120th ASEE Annual Conference & Exposition
FRANKLY, WE DO GIVE A D*MN
June 23-26, 2013

Paper ID #7900

A Systematic Approach for Development and Simulation of Digital Control Algorithms using SIMULINK

Prof. Matthew G Feemster, U.S. Naval Academy

Matthew Feemster received his Ph.D in Electrical Engineering from Clemson University in 2000. From 2000 to 2002, he was the lead Controls Engineer at WaveCrest Laboratories based in Dulles, VA. In 2002, he accepted a position at the U.S. Naval Academy where he is currently an Associate Professor. His current research interests include nonlinear/adaptive control techniques applied to marine applications.

A Systematic Approach for Development and Simulation of Digital Control Algorithms using SIMULINK

I. Abstract

In this paper, a methodology is presented to assist students in the development of a digital control algorithm. Specifically, the recent ability to embed functions within the SIMULINK environment of the software package MATLAB has facilitated the ability to simulate "C-like" digital control algorithms. The proposed methodology is presented through a laboratory exercise that develops a digital heading controller implemented within the Dynamic C environment for an autonomous ground vehicle.

II. Introduction

With the development of readily available inertial measurement units (IMUs) board such as the ArduPilot® for mobile applications, measurement of states such as position, heading, roll, pitch, and yaw is greatly simplified. As a result, developing control students need only to focus on the design and implementation of the digital control algorithm that utilizes the sensor measurements to calculate the proper actuator commands. However from review of final capstone reports and presentations over the years, students expressed continued difficulties when implementing even simple PID based control algorithms on digital processor. This difficulty in C based implementation seems surprising in since all students are required to take a two hour lab based course dedicated to the design and implementation of control algorithms on the Rabbit single board computer (a select microprocessor from Digi® shown in Figure 1). Specifically this course targets the design of classical compensators $KG_c(s)$ for a typical DC motor with implementation of the algorithm on a Rabbit single board computer.

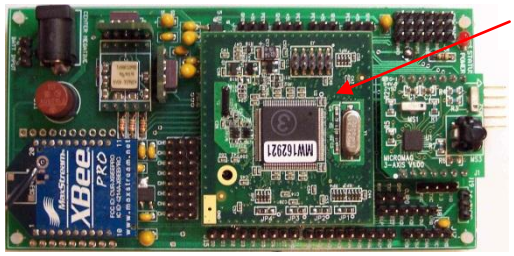


Figure 1: Rabbit single board computer

The conversion of the continuous time compensator to the corresponding digital compensator $KG_c(z)$ is accomplished via Tustin's Transformation (utilizing the `c2d` command in MATLAB). From the structure of $KG_c(z)$, students subsequently develop the corresponding difference equations that can now be transferred to the Rabbit SBC for implementation. The major oversight here is that students did not *simulate* their difference control equations to verify that the conversion from $KG_c(z)$ to difference equations was performed properly. If the control experiments did not perform as expected from the continuous time simulation, students were unsure 1) if there were hardware problems (bad connections, improper sensor calibration, *etc*) or if 2) their digital compensator was not developed correctly to difference equations.

In response to this observation, this paper will present a sequence within the MATLAB/SIMULINK software environment to provide students with assurance that the structure of their difference control algorithm is correct. Specifically, the recent ability to utilize embedded function blocks within the SIMULINK modeling environment has facilitated a straightforward method for verifying digital control algorithms. In order to showcase these capabilities, the remainder of the paper traces through a laboratory exercise (conducted over several lab periods in a

digital controls course) that targets the development of a digital heading control algorithm for an autonomous ground vehicle. In addition, a brief survey was conducted at the close of the course to gauge the effectiveness of the introduced process.

III. Digital Heading Control of the Traxxas Ground Vehicle

The primary objective of the laboratory exercise is the development of a digital control algorithm to be implemented on the Traxxas® ground vehicle of Figure 2 to promote heading tracking.

Rabbit 3000 μ processor with magnetic compass

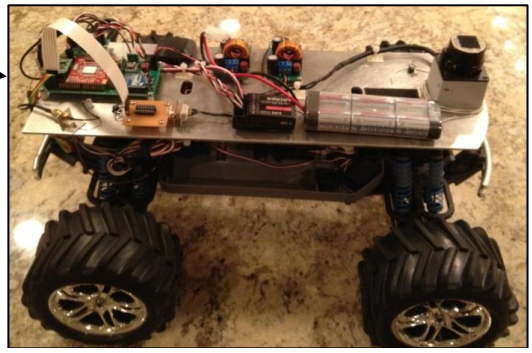


Figure 2: Traxxas EMaxx RC Vehicle

Students are provided with the following information to begin their design process:

- A linear transfer function is provided that relates the steering angle of the front wheels δ to the heading of the vehicle ψ, is given by $G(s)=\frac{\psi(s)}{\delta(s)}=\frac{(v/L)}{s}$ where v is the vehicle's longitudinal velocity, and L is the distance between the front/rear axles. For the above EMaxx vehicle, the model parameter values are given in Table 1

$v=1.5(m/\sec)$ - this constant longitudinal velocity is maintained by a separate speed control loop implemented on the vehicle.	$L=0.35(m)$	$\lvert\delta\rvert\le 0.52(rad)$. The maximum steering angle is approximately $30(\deg)$.	$G(s)=\frac{\psi(s)}{\delta(s)}=\frac{4.23}{s}$

Table 1: Model parameter values for the Traxxas EMaxx vehicle

- The students are then required to design a controller to achieve the following control objectives:

  a) Closed-loop stability.
  b) Steady state error is zero for a step heading reference command of $\psi_{\text{ref}}=90^{\circ}$.
  c) The vehicle must exhibit a settling time of approximately $T_s\approx 5.0(\sec)$ and an overshoot of approximately $\%OS\approx 2\%$ for a step heading reference command of $\psi_{\text{ref}}=90^{\circ}$.
  d) The control algorithm must not request more than 30° of steering angle ($\lvert\delta\rvert\le 30^{\circ}$).

IV. The Control Design Process

Based on the above control objectives, the following *desired* closed-loop transfer function $T(s)$ is calculated in the following manner

$$\sigma = \frac{4}{T_s},\ \omega = \frac{-\pi \cdot \sigma}{\ln\left(\frac{\%OS}{100}\right)},\ T(s) = \frac{\left(\sigma^2 + \omega^2\right)}{(s+\sigma)^2 + \omega^2} = \frac{1.053}{s^2 + 1.6s + 1.053}$$

Note: The numerator of the above closed-loop transfer function $T(s)$ was crafted in a manner to produce a DC gain of 1.0 ($\lim_{s\to 0} T(s) = 1$) such that the steady state error objective is promoted (for a step input reference command).

In order to achieve the desired closed-loop transfer function $T(s)$, the following compensator $KG_c(s)$ is calculated

$$KG_c(s) = \frac{\delta(s)}{E(s)} = \frac{T(s)}{G(s)(1-T(s))} = \frac{0.24957}{(s+1.6)}$$

where $E(s) = \psi_{\text{ref}} - \psi(s)$ represents the error signal.

Step #1: Prior to going forward, the continuous time compensator $KG_c(s)$ must be evaluated against the established control objectives (there is no purpose in developing a digital compensator based on $KG_c(s)$ if this compensator does not meet the objectives) through the utilization of the following MATLAB SIMULINK block diagram.

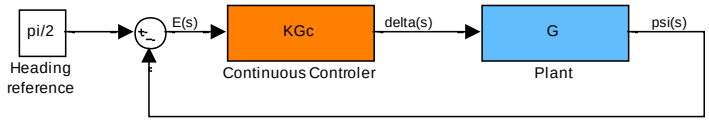
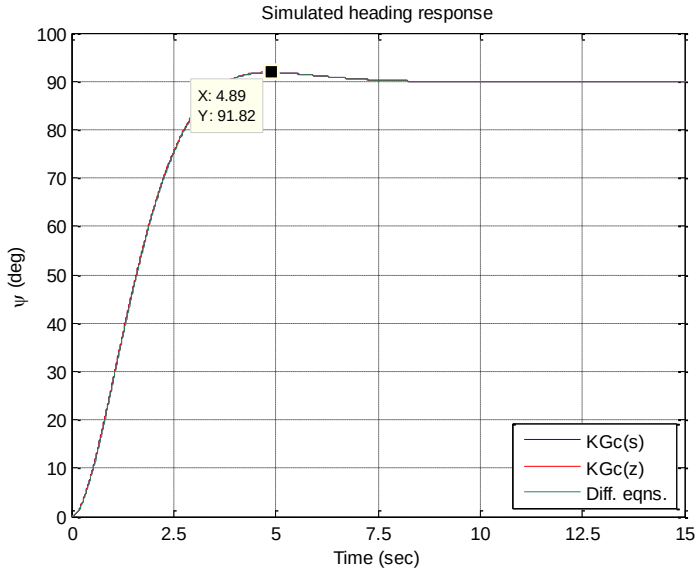


Figure 3: Continuous time simulation

Step #1 Simulation Results: The designed compensator $KG_c(s)$ meets all control objectives as shown in Figure 4 and Figure 5; therefore, one may proceed with the digitization process.



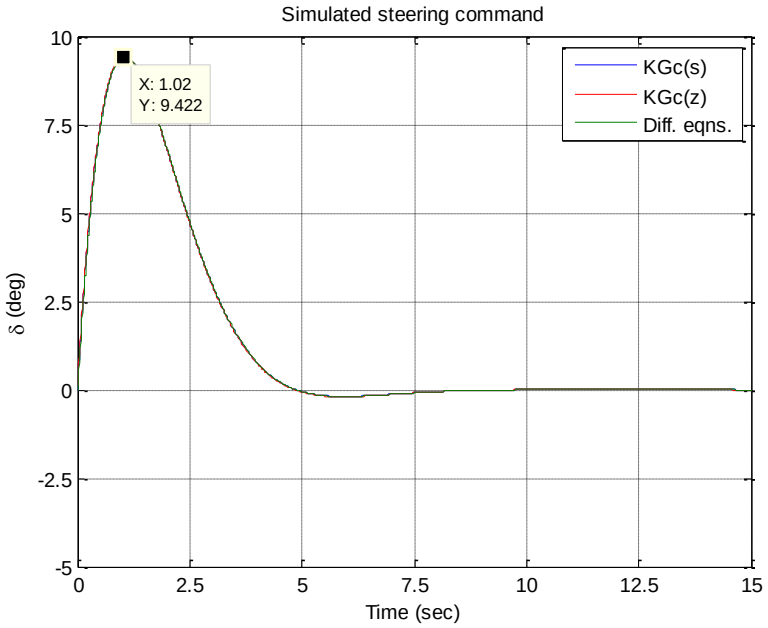
Evaluation of control objectives

$$\%OS = \frac{(91.82 - 90.0)}{90.0} \cdot 100\% = 2.02\%$$

$$T_s \approx 4.89(\text{sec})$$

$$\text{SSE} \approx 0(\text{deg})$$

Figure 4: Simulated heading response for $KG_c(s)$, $KG_c(z)$, and Difference equations



Evaluation of control objectives

$\delta_{\max} = 9.43^{\circ}$

Figure 5: Simulated steering command angles for $KG_c(s)$, $KG_c(z)$, and difference equations

Note: the response for heading and the steering command for all three approaches are identical and therefore are difficult to discern individually in Figure 4 and Figure 5.

Step #2: The digital compensator $KG_c(z)$ can be calculated via the `c2d` command in MATLAB using a sampling time of $T_{samp} = 0.01(\text{sec})$ and Tustin's approximation method. Note that other approximation methods can be employed.

$$KG_c(z) = \text{c2d}\left(KG_c, T_{samp}, \text{'tustin'}\right) = \frac{0.0012379(z+1)}{(z-0.9841)}$$

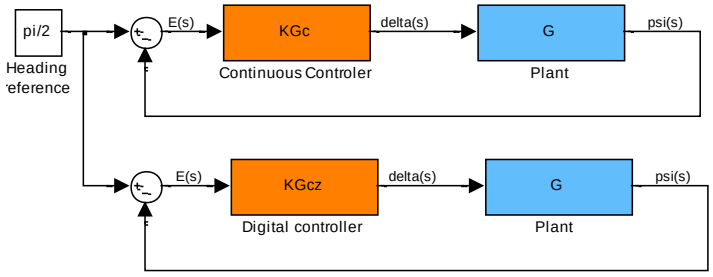


Figure 6: Digital compensator simulation

Step #2 Simulation Results: The heading response for $\psi(t)$ utilizing the digital controller $KG_c(z)$ and the sample time of $T_{samp} = 0.01(\text{sec})$ produces acceptable results (see Figure 4 and Figure 5). If acceptable results were

NOT obtained, one can now focus in on the sample time T_{samp} as being a problematic design parameter or possibly the approximation method utilized.

Step #3: At this stage, the control algorithm $KG_c(z)$ is now ready to be converted from a transfer function representation to a difference equation algorithm. After cross multiplication of $KG_c(z)$, the following control difference equation can be developed:

$$\delta[k] = 0.9841 \cdot \delta[k-1] + 0.0012379 \cdot (e[k] + e[k-1])$$

where k denotes the index number. It is at this point that the contribution of the paper is illustrated. With the recent inclusion of the embedded MATLAB function within the SIMULINK environment, the following model can now be created to simulate the response due to $\delta[k]$:

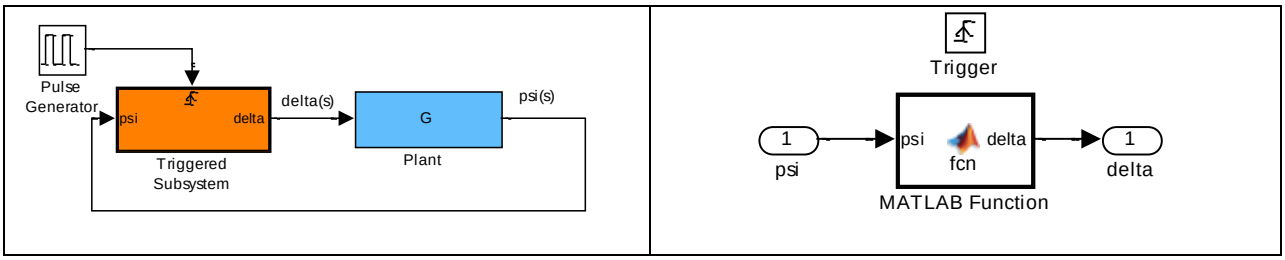


Figure 7: (a) Digital equation simulation with Triggered Subsystem block (b) MATLAB function block within the Triggered Subsystem block

The Triggered Subsystem block coupled with the pulse generator of Figure 7(a) replicates the interrupt driven approach utilized in real-time control implementation. The period of the pulse generator is set to T_{samp}; therefore, the embedded MATLAB function block contained within the Triggered Subsystem block will only be executed in the simulation every T_{samp} seconds. The commanded steering algorithm $\delta[k]$ can now be created with the following embedded MATLAB function code:

```
function delta = fcn(psi)

% Persistent definition of variables causes that variable to be remembered
between function calls. These variables would be declared as GLOBAL in C.

persistent delta1 e1

% We need to initialize variables on first function call. Check to see if the
% variables are empty. If yes, initialize them to appropriate value.
% In C environment, initialize before starting timer that generates interrupts.

if(isempty(delta1))
  delta1 = 0.0;
  e1     = 0.0;
end

% Reference heading command

psiRef = pi/2;
```

```
% Error signal
e = psiRef-psi;

% Digital steering command algorithm

delta = 0.9841*delta1+0.0012379*(e+e1);

% Age variables (these need values need to be remembered)

e1     = e;
delta1 = delta;

return;
```

Table 2: Embedded MATLAB code for Digital control algorithm

The declaration of a variable as persistent in a MATLAB function causes that variable's value to be saved between function calls (as opposed to being erased). The ability to do this greatly simplifies the simulation of difference equation within the SIMULINK environment (one could possible do it by exporting the variable from the function and utilizing a memory block). Upon declaring a variable as persistent, MATLAB subsequently creates the variable but leaves it empty; therefore on the first function call of the control program, all persistent variables will need to be given an initial value by checking to see if that variable `isempty`.

Step #3 Simulation Results: The heading response and the corresponding steering angle command for the difference algorithm of Table 2 are shown in Figure 4 and Figure 5 which demonstrates that $\delta[k]$ was correctly coded. As a result, the difference equation algorithm of Table 2 is now ready to be ported over to the C environment of the Rabbit 3000 microprocessor.

Without the ability to check their difference equation algorithm in simulation, students are often quick to blame the hardware as the culprit when their experiment does not performed as expected; however, the error typically lies within their software code. A common student mistake is to age variables in the incorrect order (youngest to oldest) or the student forgets to initialize their aged variables prior to using them in the control equation. The above simulation process allows the students to remove these common mistakes before integrating their software algorithm with the hardware.

Step #4: The final step in the exercise is to migrate the embedded MATLAB code of Table 2 into the Dynamic C environment. The complete Rabbit SBC Dynamic C code that implements the control strategy $\delta[k]$ (and the longitudinal speed controller) is included in the Appendix A (**Note:**: the boxed in portions of code in Appendix A represent the corresponding code of Table 2). After successful compiling, the EMaxx vehicle was subsequently aligned to true North ($\psi = 0^{\circ}$) and the Dynamic C program of Appendix A was executed with a reference heading change of $\psi = 90^{\circ}$ (due East) . The experimental heading response and commanded steering angle are shown in Figure 8 and Figure 9.

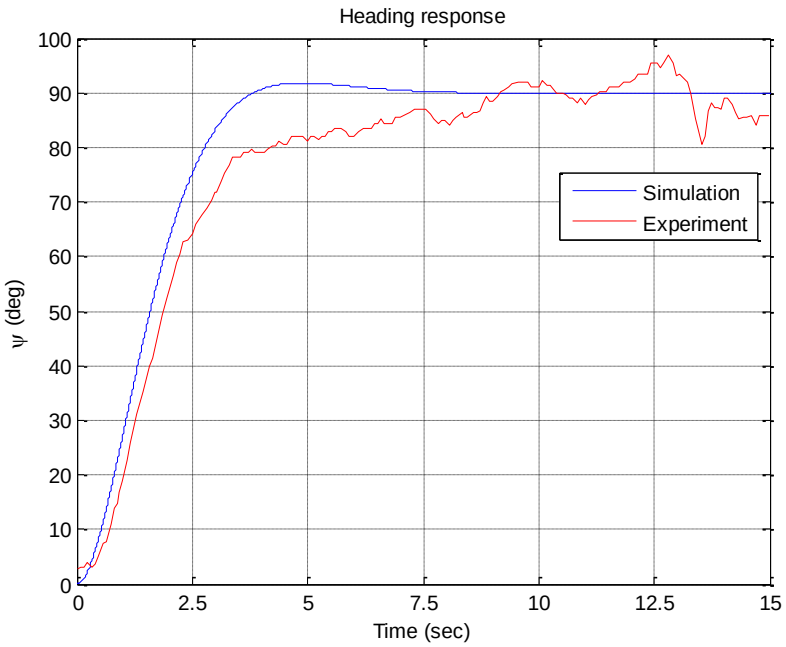


Figure 8: Experimental heading response $\psi(t)$

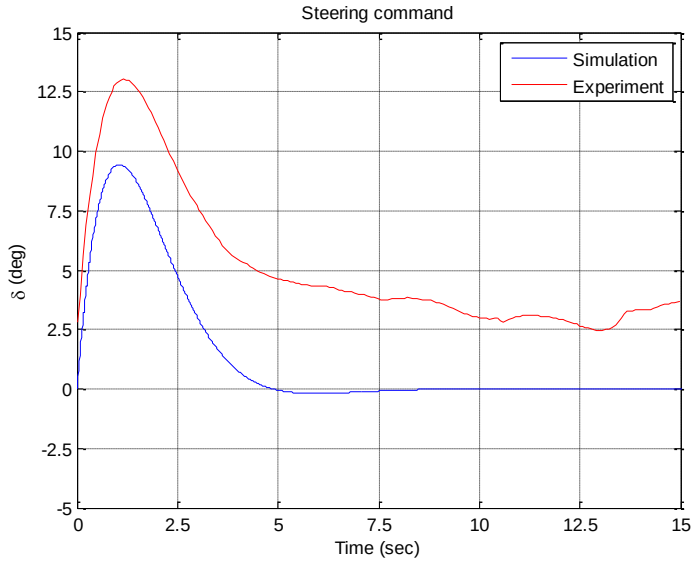


Figure 9: Simulated\experimental steering command angle $\delta(t)$

Step #4 Experimental Results: The above experimental heading response demonstrates adequate performance though differences are observed between the simulation results. The response discrepancies can be largely attributed to the mechanical setup of the EMaxx vehicle. For instance, the actuation of the steering command angle $\delta(t)$ via the stock servo motors is approximate at best. Specifically, the servos that are utilized to actuate the steering linkage are not able to accurately generate small commanded steering angle. Furthermore, the experimental test area is sprinkled with potential magnetic interferences (large iron pipes underneath utilized walkways). Though this may be viewed negatively, these difficulties during testing provided excellent observations for the students.

V. Student Evaluation

A brief questionnaire was attached to the student's final exam to gauge if the above design process had an impact on their ability to successfully develop a digital control algorithm. The following two questions were asked of the twenty-six students:

1.) Please rate your ability to simulate difference equations before/after taking this course (on a scale of 1 to 10).

    From the twenty six respondents, the average ability to simulate difference equations before taking this course was 3.2 out of 10.0 while after the course an 8.6 out of 10.0

2.) Please rate your ability to implement difference equations within a C environment prior to taking this course (on a scale of 1 to 10).

    From the twenty six respondents, the average ability to implement difference equations within C before taking this course was 3.0 out of 10.0 while after the course an 7.7 out of 10.0

Though the students seem to feel more confident of their ability to implement a digital control algorithm successfully, it will be interesting to see if this methodology is carried over into the execution of their senior capstone projects. In addition to the numerical scoring, some students offered the following comments on the questionnaire form:

"*I certainly learned how to simulate the C code within an embedded matlab function and the IMPLEMENTING within a C environment*"

"C code still gives me a little trouble."

"I am confident in my ability to simulate to simulate difference equations after taking this course."

"I now understand this topic better. Any weaknesses lie in my C-coding abilities, not in my understanding of difference equations."

VI. Conclusion

In this paper, a design process was presented that provides a straightforward methodology for undergraduate control students to verify their digital difference equations in simulation prior to involving any hardware concerns. The proposed method is facilitated by the recent addition of embedded functions to the MATLAB/SIMULINK simulation environment. The resulting embedded MATLAB function code requires little (only in variable declarations) modification to be implemented within a C environment. The process was successfully employed in digital control course exercise where a heading controller was developed for an automated ground vehicle. Initial student responses seem to indicate that the methodology is beneficial.

VII. Appendix A

```
#define SPI_SER_B               // Choose serial port B for SPI bus
#define SPI_CLK_DIVISOR 5       // Minimal clock divisor
#define MM3_PERIOD_SELECT 2

#use "spi.lib"                  // Contains the SPI functions
#use "NAVBD3_SBC_LIB.lib"       // Contains Nav Board library function
#use "ROVER_LIB.LIB"            // Library with vehicle based functions

// ============================================================================
// SBC Rabbit Parameters
// ============================================================================
#define XTAL_FREQ       (14.756) // (Mhz) - Crystal frequency
#define Timer_Freq      (100.0)  // (Hz)  - Timer B routine frequency

// ============================================================================
// Function prototypes
// ============================================================================
nodebug root interrupt void TimerRoutine(void); // Control calculations

// ============================================================================
// Global definitions
// ============================================================================
int   ControlIteration, LED_flag;

float psiRef, psi, delta, e, delta1, e1;

float time, enable, v, psiRefInit;
float Mx, My, Mz, steerPulseWidth;
float x, xDot, xOld;
float speedPW, ev, evOld, evInt, vRef;

// ============================================================================
// main()
// ============================================================================
void main()
{
  // Board initializations
  NavBd3_Init();

  // Initialize encoder
  qd_init(1); qd_zero(1);

  // Set all PWM ports back to 1.5 (msec)
  set_servo(1,1.5); set_servo(2,1.5); set_servo(3,1.5);

  // Variable initializations
  time   = 0.0;  enable = 1.0;  xOld   = 0.0;  evOld  = 0.0;  evInt  = 0.0;
  delta1 = 0.0;  e1     = 0.0;

  // Start timer routine execution
  TimerBInit(Timer_Freq);

  // Get initial heading. Use it as the reference heading
  psiRefInit = getCompass(MM3_PERIOD_SELECT,&Mx, &My, &Mz)*PI/180.0;
```

```
  // Main loop
  while(time<20.0)
    printf("%.2f,%.2f, %.1f, %.1f, %.2f
\n",time,v,psiRef*180.0/PI,psi*180.0/PI,delta*180.0/PI);

  // Stop timer routine execution
  TimerBUninit();

  // Set all PWM ports back to 1.5 (msec)
  set_servo(1,1.5); set_servo(2,1.5); set_servo(3,1.5);

  while(1);
}

// ============================================================================
// TimerRoutine() - This routine gets executed every 1/Timer_Freq (sec)
// ============================================================================
nodebug root interrupt void TimerRoutine(void)
{
  // Reset IRQ
  RdPortI(TBCSR); WrPortI(TBL1R,NULL,0); WrPortI(TBM1R,NULL,0); ipres();

  // Increment time variable
  time = time + 1.0/Timer_Freq;

  // Measure longitudinal position/velocity
  x    = ((float)qd_read(1))/(-11460.0)*(0.5016);
  xDot = (x-xOld)*Timer_Freq;
  xOld = x;
  v    = xDot;

  // Measurement of ture heading
  psi = getCompass(MM3_PERIOD_SELECT,&Mx, &My, &Mz)*PI/180.0;

  // Wait 5 seconds to command change in heading (let's vehicle get up to
speed)
  if(time>5.0)
    psiRef = psiRefInit+PI/2.0;
  else
    psiRef = psiRefInit;

  e = psiRef-psi;
  delta = 0.9841*delta1+0.0012379*(e+e1);

  // Age variables
  e1     = e;
  delta1 = delta;

  // Saturate steering command
  if(delta> (30.0)*PI/180.0)
    delta = (30.0)*PI/180.0;

  if(delta< (-30.0)*PI/180.0)
    delta = (-30.0)*PI/180.0;

  // Calculate appropriate pulse width to generate  delta steering angle
```

```
  steerPulseWidth = 1.5-enable*delta/((25.0)*PI/180.0)*(0.5);

  // Set PWM port #1 and #2 (steering servos) to a pulse width in mSec
  set_servo(1,steerPulseWidth); set_servo(2,steerPulseWidth);

  // Linear speed controller
  vRef    = 1.5; // (m/sec)
  ev      = vRef-v;
  evInt   = evInt+0.5*(1.0/Timer_Freq)*(ev+evOld);
  evOld   = ev;

  speedPW = (0.2)*ev+(0.01)*evInt;

  // Saturate command for speed
  if(speedPW>  0.5)
    speedPW =  0.5;
  if(speedPW< -0.5)
    speedPW = -0.5;

  // Set PWM port #3 (speed controller) to a pulse width in mSec
    set_servo(3,1.5+enable*speedPW);
}
```