ELE538 Microprocessor Systems
Technical Note
Motor Speed Control

Peter Hiscocks
Department of Electrical and Computer Engineering
Ryerson Polytechnic University
phiscock@ee.ryerson.ca
August 27, 2002

Contents

1 Introduction **1**

2 Speed and Steering Control **2**

3 Motor Speed Control **3**
- 3.1 Design . . . 3
- 3.2 The 68HC11 *Output Compare* Timer Mechanism . . . 4
- 3.3 Example: Square Wave Generator . . . 6
- 3.4 Duty Cycle Calculation . . . 7
- 3.5 Output Compare Interrupt Machinery . . . 8
- 3.6 The *Output Compare* Interrupt Service Routine . . . 8

4 Appendix 1: Duty Cycle Routine **10**

List of Figures

1 Motor Control, Programming Model . . . 4
2 Output Compare Mechanism . . . 5
3 Output Action . . . 6
4 Duty Cycle Waveform . . . 7
5 Output Compare Machinery . . . 8
6 Output Compare Interrupt Service Routine, Pseudocode . . . 9

1 Introduction

The *eebot* robot is driven by two 3 volt DC gearmotors. For precise control of eebot steering and navigation, it is necessary to be able control the speed of these two driving motors. To steer the robot, the gearmotors can be driven at different speeds or even in different directions. The robot can then execute very tight turns. The front

spherical wheel rolls normally when the robot is moving straight ahead, and skids sideways when the robot is executing a turn.

The circuitry of eebot was designed with motor speed control in mind: the power for each motor is controlled from a timer output on the 68HC11 that may be modulated in duty cycle, thereby controlling motor speed.

Duty Cycle Modulation is a simple concept. The motor is switched on and off at a rate much faster than the motor can respond. As a result, the mechanical inertia of the motor effectively averages the pulse waveform. The average value of the voltage across the motor, which sets the motor speed, is proportional to the duty cycle D, where

$$D = \frac{T_{on}}{T}$$

T_{on} is the time the switch is closed, applying the supply voltage to the motor, and T is the period of the repetitive waveform. (Also notice that

$$T = T_{on} + T_{off}$$

where T_{off} is the time the motor is disconnected from the supply voltage.) A duty cycle waveform is shown in figure 4 on page 7.

Because the electronic switches driving the motor are either `ON` or `OFF`, they dissapate very little power. Consequently, the motor drive circuit is economical of battery power and does not require a large heat sink.

If the duty cycle of the two motors is reduced or increased in unison, the motors will slow down or speed up. The robot will continue to move in a straight line with varying speed. For example, if the guidance software may determine that the speed should be slowed so that the robot can navigate obstacles precisely.

If one motor is slowed compared to the other motor the robot will turn toward the slower motor. This allows *proportional* control of steering. A small change in speed will lead to a gradual change in heading, so the robot may be steered smoothly and precisely.

2 Speed and Steering Control

There are two common mechanisms for steering a vehicle. We are all used to the automotive model, which consists of a steering device (the steering wheel) and a speed control device (the throttle, or gas pedal).

An alternative method is often used in tracked vehicles such as a bulldozer or army tank. A throttle is provided to set basic engine speed. Two levers, one for each track, can be used to brake one track or the other, thereby causing the vehicle to change direction. This takes some practice to master[1].

The basic mechanism of the eebot is the same as a tracked vehicle. It would therefore seem natural to provide two throttles, one for each motor. Steering is accomplished by setting the throttles to different values (differential steering). Speed control is accomplished by varying the throttles together (collective speed control). In practice however, it is very difficult to steer a vehicle with this type of control. We would say that the *operator interface* is poor.

Software can provide a better interface for steering the vehicle. The operator is provided with two separate controls, one for steering and the other for speed. The software then uses these input signals to generate the correct collective and differential signals for the two motors.

For example, the throttle control generates a signal V which specifies the velocity of the vehicle. The steering system then apportions this velocity signal to the two motors so that steering takes place correctly. For example, the steering signal S could have a value between 0 (maximum left turn) and 1 (maximum right turn). Then for

[1] And while learning to drive one of these things, it is possible to do much damage to ones surroundings.

$S = 0.5$ the vehicle would steer straight ahead. The port motor speed S_p and the starboard motor speed S_s are given by

$$S_p = V \times S$$

$$S_h = V \times (1 - S)$$

We now describe the assembly language software necessary to control the duty cycle of the motor power waveform. The objective is to construct a software routine that will accept a value of duty cycle between 1% and 99% and adjust the motor voltage waveform duty cycle to this value.

3 Motor Speed Control

The motor control circuit is a L293D integrated circuit. One pin on the L293D controls motor voltage *polarity* to control the motor direction. There is also a motor voltage *enable* pin on the L293 that enables and disables motor supply voltage. By varying the duty cycle of the enable signal, the average motor voltage may be adjusted to something less that 3 volts, controlling the speed of the motor.

The motor requires a minimum of 1 volt to operate. With a 3 volt supply voltage, the duty cycle range can therefore range between 30% (minimum) and 99%. The optimum switching frequency for the motor supply is 25Hz.

Two LED indicators are attached to each motor, one for each connection to the motor, labelled as `MOTOR+` and `MOTOR-`. These are useful for determining that the motor is receiving power and provide a visible indication of the duty cycle.

The simplified circuit for the motor control circuit is shown in figure 1.

Notice:

- The direction is controlled by an electronic DPDT switch which controls the direction of current flow through the DC motor. The state of this switch is controlled by bits D0 and D1 of the general purpose output register, which is mapped into the microprocessor memory location $1100.
- The speed of each motor is controlled by a switch in series with the power supply. Varying the duty cycle of the switch sets the average voltage across the motor, and hence its speed. The motor speeds are controlled by the Output Capture lines OC4 and OC3, which are part of the timer section of the 68HC11 microprocessor.

3.1 Design

The motor speed control waveform was experimentally found to work best at a frequency of 25Hz, a period of 40 milliseconds[2]. Consequently, the duty cycle modulated waveform will consist of an output high segment T_H followed by an output low segment T_L, such that $T_H + T_L = 0.040$ seconds. There are two motors, so we require two of these duty-cycle pulse generators. As well, the waveforms must be generated continuously. Interrupts and other delays must have no effect on the motor signals. The precision of the duty cycle should be something like 1 part in 100, or 1%.

Which timer should we use to generate this waveform? There are many timers available on the 68HC11:

- software delay
- TOF counter

[2] It should work at higher frequencies but, for reasons unknown, does not.

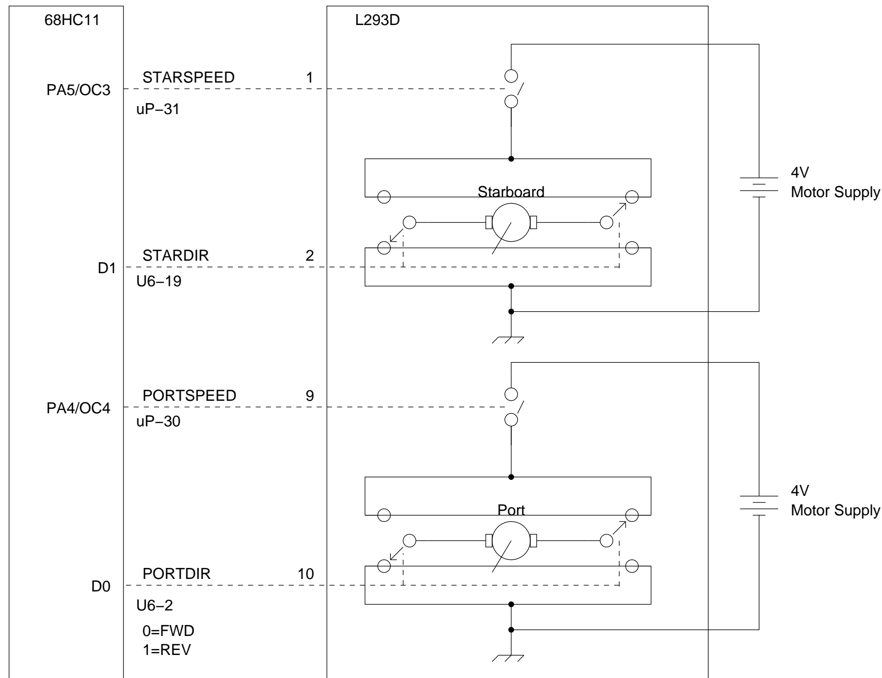

Figure 1: Motor Control, Programming Model

- Real Time Interrupt
- Hardware timers

We can immediately eliminate the *software delay* as a candidate because that would require tying up the processor in a lengthy delay loop.

The `TOF` counter is not suitable because the time resolution is not fine enough: it works in increments of 2^{16} microseconds (0.065 seconds).

Similarily, the real-time interrupt is intended for applications in the tens of milliseconds, so it is not suitable.

Fortunately, the hardware timers are ideal for this application because the output waveform is hardware controlled. Interrupts have no effect on the switching time. They work semi-automatically, and the time that the timer is serviced can vary without affecting the output waveform. The hardware timers work by comparing a 16 bit value to the 16 bit free-running counter, so their basic resolution is 1μsecond with a precision of $1 : 2^{16}$. This meets all our requirements, so is the mechanism of choice in this application[3].

3.2 The 68HC11 *Output Compare* Timer Mechanism

The 68HC11 timer system includes 5 *output compare timer channels*. This machinery may be used to generate a duty cycle modulated waveform semi-automatically. We'll now review how those timer channels work.

[3]This is not surprising, since the 68HC11 was originally designed for automotive engine control applications, which require several precise timing waveforms.

A block diagram of one of the five output compare timers is shown in figure 2.

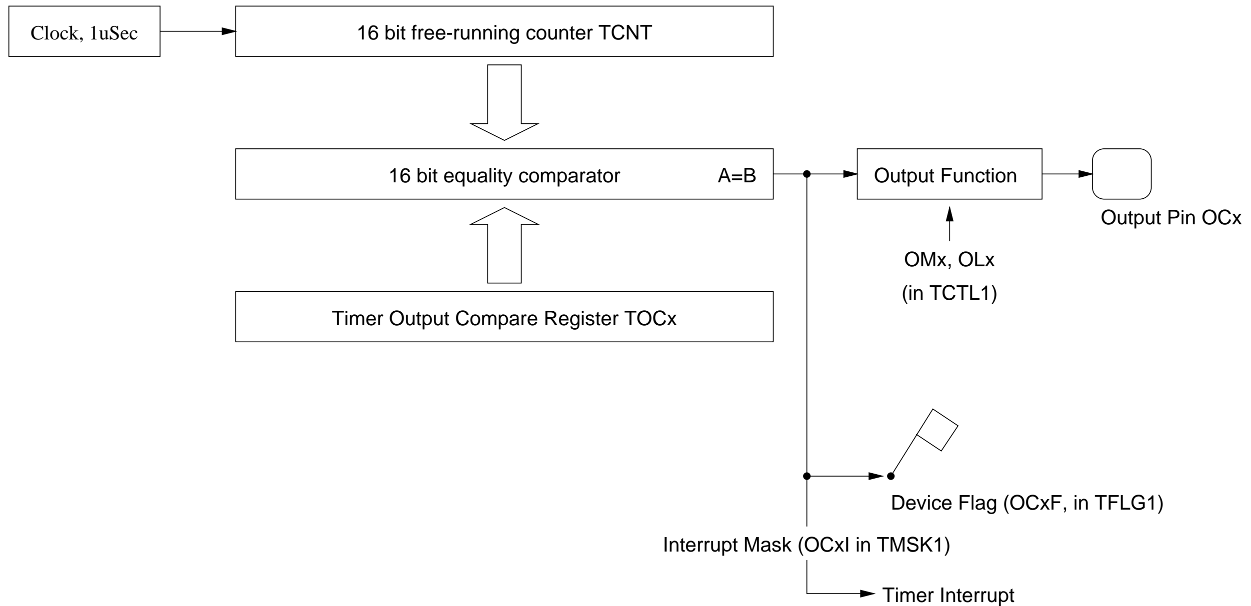

Figure 2: Output Compare Mechanism

The 16 bit free-running counter increments at a rate of 1μ second per count and rolls over to zero at a count of 2^{16}. The counter may be read from software but cannot be written.

For each of the 5 hardware output compare channels, there is a 16 bit *equality comparator* and a 16 bit *timer output compare register* TOC. The timer output compare register may be loaded from software by a sequence of instructions such as:

```
LDD #COMPARE_VALUE
STD TOC1
```

which sets the timer output compare register for timer channel 1.

Whenever the value in the free-running counter is equal to the value in the timer output compare register, two things happen:

- the *output function* causes an *output action* to happen at the timer output pin, and
- the timer device flag TOF is set.

If the interrupt is enabled for this particular timer channel, setting the TOF flag further causes an interrupt to occur.

The output action that occurs on a successful compare depends on two bits in configuration register TCTL1 as shown in figure 3 on page 6. Continuing with our consideration of timer channel 1, if OM1= 0 and OL1= 1, then the voltage at the timer 1 output pin would alternate from +5 volts to zero and vice versa each time the value in TCTNT is equal to the value in TOC1.

OMx	OLx	Output Action
0	0	No affect
0	1	Toggle (flip) Pin
1	0	Clear
1	1	Set

Figure 3: Output Action

3.3 Example: Square Wave Generator

A simple example will illustrate how this mechanism can be used to produce a square wave. As usual, we'll assume the use of output compare channel 1.

First, consider the specific case where the square wave period was to be exactly equal to the twice the repetition period of the free running counter. Then we could put *any value* into the Timer Compare register `TOC1`. The value in the free running counter `TCTNT` would be exactly equal to the value in the timer compare register `TOC1` once every 2^{16} counts, or every 65536μseconds. We would set up `OM1`$= 0$ and `OL1`$= 1$, so that the output voltage would flip between +5 volts to zero and vice versa every 65536 μseconds (≈ 65 milliseconds). The result would be a square wave with a period of 131 milliseconds at the timer channel 1 output pin[4].

This is not likely to be useful unless one needs a square wave of exactly 131 milliseconds. However, with some software intervention, the square wave can be made to be *any* period up to 131 milliseconds. The method requires the intervention of software to reload the timer output compare register `TOC1` every time a successful comparison occurs. The value loaded into `TOC1` is equal to the half-period of the desired waveform.

Now, in detail:

- Set the initial alarm time $T := 0$.
- Suppose the desired period of the square wave is T_s milliseconds (where $T_s < 131$ milliseconds).
- The timer output function is set up to cause the output pin voltage to flip with every successful compare event.
- The timer is also set up to cause an interrupt on every successful compare event.
- On interrupt, the interrupt service routine
  - clears the device flag
  - calculates $T_{new} := T_{old} + T_s/2$ and
  - puts the new time T_{new} back in the timer compare register `TOC1`: $T_{old} := T_{new}$

In other words, every time a half-period event occurs, the output flips as a result of the timer output hardware and the timer compare register is reloaded with the next comparison time.

Notice that the reloading take some time. The interrupt service routine must be called and executed, and this takes time. However, the output transition is caused by hardware, so it occurs precisely at the correct instant. The only requirement of the ISR is that it execute before the *following* compare event occurs.

[4]The first half-period will not necessarily be 65 milliseconds because it depends on the contents of the free-running counter when the process is started.

3.4 Duty Cycle Calculation

From the previous example, it is only a small step to controlling the duty cycle of the output waveform. If the duty cycle is something other than 50% the timer software alternately loads two different values of time interval into the timer compare register depending on the required length of the output interval.

In our case, we want the output waveform to have a frequency of 25 Hz, ie, a period of 40 milliseconds. Then the waveform will consist of a time interval T_H in which the output voltage is *high* followed by a time interval T_L in which the output voltage is *low*, and the total of T_H plus T_L will be 40,000 microseconds, as shown in figure 4. (Time T_H corresponds to the previously discussed T_{on}. T_L corresponds to T_{off}. The terms *high* and *low* refer to the the motor voltage during T_{on} and T_{off}.)

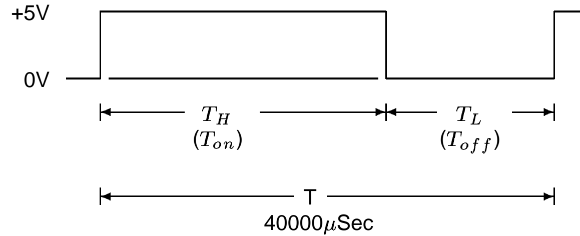

Figure 4: Duty Cycle Waveform

In setting up the values of T_H and T_L, we must calculate values that satisfy the two requirements of duty cycle and waveform period. Expressing the period of the waveform in microseconds (the basic unit of the timer circuits), we have

$$\begin{aligned} T &= T_H + T_L \\ &= 40000 \end{aligned}$$

As well, the duty cycle D (in percent) is given by

$$D = \frac{T_H}{T} \times 100\%$$

Putting these two equations together and solving for T_H and T_L in terms of D we have

$$T_H = 400D \tag{1}$$

and

$$T_L = 400 - T_H \tag{2}$$

Every time the motor duty cycle is changed, foreground software routine recalculates the values of T_H and T_L. The timer output compare interrupt service routine then alternately loads T_H and T_L into `TOCx` to generate the duty cycle waveform.

Notice that this has to be done twice, once for each motor. The timer `TOCx` is `TOC3` for the starboard motor and `TOC4` for the port motor[5].

[5] We have recently moved the starboard motor from PA3/OC5 to PA5/OC3.

3.5 Output Compare Interrupt Machinery

Each output compare timer channel uses the machinery shown in figure 5 on page 8:

Name	Function	Register
Output Compare	Contains the timer value to be compared to the 16-bit free-running counter. Loaded with the correct value by the timer ISR on each timer interrupt.	`TOC3` (Starboard) `TOC4` (Port)
Output Action	Specification of the *output action*, the output voltage level that occurs with every interrupt. Two bits must be setup by the timer initialization software.	`TCTL1` (See figure 3)
Device Flag	Set when a output compare occurs, must be cleared by the interrupt service routine.	`TFLG1`
Interrupt Mask	An *interrupt enable mask bit*, that enables and disables interrupts for this output compare channel. One bit must be set by the timer initialization software.	`TMSK1`
Interrupt Vector	A two-byte address at the top of memory which points to a pseudo-interrupt vector.	No setup required
Pseudo-Interrupt Vector	Three-byte jump instruction, must be set up by the interrupt initialization software.	See table, Lab 6 $00D9 (Timer 3) $00D6 (Timer 4)
Interrupt Service Routine	Interrupt Service Routine: One required for each output compare timer.	See following section
Interrupt Enable	Initializes the channel machinery and turn on its interrupts.	See following section
Interrupt Disable	(Optional) A subroutine to turn off the interrupts on that channel.	See following section

Figure 5: Output Compare Machinery

3.6 The *Output Compare* Interrupt Service Routine

Now we will discuss the design of the timer interrupt subroutine. We assume that there is one ISR for each motor, so we'll describe only the starboard version.

The routine alternates between two principal states: `OUTPUT_LOW` and `OUTPUT_HIGH`. On each timer interrupt, the state changes. Consequently, the state diagram is very simple: the state machine simply transitions unconditionally from one state to another each time the interrupt subroutine is called. The pseudocode for the Interrupt Subroutine shown in figure 6 on page 9.

Notice the following logic in the ISR:

- If the state is currently `OUTPUT_HIGH` when the interrupt occurred, the output switched (under hardware control) to the output *low* state. So `OMx:OLx` was and is currently set to `1:0`. (See figure 3).
- The pin voltage is currently zero volts (*low*).
- Obviously, the next state should be `OUTPUT_LOW`.

```
        Clear the timer flag.
        If STATE == OUTPUT_HIGH then
                STATE := OUTPUT_LOW
                T := (T + T_LOW) MOD 2^16
                TOCx  := T
                OMx:OLx := 1:1 ;Transition on interrupt to output high
                Return
        If STATE == OUTPUT_LOW then
                STATE := OUTPUT_LOW
                T := (T + T_HIGH) MOD 2^16
                TOCx  := T
                OMx:OLx := 1:0 ;Transition on interrupt to output low
                Return
        Else                   ; Illegal state!
                Set error flag in RAM
                Loop here forever

Note:    := means 'is assigned to'
         == means 'is equal to'
```

Figure 6: Output Compare Interrupt Service Routine, Pseudocode

- We now want the timer to delay for the interval `T_LOW`. If the present time is `T`, then we wish the next interrupt and output transition to occur at time `T:=T+T_LOW`. This sum is *modulo* 2^{16}, that is, the result is truncated to 16 bits, ignoring any carries. The time `T` must be maintained somewhere as a 16-bit variable.
- Next, we load the truncated sum `T` into the timer compare `TOC` register.
- At the end of that interval, we want the output to switch (under hardware control) to output *high*, so we must load the output action bits `OMx:OLx` with `1:1`. This will cause the output to transition *high* on the next timer interrupt.

Error Checking

The design of routines to detect, signal and recover from errors is a major challenge in software design. In this case, we take the easy way out: if the machine somehow gets into an illegal state, the state machine sets a bit in a pre-determined RAM register and then loops forever. The *alive* indicator extinguishes, which signals that the program has crashed.

The main loop clears this location each circuit around the main loop. Consequently, if the machine appears to crash, a dump of the 'error flag register' would show a non-zero result if this were the cause. In other words, if this error occurs, we deliberately hang up the program. In the robot operating system, this is probably acceptable. In an aircraft flight control system, it would definitely not be sufficient.

4 Appendix 1: Duty Cycle Routine

The complete working routine for duty cycle control of the `OC3` (eebot starboard motor) is shown on the following pages.

Page	Description
11	Program header, equates and RAM register definitions. These would be modified with the addition of registers and equates when adding port motor control.
12	This page includes the Main Loop for the program, which simply starts the `OC3` waveform running and then loops endlessly. This routine would be modified to call the initialization for the `OC4` timer channel. It also includes the routine to calculate timer intervals based on the desired duty cycle. It should not be necessary to modify this routine. It also includes the routine to initialize duty cycle modulation on timer channel 5. A second version of this is required for the port motor.
13	The duty cycle modulation interrupt service routine. A second version of this is required for the port motor.
14	A 16×16 bit multiplication routine. It should not be necessary to modify this routine when adding the second timer control channel.

```
*---------------------------------------------------------------
*                  Duty Cycle Demonstration
*
* This program demonstrates the use of the output compare timer channel
*  on the 68HC11 microprocessor to produce a waveform of specified duty
*  cycle.
* The code shown drives the starboard motor of the eebot robot, which is on
*  output compare channel 5. Similar code would be used for the port motor
*  on output compare channel 4.
* To operate this test program:
*   - load the .s19 file into the HC11 RAM as usual
*   - use MM 6000 to set the duty cycle to something between 01 and 99
*   - run the program with G 6050
*   - the waveform at the MPP board pin PA3/OC3 should show the correct
*      duty cycle
*   - if eebot is connected, the motor speed should be affected.

* Tested: 6 October 2001
* Peter Hiscocks
* Modified to change OC5 to OC3 August 2002. Not tested!
*----------------------------------------------------------------
*
* Equates
* See the Pink Book for the bit assignments
TOC3    equ $101A       ; Timer compare hardware register (2 bytes)
TCTL1   equ $1020       ; Output action control register
TMSK1   equ $1022       ; Output compare, interrupt mask
TFLG1   equ $1023       ; Output compare, device flag
TOCVEC3 equ $00D9       ; Pseudo-interrupt vector location (3 bytes)
*
OUTPUT_LOW      equ 0     ; Definition of the Output Compare timer states
OUTPUT_HIGH     equ 1     ;
MOTOR_PERIOD    equ 40000 ; Period of motor frequency waveform, microsec
TOC3_BADSTATE   equ %00000001 ; Error bit
*
* Variables
 org $6000

* Starboard duty cycle parameters
ST_DUTY_CYCLE   fcb 75  ; Duty cycle in % (1 to 99)
*                       ;  Initialized to 75%
ST_T_LOW        rmb 2   ; Time to spend in low state (microsec)
ST_T_HIGH       rmb 2   ; Time to spend in high state (microsec)
ST_TIME_TOTAL   rmb 2   ; Time total
ST_STATE        rmb 1   ; State of the duty cycle state machine

ERROR_REG       rmb 1   ; Contains error messages
TEMP            rmb 2   ; General purpose working register
*
* Working Register Area for double-precision multiply
OP1_HI  rmb 1   ; First operand, high byte
OP1_LO  rmb 1   ; First operand, low byte

OP2_HI  rmb 1   ; Second operand
OP2_LO  rmb 1

RES_HI   rmb 1  ; Result registers, high to low.
RES_NSB2 rmb 1
RES_NSB1 rmb 1
RES_LO   rmb 1
```

```
* Text (Code)
 org $6050
*-------------------------------------------------------------------
*                      Main Loop
*
START   CLR ERROR_REG      ; No errors yet

        LDAA ST_DUTY_CYCLE ; Calculate the timer intervals
        JSR CALC_TIMER_INT
        STX ST_T_LOW
        STY ST_T_HIGH

        JSR ST_INIT_PWM ; Set up the interrupt machinery for OC3
        CLI             ; Enable global interrupts, PWM is running
LOOP    BRA LOOP        ; Loop here forever

*-------------------------------------------------------------------
*                 Calculate Timer Intervals

* Passed: Duty cycle D in ACCA
*         Motor waveform period Tm (microsec) in variable 'MOTOR_PERIOD'
* Returns: T_High in Y index register
*          T_Low in X index register

* Algorithm: T_High = D * Tm/100
*            T_Low  = Tm - T_high
*

CALC_TIMER_INT  PSHA            ; Save the duty cycle to the stack
                LDD #MOTOR_PERIOD   ; Calculate Tm/100
                LDX #100
                IDIV            ; Quotient Tm/100 is now in X

                LDAA #0         ; Get Duty Cycle D into Y
                PULB            ; by retrieving it from the stack
                XGDY

                JSR MUL16       ; Find T_High = D * Tm/100
                STX TEMP        ; Lower 16 bits are in X index register
*                               ; Upper 16 bits are in Y but can be ignored
*                               ;  since they are known to be zero
                LDD #MOTOR_PERIOD       ; Now calculate Tm-T_high
                SUBD TEMP       ;
                XGDX            ; putting the T_low result in X
                LDY TEMP        ; and the T_high result in Y
                RTS
*-------------------------------------------------------------------
*                 Duty Cycle Modulation: Initialization
*
* Note: PWM is 'Pulse Width Modulation', the same as 'duty cycle
* modulation'.
ST_INIT_PWM     LDAA #$7E       ; Set up the pseudo-interrupt jump instruction
                STAA TOCVEC3
                LDD #TOC3_ISR    ; to point at the TOC3 ISR
                STD TOCVEC3+1

                LDAA #OUTPUT_HIGH ; Initialize the state
                STAA ST_STATE

                LDAA #%00100000 ; Clear the timer compare flag
                STAA TFLG1      ; See pink book page 10-14

                LDAA TMSK1      ; Enable the TOC3 interrupt
                ORAA #%00100000
                STAA TMSK1

                RTS
```

```
*--------------------------------------------------------------------
*               Duty Cycle Modulation: Interrupt Service Routine
*
* This routine handles the timer interrupt for Output Compare channel 3.
* In the eebot, this is the starboard motor speed control pin.

* Algorithm:
*        Clear the timer flag.
*        If STATE == OUTPUT_HIGH then
*                STATE := OUTPUT_LOW
*                T := (T + T_LOW) MOD 2^16
*                TOCx  := T
*                OMx:OLx := 1:1 ;Transition on interrupt to output high
*                Return
*        If STATE == OUTPUT_LOW then
*                STATE := OUTPUT_LOW
*                T := (T + T_HIGH) MOD 2^16
*                TOCx  := T
*                OM3:OL3 := 1:0 ;Transition on interrupt to output low
*                Return
*        Else                   ; Illegal state!
*                Set error flag in RAM
*                Loop here forever

TOC3_ISR        LDAA #%00100000 ; Clear the timer compare flag
                STAA TFLG1      ; See pink book page 10-14

                LDAA ST_STATE           ; If the state
                CMPA #OUTPUT_HIGH       ; is OUTPUT_HIGH then
                BNE TOC3_ISR_1

                LDAA #OUTPUT_LOW        ; set state to output_low
                STAA ST_STATE

                LDD ST_TIME_TOTAL       ; Calculate the next event time
                ADDD ST_T_LOW
                STD ST_TIME_TOTAL       ; Update the running total
                STD TOC3                ; and set up the compare register
                LDAA TCTL1              ; Transition on interrupt to high
                ORAA #%00110000         ; Set both OM3 and OL3
                STAA TCTL1

                BRA TOC3_ISR_EXIT       ; Exit

TOC3_ISR_1      CMPA #OUTPUT_LOW        ; Else if the state is OUTPUT_LOW
                BNE TOC3_ISR_2

                LDAA #OUTPUT_HIGH       ; set the state to output_high
                STAA ST_STATE

                LDD ST_TIME_TOTAL       ; Calculate the next event time
                ADDD ST_T_HIGH
                STD ST_TIME_TOTAL       ; Update the running total
                STD TOC3                ; and set up the compare register

                LDAA TCTL1              ; Transition on interrupt to low
                ORAA #%00010000         ; Set bit OM3
                ANDA #%11101111         ; Clear bit OL3
                STAA TCTL1

                BRA TOC3_ISR_EXIT        ; Exit

TOC3_ISR_2      LDAA ERROR_REG          ; Else state is in error
                ORAA #TOC3_BADSTATE     ; so set the bit
                STAA ERROR_REG
                SWI                     ; and break to the monitor

TOC3_ISR_EXIT   RTI
```

```
*-------------------------------------------------------------------
*                      Double Precision Multiply

* This routine accepts two 16 bit operands and generates a 32 bit result.

* Passed: Multiplier word in X index
*         Multiplicand word in Y index
* Returns: Result low word in X
*          Result high word in Y

* Algorithm by analogy:
*       27
*     * 68
*     ----
*       56 Product     * multiply low digits
*      16  Product     * first cross multiply
*     ----
*      216 Partial sum
*      42  Product     * second cross multiply
*     12   Product     * multiply high digits
*     ----
*     1836 Final sum

MUL16   STX OP1_HI      ; Set up the multiplier in work area
        STY OP2_HI      ; Set up the multiplicand in the work area

        CLR RES_HI      ; Clear the result registers
        CLR RES_NSB2
        CLR RES_NSB1
        CLR RES_LO

        LDAA OP1_LO     ; Multiply the low bytes
        LDAB OP2_LO
        MUL             ; Result is in D
        STD RES_NSB1    ; and save the result in the low 16 bit reg

        LDAA OP1_LO     ; First cross multiply
        LDAB OP2_HI
        MUL             ; Result is in D
        ADDD RES_NSB2   ; Find partial sum
        STD RES_NSB2
        BCC MUL16_1     ; If the add generated a carry
        INC RES_HI      ;  increment the high byte

MUL16_1 LDAA OP1_HI     ; Second cross multiply
        LDAB OP2_LO
        MUL             ; Result is in D
        ADDD RES_NSB2   ; Find partial sum
        STD RES_NSB2
        BCC MUL16_2     ; If the add generated a carry
        INC RES_HI      ;  increment the high byte

MUL16_2 LDAA OP1_HI     ; Multiply the high bytes
        LDAB OP2_HI
        MUL             ; Result is in D
        ADDD RES_HI     ; Add the two upper bytes
        STD  RES_HI

        LDX RES_NSB1    ; Move the low word to X
        LDY RES_HI      ; Move the high byte to Y

        RTS
```