University of Florida

Intelligent Machine Design Lab

Experminental Platform for the Control of Unstable Systems, EPCUS: An Inverted Pendulum

By: Randy Wells

Date: 6 December 2000

Table of Contents

1.0 Abstract

This report describes the construction of an experimental platform for the control of unstable systems, named EPCUS. This will be realized through the stabilization of an inverted pendulum using a Motorola 68HC11 microcontroller. A full state feedback controller was designed to output a pulse width modulated signal to a DC motor controller, which will in turn apply a tractive force to drive a cart on wheels forward and reverse to maintain the upright position of the pendulum as well as maintain the cart at its original position. The system will be able to detect when the pendulum cannot be stabilized through the use of bump switches as well as detect obstacles in its path using infrared emitters and receivers. Data will be collected into external RAM for later performance analysis in MATLAB.

2.0 Acknowledgement

I would like to thank Dr. Arroyo, Rand Chandler, and Scott Nortman for their valuable advice and expertise, Jeremy Anderson for helping me trouble shoot the electronic speed control, Dr. Gloria Wiens for the use of her HP digital scope and lab space to setup my lathe / mill, and Chris Hall and Walker Racing for supplying the 3 rotary potentiometers

3.0 Introduction

The inverted pendulum is a common non-linear, unstable mechanism often used to experiment with new control techniques. This is a common mechanism because it has similar dynamics to many physical systems. One such system is a rocket during launch where the vectored thrust of the rocket nozzle aims to keep the rocket upright or pointed in a desired direction. This project is intended as a platform to implement different control strategies for the stabilization of the pendulum as well as the position control of the moving cart.

4.0 Integrated System

The Experimental Platform for the Control of Unstable Systems, or EPCUS, consists of a moving cart that carries the pendulum pivot. This cart is moved forward and reverse with a belt driven DC motor. Control is implemented using a Motorola 68hc11 microcontroller sensing pendulum angle and cart position. The system can detect when the pendulum can not be stabilized or is in danger of contacting an obstacle and react accordingly.

5.0 Mobile Platform

Success of this project relies on a quality mobile platform to ensure accurate, repeatable sensing and actuation. The platform consists of two aluminum rails connected by a plywood chassis plate as shown in Figure 1. This creates the base for the mounting of sensors and motors. Wheels, bearings, motor, batteries, electronic speed control and belt drive where taken from radio control cars. Wheel and pulley hubs, pendulum pivot, motor mount, and pivot sensor mount were machined out of aluminum.

Bearings were used on all rotating shafts to reduce friction as well as provide consistent motion. It was critical for the location of the axle bearings in the aluminum rails be equal distant in each rail to ensure that the axles are parallel and the cart will travel in a straight line.

The batteries are mounted below the chassis plate to leave room for mounting the electronic speed control, microcontroller board, and other electronics.

Mobile Platform

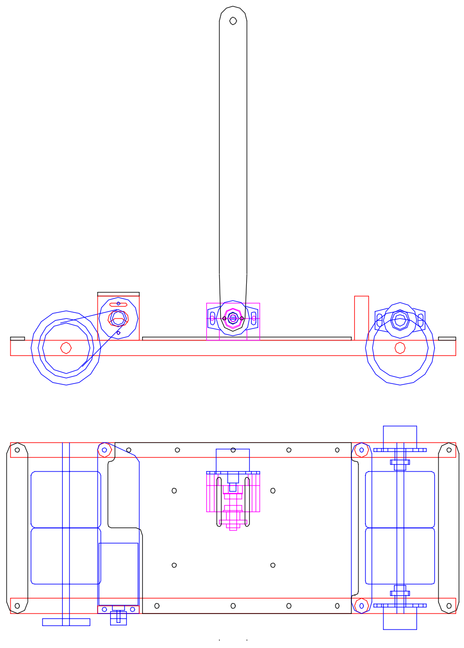

FIGURE 1.

6.0 Actuation

Actuation is implemented by a DC motor connected by belt drive to the rear axle. A pulse width modulated signal produced by the microcontroller is used by a hobby electronic speed control to adjust the voltage output to the motor. The electronic speed control uses the same input signal as a hobby servo. This pulse width modulated signal has a 20 ms. period and has a range of 1.0 ms. to 2.0 ms. high time for control.

6.1 Electronic Speed Control

A motor driver kit used previously in IMDL courses was first attempted but to the unavailability of some components, an alternate device was sought.

A Traxxas XL-1 electronic speed control was ultimately selected for its ability to output forward and reverse motion equally and a price of $40. Most electronic speed controls are designed primarily for forward motion with minimal reverse capabilities. This device has proven to be somewhat unreliable and has caused delays in development.

Interface to the HC11 microcontroller was done through output capture port OC2. Opto-isolator circuitry was used to isolate the microcontroller form the electronic speed control. Control of the output capture was performed with code modified from the Mekatronix servo routines as an output compare interrupt service routine. The code used follows below.

```
/***** TOC2_isr ********************************/
void TOC2_isr()
{

/* Keep the motor off if no duty cycle specified.*/
  if(duty_cycle == 0)
   {
    CLEAR_BIT(TCTL1, 0x40);
   }
  else
    if(TCTL1 & 0x40)
     {
      TOC2 += PERIOD_MIN+duty_cycle;  /* Keep pulse width up for
duty_cycle */
      CLEAR_BIT(TCTL1,0x40);  /*  Prepare to turn off next OC2 inter-
rupt*/
     }
    else
     {
      TOC2 += (PERIODM - (PERIOD_MIN + duty_cycle));
       SET_BIT(TCTL1,0x40);   /* Prepare to raise signal next OC2 intr. */
     }

CLEAR_FLAG(TFLG1,0x40);     /* Clear OC2F interrupt Flag */

}
/*******************************************/
```

6.2 Calibration

For control of the pendulum and cart, the relationship between speed control pulse width and tractive force at the wheel must be determined. This is determined experimentally. The cart was modeled by the 2nd order model shown by Equation 1.

$$m\ddot{x} + c\dot{x} = u \qquad \text{(EQ 1)}$$

Least squares was used to fit measured data produced by step inputs to the model. Least squares minimizes the squared error between the measurements and the model. The model represented by Equation 1 above can be factored into the form shown in Equation 2.

$$\begin{bmatrix} \ddot{x}_1 & \dot{x}_1 \\ \ddot{x}_2 & \dot{x}_2 \\ | & | \\ \ddot{x}_n & \dot{x}_n \end{bmatrix} \begin{bmatrix} m \\ c \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \\ | \\ u_n \end{bmatrix} \qquad \text{(EQ 2)}$$

In this representation, $\ddot{x}$ is the cart acceleration, $\dot{x}$ is the cart velocity, and u is the input to the speed control. For an arbitrary test input with n samples and representing Equation 2 as

$$Ax = b \qquad \text{(EQ 3)}$$

the vector of unknowns, x can be solved by

$$x = [A^T A]^{-1} A^T b \qquad \text{(EQ 4)}$$

The cart was commanded to perform a step function as shown in Figure 2.

Cart Step Response

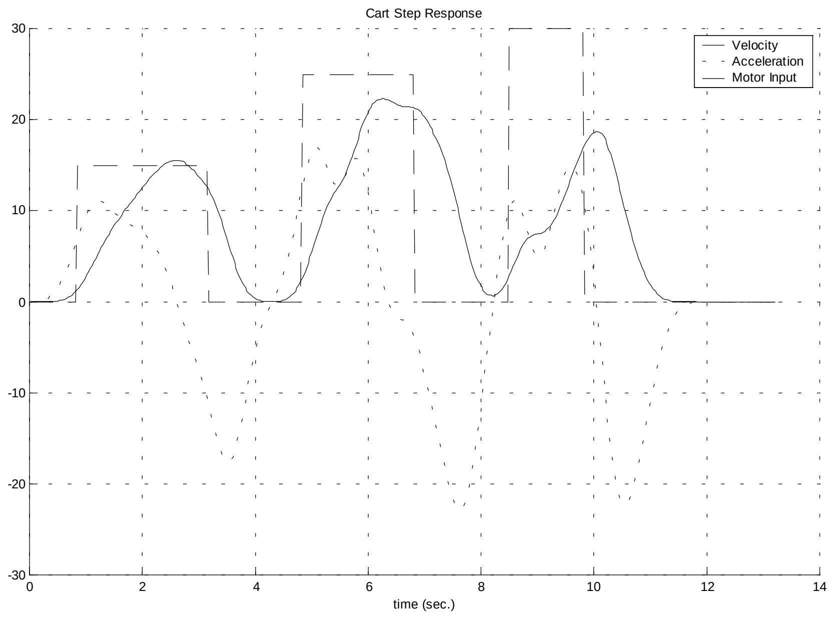

FIGURE 2.

Values of m and c were determined from repeated tests of this type. A value of 0.65 and 0.82 were determined for m and c respectively.

7.0 Sensors

7.1 Scope

Equally important as the mobile platform to the success of this project is electronic sensing. Repeatability and accuracy of the sensing will reduce problems associated with differentiating signals to calculate velocity and will reduce inaccurate feedback of the control system.

7.2 Infrared

Infrared emitters, modulated at 40 kHz, and receivers are used to detect obstacles in the path of the cart. The emitters were modulated using a MC74HC390 divider ic to divide the e clock output of the microcontroller. The infrared receivers have been hacked to output an analog signal that will approximate distance. This allows the obstacle avoidance threshold to be adjusted. The infrared emitters and detectors are mounted at the front and rear of the cart and point in the direction of travel.

7.3 Bump

"Momentary On" switches are interfaced with the digital inputs of PORT C of the microcontroller to sense when the pendulum has reached an angle that can not be stabilized. Reaching this angle, the pendulum will rest on a lever that will depress the bump switch as shown in Figure 3.

Bump Switches

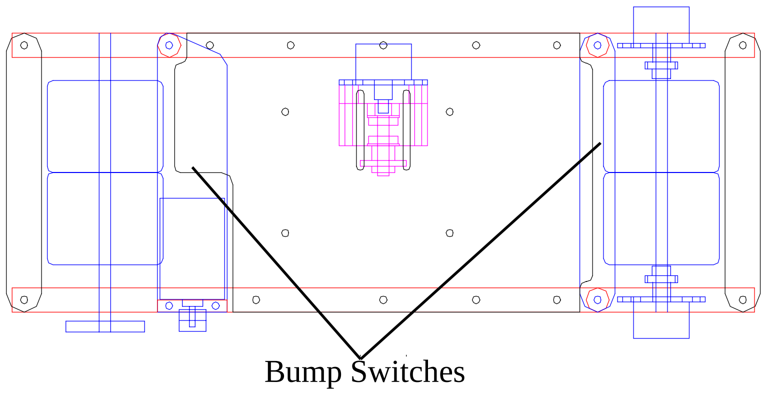

FIGURE 3.

7.4 Angular Potentiometer

Angular potentiometers were donated by Chris Hall of Walker Racing Inc. They were used to measure damper length through bell crank rotation and transmission selector position on a CART Champ race car. These sensors have reached their mileage limit for use on a CART Champ car but still perform to original specifications. These potentiometers measure rotation of 300 degrees. Each sensor will be interfaced with one of the eight analog inputs of the microcontroller. Typical sensor output can be found in Figure 4. This corresponds to 0.02093 radians / lsb.

.

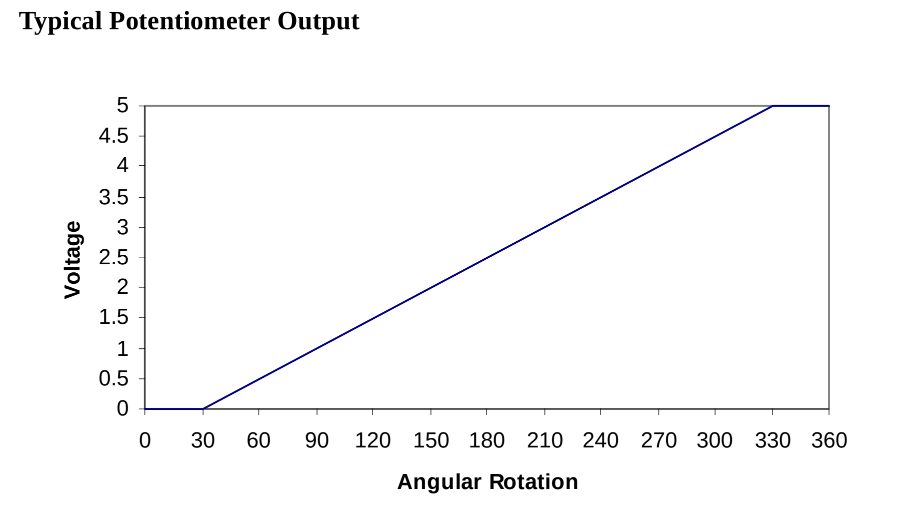

FIGURE 4.

7.4.1 Pendulum Angle

Pendulum angle is measured by directly connecting the pendulum pivot to the shaft of an angular potentiometer. The range of motion of the pendulum is approximately 150 degrees before the bump switches are activated. When connected to the analog port of the microcontroller this results in an output range from 0x64 to 0x179. This signal is differentiated to obtain pendulum angular velocity. Since the resolution of this measurement is limited, differentiating to obtain velocity causes problems by magnifying any measurement noise. A 4 measurement rolling average appears to be an acceptable compromise.

Sample pendulum angle and angular velocities were uploaded from the microcontroller and are shown in Figure 5. In this figure, the “measurement val.” is as calculated with the rolling average on board the microcontroller and the “calc. val” is a direct differentiation in MATLAB of the pendulum angle.

Pendulum Angle

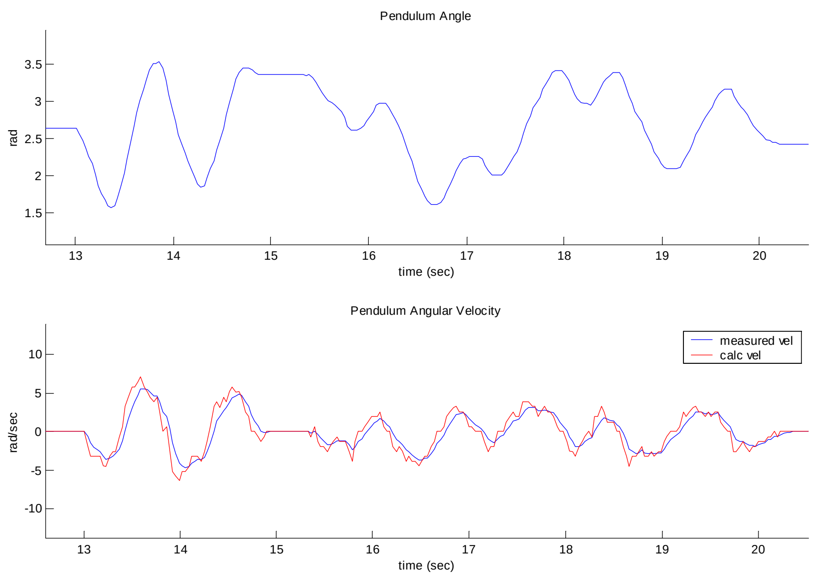

FIGURE 5.

7.4.2 Cart Position

Cart position is also measured using angular potentiometers. This is accomplished by gearing the sensor input shaft to the non-drive axle of the cart. A ratio of 2:1 sensor to wheel revolutions was selected due to availability of gears. This provides a resolution of 0.0125 inches / lsb. This is much more resolution than necessary. The non-drive axle is used for this measurement to prevent measuring wheel slip caused by loss of traction at the drive wheels. Since each sensor has an angular range of only 300 degrees, two sensors are used to continuously track cart position. Mounting of these sensor can be seen in Figure 6.

Cart Position Sensor Mounting

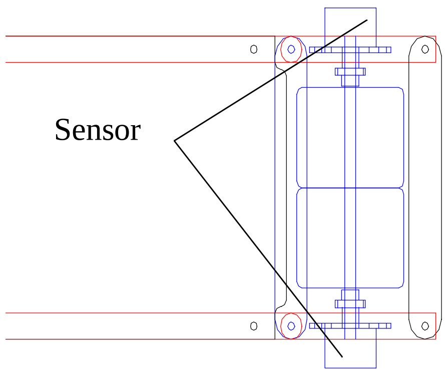

FIGURE 6.

The algorithm used to output the displacement measurement from these two sensors is shown in Figure 7.

Cart Position Algorithm

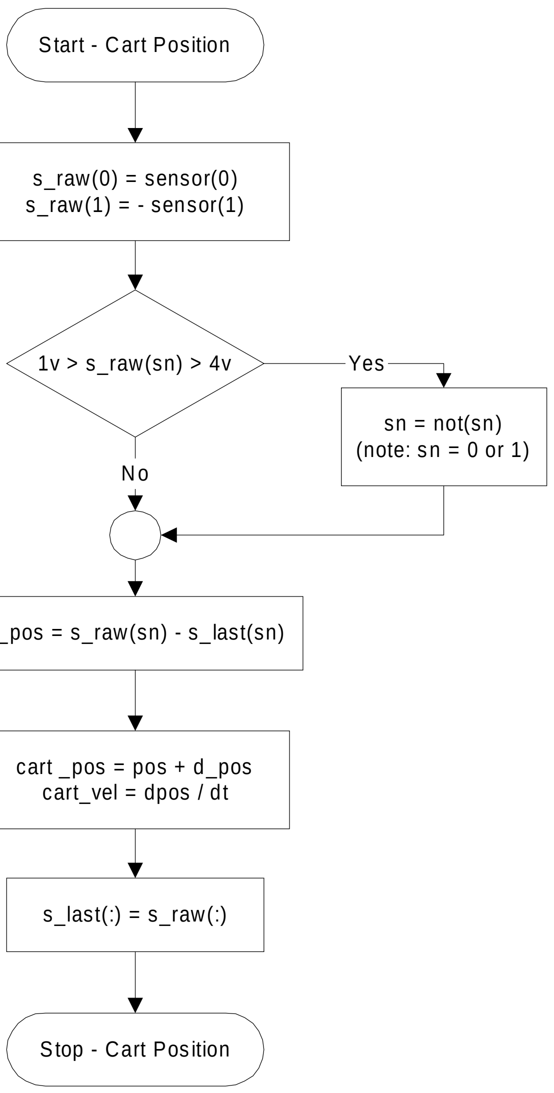

FIGURE 7.

The above figure has been implemented using the following code.

```
/******** Measure Cart Position ********************/
void sense_cart(){
extern unsigned int sens_num;
extern int cart_last[2];
extern float cartp;
extern float cartv;
extern float d_time;
extern unsigned int cartv_avg;
int s_raw[2];
int d_pos;
float tempv;

s_raw[0] = analog(0);//read cart sensor 0
s_raw[1] = -analog(1);//read cart sensor 1

// check for out of sensor limit

if ((s_raw[0] < 45) || (s_raw[0] > 210))
sens_num=1;
else if ((s_raw[1] > -45) || (s_raw[1] < -210))
sens_num=0;

d_pos= s_raw[sens_num] - cart_last[sens_num];

cartp += d_pos * cart_calib[sens_num];
tempv = d_pos * (cart_calib[sens_num] / d_time);
cartv = ((cartv * cartv_avg) + tempv)/(cartv_avg+1);

cart_last[0] = s_raw[0];
cart_last[1] = s_raw[1];

}
/****************************************************/
```

The effect of the sensor switching can be seen in Figure 8, which shows the value from the analog to digital convertor that would be used in calculating the cart displacement.

Displacement Output

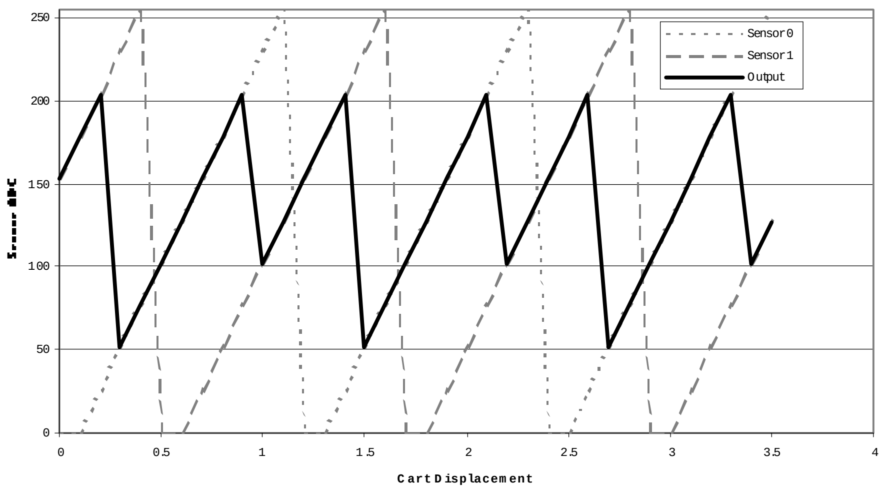

FIGURE 8.

Cart position is calculated using Equation 5 and Equation 6.

$$\Delta pos = \Delta\theta_{sensor}\left(\frac{n_{axle}}{n_{sensor}}\right)(\pi \cdot D_{tire}) \qquad \text{(EQ 5)}$$

$$\Delta\theta_{sensor} = (\text{analog}(n)_k - \text{analog}(n)_{k-1})\left(\frac{lsb}{\theta°}\right)_{ADC} \qquad \text{(EQ 6)}$$

As with pendulum velocity, cart velocity will be obtained through differentiation. Because of the higher resolution, the problem of differentiating to calculate velocity doesn’t appear to be as much of a problem. Therefore, a rolling average will not be used in the cart velocity calculation.

Sample cart position and velocity measurements were uploaded from the microcontroller and are shown in Figure 9. In this figure, the “measurement val.” is as calculated with no rolling average, on board the micorcontroller and the “calc. val” is a direct differentiation in matlab. The two traces are nearly identical as would be expected.

Cart Position

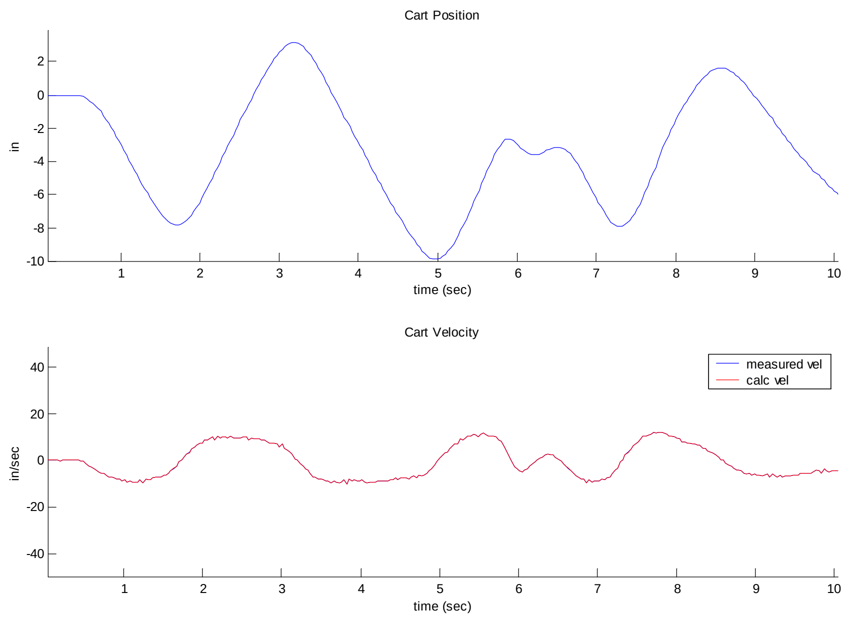

FIGURE 9.

It was discovered, once the motor driver was interfaced, that it was possible for the cart to travel fast enough that the sensor dead band can be skipped within one sample time giving the impression that the cart has now changed direction when it has not. This occurs at approximately 47 in/sec. The cart velocity can exceed well over 100 in/sec. An algorithm was attempted to detect and compensate for this problem but was found not to be robust enough to provide usable control. Therefore, new gear hubs will be made to swap the gears between axle and sensor. This will result in a gear ration of 1:2 extending the threshold before the dead band is skipped to almost 200 in/sec.

8.0 Behaviors

8.1 Pendulum Stabilization

The most important task will be to stabilize the inverted pendulum. State feedback control will be used to replace the unstable pendulum dynamics with a desired set of stable dynamics.

8.1.1 System Model

Linearizing the system dynamics about the proposed operating conditions results in the following system of equations.

$$\ddot{\theta} = \frac{(m \cdot r)^2 g \cdot \theta - c \cdot \dot{x} + I_o \cdot u}{(M+m)I_o - (m \cdot r)^2}$$

$$\ddot{x} = \frac{-\dot{x} \cdot c \cdot (m \cdot r) + (M+m) \cdot (m \cdot r)^2 g \cdot \theta + (m \cdot r) \cdot u}{(M+m)I_o - (m \cdot r)^2} \qquad \text{(EQ 7)}$$

Where *m* is the mass of the pendulum, *r* is the distance from pivot to the center of mass of the pendulum, *g* is the gravitational constant, *c* is the coefficient of viscous friction, *I* is the moment of inertia of the pendulum, *u* is the tractive force provided by the motor, and *M* is the mass of the cart. These equations can be represented in state space form as demonstrated in Equation 8.

$$\dot{X} = AX + Bu \qquad \text{where} \qquad X = \begin{bmatrix} x \\ \theta \\ \dot{x} \\ \dot{\theta} \end{bmatrix}$$

$$A = \frac{1}{(M+m)I_o - (m \cdot r)^2} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & (m \cdot r)^2 g & -c & 0 \\ 0 & (M+m) \cdot (m \cdot r)^2 g & -c \cdot m \cdot r & 0 \end{bmatrix} \qquad \text{(EQ 8)}$$

$$B = \begin{bmatrix} 0 \\ 0 \\ I_o \\ m \cdot r \end{bmatrix}$$

8.1.2 State Feedback Controller

State feedback requires that all states be available for control either through measurement or observers. Since only pendulum angle and cart position are directly measured, an observer is usually designed to estimate the velocity without the inherent problems of differentiation. Since differentiation is much easier to program and requires less computation time it was used instead of an observer. Any errors caused by differentiation appear to be negligible at this point

Through simulation, the desired dynamics were decided on based on actuator and traction limitations. The eigenvalues of the desired dynamics were used to calculate the state feedback gains.

First, the characteristic polynomial of the original system was calculated.

$$a(s) = det(sI - A) = s^4 + a_1 s^3 + a_2 s^2 + a_3 s^1 + a_4 \qquad \text{(EQ 9)}$$

The desired characteristic polynomial was created form the desired eigenvalues.

$$\begin{aligned} \alpha(s) &= (s - \lambda_1)(s - \lambda_2)(s - \lambda_3)(s - \lambda_4) \\ \alpha(s) &= s^4 + \alpha_1 s^3 + \alpha_2 s^2 + \alpha_3 s + \alpha_4 \end{aligned} \qquad \text{(EQ 10)}$$

The state feedback gains were then calculated.

$$k = (\alpha - a)(W^T)^{-1} C^{-1}$$

$$\text{where} \qquad W = \begin{bmatrix} 1 & 0 & 0 & 0 \\ a_1 & 1 & 0 & 0 \\ a_2 & a_1 & 1 & 0 \\ a_3 & a_2 & a_1 & 1 \end{bmatrix} \qquad \text{(EQ 11)}$$

$$\text{Controllability Matrix} \qquad C = \begin{bmatrix} b & Ab & A^2 b & A^3 b \end{bmatrix}$$

$$a = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \end{bmatrix}$$

$$\alpha = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 \end{bmatrix}$$

The resulting output is then

$$u = -k(X - X_{desired}) \qquad \text{(EQ 12)}$$

This output calculation occurs approximately every 32 ms. when a real time interrupt service routine sets a calculation flag.

8.2 Obstacle Avoidance

Obstacle avoidance behavior will detect when the cart nears an object detected by the infrared sensor. When this occurs, the cart will disable the pendulum stabilization behavior and either stop the cart motion or return to its initial starting position. This is implemented by comparing the analog output of the sensor to a predetermined threshold. Upon exceeding the threshold, the motor output is disabled. It was found that a threshold of 95 lsb was adequate.

8.3 Instability Fail-safe

When the angular motion of the pendulum becomes too great it will not be possible for the cart to stabilize the pendulum. This can be detected by the bump sensors which will signal when the pendulum has fallen over and cannot be re-righted. When this occurs, the cart will disable the pendulum stabilization behavior and stop the cart motion.

8.4 Data Acquisition

The microcontroller will record its performance and save this information to the external RAM. This information will include pendulum angle, pendulum velocity, cart position, cart velocity, and output duty cycle. This information along with a short header can be uploaded to a PC and saved to a file with the terminal capture feature of ICC11. This file can then be read by Matlab with decoding information provided in the uploaded header.

The data is uploaded through the serial port using the put_int and put_char functions to send comma delimited data. These functions were selected because of their small size and simplicity. The limitation when sending floating point numbers as integers is that they must be scaled manually to retain as many significant digits as possible. The scaling values used are uploaded in the header. The format of the header is as follows:

[*number of channels (n), scale value (1), ... , scale value(n)*]

Once this data has been read by Matlab it can be manipulated and graphed.

9.0 Results

While implementing the cart position control it was discovered that there was a sensing problem with the cart position measurement that had not been foreseen. This problem is based on the fact that since the rotary sensors used to measure the cart position have a 300 degree measurement range. This sensor limitation required an algorithm to detect the dead band and switch sensors. Due to the gear ratio between axle and sensor and the sampling rate, this cannot be reliably detected over 47 in / sec. This problem results in control of the cart to easily become unstable.

Because of the instability of the cart control due to a sensing problem. Pendulum balance could not be realized. Pendulum instability detection and infrared obstacle avoidance were successfully tested but without stable cart control, the integrated system could not be demonstrated.

The data acquisition function was indispensable at diagnosing this sensing problem.

10.0 Conclusion

A robust mobile platform has successfully been completed, the 40 kHz modulation has successfully been tested, the motor and motor driver has successfully been interfaced, and step inputs have been performed by the cart. Performing cart position control it was discovered that there were some fundamental problems in the sensing of the cart position. These problems will hopefully be rectified with different gear ratios for the sensors.

11.0 Appendix

11.1 C Code

All Programming was done using ICC11. The following is listing of the code used.

11.1.1 main.c

```
/**********************************************

Title:main.c
Description:main inverted pendulum program
Programmer:Randy Wells
Date:10 October 2000
Revision History:
Ver.1new support
**********************************************/

/**************************** Includes ********/
#include <hc11.h>
#include <mil.h>

/**********************************************/

/************* Constants ***********************/
/*Full period for a motor in clock cycles.
A clock cycle is 0.5 microseconds*/
//#define PERIODM 40000
#define PERIODM 0x9c40
#define PERIOD_MIN 2400// min pulse width 1.2ms
/***********************************************/

/******** Set up interrupt vectors *************/
extern void _start(void);/* entry point in crt??.s */
#pragma interrupt_handler RTI_isr
#pragma interrupt_handler TOC2_isr;
#define DUMMY_ENTRY(void (*)(void))0xFFFF
/**********************************************/

/************ Prototypes ************************/
float fsfb_control(float cartp, float cartv, float pendp, float pendv);
float fsfb_cart_control(float cartp, float cartv, float pendp, float pendv);
float calib_control(float cartp, float cartv, float pendp, float pendv);
/**********************************************/

/************ Globals Variables ****************/
int RTIcalc=0;
```

```
unsigned int sens_num = 0;

float cartp = 0.0;// cart position
float cartv = 0.0;// cart velocity
float pendp = 0.0;// pendulum angle
float pendv = 0.0;// pendulum velocity
float debug = 0.0;// debug parameter

float k_pendp = 20.0;// gain:  pendulum angle
float  k_pendv = 0.0;// gain:  pendulum angular vel.

int cart_last[2] = {0,0};
float vel_last = 0.0;
float pend_last = 0.0;
float mduty;// motor pwm duty cycle
float d_time = 0.032686;// interrupt interval seconds
unsigned int duty_cycle;

// system mode, 0: standby, 1: pendulum
int s_mode = 0;

/********* Initialize RTI **********************/
void init_RTI() {
PACTL |= 0X03;//clock divide by 8 = 32.768 ms

TMSK2 |= 0X40;//interrupt on rtif
CLEAR_FLAG(TFLG2,0x40);// clear interrupt flag

}// interrupts must still be turned on using INTR_ON
/**********************************************/

/***** TOC2_isr ********************************/
void TOC2_isr()
{

/* Keep the motor off if no duty cycle specified.*/
  if(duty_cycle == 0)
   {
    CLEAR_BIT(TCTL1, 0x40);
   }
  else
    if(TCTL1 & 0x40)
     {
      TOC2 += PERIOD_MIN+duty_cycle;  /* Keep pulse width up for
duty_cycle */
```

```
     CLEAR_BIT(TCTL1,0x40);  /*  Prepare to turn off next OC2
interrupt*/
    }
   else
    {
     TOC2 += (PERIODM - (PERIOD_MIN + duty_cycle));
//TOC2 += (0x9c40 - duty_cycle);
      SET_BIT(TCTL1,0x40);   /* Prepare to raise signal next OC2 intr. */
    }

CLEAR_FLAG(TFLG1,0x40);     /* Clear OC2F interrupt Flag */

}

/*********************************************/

/***** RTI_isr Interrupt Service Routine *******/
void RTI_isr()
{
extern int RTIcalc;
RTIcalc=1;// signal calculation
CLEAR_FLAG(TFLG2,0x40); //clear flag, enables next interrupt
}
/*********************************************/

/**************** Main ***********************/
void main()
{

INTR_OFF();
// Declare Variables

// Initializations
init_pwm();//initial pwm parameters
init_analog();//initialize analog input
init_pend();
init_serial();//initialize serial coms.
init_RTI();// initialize rti
init_daq();// initialize data acquisition
// set Port C data direction
CLEAR_BIT(DDRC,0x07);// bit 0,1,2: input
SET_BIT(DDRC,0x80);// bit 7:  output

s_mode=0;
INTR_ON();

pwm_motor2(50.0);
```

```
/************* Start Main Loop ************/
write("start\n\r");
//put_char(13); put_char(10);// line feed
while(1){

//s_mode=1;


// use toggle switch on portc bit 1
if (!(PORTC & 0x02))//bit 1, port c
{
//put_int(s_mode); put_char(13); put_char(10);
if (s_mode == 1) // standby mode switch commands
{
s_mode = 0;// standby
disable_motor();
}
// standby mode continous commands

// upload memory block to serial port
if (!(PORTC & 0x04))
{
daq_up();

}//end upload commands
}
else
{
if (s_mode == 0)// active mode switch commands
{
s_mode = 1;
enable_motor();
}

// active mode continuous commands
if (RTIcalc == 1)
{
SET_BIT(PORTC,0x80);//used to check isr calculation time

sense_cart2();
sense_pend();

if ((pendp > 1.5) | (pendp < -1.5))
{
k_pendp=0;
k_pendv=0;
}
//mduty=fsfb_control(cartp, cartv, pendp, pendv);
mduty=fsfb_cart_control(cartp, cartv, pendp, pendv);
//mduty=calib_control(cartp, cartv, pendp, pendv);

put_int((int)(mduty));
put_char(13); put_char(10);// line feed

pwm_motor2(mduty);

daq_store();
RTIcalc=0;
CLEAR_BIT(PORTC,0x80);//used to check isr calculation time

}

// bump and obstacle detection
if (bump() || obstacle())
{
disable_motor();
}// end bump and obstacle detect


}
} // End Main Loop
}//  End Program
/***************************************/





/
************************************************************

  Initialize interrupt vectors

************************************************************
*/

#pragma abs_address:0xffd6
void (*interrupt_vectors[])(void) =
{
DUMMY_ENTRY,/* SCI */
DUMMY_ENTRY,/* SPI */
DUMMY_ENTRY,/* PAIE */
DUMMY_ENTRY,/* PAO */
```

```
DUMMY_ENTRY,/* TOF */
DUMMY_ENTRY,/* TOC5 */
DUMMY_ENTRY,/* TOC4 */
DUMMY_ENTRY,/* TOC3 */
TOC2_isr,/* TOC2 */
DUMMY_ENTRY,/* TOC1 */
DUMMY_ENTRY,/* TIC3 */
DUMMY_ENTRY,/* TIC2 */
DUMMY_ENTRY,/* TIC1 */
RTI_isr,/* RTI */
DUMMY_ENTRY,/* IRQ */
DUMMY_ENTRY,/* XIRQ */
DUMMY_ENTRY,/* SWI */
DUMMY_ENTRY,/* ILLOP */
DUMMY_ENTRY,/* COP */
DUMMY_ENTRY,/* CLM */
_start/* RESET */
};
#pragma end_abs_address
```

11.1.2 sensor.c

```
/*************************************

Title:sensor.c
Description:low level sensor stuff
Programmer:Randy Wells with parts
borrowed from Mekatronix
Date:10 October 2000
Revision History:
Ver.1new support
**************************************/

/************ Includes ***************/
#include <hc11.h>
#include <mil.h>
/************************************/

/*********** Constants *************/
#define IR_thresh 125//Infrared Threshold
float cart_calib[2]={0.0125, 0.0125};// inch/count
//#define pend_calib_k 1.2// deg/count
#define pend_calib_k 0.02093// rad/count
#define pend_calib_o -2.5

#define cartv_avg 2
```

```
#define pendv_avg 2
/**********************************/

/*********** Prototypes ************/
void init_analog(void);

int analog(int port);
/* Takes one reading from the analog port and returns the value read
 * Inputs : Port
 * Output : None
 * Return Value : Value read from A/D port
 */
/***********************************/


/*************************************
 MEKATRONIX Copyright 1998
Title        analog.c
Programmer   Keith L. Doty
Date         February 9, 1998
Version      1

Description: Routines to power up the analog port
and to read the 8 channels.
**************************************/

/************** Prototypes ************/
// from analog.c
void init_analog(void);   /* Power up A/D */
int analog(int);     /* Read analog channel 0<= int <= 7 */
// from Randy Wells

/************************************/
int bump(void);
int obstacle(void);


void init_analog(void)
/* This routine must be executed before
 the function analog is called.  Power up A/D */
{
  SET_BIT(OPTION,0x80);
}
/******End of init_analog ************/


int analog(int port)
```

```
/
************************************************************
************
 * Function: Takes one reading from the analog port and returns the*
 *     value read. *
 * Returns:  Analog value read from A/D PORTE                    *
 *                                                 *
 * Inputs                                             *
 *   Parameters: None                                     *
 *   Globals:    None                                   *
 *   Registers:  PORTE                                    *
 * Outputs                                            *
 *   Parameters: None                                     *
 *   Globals:    None*
 *   Registers: None                     *
 * Functions called: None                                  *

************************************************************
************/
{
  ADCTL=port;/* Address the selected channel */
  while((ADCTL & 0x80) != 0x80); /* Wait for A/D to finish */
  return(ADR1);/* Return analog value */
}

/**********************End of analog
*****************************/


int bump(){// detect bump switch press
return (!(PORTC & 0x01));//bit wise

}

/********** IR obstacle detect ********************/
int obstacle(){ // detect infrared over threshold
extern unsigned int IRthresh;
int temp3;
int temp4;
temp3=analog(3);
temp4=analog(4);
//put_int(temp3); put_char(‘ ‘); put_int(temp4); put_char(‘ ‘);
//put_char(13); put_char(10);
return(0);
//return ((analog(3) > IR_thresh)||(analog(4) > IR_thresh));

}
```

```
/**************************************************/

/***** Initialize Pendulum Measurements ************/
void init_pend(){
extern int cart_last[2];
extern float pend_last;
extern float pendp;
extern float pendv;
extern float cartp;
extern float cartv;
extern float cartvdt[2];
extern float mduty;
extern float d_time;

cart_last[0] = analog(0);
cart_last[1] = -analog(1);
cartp=0.0;
cartv=0.0;

pendp = (pend_calib_k * analog(2)) + pend_calib_o;
pend_last=pendp;
pendv=0.0;
mduty=0.0;
}

/**************************************************/

/********* Measure Pendulum Angle ******************/
void sense_pend(){
extern float pend_last;
extern float pendp;
extern float pendv;
extern float d_time;


int s_raw;
int d_pos;
float tempv;

s_raw = analog(2);

pendp = (pend_calib_k * s_raw) + pend_calib_o;
tempv = (pendp - pend_last) / d_time;
pendv = ((pendv*pendv_avg) + tempv)/(pendv_avg+1);
pend_last = pendp;

//put_int((int)(pendp*10)); put_char(‘ ‘);
//put_int((int)(pendv)); put_char(' ');
//put_char(13); put_char(10);
}
/**************************************************/

/******** Measure Cart Position *******************/
/*
void sense_cart(){
extern unsigned int sens_num;
extern int cart_last[2];
extern float cartp;
extern float cartv;
extern float debug;
extern float d_time;

int s_raw[2];
int d_pos;
float tempv;

s_raw[0] = analog(0);//read cart sensor 0
s_raw[1] = -analog(1);//read cart sensor 1

// check for out of sensor limit

if ((s_raw[0] < 65) || (s_raw[0] > 190))
sens_num=1;
else if ((s_raw[1] > -65) || (s_raw[1] < -190))
sens_num=0;

debug=sens_num;

d_pos= s_raw[sens_num] - cart_last[sens_num];

cartp += d_pos * cart_calib[sens_num];
tempv = d_pos * (cart_calib[sens_num] / d_time);
cartv = ((cartv * cartv_avg) + tempv)/(cartv_avg+1);

//put_int((int)(cartp)); put_char(' ');
//put_int((int)(cartv)); put_char(' ');
//put_char(13); put_char(10);

cart_last[0] = s_raw[0];
cart_last[1] = s_raw[1];

}
/**************************************************/
```

```
/******** Measure Cart Position ********************/
/*
void sense_cart2(){
extern unsigned int sens_num;
extern int cart_last[2];
extern float vel_last;
extern float cartp;
extern float cartv;
extern float mduty;
extern float debug;
extern float d_time;

int s_raw[2];
int d_pos;
float tempv;

s_raw[0] = analog(0);//read cart sensor 0
s_raw[1] = -analog(1);//read cart sensor 1

// check for out of sensor limit

if ((s_raw[0] < 65) || (s_raw[0] > 190))
sens_num=1;
else if ((s_raw[1] > -65) || (s_raw[1] < -190))
sens_num=0;

//debug=sens_num;
debug=0.0;

d_pos= s_raw[sens_num] - cart_last[sens_num];

cartp += d_pos * cart_calib[sens_num];
tempv = d_pos * (cart_calib[sens_num] / d_time);

// test if sensor skipped dead band going forward
if ((vel_last > 20.0) & (tempv < 10.0) & (mduty > 50.0))
{
d_pos += 300;// lsb/revolution
//recalculate
cartp += d_pos * cart_calib[sens_num];
tempv = d_pos * (cart_calib[sens_num] / d_time);
debug=1.0;
}
// test if sensor skipped dead band going in reverse
else if ((vel_last < -20.0) & (tempv > -10.0) & (mduty < 50.0))
{
```

```
d_pos -= 300;// lsb/revolution
// recalculate
cartp += d_pos * cart_calib[sens_num];
tempv = d_pos * (cart_calib[sens_num] / d_time);
debug=2.0;
}

cartv = ((cartv * cartv_avg) + tempv)/(cartv_avg+1);

//put_int((int)(cartp)); put_char(‘ ‘);
//put_int((int)(cartv)); put_char(‘ ‘);
//put_char(13); put_char(10);

cart_last[0] = s_raw[0];
cart_last[1] = s_raw[1];
vel_last = cartv;

}
/**************************************************/


/******** Measure Cart Position ********************/
void sense_cart2(){
extern unsigned int sens_num;
extern int cart_last[2];
extern float vel_last;
extern float cartp;
extern float cartv;
extern float mduty;
extern float debug;
extern float d_time;

int s_raw[2];
int d_pos;
float tempv;

s_raw[0] = analog(0);//read cart sensor 0
s_raw[1] = -analog(1);//read cart sensor 1

// check for out of sensor limit

if ((s_raw[0] < 65) || (s_raw[0] > 190))
sens_num=1;
else if ((s_raw[1] > -65) || (s_raw[1] < -190))
sens_num=0;

//debug=sens_num;
```

```
debug=0.0;

d_pos= s_raw[sens_num] - cart_last[sens_num];

tempv = d_pos * (cart_calib[sens_num] / d_time);

// test if sensor skipped dead band going forward
//if ((vel_last > 10.0) & (tempv < 0.0) & (mduty > 50.0))
if (((vel_last-tempv) > 10.0) & (mduty > 50.0))
//if ((vel_last > 20.0) & (tempv < -5.0))
{
d_pos += 300;// lsb/revolution
debug=1.0;
}
//else if ((vel_last < -10.0) & (tempv > 0.0) & (mduty < 50.0))
//else if ((vel_last < -20.0) & (tempv > 5.0))
else if (((vel_last-tempv) < -10.0)  & (mduty < 50.0))
{
d_pos -= 300;// lsb/revolution
debug=2.0;
}

//calculate
cartp += d_pos * cart_calib[sens_num];
tempv = d_pos * (cart_calib[sens_num] / d_time);
cartv = ((cartv * cartv_avg) + tempv)/(cartv_avg+1);

cart_last[0] = s_raw[0];
cart_last[1] = s_raw[1];
vel_last = cartv;

}
/***************************************************/
```

11.1.3 motor.c

```
/
***********************************************************
**********

Title:motor.c
Description:low level pwm output for servos and motordrivers
Programmer:Randy Wells
Date:10 October 2000
```

```
Revision History:
Ver.1new support
*************************************************************
***********/

/**************************** Includes
**********************************/
#include <hc11.h>
#include <mil.h>
/
*************************************************************
************/

/******** Constants ***************/
//A clock cycle is 0.5 microseconds

// one percent of output range:  (max 3600 (1.8ms) - min 2400 (1.2ms))/100
#define PERIOD_1PC 12
/**********************************/

/*************************** Prototypes
*********************************/
void init_pwm(void);
float limit_range(float val, float low, float high);
void pwm_motor(float vel);
/
*************************************************************
************/

/**************************** Globals
**********************************/

/
*************************************************************
************/

/******* PWM Initialization *********************/
void init_pwm()
{
extern float mduty;


/* Set OC3 to output low */
SET_BIT(TCTL1,0x80);

CLEAR_BIT(TCTL1,0x040);
```

```
/* Set PWM duty cycle to 0 first */
  mduty=0.0;

/* Enable motor interrupts on OC3 */

SET_BIT(TMSK1,0x40);


}
/*********************************************/

/*********** motor control new ******************/
void pwm_motor2(float vel)
{
extern unsigned int duty_cycle;

vel = limit_range(vel, 1.0, 99.0);
duty_cycle = (unsigned int)(vel*PERIOD_1PC);
}
/*********************************************/
/***********Limit motor output pwm **************/
float limit_range(float val, float low, float high)
{
if (val < low)
return(low);
else if (val > high)
return(high);
else
return(val);
}
/***********************************************/

/********* Output motor pwm ********************/
void pwm_motor(float vel)
{
vel = limit_range(vel, 1.0, 99.0);
TOC2 = (unsigned int) (655.36 * vel);
}
/***********************************************/

/********** Motor Enable ************************/
void enable_motor()
{
init_pend();
// enable interrupt
//INTR_ON();// clear interrupt mask
write("motor_on");
```

```
put_char(13); put_char(10);// line feed
}
/***********************************************/

/********** Motor Disable ***********************/
void disable_motor()
{
pwm_motor2(50.0);// stop motor
// disable interrupt
//INTR_OFF();// set interrupt mask
write("motor_off");
put_char(13); put_char(10);// line feed

}
/***********************************************/
```

11.1.4 control.c

```
/************************************

Title:control.c
Description:control calculations
Programmer:Randy Wells with parts
Date:10 October 2000
Revision History:
Ver.1new support
*************************************/


/************ Includes ***************/
#include <mil.h>
#include <hc11.h>
/************************************/

/************ Constants ***********/
#define k_cartp 5.27// gain:  cart position
//#define k_cartv -2.3// gain:  cart velocity
#define k_cartv -1.3// gain:  cart velocity
// pend gains are now globals in main
#define k_motor 1.0// gain:  motor

//#define c_step 10// 32 msec step size:  320 msec.
#define c_step 5// 32 msec step size:  160 msec.
/*********************************/

/********** Prototypes ***********/
float fsfb_control(float cartp, float cartv, float pendp, float pendv);
float calib_control_test(float cartp, float cartv, float pendp, float pendv);
float calib_control(float cartp, float cartv, float pendp, float pendv);
/***********************************/

/*********** Global Variables *****/

int c_count = 0;// calibration sequence counter
int c_mode = 0;
int c_pos=0;
float c_out = 50.0;
int c_wait = 0;
/*
float k_cartp = 0.0;// gain:  cart position
float k_cartv = 0.0;// gain:  cart velocity
float k_pendp = 20.0;// gain:  pendulum angle
float k_pendv = 0.0;// gain:  pendulum angular vel.

float k_motor = 1.0;// gain:  motor
*/
/*********************************/


/****** calibration control test *******/
/*
float calib_control(float cartp, float cartv, float pendp, float pendv)
{
extern int c_count;
extern float c_out;

if (c_count >= c_step)
{

// neutral
if (c_mode == 0)
{
c_out=50.0;
if (c_wait == 20)
{
c_mode=1;
c_wait=0;
}
else
{
++c_wait;
}
}
```

```
// step output
else if (c_mode == 1)
{
c_out=70.0;
if (c_wait == 70)
{
c_mode=2;
c_wait=0;
}
else
{
++c_wait;
}
}
else if (c_mode == 2)
{
c_out=50.0;
if (c_wait == 50)
{
c_mode=3;
c_wait=0;
}
else
{
++c_wait;
}
}

// step output
else if (c_mode == 3)
{
c_out=80.0;
if (c_wait == 60)
{
c_mode=4;
c_wait=0;
}
else
{
++c_wait;
}
}

else if (c_mode == 4)
{
c_out=50.0;
if (c_wait == 50)
{
c_mode=5;
c_wait=0;
}
else
{
++c_wait;
}
}

// step output
else if (c_mode == 5)
{
c_out=85.0;
if (c_wait == 50)
{
c_mode=6;
c_wait=0;
}
else
{
++c_wait;
}
}

// neutral
else if (c_mode==6)
{
c_out=50.0;
}
}
else
{
++c_count;
}

return(c_out);
}

/*************************************/



/******* state feedback cart postion control ****/
float fsfb_cart_control(float cartp, float cartv, float pendp, float pendv)
{
```

```
float temp;

// step cart position
if (c_count >= c_step*60)
{

// cart position #1
if (c_pos == 0)
c_pos=1;
else
c_pos=0;

c_count=0;
}
else
{
++c_count;
}
/*****/

temp = 0.0;
temp += ((c_pos*24)-cartp)* k_cartp;
temp += cartv * k_cartv;
temp += 50.0;

// remove dead band
if (temp < 48.0) temp = (temp*0.8) - 10;
else if (temp > 52.0) temp += 4;
else temp = 50.0;

// limit output
if (temp > 80.0) temp = 80.0;
if (temp < 10.0) temp = 10.0;


put_int((int)(c_pos*5)); put_char(‘ ‘);
put_int((int)(cartp)); put_char(‘ ‘);
//put_int((int)(temp)); put_char(‘ ‘);
//put_char(13); put_char(10);

return(temp);
}
/*****************************************/
```

```
/******** state feedback control *******/

/*
float fsfb_control(float cartp, float cartv, float pendp, float pendv)
{
extern float k_pendp;// gain:  pendulum angle
extern float k_pendv; // gain:  pendulum angular vel.
float temp;

temp = 0.0;
temp += cartp * k_cartv;
temp += cartv * k_cartv;
temp += pendp * k_pendp;
temp += pendv * k_pendv;
temp *= k_motor;
temp += 50.0;

//put_int((int)(temp)); put_char(‘ ‘);
//put_char(13); put_char(10);

return(temp);
}


*/
```

11.1.5 daq.c

```
/***********************************************

Title:main.c
Description:main inverted pendulum program
Programmer:Randy Wells
Date:10 October 2000
Revision History:
Ver.1new support
***********************************************/

/*************************** Includes ********/
#include <hc11.h>
#include <mil.h>
/***********************************************/

/*************** Constants **********/
#define mem_size 1000
/************************************/
```

```
/************* Prototypes ***********/


/**********************************/

/********* Global Variables *********/
int daq_cartp[1000];
int daq_cartv[1000];
int daq_pendp[1000];
int daq_pendv[1000];
int daq_mduty[1000];
int daq_debug[1000];
unsigned int daq_ptr = 0;

unsigned int daq_scale[6] = {10, 10, 1000, 10, 1, 1};
/************************************/

/****** DAQ Upload **************************/
void daq_up(){
extern int daq_cartp[1000];
extern int daq_cartv[1000];
extern int daq_pendp[1000];
extern int daq_pendv[1000];
extern int daq_mduty[1000];
extern int daq_debug[1000];

int n;

put_int(6); put_char(‘,’);

for (n=0; n <= 5; n++)
{
put_int(daq_scale[n]);
put_char(‘,’);
}

for (n=0; n != mem_size-1; n++)
{

put_int(daq_cartp[n]);
put_char(‘,’);

put_int(daq_cartv[n]);
put_char(‘,’);

put_int(daq_pendp[n]);
put_char(‘,’);
put_int(daq_pendv[n]);
put_char(‘,’);

put_int(daq_mduty[n]);
put_char(‘,’);

put_int(daq_debug[n]);
put_char(‘,’);

}
//put_int(999);
}
/********************************************/

/******** Initialize DAQ *********************/
void init_daq(){

extern int daq_cartp[1000];
extern int daq_cartv[1000];
extern int daq_pendp[1000];
extern int daq_pendv[1000];
extern int daq_mduty[1000];
extern int daq_debug[1000];

int ii;

daq_ptr = 0;
write(“daq_init start”);
put_char(13); put_char(10);// line feed

for(ii=0; ii!=mem_size; ++ii)
{
daq_cartp[ii]=0;
daq_cartv[ii]=0;
daq_pendp[ii]=0;
daq_pendv[ii]=0;
daq_mduty[ii]=0;
daq_debug[ii]=0;
}
write(“daq_init finish”);
put_char(13); put_char(10);// line feed
}
/*********************************************/

/******* DAQ Store Data *********************/
void daq_store()
```

```
{
extern float cartp;
extern float cartv;
extern float pendp;
extern float pendv;
extern float mduty;
extern float debug;
extern int daq_cartp[1000];
extern int daq_cartv[1000];
extern int daq_pendp[1000];
extern int daq_pendv[1000];
extern int daq_mduty[1000];

extern int daq_debug[1000];

extern unsigned int daq_ptr;
extern unsigned int daq_scale[6];

// reset memory pointer when array is filled
if (daq_ptr == mem_size)
daq_ptr=0;

// store data
daq_cartp[daq_ptr] = (int)(cartp * daq_scale[0]);
daq_cartv[daq_ptr] = (int)(cartv * daq_scale[1]);
daq_pendp[daq_ptr] = (int)(pendp * daq_scale[2]);
daq_pendv[daq_ptr] = (int)(pendv * daq_scale[3]);
daq_mduty[daq_ptr] = (int)(mduty * daq_scale[4]);
daq_debug[daq_ptr] = (int)(debug * daq_scale[5]);

//put_int((int)(daq_cartp[daq_ptr]));
//put_char(13); put_char(10);// line feed


// remove this condition for continuos loggin
if (daq_ptr < mem_size)
++daq_ptr;

}
```