Microprocessor Systems
Motor Control
and
Using the Hardware Timer

Contents

[1]This lab was adapted to be used with the HCS12 microcontroller by V. Geurkov.

List of Figures

1 Objectives

In this lab, we will develop

- Software routines to control the *eebot* motors.
- A software clock using the HCS12 interrupt-driven *hardware timer overflow flag*.
- A *timed alarm* mechanism that can be useful in controlling the *eebot* in various manoeuvres such as turns.
- A routine to read information from the *eebot* bumper swithches.

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

2 Overview

Robot *roaming* behavior can be obtained with a very simple set of rules. Initially, the robot drives in a straight line[2] 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 HCS12 hardware timer that will be used to create the various time delays.

3 Motor Control Routines

A microcontroller communicates with output devices (such as motors) through PORTS. In the simplest case, a port can be considered as a data register. The inputs of this register are connected to the internal data bus, while the outputs are tied to external pins of the microcontroller. Each port is *mapped* into the address space of the microcontroller. This means that 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 0 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 HCS12 *eebot* microcontroller system, we will use two ports that are controlled in this fashion.

3.1 Controlling individual bits

~~The first of these ports, PORTA is located at address $0000.~~ In the *eebot* system, two bits of PORTA control the direction of the two robot drive motors. Three other bits, as we will see later, determine which of the 6 optical sensors is read on channel 1 of the A/D converter. One more bit enables the read operation. And the remaining 2 bits are not used.

The second of these ports, PORTT is located at $0240. Two bits of this port are used to control the speed (On/Off) of two motors.

[2]Well, 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.

When individual control lines are mapped into the bits of a memory byte, we need routines that can set and clear each of the individual bits. When you write to a location such as PORTA or PORTT, 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 (PORTA or PORTT) 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 and figure (see *The eebot Technical Description* document):

Motor	Function	Pin	Notes
6WDUERDUG	6SHHG	375	1 IRU RQ, 0 IRU RII
	DiUHcWiRQ	3$1	0 IRU IZG, 1 IRU UHY
3RUW	6SHHG	374	1 IRU RQ, 0 IRU RII
	DiUHcWiRQ	3$O	0 IRU IZG, 1 IRU UHY

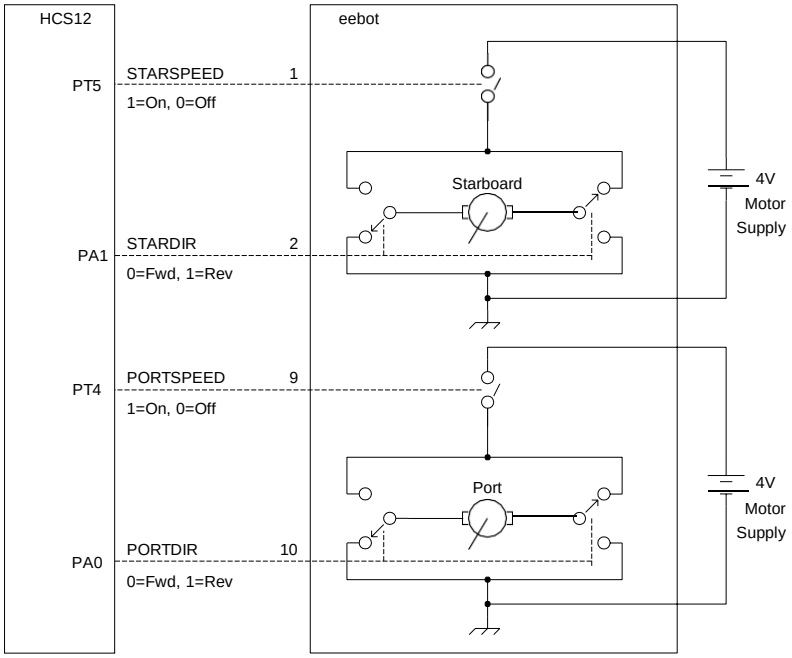

Figure 1: Motor Control, Programming Model

3.2 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 of the output register PORTA or PORTT. However, when changing bits in PORTA or PORTT, 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 Serial Monitor "Step Over" instruction to execute the various subroutines and drive the *eebot* motors. For example, with different subroutines, you should be able to start the Starboard motor in forward direction and then start the Port motor in forward direction.

To energize 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.

Note: You can use PORTH of the bench microprocessor system to imitate (emulate) PORTA & PORTT of the *eebot* system for debugging purposes. The PORTH pins are connected to LEDs (see the 9S12DG128 board schematic), so that you can visually verify if the pins are controlled properly.

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. Conseqently, 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 this lab, we will take a *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 HCS12 automatically call a certain routine after a specified delay. We'll tackle that later.)

4.1 The HCS12 Timer Mechanism

The timer subsystem of the HCS12 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 `$0044`). `TCNT` is driven by the system clock, so the basic clock tick is 1/24 MHz = 41.67 nanoseconds. The counter *free-runs*, that is, there is no way to stop it or reset it. The location `$0044` is *read-only*[3].

The TCNT must be enabled before in can run. Setting bit 7 of the TSCR1 register (memory location `$0046`) enables the TCNT to count.

(Use the Serial Monitor to enable TCNT with the `WB` command and examine location `$0044` with the `DW` 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 41.67 nanoseconds clock, it will overflow every $2^{16} \times (41.67 \times 10^{-9}) = 0.00273$ seconds (= 2.73 ms). The clock input to the TCNT can be prescaled by a factor of 128 at most. Therefore, the overflow will occur every $(128 \times 2.73 =)$ 349.5 ms. This is still 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 (Freescale) had provided an additional 8 bit hardware counter stage. That would have provided a clock with a resolution of 349.5 milliseconds and a maximum count of $2^8 \times 0.3495 = 89.5$ seconds. If we selected the prescale factor equal to 16, the resolution and time count would accordingly change to $(16 \times 2.73 =)$ 43.7 ms and $2^8 \times 0.0437 = 11.2$ seconds, which is more useful for this application. Therefore, we will consider 16 as a default prescale value.

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 43.7 milliseconds, 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 `$004F`). It is *set* every time the free running counter overflows[4].

If you read location `$004F`, you will find that the high bit is effectively always set because it gets set every 0.0437 seconds. Taking the reciprocal, this is 23 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 occured.

There are a number of interrupt systems on the HCS12 microprocessor. Each system has three components:

[3] It 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 41.67 nanoseconds.

[4] See the textbook or reference manual.

- A *device flag* that is *set* on occurance 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.
- 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 0.0437 seconds, or 23 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  $004F        ; MSB is the timer overflow flag
TOF_COUNTER     DS.B  1           ; Overflow counter

TOF_ISR         INC  TOF_COUNTER  ; Increment the overflow count
                LDAA #%10000000   ; Clear the TOF flag
                STAA TFLG2        ;   --"--
                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 memory of the microprocessor. The exact incantation is:

  ```
  ORG $FFDE
  FDB TOF_ISR
  ```

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

- Set the timer overflow interrupt enable bit, which is known as `TOI`. This has the effect of enabling the TOF interrupt. This bit is found in the MSBit of the register `TSCR2` (memory address `$004D`). The three LSBits of the `TSCR2` define the prescale factor for the TCNT. If we choose 100 for these bits, than the factor will be 16. Therfore, the constand that must be written to `TSCR2` is %10000100[5].

- 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 2 on page 7 and figure 3 on page 8.

[5] See the Reference Manual.

```
*******************************************************
*          Timer Overflow Demonstration
*
* This program contains routines to demonstrate the timer overflow
*   interrupt on the HCS12.
* Instructions:
*   Assemble this program and load it into memory.
*   Use the Serial Monitor WW command to zero the TOF_COUNTER.
*   Use the monitor GO 4000 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 23 counts per second.)
*   Manually reset the HCS12 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
*******************************************************
*         Register Definitions

TFLG2        EQU $004F          ; Contains the TOF at MSB
TSCR1        EQU $0046          ; Contains the TEN (timer enable bit)
TSCR2        EQU $004D          ; Contains the TOI (timer overflow interrupt enable bit)

             ORG $3000          ; Where our TOF counter register lives
TOF_COUNTER  RMB 1              ; (or DS.B) One byte overflow counter
             ORG $4000          ; 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 2: 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.
- As shown in figure 2 and figure 3, construct two other small subroutines, one 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 interrupt vector address and any other initial conditions (e.g. for the hardware) are set up.
- 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 `WB` 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` should be incrementing at a rate of 23 counts per second.
- Let the microprocessor run for a second or so and then manually reset it back to the Serial Monitor. Check the overflow counter register and see if it is incrementing properly.

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

ENABLE_TOF  LDD #TOF_ISR        ; Setup the interrupt vector for timer overflow
            STD $FFDE

            LDAA #%10000000
            STAA TSCR1          ; Enable TCNT by setting bit 7

* 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.

            STAA TFLG2          ; Clear the TOF flag by writing to bit 7
            LDAA #%10000100     ; Turn timer overflow interrupt on by setting bit 7
            STAA TSCR2          ; in TSCR2 and select prescale factor equal to 16
            RTS
*************************************************************
*         Timer Overflow Interrupt Service Routine

*  This routine is called on interrupt each time the free-running
*   16 bit counter overflows.
*  Assuming that the timer prescaler bits PR0, PR1 and PR2 have
*   been changed, the basic rate of the 16 bit free running
*   counter is 667 nanoseconds, so overflows occur about 1/23
*   second apart. The TOF_COUNTER may be used in time delays.

TOF_ISR     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 monitor 'trace' function doesn't
*  work otherwise.

DISABLE_TOF LDAA #%00000100     ; Turn timer overflow interrupt off by clearing bit 7
            STAA TSCR2          ;  in TSCR2 and leave prescale factor at 16
            RTS
```

Figure 3: Timer Overflow Routines (Part 2)

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 0.0437 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 3 should be copied to your library and included in your robot guidance computer program. When the program first starts up, it should call the `ENABLE_TOF` subroutine to start the timer-overflow interrupt running. It should never be necessary in normal operation to call the routine `DISABLE_TOF`, 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 will 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 HCS12 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 23 times a second without our intervention. The necessary code to do this must be included in this program and was shown in figure 3.

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 HCS12 free running counter TCNT

            ORG $3000
TOF_COUNTER RMB 1               ; The timer, incremented at 23Hz
AT_DEMO     RMB 1               ; The alarm time for this demo
DT_DEMO     EQU 115             ; 5 second delay (at 23Hz)

            ORG $4000
INIT_DELAY  LDAA TOF_COUNTER    ; Initialize the alarm time
            ADDA #DT_DEMO       ; 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 adviseable 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 ($Tc > Ta$) 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 44 milliseconds or so. If each computer instruction takes approximately 167 nanoseconds to execute, then the computer can execute something like $44 \times 10^{-3} / 167\ 10 \times^{9} = 263,500$ 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 $263,500$ 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 *eebot* or bench microcontroller board, and then activating the routines with the monitor "Step Over" instruction. Or, you can assemble and install a timed (5 second) delay program. Activate it by the monitor "Go" 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 seconds 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.

Some hints on creating the programs can be found in Appendices A, B and C.

6 Appendix A

The Appendix A refers to assignment 1 for *eebot*.

```
*************************************************************
*           Motor Control                                   *
*************************************************************
            BSET    DDRA,%00000011
            BSET    DDRT,%00110000
            JSR     STARFWD
            JSR     PORTFWD
            JSR     STARON
            JSR     PORTON
            JSR     STARREV
            JSR     PORTREV
            JSR     STAROFF
            JSR     PORTOFF
            BRA     *

STARON      LDAA    PTT
            ORAA    #%00100000
            STAA    PTT
            RTS

STAROFF     LDAA    PTT
            ANDA    #%11011111
            STAA    PTT
            RTS

            ...

PORTFWD     LDAA    PORTA
            ANDA    #%11111110
            STAA    PORTA
            RTS

PORTREV     LDAA    PORTA
            ORAA    #%00000001
            STAA    PTH
            RTS
```

7 Appendix B

The Appendix B refers to assignment 2 for *eebot*.

```
*************************************************************
*           5 Second Delay                                  *
*************************************************************
DT_DEMO     EQU     115             ; 5 second delay

            ORG     $3850
TOF_COUNTER RMB     1
AT_DEMO     RMB     1

            ORG     $4000
Entry       LDS     #$4000

            JSR     ENABLE_TOF      ; Jump to TOF init
            CLI

            LDAA    TOF_COUNTER
            ADDA    #DT_DEMO
            STAA    AT_DEMO

CHK_DELAY   LDAA    TOF_COUNTER
            CMPA    AT_DEMO
            BEQ     STOP_HERE

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

STOP_HERE   SWI
*************************************************************
ENABLE_TOF  LDAA    #%10000000
            STAA    TSCR1           ; Enable TCNT
            STAA    TFLG2           ; Clear TOF
            LDAA    #%10000100      ; Enable TOI and select prescale factor equal to 16
            STAA    TSCR2
            RTS
*************************************************************
TOF_ISR     INC     TOF_COUNTER
            LDAA    ...             ; Clear
            STAA    TFLG2           ;  TOF
            RTI
*************************************************************
DISABLE_TOF LDAA    #%00000100      ; Disable TOI and leave prescale factor at 16
            STAA    TSCR2
            RTS
*************************************************************
*           Interrupt Vectors                               *
*************************************************************
            ORG     $FFFE
            DC.W    Entry           ; Reset Vector

            ORG     $FFDE
            DC.W    TOF_ISR         ; Timer Overflow Interrupt Vector
```

8 Appendix C

The Appendix C refers to assignment 3 for *eebot.*

```
***********************************************************
*           Timer Alams                                   *
***********************************************************
;definitions
OneSec      EQU   23               ; 1 second delay (at 23Hz)
TwoSec      EQU   46               ; 2 second delay (at 23Hz)
LCD_DAT     EQU   PORTB            ; LCD data port, bits - PB7,...,PB0
LCD_CNTR    EQU   PTJ              ; LCD control port, bits - PJ7(E),PJ6(RS)
LCD_E       EQU   $80              ; LCD E-signal pin
LCD_RS      EQU   $40              ; LCD RS-signal pin

;variable/data section
            ORG   $3850            ; Where our TOF counter register lives
TOF_COUNTER RMB   1                ; The timer, incremented at 23Hz
AT_DEMO     RMB   1                ; The alarm time for this demo

;code section
            ORG   $4000            ; Where the code starts
Entry:
_Startup:
            LDS   #$4000           ; initialize the stack pointer
            JSR   initLCD          ; initialize the LCD
            JSR   clrLCD           ; clear LCD & home cursor
            JSR   ENABLE_TOF       ; Jump to TOF initialization
            CLI                    ; Enable global interrupt

            LDAA  #'A'             ; Display A (for 1 sec)
            JSR   putcLCD          ;  --"--

            LDAA  TOF_COUNTER      ; Initialize the alarm time
            ADDA  #OneSec          ;  by adding on the 1 sec delay
            STAA  AT_DEMO          ;  and save it in the alarm
CHK_DELAY_1 LDAA  TOF_COUNTER      ; If the current time
            CMPA  AT_DEMO          ;  equals the alarm time
            BEQ   A1               ;  then display B
            BRA   CHK_DELAY_1      ;  and check the alarm again

A1          LDAA  #'B'             ; Display B (for 2 sec)
            JSR   putcLCD          ;  --"--

            LDAA  AT_DEMO          ; Initialize the alarm time
            ADDA  ...              ;  by adding on the 2 sec delay
            STAA  ...              ;  and save it in the alarm
CHK_DELAY_2 LDAA  TOF_COUNTER      ; If the current time
            CMPA  ...              ;  equals the alarm time
            BEQ   A2               ;  then display C
            BRA   CHK_DELAY_2      ;  and check the alarm again

A2          LDAA  #'C'             ; Display C (forever)
            JSR   putcLCD          ;  --"--
            SWI

;subroutine section
initLCD     ...                    ; same as in lab3
clrLCD      ...                    ;  --"--
del_50us    ...                    ;  --"--
cmd2LCD     ...                    ;  --"--
putcLCD     ...                    ;  --"--
dataMov     ...                    ;  --"--
ENABLE_TOF  ...                    ;  same as in Appendix B of this lab
TOF_ISR     ...                    ;  --"--
****     Interrupt Vectors   *****  ;  --"--
```

9 References

Schematics of the HSC12 Evaluation Board
Available on-line.

CPU12 Reference Manual
Motorola Document CPU12RM/AD REV 3, 4/2002
The authoritative source of information about the 68HC12 & HCS12 microcontrollers.

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

HCS12/9S12: An Introduction to Software and Hardware Interfacing
Han-Way Huang
Delmar Cengage Learning, 2010
A basic text on the HCS12 microcontroller.