the timer overflow can be used to cause an *interrupt*, which will service the counter routine automatically.

4.3 Introduction to Interrupts

We will discuss interrupts in detail in the lecture portion of this course. In these notes, we'll simply provide a brief overview and a cookbook approach to using the timer overflow interrupt.

You may think of the interrupt mechanism as a *hardware triggered subroutine*. A subroutine is normally called with a JSR instruction. It may be called from any point in the program, and execution will return to the same point when the subroutine executes the RTS instruction.

When an interrupt is triggered, the hardware causes execution to switch to an *interrupt service routine* (ISR). The interrupt service routine does something useful. Then the last instruction in the ISR, which is always RTI (*Return From Interrupt*), returns execution to the point in the program where the interrupt occurred.

There are a number of interrupt systems on the 68HC11 microprocessor. Each system has three components:

- A *device flag* that is *set* on occurrence of the interrupt. (In our case, this is the TOF flag).
- An *interrupt enable/disable bit*, which determines whether an interrupt occurs when the device flag is set.

²It is possible to change the basic clock tick rate, but it turns out to be inconvenient and unhelpful to do so, so we'll stick to the default of 1 μ second.

³See page 10-12 of the Pink Book.

- An *interrupt vector*, which is the two-byte address of the interrupt subroutine.

When the timer interrupt is set up properly, each time a timer overflow occurs (every 0.065 seconds, or 15 times per second), the timer overflow interrupt service routine will be called. This will simply increment an 8 bit software register (some location in RAM), and then return. Once this is running, we can access the software counter as if it were an extension of TCNT, and use it for determining time delays.

One of the tasks of the interrupt service routine is to *clear the device flag*. If this is not done properly, the interrupt will recur immediately the interrupt service routine returns (ie, when the RTI instruction executes), effectively crashing the machine.

4.4 Setting up the Timer Overflow Interrupt

The interrupt mechanism is very powerful. It allows things to happen *in the background* of a computer program at the whim of asynchronous hardware events. The foreground computer program does not need to be involved in servicing the interrupt-driven routines.

Now we will provide instructions for setting up the timer overflow interrupt:

The interrupt service routine simply increments the overflow counter, clears the device flag and returns. The clearing of the device flag is counter-intuitive:

To clear the device flag, you must write a logic 1 to the flag. You would *think* that it would be correct to write a logic 0 to the flag to clear it, but not so with this device. Lesson: always read the manual.

So the TOF interrupt service routine is something like this:

```
TFLG2      EQU $1025    ; MSB is the timer overflow flag
TOFCOUNTER RMB 1       ; Overflow counter

TOF_ISR    INC TOFCOUNTER ; Increment the overflow count
           LDAA TFLG2
           ORAA #%10000000 ; Clear the TOF flag
           STAA TFLG2      ; (Pink Book, page 10-14)
           RTI
```

Now we need a routine to turn the TOF interrupt on. There are three stages to this: we need to

- Set up the interrupt vector that points to TOF_ISR. This, it turns out, consists of putting an instruction to jump to the TOF interrupt service routine at a magic location in the internal RAM of the microprocessor. The exact incantation is:

```
ORG $00D0
    JMP TOF_ISR
```

We'll explain exactly why this works at a later time.

- Set the TOF device interrupt flag, which is known as TOI. This has the effect of enabling the TOF interrupt. This bit is found in the MSBbit of the register TMSK2 (memory address \$1024)⁴.

⁴See the Pink Book, page 10-12.

- Enable the Global Interrupt Flag. The *interrupt mask* bit in the Condition Code Register enables and disables *all* interrupts, hence the term *Global Interrupt flag*. This is accomplished with the CLI instruction. Interrupts are disabled globally with the SEI instruction.

Putting this together, we have the routines for servicing the TOF interrupt, enabling the TOF interrupt, and disabling the TOF interrupt, as shown in figure 1 on page 7 and figure 2 on page 8.

```
*****
*           Timer Overflow Demonstration
*
* This program contains routines to demonstrate the timer overflow
* interrupt on the 68HC11.
* Instructions:
* Assemble this program.
* Load the .s19 file into memory
* Use the monitor MM command to zero the TOF_COUNTER.
* Use the Buffalo monitor GO 6010 command to start the TOF interrupt
* running. Then let the machine run for a few seconds
* so the TOF count accumulates to something.
* (It should be increasing about 15 counts per second.)
* Manually reset the HC11 and examine TOF_COUNTER. It should
* contain some non-zero count value.
* Run the program again and manually reset again. The TOF_COUNTER
* should have increased from the previous value.
* Peter Hiscocks
* 1 October 2001
*****
*           Register Definitions

TOF_JMP_VECTOR EQU $00D0 Pseudo-vector location in RAM
TFLG2 EQU $1025 Contains the TOF device flag at MSB
TMSK2 EQU $1024 Contains the TOF interrupt enable flag

ORG $6000 ; Where our TOF counter register lives
TOF_COUNTER RMB 1 One byte overflow counter
ORG $6010 EQU * Where the code starts

* The following is the main routine. Start here.
START JSR ENABLE_TOF ; Start the TOF interrupt
      CLI ; Enable global interrupts
LOOP  BRA LOOP ; and then loop until manual reset
      SWI ; Should never get here
```

Figure 1: Timer Overflow Routines (Part 1)

4.5 Debugging Interrupts

Interrupt routines can be *very* difficult to debug, and so some care is required in ensuring that they are set up properly.

The best way to do this is

- Construct the ISR as a subroutine (the last statement is RTS), drive it from a stub, and as much as possible verify that it works.

```

*****
*           Enable Timer Overflow
*
* This routine sets up the ISR pseudo-vector and enables
* the TOF interrupt mask bit.

ENABLE_TOF LDAA #$7E Setup the interrupt vector for timer overflow
           STAA TOF_JMP_VECTOR
           LDD #TOF_SERVICE
           STD TOF_JMP_VECTOR+1
* When enabling the Timer Overflow interrupt, it is prudent to clear
* the TOF flag so than an interrupt does not occur immediately, but
* rather on the next timer overflow.
           LDAA #%10000000 Clear the TOF flag
           STAA TFLG2      by writing to bit 7
           LDAA TMSK2      Turn timer overflow interrupt on
           ORAA #%10000000 by setting bit 7 in TMSK2
           STAA TMSK2
           RTS
*****
*           Timer Overflow Service Routine

* This routine is called on interrupt each time the free-running
* 16 bit counter overflows.
* Assuming that the timer prescaler bits PR0 and PR1 have not been
* changed, the basic rate of the 16 bit free running
* counter is 1 microsecond, so overflows occur about 1/15 second
* apart. The TOF_COUNTER may be used in time delays.

TOF_SERVICE INC TOF_COUNTER Increment the overflow counter
            LDAA #%10000000 Clear the TOF flag
            STAA TFLG2      by setting(!) bit 7
            RTI             Restore machine state, enable interrupts
*****
*           Disable the TOF interrupt
*
* The routine to disable the timer overflow interrupt is useful
* during debugging since the Buffalo monitor 'trace' function doesn't
* work otherwise.

TOF_INT_OFF LDAA TMSK2      Turn timer overflow interrupt off
            ANDA #%01111111 by clearing bit 7 in TMSK2
            STAA TMSK2
            RTS

```

Figure 2: Timer Overflow Routines (Part 2)

- As shown in figure 1 and figure 2, construct two other small subroutines, one that that turns the interrupt ON, and the other one that turns the interrupt OFF. The routine that enables the interrupt is also responsible for making sure that the vector is set up and any other initial conditions (such as setting up the hardware).
- After a careful check of the interrupt service routine code, replace the RTS instruction with RTI.
- Load the code into the microprocessor and use the monitor MM command to initialize the TOF_COUNTER to zero.

- Use the monitor GO instruction to run the main routine at address START. This calls the subroutine ENABLE_TOF that sets up the interrupt. The main routine then turns on global interrupts with the CLI instruction and enters an endless loop: LOOP BRA LOOP. At this point, the TOF interrupt should be running correctly and the TOF_COUNTER counter should be incrementing at a rate of 15 counts per second.
- Let the microprocessor run for a second or so and then manually reset it back to the Buffalo Monitor. Check the overflow counter register and see if it is incrementing properly.

4.6 Timer Overflow Counter, Summary

Once we have the timer overflow interrupt subroutine up and running we have created an 8 bit clock with a resolution of 0.065 seconds per count that will run automatically behind whatever computer program we wish to run.

The foreground computer program can refer to the 8 bit clock that is running in the background and use it to determine the progress of a delay.

All three of the routines shown in figure 1 should be copied to your library and included in your robot guidance computer program. When the program first starts up, it should call the TOF_ISR_ON subroutine to start the timer-overflow interrupt running. It should never be necessary in normal operation to call the routine TOF_ISR_OFF, but it could be useful for debugging purposes.

4.7 Alarm Time

In this section, we show how to use the overflow counter as a general purpose delay counter. This will enable us to specify that something happen at some time in the future.

If the present time is T_p , and the time to the event is the *delay time*, then something should happen at time $T_a = T_p + T_d$, where T_a will be known as the *alarm time*.

There are two ways to detect when the alarm time has occurred: by *polling* and by *interrupt*. If the alarm is polled, we check the time periodically to see if it has exceeded the alarm time – if it has, we do the required event. In pseudocode:

```
If Tp > Ta then
  Do the event
```

Alternatively, when the present time exceeds the alarm time, this causes an interrupt and the corresponding interrupt service routine performs the required event. This is how the hardware timers of the 68HC11 work, but we will leave that to a future exercise.

These mechanisms have counterparts in the world of human behaviour. When you have an important appointment, you have two options for leaving on time: watch the clock closely (polling) or set the clock alarm so that it gets your attention (an interrupt).

Polling is simple, but it requires that the present time be compared frequently against the alarm time. Otherwise, there will be excessive *latency* between the time that the alarm should be discovered and the time that it is actually discovered.

In this application, the time intervals need not be particularly precise and it is a simple matter to compare the actual time against the alarm time so we will use polling to check the alarms.

4.8 Example: A Timed Delay

In this section, we'll show an example of a timed delay, using the overflow counter, in assembly language.

If you were to run this program from the monitor, you would see the following: nothing would appear to happen for 5 seconds, and then the monitor prompt would appear.

There are two parts to the routine: one to initialize the timer, and one to check it. We assume that the TOF_COUNTER is maintained by its own interrupt service routine, so that it increments 15 times a second without our intervention. The necessary code to do this must be included in this program and was shown in figure 1.

We need a naming convention for various registers, so we'll use DT_ for *delay time* and AT_ for *alarm time*.

```
*****
*
*           Demo: Alarm using TOF Counter
*
* Delays for 5 seconds and then breaks to the monitor.
* Requires that the TOF_COUNTER be interrupt driven by overflow
* from the 68HC11 free running counter TCNT

ORG 6000  TOF_COUNTER  RMB 1  ; The timer, incremented at 15Hz
          SECONDS     EQU 15  ; Overflows per second at 1MHz crystal
          DT_DEMO     EQU 5*SECONDS
          AT_DEMO     RMB 1  ;

ORG 6010
INIT_DELAY  LDAA TOF_COUNTER  ; Initialize the alarm time
            ADDA T_DELAY      ; by adding on the delay
            STAA AT_DEMO     ; and save it in the alarm

CHK_DELAY  LDAA TOF_COUNTER  ; If the current time
            CMPA AT_DEMO     ; equals the alarm time
            BEQ STOP_HERE    ; then stop here

            NOP              ; Do something during the display
            BRA CHK_DELAY    ; and check the alarm again

STOP_HERE  SWI              ; Done, break to the monitor
```

4.9 A Potential Bug: Overflow of the Overflow Counter

There is the possibility of a serious bug in this design.

When we are comparing numbers in a computer program, we are usually advised to avoid a test for *equality* and instead use a test for *greater than* or *less than*. This is suggested because the representation of numbers in a high level language is often in floating point, and so two numbers which are essentially equal may in fact differ in one or more of the least significant bit positions.

So at first blush it would seem advisable to adopt that practice here. Moreover, we might reason that if we're a little late in checking the TOF counter it won't matter because the current time will have exceeded the alarm time ($T_c > T_a$) and the comparison will detect that.

This will work in some situations but not in others. Let's see why. Consider the following situation:

- The current time plus the delay time creates an alarm equal to %11111111 (\$FF).
- The counter increments towards this value, but we don't get around to checking it until it has gone a couple of clock ticks past the alarm. The counter overflows at a count of %11111111, and so when we compare it to the alarm it contains %00000001 (\$01).
- At this point, the counter has passed the alarm, but because it has overflowed, it compares as *less than* the alarm. So the delay would not terminate when it should, a serious error. In fact, if this situation continued, the delay would *never* terminate!

(There is a human analogy to this problem. Suppose someone has asked you to meet them at noon, when the clock reads **12**. You happen to check the clock, and it reads **1**. A literal interpretation of this is that you are not late, because 1 is less than 12. In fact, you are 11 hours early! But the clock has overflowed, so you are actually an hour late. We humans clock-readers take that into consideration. Clock overflow occurs twice a day under this system, which adds to the possible confusion. One reason for the European convention of numbering the hours on a 24 hour basis, is the fact that overflow occurs only once a day and at a time when not much is happening.)

There are various fixes for this problem. One way is to set up the counter so that it generates a carry bit every time it overflows, and then take the carry bit into consideration. This then requires a two byte comparison, which is a bit complicated.

There is a simpler alternative that will work in this situation. The counter is incremented every 65 milliseconds or so. If each computer instruction takes approximately 4 microseconds to execute, then the computer can execute something like $65 \times 10^{-3} / 4 \times 10^{-6} = 16 \times 10^3$ instructions per counter tick. If the alarm is checked more frequently than this, we will be sure to catch it at the instant it is exactly equal to the alarm time, which is what we want. Assuming that the program will execute no more than 16000 instructions between each check of the alarm, we should catch detect when the counter and alarm are equal.

If the original calculation of alarm time (present time plus delay time) overflows, then the counter will also overflow, and the equality will still be caught.

It would be *safer* to check for the counter greater than the alarm time, and in a mission critical application we would have to do that. But in this case, it shouldn't be a problem.

5 The Assignment

There are three components to this lab exercise:

1. Motor control subroutines
2. The interrupt-driven timer overflow counter routines
3. Timer Alarms

Demonstrating any one of these components will result in a pass grade. Demonstrating both will result in a full grade.

5.1 Motor Control

Create subroutines to control the *eebot* drive motors according to the information given previously in section 3. Be prepared to demonstrate one or two of them when requested by the lab supervisor.

5.2 Timer Overflow

To get a mark for this section of the assignment, demonstrate that you have a working, interrupt driven overflow counter. You can do this by assembling and installing the subroutines on the MPPV1 computer, and then activating the routines with the monitor CALL instruction.

5.3 Timer Alarms

For this section, you must create a three-stage alarm with displays on the LCD. The program should show **A** at the start of the program, **B** after 1 second and then **C** after a further 2 seconds. To get credit, the alarm program must be based on the interrupt timer concepts discussed in this lab. Software loops are not acceptable. If you demonstrate this program, you do not need to demo that Timer Overflow is working.

6 References

M68HC11 Reference Manual

Motorola Document M68HC11RM/AD REV 3, 1991

The authoritative source of information about the 68HC11 microprocessor.

Available from Motorola on request.

68HC11 Microcontroller, Construction and Technical Manual

Peter Hiscocks, 2001

Technical information on the MPP Board, 68HC11 Microprocessor Development System

Information on programming and interfacing the MPP Board used at Ryerson and elsewhere.

Available from Active Electronics, at the Victoria Park-Gordon Baker store in Toronto.

M68HC11: An Introduction, Software and Hardware Interfacing

Han-Way Huang

Delmar Thompson, 2001

A basic text on the 68HC11 microprocessor.

Microcomputer Technology: The 68HC11

Second Edition

Peter Spasov

Prentice Hall, 1996