Final Project

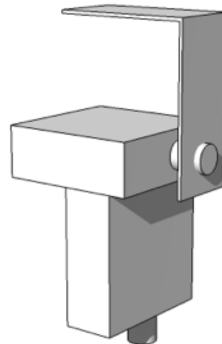
Chad Huard :: ee6681@wayne.edu :: 5/1/10

1. Description:

This report describes the use of the Motorola 68HC11 microprocessor (HC11) to control a 2 degree of freedom servo 'arm' based on input from an accelerometer. In order to encourage user interaction with the project, I have attached a laser pointer to the servo arm and implemented a light sensor that sounds a buzzer when the laser is correctly aimed at the sensor. All code is written in Assembly.

The HC11 microprocessor is mounted in Wytec's EVBPlus evaluation board (EVB). This board provides a simple way to program the microprocessor from a computer via serial connection and also gives the chip access to various peripheral devices. The on-board peripherals used in the project include: PC0 button, PE7 trim pot, PA5 buzzer and the array of 7-segment displays.

The accelerometer is Freescale's MMA7260QT. The accelerometer IC is mounted on a Sparkfun Electronics breakout board to simplify implementation. This accelerometer requires 3.3V power while the evaluation board operates on 5V, so the accelerometer is placed on a separate breadboard along with a basic 3.3V regulator drawing power from the EVB. The X and Y outputs are connected to PE4 and PE5 by jumpers long enough to allow the user to tilt the entire breadboard in space to control the arm. These outputs are analog signals which are processed by the HC11's analog to digital converter (ADC).



The arm is constructed of two TowerPro SG-5010 servo motors mounted at 90° to one another. The bottom servo's shaft is attached to a base, and the top servo is attached to a yoke. This arrangement gives two degrees of freedom allowing the yoke to rotate through all 3 axis. The laser pointer is mounted on top of the yoke, allowing the user to point it ±45° up, down, left or right from its center position. The power supply on the EVB can not provide enough current to move the servo quickly, so the servos require their own dedicated power supply provided by a separate 5V 'wall-wart'. The bottom servo is connected to PA3 and the upper servo is driven by PA4.

2. Project Goals:

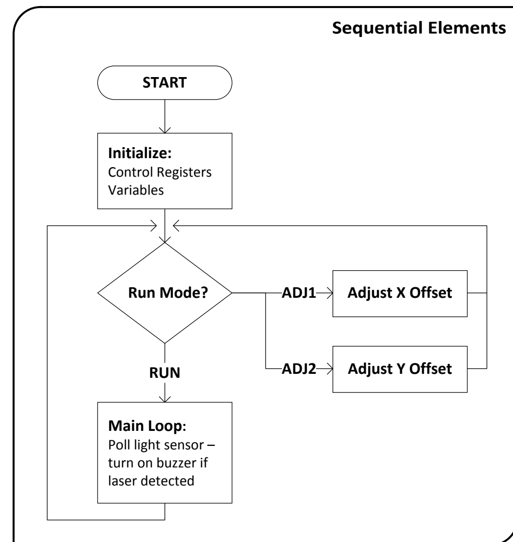
The main goal of this project is to demonstrate a working knowledge of Assembly language, the HC11 and its architecture. My specific objective goals for this project are:

- Accurately measure and calibrate input from accelerometer
- Smoothly and accurately control servo motors based on accelerometer input
- Produce clean and modular assembly code

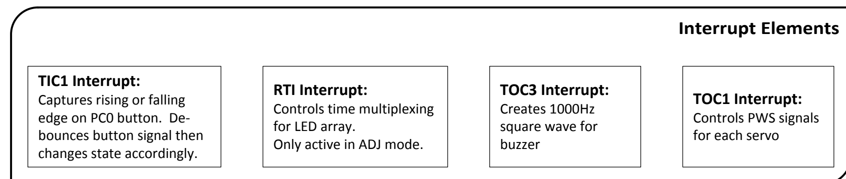
3. Solution:

3.1. Overview:

In order for this system to function, the microprocessor must handle many time sensitive functions simultaneously. The servos require a very specific pulse width modulation (PWM) signal, the buzzer requires a square wave output and the 7-segment displays require accurate time multiplexing. Because of these requirements, the program relies heavily on the HC11's interrupt system. Specifically, this program uses four interrupts: TIC1, RTI, TOC3 and TOC1. Using the interrupts allows each of the functions to be turned on or off with minimal influence on the other timing systems.



Since most of the functionality is handled by interrupts, the main loop of the program is quite simple. There are three 'modes' the system can be in: RUN, ADJ1 and ADJ2. RUN mode is the standard mode where the servo arm reacts to the input of the accelerometer. In this mode, the only thing that is done by the main loop is the monitoring of the light sensor to detect the laser. The calibration modes ADJ1 and ADJ2 allow the user to set the zero point of the arm when the accelerometer sensor his held flat. In these modes the main loop becomes responsible for polling the analog to digital converter and converting the reading to an offset for one of the two servos. ADJ1 adjusts the bottom servo, and ADJ2 adjusts the upper servo. Switching from mode to mode is accomplished by pressing the PC0 button on the EVB.



3.2. Input Processing:

The accelerometer provides an analog output based on the force of acceleration felt by the chip in each of the three axis. This system utilizes only the X and Y axis measurements, ignoring the Z axis. The output voltage from the accelerometer is between 0.8V and 2.3V, and represents the tilt of the sensor in the X or Y axis. 0.8V represents a 90° tilt in the negative direction, 2.3V is 90° in the positive direction and 1.55V is the output at rest horizontally. This voltage is read and converted to a digital value by the ADC in the HC11. Both X and Y signals are converted simultaneously and continuously by using the ADC in MULTI and SCAN modes.

The signal provided by the accelerometer is fairly noisy when being held in hand because small rapid movements can account for fairly large accelerations, often on the order of the force of gravity. Most of these movement based accelerations are brief and fairly random in direction, so they can be smoothed out using signal processing (leaving a good approximation to the tilt of the sensor due to the force of gravity).

Since the servo motors operate on a signal with a 20ms period, it is impossible to update their position more frequently than that; therefore, it is counterproductive to read the accelerometer input more frequently than once every 20ms. The result of the analog to digital conversion is an 8-bit digital value. This reading from the ADC is then converted to a delay by using a lookup table. This value of this delay is between 2000 and 4000, corresponding to the number of clock

cycles in the pulse width of the servo PWM signal. Once the delay is found it is stored in the an array of 8 measurements, the system uses the average of these 8 most recent measurements as the actual delay. This rolling average serves the dual purpose of smoothing noise and interpolating between values in the lookup table.

The last element in the signal processing as a maximum change in delay. Once the average delay has been calculated it is checked against the previous pulse's delay. If the difference between delays is more than 100µs then the old delay ±100µs is used instead of the new delay reading. This limits the amount the servo is asked to move in a single pulse cycle and smooths the motion when the accelerometer is being moved rapidly.

3.3. PWM Output Processing:

In order to control the position of the servos the microprocessor must provide a PWM signal with a period of 20ms and a pulse width ranging from 1ms to 2ms. A pulse width of 1ms correlates to full left rotation in the servo, 2ms is full right and 1.5ms centers the servo. This signal is created using the TOC1 interrupt and the TOC4 and TOC5 output compare registers, one for each servo. The TOC1 interrupts every 20ms and sets both PA4 (TOC4) and PA3 (TOC5) to high, creating the period of the signal. The service routine then loads the appropriate delay into TOC4 for one servo and TOC5 for the other. When the TOC4 or TOC5 output compare occurs the system pulls the respective pin low, setting the pulse width of the PWM signal, but no interrupt is called. The distinction between the use of TOC1 and the two output compare registers TOC4 and TOC5 is critical. TOC1 is used as an interrupt and is responsible for the leading edge of the PWM signal, creating the period. The two other output compare registers TOC4 and TOC5 do not interrupt; they are used to automatically clear their respective pins based on the delay settings derived from the accelerometer input, setting the pulse width. The TOC1 interrupt service routine is also responsible for calling the sub-routines that actually read the input from the accelerometer and calculate the delays to ensure that the input is read exactly once per PWM cycle.

3.4. Calibration Mode:

The calibration modes (ADJ1 and ADJ2) are accessed by pressing the PC0 button on the EVB. This button is connected to PC0 by a trace on the board, but in this system it is also connected to PA2 using a jumper from PC0 to PA2. TIC1 is configured to interrupt on a rising or falling edge on PA2, detecting a button press or release from the PC0 button. One millisecond after the interrupt occurs PC0 is polled to make sure the button state has actually changed and was not just bounce. This technique allows for de-bouncing the rising and falling edge of the button press. De-bouncing the button release was found to be critical in avoiding false button presses.

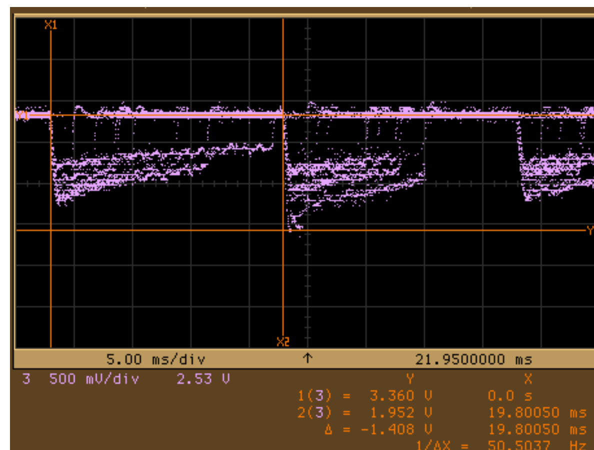
After detecting and confirming a button press the interrupt service routine increments the run mode (RUN → ADJ1 → ADJ2 → RUN) so when the program returns to the main loop it is in the new mode. If the mode is ADJ1 or ADJ2 the ADC is re-configured to read PE7 instead of the accelerometer. Each time through the main loop the trim pot on PE7 is polled and its value is used as an 'offset' for the servo position. In calibration mode the reading from PE7 is also displayed in hex on the 7-segment displays. Time multiplexing of the 7-segment displays is achieved by using the RTI interrupt to cycle through the displays.

4. Discussion:

4.1. Accelerometer

4.1.1. Power

Getting the accelerometer to operate properly and then making the signal useful were the most challenging parts of this project. In my first attempt at operating the accelerometer I built the regulated 3.3V power supply and ran it on the same wall wart as I was using for the servo motors. This created a problem when the servos rotated. If the servo rotation was even remotely fast they would draw too much current thus dropping the voltage delivered to the accelerometer. Since the accelerometer output is the ratio of its input voltage to the force of acceleration, changing the input voltage directly affects the output. This quick change in output led to the servos repositioning even faster and into a vicious cycle that required the board to be reset. Below is an example of the accelerometer's input voltage during this phenomenon:



This shows that the input voltage could drop as much as 1.4V causing dramatically wrong readings and continuing the vicious cycle. I resolved this issue by driving the sensor from the EVB power supply and using the extra wall wart only for the servos

4.1.2. Resolution

Another major problem I encountered with the accelerometer was resolution. The output of the accelerometer has a 1.5V range and the ADC reads voltages in the range of 0V to 5V. This problem could be partially addressed by using the VRH and VRL pins on the HC11, but after researching the EVB I found that they are missing the jumper that would allow these pins to be used. The way our EVB is configured the VRH is always 5V and VRL is always 0V. In order to change the configuration you must reposition the jumper J11, which is not installed on our board, eliminating the possibility of using the ADC with a smaller voltage range.

The HC11's analog to digital converter produces an 8-bit result, but since we are only using 1.5V of the 5V that make up the 8-bit range we get:

$$\frac{5\text{V}}{256} = 0.0195 \text{ Volts / Division}$$

$$\frac{1.5\text{V}}{0.0195 \text{ V/D}} = 76 \text{ Divisions}$$

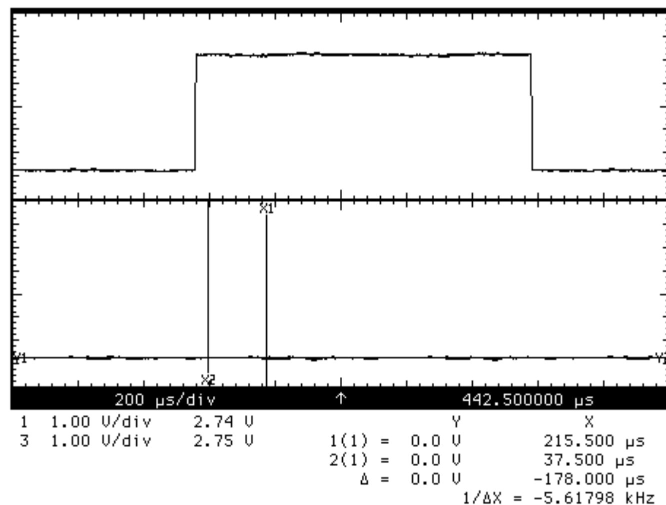
$$2^n = 76 \quad \Rightarrow \quad n = 6.25 \text{ Effective Bits}$$

With only 6.25 effective bits resolution I found that the smallest increment of motion in the servo as quite large. This made it very difficult to accurately aim the laser mounted on the arm at the light sensor. When I discovered this I was already using a rolling average to smooth the input signal (as described in the solution section) but I was averaging the 8-bit ADC readings. I found that if I converted to the 16-bit delay before taking the average I could smooth the signal and interpolate between readings, thus raising the effective resolution to 9.25-bit. This creates much smoother motion in the servos and makes accurate aiming much easier.

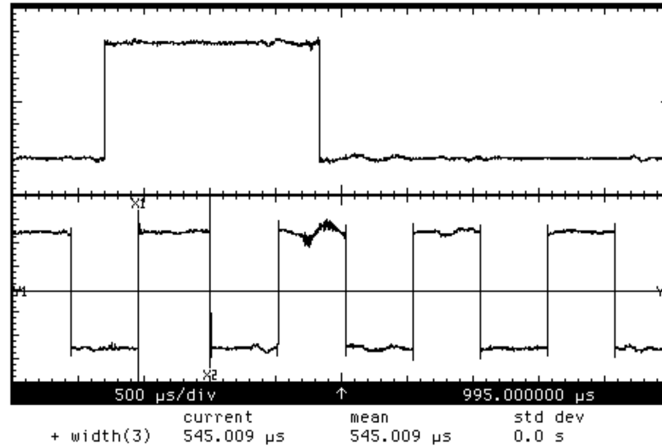
While the interpolation technique greatly increases smoothness and accuracy, this system still provides borderline performance in an absolute sense. The servo motors are able to move in much smaller increments than the system is able to resolve. A much better solution in the long term would be to use a different accelerometer with a digital output that would communicate with the HC11 over a serial bus (SPI). Such devices are capable of 12-bit resolution without the use of interpolation and would produce much smoother and more accurate motion.

4.2. Timing

Timing issues turned out to be a major concern in this system. In the solution section I discussed that interrupts were used for most of the time critical functions. Unfortunately, there are so many different timing systems all working simultaneously that some interrupts actually begin to overlap causing timing errors. Below is the timing of the PWM signal and its interrupt routine:



The top trace is the actual pulse width of the PWM signal. The markers on the bottom trace shows the moment when the reading is taken from the ADC (X2) and when the TOC1 interrupt service routine ends (X1). The timing markers were created by clearing PORTB causing a STRB which was monitored by the scope. This in and of itself does not indicate any problem, but the following graph shows the pulse width along with the square wave driving the buzzer:

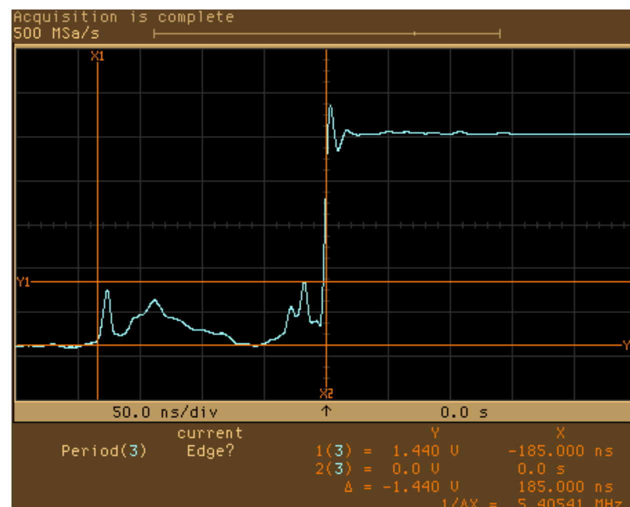


Here you can see that the RTI interrupt (which is responsible for toggling the buzzer pin) is occurring while the program is still inside the TOC1 interrupt routine. Because of this, the pulse between the markers X1 and X2 is 545µs, while all of the other pulse widths are 512µs. This anomalous pulse occurs once every PWM pulse, or 20ms, causing the buzzer to sound slightly harsh instead of having a clean sine wave tone.

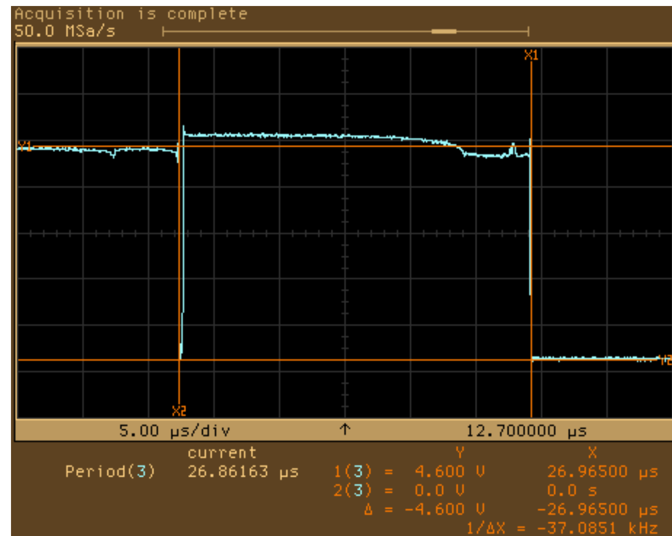
After careful re-examination of my code I found a simple solution to these timing errors. My initial code read the TCNT register upon entering each of the timing interrupts; therefore, if the interrupt was entered 50µs late because the board was executing a long service routine the new pulse would be 50µs too long. This is exactly what was happening with the buzzer. The simple solution is to not use the TCNT at all, but load the previous value of the timing register itself. Using this method, even if the interrupt is called 50µs late the pulse duration is still correct. It is such an obvious solution that it seems like it should have never been a problem in the first place, but sometimes the most obvious solutions are the hardest to find.

4.3. Button De-Bounce

De-bouncing mechanical switches and buttons is a necessity in any design, since all such devices will exhibit some amount of bounce. In class we explored a few techniques for de-bouncing using delays and flip-flops, but I found that these techniques were not enough. The techniques described in class were for de-bouncing the leading edge of a button press, but I found that the trailing edge was just as frequently to blame for false button readings. Here is the leading edge of the button:



Out of ten trials this was the worst bounce I captured. Notice that the ΔT is only 185ns which is significantly faster than the HC11 could possibly re-read the switch. Here is the glitch I found to be fairly common (aprox. 10% of button presses) on the trailing edge:



The glitch is about 27µs before the final falling edge, which is long enough for the microprocessor to consider the plateau after the glitch to be a second button press. I de-bounced the button by using a variable to keep track of the button state (up or down). When the TIC1 interrupt occurs, indicating that the state has changed, the button state is not changed until the button is polled on PC0 1ms later. Only if the button remains in its new position after the 1ms delay does the button state variable change and the program run mode is incremented. This technique solves the de-bouncing problem completely for both the rising and falling edge.

5. Conclusion:

This project was challenging and informative. This project required viewing the Motorola 68HC11 as a true micro-controller as opposed to simply a microprocessor. Almost all of the HC11 sub-systems are pressed into service with the exception of the serial port. Throughout the course of this project I successfully used techniques from every single unit of the class, from parallel I/O for the 7-segment displays, to analog to digital conversion for the accelerometer, to interrupts and timing port registers for the creation of the PWM signals.

Evaluating this project on the basis of the project goals I find it to be quite successful. Within the constraints of the EVB and accelerometer chosen the servo arm is quick to respond to user input and its motion is smooth. This project greatly expanded my knowledge of embedded design and was therefore quite successful in that regard as well.

6. Appendix: Code

6.1. MAIN.asm

```
*============================================
* Accelerometer Controled 2-DOF Servo 'Arm'
* By: Chad Huard  4/30/10  ECE2620
*
* Pin Hookups:
* PC0 --> PA2
* PA3 --> Yellow Servo
* PA4 --> Blue Servo
* PE5 --> X Accel (yellow)
* PE4 --> Y Accel (blue)
*============================================
#include            "EQU.asm"

            ORG     $C000
#include            "PWM_VAR.asm"
#include            "ADC_VAR.asm"
#include            "GEN_VAR.asm"



            LDS     #$8FFF
            JSR     MAIN_INIT
            JSR     PWM_INIT
            CLI


*============================================
* MAIN PROGRAM
*============================================
LOOP        LDAA    R_STATE
            BEQ     RUN         if in Run mode, skip the ADC

* Calibration Mode:
* Load the value of ADR1 and change the value of
* the offsets:
********************************************
            LDAA    ADR1
            LDX     ADJ_PTR
            STAA    0,X

            TAB
            LSRA
            LSRA
            LSRA
            LSRA
            ANDB    #$0F
            JSR     SEV_SEG
            STAA    LED+1
            STAB    LED

* Run Mode:
* Check to see if AR3 (light sensor) has
* dropped below threshold:
********************************************
RUN         LDX     #$1000
            LDAA    ADR3            read light sensor
            CMPA    #LT_THRESH
            BHI     HIGH
            BSET    TCTL1_OFF,X     %00010000
            BRA     LOOP
```

```
HIGH         BCLR    TCTL1_OFF,X     %00010000
             BRA     LOOP


*============================================
* FUNCTION INCLUDES
*============================================
#include             "PWM_FUNC.asm"
#include             "ADC_FUNC.asm"
#include             "GEN_FUNC.asm"

#include             "PWM_JMP.asm"
#include             "GEN_JMP.asm"
                     END
```

6.2. PWM_VAR.ams

```
*==========================================
* PWM VARIABLES:
*==========================================

DELAY1       FDB     3000
DELAY2       FDB     3000
FREQ         EQU     $9C40
HI_LIM1      FDB     0
LOW_LIM1     FDB     0
HI_LIM2      FDB     0
LOW_LIM2     FDB     0
```

6.3. ADC_VAR.asm

```
*============================================
* Variables for ADC_FUNC
*============================================

X_AVE        RMB     2       These are actually delays, not readings
Y_AVE        RMB     2
X_PREV       RMB     2       This is used to store the current location in the
Y_PREV       RMB     2       array, so it can be changed from theinterrupt
X_ARRAY      RMB     16
Y_ARRAY      RMB     16
```

6.4. GEN_VAR.asm

```
*============================================
* Variables for GEN_FUNC
*============================================

LED          FCB     $06,$4F,%01001000,$66
LED_DISP     FCB     %00111000,%00110100,%00101100,%00011100
LED_CUR      FCB     3
B_STATE      FCB     $00     $80=DOWN    $00=UP
R_STATE      FCB     $00     $00=NORMAL  $80=ADJ1    $81=ADJ2
ADJ_PTR      FDB     $0000
X_OFFSET     FCB     $00
Y_OFFSET     FCB     $00
SEV_TEMP     RMB     2
LT_THRESH    EQU     10
```

6.5. PWM_FUNC.asm

```
*////////////////////////////////////////////////
*//                                           //
*// PWM FUNCTIONS:                            //
*//                                           //
*////////////////////////////////////////////////

*============================================
* PWM:
*============================================
* This function handles converting the
* average readings to delays using the
* delay table.  It accesses the averages
* directly from memory.
*
* Delay times are not allowed to change
* by more than 100 clock cycles in each
* pwm pulse
*============================================
PWM:        PSHX
            PSHY

*////////////////////////////////////////////////
*// X Delay:                                  //
*////////////////////////////////////////////////
********************************************
* Set High and Low limits:
********************************************
            LDD     DELAY2
            ADDD    #200
            STD     HI_LIM2
            SUBD    #400
            STD     LOW_LIM2

* Load value of X_AVE
********************************************
            LDD     X_AVE

* Check for high or low:
********************************************
HI_DLY2:    CPD     HI_LIM2
            BLO     LOW_DLY2
            LDD     HI_LIM2
            STD     DELAY2

LOW_DLY2:   CPD     LOW_LIM2
            BHI     MID_DLY2
            LDD     LOW_LIM2
            STD     DELAY2

MID_DLY2:   STD     DELAY2


*////////////////////////////////////////////////
*// Y Delay:                                  //
*////////////////////////////////////////////////
********************************************
* Set High and Low limits:
********************************************
            LDD     DELAY1
            ADDD    #200
            STD     HI_LIM1
            SUBD    #400
            STD     LOW_LIM1
```

```
* Load value of Y_AVE
********************************************
            LDD     Y_AVE

* Check for high or low:
********************************************
HI_DLY1:    CPD     HI_LIM1
            BLO     LOW_DLY1
            LDD     HI_LIM1
            STD     DELAY1

LOW_DLY1:   CPD     LOW_LIM1
            BHI     MID_DLY1
            LDD     LOW_LIM1
            STD     DELAY1

MID_DLY1:   STD     DELAY1


            PULY
            PULX
            RTS

*============================================
* INITALIZE TIMERS FOR PWM
*============================================

* Configure toc4-toc5 to clear on flag:
PWM_INIT    LDAA    #%00001010
            STAA    TCTL1

* Configure toc1 to set toc4-toc5:
            LDAA    #%00011000
            STAA    OC1M
            STAA    OC1D

            RTS

*============================================
* OC1 INTERRUPT ROUTINE:
*============================================
* This routine controls the actual pulse
* widths.  It is also used as a timer to call
* the get_accel and pwm subroutines so they
* only happen once every pulse cycle (otherwise
* the processor updates the delay hundreds of
* times between each pulse making the averaging
* ineffective)
*============================================

TOC1_I:     LDD     TOC1
            ADDD    DELAY1
            ADDB    X_OFFSET
            ADCA    #0
            STD     TOC4
            SUBD    DELAY1
            SUBB    X_OFFSET
            SBCA    #0
            ADDD    DELAY2
            ADDB    Y_OFFSET
            ADCA    #0
            STD     TOC5
            SUBD    DELAY2
```

```
            SUBB    Y_OFFSET
            SBCA    #0
            ADDD    #FREQ
            STD     TOC1
            LDAA    #%10000000
            STAA    TFLG1
            LDAA    R_STATE
            BNE     NO_ACCEL        skip GET_ACCEL if not in 'run' mode
            JSR     GET_ACCEL
NO_ACCEL    JSR     PWM
            RTI

*===========================================
* DELAY TABLE:
*===========================================
#include    "DELAY_TBL.asm"
```

6.6. ADC_FUNC.asm

```
*====================================
* A/D Conversion on PE4 and PE5:
*====================================
* Each measurement is converted to a
* delay time using the lookup table
* then stored in an array.  The array
* is then used to take a rolling average
* and the average delay stored in
* memory.
*====================================

* Get reading from ADC and store in array:
*******************************************
GET_ACCEL   LDX     X_PREV      Called from interrupt, so get previous value
            LDY     #DELAY_TBL
            CLRA
            LDAB    ADR1
            ABY
            ABY
            LDD     0,Y
            STD     0,X
            INX
            INX
            CMPX    #X_ARRAY+16
            BNE     NO_RE_X
            LDX     #X_ARRAY
NO_RE_X     STX     X_PREV      Store place in array for next time

            LDX     Y_PREV
            LDY     #DELAY_TBL
            CLRA
            LDAB    ADR2
            ABY
            ABY
            LDD     0,Y
            STD     0,X
            INX
            INX
            CMPX    #Y_ARRAY+16
            BNE     NO_RE_Y
            LDX     #Y_ARRAY
NO_RE_Y     STX     Y_PREV

* Find average values of X_ARRAY and Y_ARRAY:
*******************************************
```

```
            LDX     #X_ARRAY
            LDD     0,X
            INX
            INX
            ADDD    0,X
            INX
            INX
            ADDD    0,X
            INX
            INX
            ADDD    0,X
            INX
            INX
            ADDD    0,X
            INX
            INX
            ADDD    0,X
            INX
            INX
            ADDD    0,X
            INX
            INX
            ADDD    0,X
            LSRD
            LSRD
            LSRD
            STD     X_AVE

            LDX     #Y_ARRAY
            LDD     0,X
            INX
            INX
            ADDD    0,X
            INX
            INX
            ADDD    0,X
            INX
            INX
            ADDD    0,X
            INX
            INX
            ADDD    0,X
            INX
            INX
            ADDD    0,X
            INX
            INX
            ADDD    0,X
            INX
            INX
            ADDD    0,X
            LSRD
            LSRD
            LSRD
            STD     Y_AVE

            RTS
```

6.7. GEN_FUNC.asm

```
*============================================
* Main initialization and config of registers
*============================================
MAIN_INIT   LDX     #Y_OFFSET
```

```
            STX     ADJ_PTR
            LDX     #X_ARRAY
            LDY     #Y_ARRAY
            STX     X_PREV
            STY     Y_PREV

* Config Port D for output to LEDs
*============================================
            LDAA    #%00111100
            STAA    DDRD

* Config TIC1 AND TMSK1:
*============================================
            LDAA    #%00110000
            STAA    TCTL2
            LDAA    #%10100100          SET TIC1 TO CAPTURE ANY EDGE
            STAA    TMSK1               TURN ON INPUT COMPARE


* TURN ON A/D CONVERTER:
*============================================
            LDAA    OPTION
            ORAA    #%10000000
            STAA    OPTION
            LDAA    #%00110100      SCAN=ON  MULTI=ON
            STAA    ADCTL
            RTS

*============================================
* Seven segment display lookup:
*============================================
* Convert binary nibble to its seven segment
* display code.
*============================================
SEV_SEG     PSHX
            PSHY
            STAA    SEV_TEMP
            STAB    SEV_TEMP+1
            LDD     #SEV_LOOKUP
            ADDB    SEV_TEMP
            ADCA    #0
            XGDX
            LDD     #SEV_LOOKUP
            ADDB    SEV_TEMP+1
            ADCA    #0
            XGDY
            LDAA    0,X
            LDAB    0,Y
            PULY
            PULX
            RTS


*============================================
* Button Handling / Run States:
*============================================
* To switch between modes the PC0 button on
* the EVB is used.  It is connected to the
* PA2 pin so the input compare register can be
* used as an interrupt.
*
* There are three run states:
* R_STATE=$00:  Run. Standard mode positions
*               arm based on accelerometer
```

```
* R_STATE=$80:  ADJ1. Adjust X offset value
* R_STATE=$81:  ADJ2. Adjust Y offset value
*============================================

*============================================
* 1ms Delay Routine
*============================================
DELAY_1MS   PSHY
            LDY     #200
LOOP_1MS    DEY
            BNE     LOOP_1MS
            PULY
            RTS

*============================================
* Down --> Up
* Change B_STATE to UP ($80)
*============================================
DN2UP:      CLR     B_STATE
            RTS

*============================================
* Up --> Down
* Turn on LEDs & change B_STATE to down ($00)
*============================================
UP2DN:      PSHY

* Change Button state:
            LDAA    #$80
            STAA    B_STATE

* Change Run state:
            LDAA    R_STATE
            BNE     NOT_NORM

NORM:       LDAA    #$80
            STAA    R_STATE
            JSR     CFG_ADJ1
            PULY
            RTS

NOT_NORM:   LSRA
            BCS     ADJ2

ADJ1:       LDAA    #$81
            STAA    R_STATE
            JSR     CFG_ADJ2
            PULY
            RTS

ADJ2:       CLR     R_STATE
            JSR     CFG_NORM
            PULY
            RTS

*============================================
* Configure ADJ1
* Set up board to adjust x_offset
*============================================
CFG_ADJ1:   PSHX
            LDX     #$1000

* Turn on LEDs:
            BSET    TMSK2_OFF,X,%01000000
```

```
* Configure A/D:
            LDAA    #%00100111
            STAA    ADCTL

* Set pointer to X offset:
            LDX     #X_OFFSET
            STX     ADJ_PTR

* Configure Display:
            LDAA    #$06        character '1'
            STAA    LED+3

            PULX
            RTS

*============================================
* Configure ADJ2
* Set up board to adjust Y_offset
*============================================
CFG_ADJ2:   PSHX

* Set pointer to Y offset:
            LDX     #Y_OFFSET
            STX     ADJ_PTR

* Configure Display:
            LDAA    #$5b        character '2'
            STAA    LED+3

            PULX
            RTS

*============================================
* Configure NORM
* Set up board for normal run
*============================================
CFG_NORM:   PSHX
            LDX     #$1000

* Turn OFF LEDs:
            BCLR    TMSK2_OFF,X,%01000000

* Configure A/D:
            LDAA    #%00110100
            STAA    ADCTL

            CLR     PORTB

            PULX
            RTS

*============================================
* Button Interrupt Service Routine
*============================================
BUTTON_I    LDAA    B_STATE
            BMI     DOWN
UP:         JSR     DELAY_1MS
            LDAA    PORTC
            LSRA
            BCS     B_END
            JSR     UP2DN
            BRA     B_END
```

```
DOWN:       JSR     DELAY_1MS
            LDAA    PORTC
            LSRA
            BCC     B_END
            JSR     DN2UP


B_END:      LDX     #$1000
            BSET    TFLG1_OFF,X,%00000100
            RTI


*===========================================
* LED Interrupt Service Routine
*===========================================
* LEDs use time multiplexing based on RTI
* interrupts.  This service routine switches
* which LED is illuminated and changes PORTB
* to display the appropriate digit.
*===========================================
LED_I       LDD     #LED
            ADDB    LED_CUR
            ADCA    #0
            XGDX
            LDD     #LED_DISP
            ADDB    LED_CUR
            ADCA    #0
            XGDY
            LDAA    0,Y
            LDAB    0,X
            CLR     PORTB
            STAA    PORTD
            STAB    PORTB
            DEC     LED_CUR
            BPL     END_LED
            LDAA    #3
            STAA    LED_CUR
END_LED     LDAA    #%01000000
            STAA    TFLG2
            RTI


*===========================================
* Buzzer Interrupt Service Routine
*===========================================
* The buzzer uses an output compare timer to
* create a square wave of 1000Hz
*===========================================
TOC3_I      LDX     #$1000
            LDD     TOC3
            ADDD    #1000
            STD     TOC3
            BSET    TFLG1_OFF,X     %00100000
            RTI
```

6.8. PWM_JMP.asm

```
*===========================================
* PWM INTERRUPT JUMP TABLE:
*===========================================

            ORG $00DF
            JMP TOC1_I
```

6.9. GEN_JMP.asm

```
*============================================
* JUMP VECTORS:
*============================================
            ORG     $00EB
            JMP     LED_I

            ORG     $00E8
            JMP     BUTTON_I

            ORG     $00D9
            JMP     TOC3_I
```