Using The PC In Hardware Control Projects

By John Blankenship and Samuel Mishal

The history of using the Personal Computer (PC) in electronic control projects parallels the history of using the mainframe computer. In the past, when only mainframes were used, the processing power had to be time-shared among multiple users. With the advent of the microcomputer every user had a dedicated processor. However, engineers quickly realized that networking these dedicated systems using the mainframe as a networking hub, database storage, and coordinator allowed for more powerful computing than would be possible with the mainframe or the microcomputer alone.

The early PC, with its parallel port, ISA bus, and serial port provided a viable controller for electronic hardware projects. However, programming hardware I/O on the PC has become progressively more complicated with each new version of the Windows OS. In addition, with the availability of powerful and easy-to-use microcontrollers (µCs) many hobbyists have found it a lot easier to use them for their projects and nowadays are only using the PC as a cross-compiler.

Many researchers use the PC as a controller for their projects, often with a wireless connection to communicate to various systems on a mobile platform (see this web site for an example: mit.edu/whall/www/heli/paper/node3.html#SECTION00030000000000000000). The systems are organized with µCs to handle the hardware level tasks and the PC acting as the Artificial Intelligence (AI) processor and user interface node. Often, these projects are robotic in nature, but the methodology does not have to be limited to robotics. You can use the same strategy with any control project you wish.

As a hobbyist you may think that the above is too complicated; we hope this article will convince you of the contrary. To demonstrate these principles this article discusses a project that emulates controlling the orientation of a satellite. It shows how a PC along with µCs in a pseudo-networked arrangement can provide processing capabilities that are otherwise difficult to achieve. The programs developed for the project demonstrate that programming a PC can be easily accomplished with the right tools and strategies. The simulation portion of the project shows that incorporating a PC can provide functionalities that would be impossible with most µCs.

Distributed Processing

A control project can be thought of as a set of subtasks. Each subtask can be controlled by a dedicated µC along with some additional circuitry. The overall project is coordinated by a master controller (MC) that communicates with the various µCs. The MC could be a PC or just another µC. The distributed processing provided with this *divide-and-conquer* strategy, allows the MC to have less I/O pins than would have been required if it had to control all the sub-processes directly. Also due to *concurrent processing* provided by the various µCs, multitasking can be readily implemented.

Often you may find that there are modules available on the market that can control the various subtasks of your project. We call these modules Helper Modules (HMs). These HMs themselves are often µCs that use their own I/O pins and memory to accomplish their task and are capable of being controlled using a serial protocol (SPI, I^2C, RS232 etc.). If you find that you need an HM that does not exist, you can design one using a µC and supporting circuitry.

Incorporating A PC

A system of µCs can be very powerful, but it does have its limitations. Most µCs are limited in their ability to manipulate arrays and perform *floating-point* as well as other high-level math operations.

Even simple *multiplication* and *division* are limited to 8 or 16 bits on many μCs. Simple projects may not require many mathematical calculations, but more complex projects that involve PID control or signal processing (DSP) for example, will usually require the processing power of a PC. Using a distributed processing strategy, the PC can become another element of the overall project or can act as the overall controller.

Another advantage to using a PC in control projects is that with its graphics capabilities you can create an *ergonomic user interface.* Just as hardware HMs help in quickly and easily accomplishing subtasks, RobotBASIC, the language used on this project, provides many *software HMs* such as matrix mathematics, bitmap manipulations, and extended graphics operations for flicker free animation (see the help files for many more HMs). Using the graphics capabilities you can create an effective user interface and with the math and matrix operations you can program complex *AI algorithms*.

Many hobbyists opt to avoid using the PC in their projects because the Window's OS makes it very difficult to program I/O communications. RobotBASIC provides many HMs for sending and receiving data through the PC's serial and parallel ports, or over Bluetooth wireless communications, thus eliminating this obstacle.

Satellite Heading Control

As mentioned earlier, we will model the control of an artificial satellite in order to illustrate the principles of distributed processing using μCs along with the PC. Only heading (yaw) control will be implemented (pitch and roll control follow along the same principles and can be implemented as separate systems). The project consists of a simulation of the satellite that can be controlled manually or automatically (using PID see later), and a real physical satellite model that can also be controlled manually or automatically using *the same* control program. Switching over from controlling the simulation to controlling the physical model or from manual to automatic control is accomplished with a click of a mouse button.

In order to control a satellite's orientation in space, a terrestrial control station (TCS) needs to obtain information about the state of the satellite in order to calculate what actions to take (see Figure 1). This information is gathered and transmitted to the TCS by a master controller (MC) aboard the satellite from HMs that interrogate the appropriate transducers (sensors). The TCS uses the received data to calculate the necessary actuator settings (motor speeds and so on) and transmits them to the MC as well as displaying information to the operator of the system. The MC uses the received settings to command HMs to activate actuators in order to move the satellite to the desired state. A major advantage of this setup is that the HMs can operate concurrently and independently of each other.

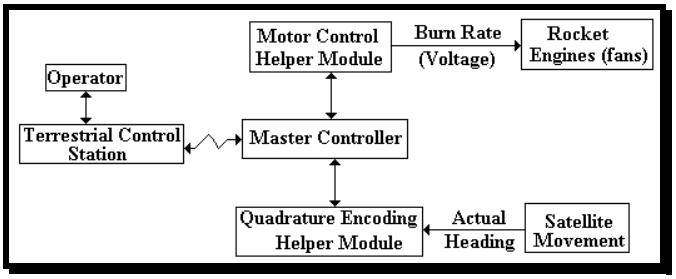

Figure 1: Satellite Control System

PID Control

To control the heading of the satellite two opposing retrorockets are used. One rocket is fired to start the satellite rotating. Since in space there is no friction or wind resistance, the second rocket has to be fired to stop the rotation. The burn rate of the fuel along with its energy value, determine the amount of

force the rocket exerts. The control algorithm calculates the burn rate to use, the duration of the burn, as well as when to start the opposing rocket.

The simplest form of control is *proportional-control* and is implemented as follows. The actual heading of the satellite is compared to the desired heading. The difference between the actual and the desired heading (the error) is used to determine the amount and direction of burn rate (force) needed in order to correct the error. This means that the bigger the error, the higher the burn rate required. In many systems this control method alone will not suffice.

In a friction-free satellite, proportional control alone will generally overshoot the desired heading, causing the controller to reverse the force being applied resulting in an oscillation. There are four types of oscillations, as shown in Figure 2. In Figure 2(a) the satellite overshoots the desired heading but goes back and reaches the desired value (settles). In Figure 2(b) the same happens but after multiple oscillations that decrease in amplitude. In Figure 2(c) the satellite never settles at the required heading but continues to oscillate about that value with a constant amplitude. In Figure 2(d) an unstable situation occurs where the satellite oscillates about the desired heading with an ever-increasing amplitude.

Obviously the responses in Figure 2(b, c, d) are always unacceptable. Some applications can accept a small amount of overshoot (Figure 2(a), perhaps to achieve a reduced overall settling time), while others cannot. An optimal behavior is shown in Figure 3(a). In this response the satellite reaches the desired heading as quickly as possible but without any overshoot or oscillation. In Figure 3(b) the settling time is a little longer (over damped). This may be required if we need to limit the acceleration the satellite is allowed to undergo due to equipment or other considerations.

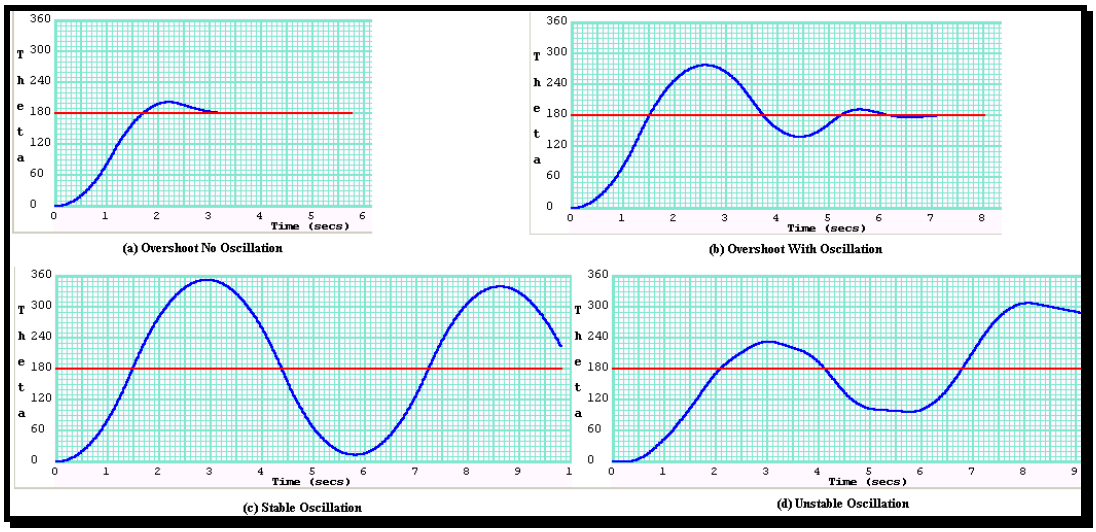

Figure 2: Types Of Undesirable Responses

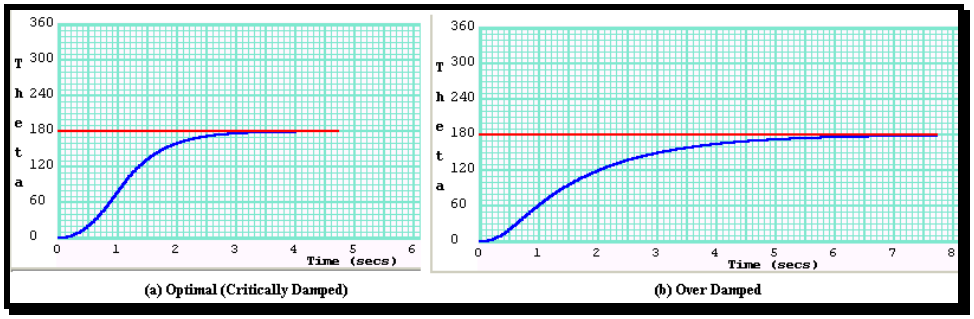

Figure 3: Desirable Responses

To achieve the kind of response shown in Figure 3 the rate of change of the heading of the satellite and the accumulated error should be taken in consideration. Lets suppose that the error is small – that is the satellite is close to its desired destination. With proportional-control alone, a small force would be applied (small error) but still continuing to move the satellite toward the desired heading. However, if the speed of the movement is fast, the opposing rocket should be fired to slow down and prevent an overshoot. Since speed is the rate of change (derivative) of the position, this form of control is called *derivative-control*.

Another aspect of the movement of the satellite that we need to consider is its accumulated error (over time). Summing the error over time is the calculus concept of integration and therefore this control method is called *integral-control*. If the error is very small, proportional-control alone might not cause enough force to actually move the satellite. Applying a force in relation to the sum of the error over time, no matter how small the error, would eventually command a force big enough to move the satellite to reduce the error.

A system that utilizes all three of these control mechanisms is called a *PID-Controller* (proportional, integral, derivative). If the controller monitors the current position of the satellite it can calculate the rate of change (derivative) and a running sum (integral) of the error. These three pieces of information can be used to calculate the amount of burn rate (force) that should be applied and in what direction.

The three control methods described above will contribute to the value of the burn rate but not in equal proportions. Three factors, K_I, K_P, and K_D will be used to specify the proportion of each method. The behavior of the system (overshoot, oscillation, settling time, steady state error, and so forth) can be controlled by adjusting these three parameters. Calculus and convoluted mathematical methods are traditionally used to determine these factors, but if you have a good simulation (see later) you can determine the values by informed trial and error.

The Satellite Model

To model a satellite we used a disc of 1/4-in foam-board (found at many craft and office supply stores). The disc needed to be supported in such a manner to provide minimum friction. To rotate the disc we used balsa wood pieces mounted perpendicular to the foam-board so as to provide a surface for two fans (opposing) to blow over and thus generate the necessary torque. The fans where mounted apart from the wheel so as not to contribute more weight (and thus friction). The arrangement is shown in Figure 4. See the YouTube video of the system at: http://www.youtube.com/watch?v=LwvspYFXJMM.

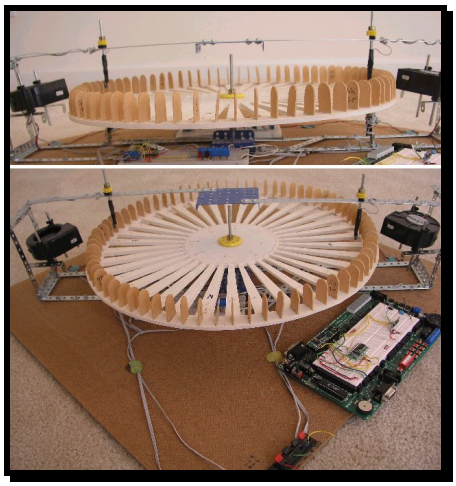

Figure 4: Satellite Model

To control the fans' motors we used a Pololu module from Parallax Inc. (www.parallax.com). This module can be controlled using a TTL RS232 input to perform pulse wave modulated (PWM) bi-directional control of one or two DC motors. We also added opto-isolation circuitry to provide noise free control (this was essential). The circuit schematic is shown in Figure 5.

To measure the actual heading of the wheel we used a quadrature encoder that utilizes a μC and two pairs of infrared transceivers as described in a previous article (see Figure 6). The wheel was cut to create 36 spokes (5º each) to provide blocking for the encoder's infrared beams giving a resolution of 2.5º.

To coordinate the communication with the PC system (TCS) we used a Basic Stamp (BS2) as a MC (see Figure 7). The BS2 receives the voltage level for the fans (and direction) then commands the Pololu module with the required value. Afterwards, the BS2 interrogates the quadrature encoder to find out the quadrature count and sends it to the PC. For simplicity and lesser cost, the communication between the PC and BS2 was implemented using a wired serial line. This can be easily substituted with an EB500 module on the BS2 side and a USB Bluetooth module on the PC side.

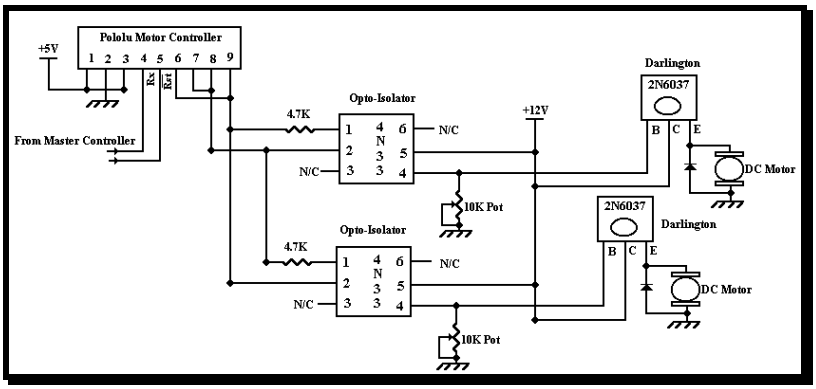

Figure 5: Fans' Controller

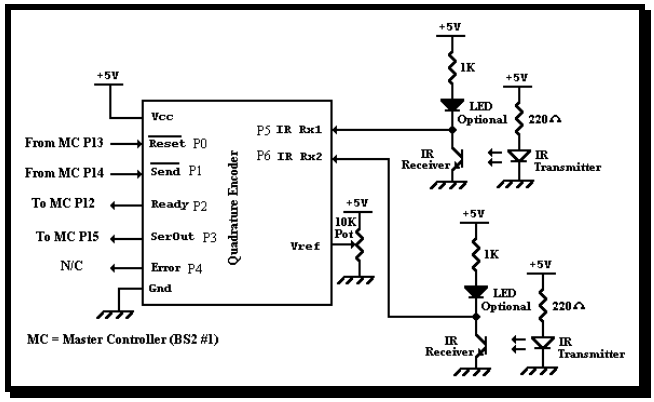

Figure 6: Quadrature Encoder

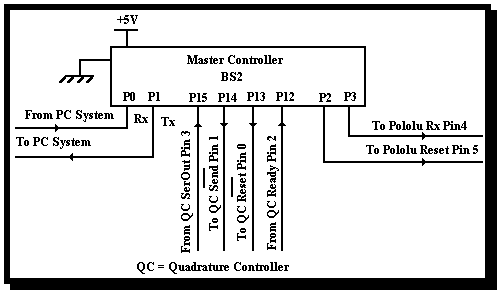

Figure 7: Master Controller.

The Simulation

Simulations are an indispensable part of a project such as this one. Design engineers use the simulation to hone their control algorithms and parameters. Cost engineers use the simulation to decide feasibility and costs. Training engineers use the simulation to train operators and to determine the optimum *human-machine* interface parameters. All this can be achieved long before the actual physical project is even started.

To create a realistic simulation you need to use physics formulas and calculus to determine the behavior of the system. The values of importance are shown in Figure 8. The velocity coefficient (VCOF meters/second) multiplied by the Burn Rate (Firing kilograms/second) gives the force of the rocket. This multiplied by the Radius (position of rocket's force from center of rotation in meters) gives the Torque applied. Dividing the Torque by the rotational inertia (J kilograms times meters squared) gives the rotational acceleration (α radians per seconds squared) applied to the satellite. Integrating this value once gives the rotational speed of the satellite (ω radians per second), integrating again gives the current heading of the satellite (θ radians). The radians value of the heading is then converted to degrees for display purposes. All these values were calculated from the characteristics of the wheel and fans in order to make the simulation match the physical model.

During the control of the simulation the current heading θ is calculated (by integration) from the calculated acceleration. During control of the physical model the current heading θ is calculated from the quadrature encoder count. Thus to get ω and α, we need to differentiate.

Integrating and differentiating are performed using discrete (digital) techniques, and the sampling interval (T seconds) becomes extremely important. The sampling rate must not be less than twice as fast as the maximum frequency of the system. The time between samplings (T) is constrained by the speed with which the PC can communicate with the MC on the model. The value that is used for the simulation is calculated for the speed of the BS2 processor. The behavior of the system is greatly affected by the PID factors and these in turn, are affected by the particular T used.

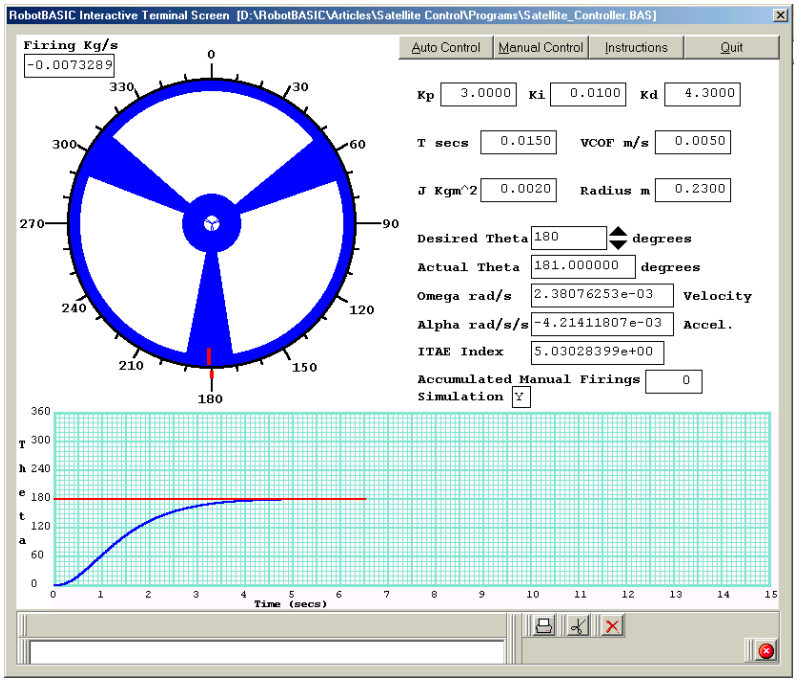

Figure 8: The Simulation & Control Program

The Control Program

The control program provides an interface (Figure 8) to allow the user to command the firing of the fans (rockets) during the manual control process by using the arrow keys on the keyboard or the mouse buttons. During automatic control, the program allows the user to indicate the desired heading either by typing it as a number or by gradually increasing or decreasing the value by clicking the mouse on an up or down scroller next to the field. Also the field labeled "Simulation" allows the user to specify whether the control procedure should be applied to the simulated satellite or to the physical model. If

this field is set to 'N' the program will use serial communications (wired or wireless) to command the physical model described above.

The program also provides the user with the ability to modify the system's specifications by changing the PID factors and the values of the physical characteristics. Other fields show the current heading (θ), the rotational speed (ω), acceleration (α), the ITAE value (see later) and firing (burn rate). A graph shows a visual history of the system's response where the desired heading is drawn in red and the actual heading in blue along a time axis. This can be helpful in calculating the settling time and overshoot as well as giving a visual indication of the system's response. Figures 2 to 11 are screen captures of this graph.

The ITAE value is the integral over time of the absolute value of the error multiplied by time. This value provides a numerical measure for the effectiveness of the control system. Smaller numbers indicate a more effective system. This can be used (along with the response graph) during the determination of the PID factors to decide which factors are better. It can also be used to give a comparison reference between automatic and manual control effectiveness.

There are three programs, the TCS program on the PC written in RobotBASIC and MC program written in PBasic as well as the Quadrature Encoder program in PBasic. The TCS program is too long to show in full here and only a part of it is listed in Figure 12 to illustrate how the calculations mentioned above are implemented in code. Figures 13 and 14 are listings of the MC and Quadrature Encoder programs respectively. You can download all the programs from www.RobotBASIC.com along with a copy of the RobotBASIC interpreter and numerous demo programs. The programs are well commented and should be very easy to study and understand. You do not need to have a physical model to be able to use the TCS program. The simulation is fun to use by itself and is very informative. You can experiment with changing the PID parameters to gain a feel for how they affect the system's response.

The values used in Figure 8 represent the characteristics of the model we built. If you build your own satellite model you will have to change the PID factors as well as the values of the physical characteristics. Of course, the physical characteristics only matter during the simulation, but to have a simulation that resembles your physical model you need to change the values so as to have a good representation of your physical model. The PID factors will also have to be changed if the model is different, but you can use the simulation to try different values and see how they affect the response and then try them on the real model.

Conclusion

Figures 9 and 10 show how the simulated system and the physical model (respectively) responded to commanded step and gradual heading changes. Figure 11 shows how during the control of the physical model, if a disturbance (e.g. moving the wheel by hand) is experienced, the control system manages to restore the wheel back to the desired heading. The slight jerkiness and lag in the response of the physical model is due to friction, which, of course, cannot be totally eliminated.

We hope you can see how this project has been enhanced by using the PC as a controller. The user interface provided pertinent information for monitoring the status of the system, and the math capabilities facilitated programming of the necessary complex algorithms. With the right programming language and distributed processing, using a PC was not a difficult task and has provided a new level of hardware control methodology not often considered as viable by many hobbyists.

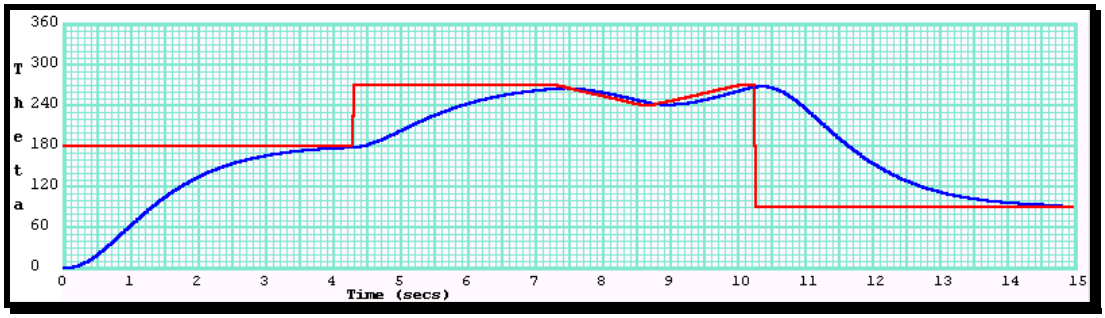

Figure 9: Simulation Control Response

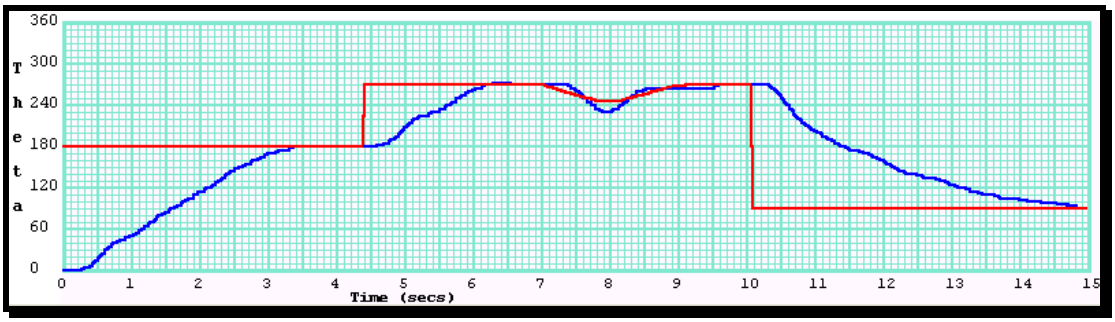

Figure 10: Physical Model Control Response

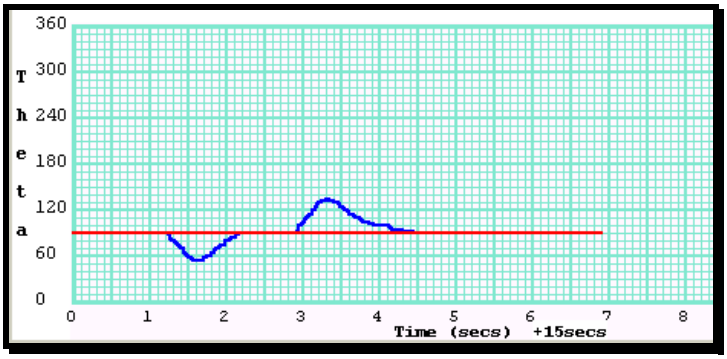

Figure 11: Disturbance Rejection Response

```
MainProgram: //only some subroutines are shown
  gosub SetUp
  gosub Instructions
  gosub MonitorInputs
  gosub FinishUp
End
//=====================================================================
CalcPID: //A0,A1,A2 are calculated from Ki,Kp,Kd in another subroutine
   PID_output = A0*error+A1*PID_errorZ1+A2*PID_errorZ2+PID_outputZ1
   PID_outputZ1 = PID_output
   PID_errorZ2 = PID_errorZ1
   PID_errorZ1 = error
   BurnRate = Limit(PID_output,-5,5)  //limit to +/-5V
Return
//=====================================================================
CalcError:
   error = desired -actual
   if abs(error) > Pi() then error = error - sign(error)*Pi(2)
       //make the error <= 180 degrees
Return
//=====================================================================
CalcResponse: //simulation response
   //----Calc Theta, dTheta (W), ddTheta (Alpha)
   Torque = BurnRate*VCOF*Radius
   ddThetaZ1 = ddTheta
```

```
   dThetaZ1 = dTheta
   ThetaZ1 = Theta
   ddTheta = Torque/J  //calc acceleration
   dTheta = dThetaZ1+T/2*(ddTheta+ddThetaZ1) //integrate to get velocity
   Theta = ThetaZ1+T/2*(dTheta+dThetaZ1)  //again to get heading
   Alpha = ddTheta
   W = dTheta
   actual = frac((iactual+Theta)/Pi(2))*Pi(2) //limit heading to 0-359 deg
   if actual < 0 then actual = actual+Pi(2)
   aTheta = round(rtod(actual))
Return
//=====================================================================
CalcSatelliteResponse: //physical model response
   if within(BurnRate,-0.3,0.3) then BurnRate = 0 //make a dead zone
   if BurnRate < 0
      CS_Speed =round(BurnRate/5.0*55-65) //left and right fans need
   elseif BurnRate > 0                    //different voltage levels
      CS_Speed =round(BurnRate/5.0*70+50) //to produce the same speeds
   Else
      CS_Speed = 0
   Endif
   CS_Sp = CS_Speed
   if CS_Speed < 0 then CS_Sp = abs(CS_Speed) | 128
   serout char(CS_Sp)  //send the voltage level (coded with direction)
   serbytesin 2,CS_dAngle,CS_NoIn //receive the quadrature count
   if CS_NoIn < 2                 //as two bytes
      n=ErrMsg(Msgs[10],Msgs[0],MB_OK|MB_ERROR)
      GoSub StopSatellite
      CommsError = true
      return
   endif
   CS_HB = ascii(substring(CS_dAngle,1,1)) //assemble the encoder count
   CS_LB = ascii(substring(CS_dAngle,2,1))
   CS_dAngle = (CS_HB<<8)+CS_LB
   if CS_HB & 128  //if negative do tow's complement
      CS_HB= ~CS_HB & 255
      CS_LB = ~CS_LB & 255
      CS_dAngle = -((CS_HB<< 8)+CS_LB+1)
   endif
   ddThetaZ1 = ddTheta
   dThetaZ1 = dTheta
   ThetaZ1 = Theta
   Theta = CS_dAngle*dtor(2.5)    //heading = count * 2.5 degrees
   dTheta = (Theta-ThetaZ1)/T     //differentiate to get omega
   ddTheta = (dTheta-dThetaZ1)/T  //again to get alpha
   Alpha = ddTheta
   W = dTheta
   actual = frac(Theta/Pi(2))*Pi(2) //make sure heading is 0-359 deg
   if actual < 0 then actual = actual +pi(2)
   aTheta = round(rtod(actual))
Return
//=====================================================================
```

Figure 12: Partial Listing Of The Control Program

```
' {$STAMP BS2}
' {$PBASIC 2.5}
'~~~~~~~~~~~~~Master Controller ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'============================================================
'=============    Variables         ==========================
'============================================================
    ReceivePin    PIN 0
    SendPin       PIN 1
    MotorReset    PIN 2
    MotorTx       PIN 3
    QuadratureRx  PIN 15
    QuadratureInt PIN 14
    QuadratureRst PIN 13
    QuadratureRdy PIN 12
    dAngle        VAR Byte(2)
    MotorSpeed    VAR Byte
    oMotorSpeed   VAR Byte
    Direction     VAR Nib
'============================================================
'=============    MainProgram       ==========================
'============================================================
Main:
  GOSUB Initialize
  DO
    SERIN ReceivePin,84,[MotorSpeed]
    IF MotorSpeed = 255 THEN GOTO Main  'reset
    GOSUB SetMotors
    GOSUB Quadrature
    SEROUT SendPin,84, [dAngle(0),dAngle(1)]
  LOOP
END
'============================================================
'========   Subroutines                  ====================
'============================================================
Initialize:
  HIGH QuadratureInt   'No interrupt on quadrature
  LOW  QuadratureRst
  PAUSE 10
  HIGH QuadratureRst  'Reset Quadrature
  LOW  MotorReset
  HIGH MotorReset  'reset motor
  PAUSE 100
  SEROUT MotorTx,84,[$80,0,0,0] 'motor 0 brake
  PAUSE 20
  MotorSpeed  = 0
  oMotorSpeed = 0
  Direction = 0
  dAngle = 0
RETURN
'============================================================
SetMotors:
  IF MotorSpeed.BIT7 = 1 THEN
     SEROUT MotorTx,84,[$80,0,0,MotorSpeed & 127] 'backwards (ccw)
  ELSEIF MotorSpeed = 0 THEN
     SEROUT MotorTx,84,[$80,0,0,0] 'brake
  ELSE
     SEROUT MotorTx,84,[$80,0,1,MotorSpeed & 127] 'forward (cw)
  ENDIF
RETURN
'============================================================
Quadrature:
    LOW  QuadratureInt
    DO WHILE (QuadratureRdy = 0)
    LOOP
    HIGH QuadratureInt 'interrupt the quadrature
    SERIN QuadratureRx,84,[STR dAngle\2]
RETURN
'============================================================
```

Figure 12: The Master Controller Program in PBasic for the BS2.

```
' {$STAMP BS2}
' {$STAMP BS2}
' {$PBASIC 2.5}
'~~~~~~~~~~~~~~Quadrature Controller~~~~~~~~~~~~~~~~~~~~~~~~~~~
'=============================================================
'=============    Variables         ==========================
'=============================================================
    Interrupt   PIN 0
    SendPin     PIN 1
    ResetPin    PIN 2
    ReadyPin    PIN 3
    LeftIR      PIN 15
    RightIR     PIN 14
    OldIRS      VAR Nib
    IRS         VAR Nib
    dA          VAR Nib
    dAngle      VAR Word
    HdAngle     VAR dAngle.HIGHBYTE
    LdAngle     VAR dAngle.LOWBYTE
    Error       VAR Bit
    Direction   VAR Nib

'=============================================================
'=============    MainProgram       ==========================
'=============================================================
Main:
  GOSUB Initialize
  DO
    GOSUB Quadrature
    IF ResetPin = 0 THEN GOTO Main
    IF Interrupt = 0 THEN
       HIGH ReadyPin
       GOSUB Quadrature
       LOW ReadyPin
       SEROUT SendPin,84, [HdAngle,LdAngle]
    ENDIF
  LOOP
END
'=============================================================
'========   Subroutines                    ====================
'=============================================================
Initialize:
  OldIRS.BIT0 = RightIR
  OldIRS.BIT1 = LeftIR
  dAngle = 0
  Direction = 0
  LOW ReadyPin
  INPUT Interrupt
RETURN
'=============================================================
Quadrature:
    Error = 0
    dA = 0
    IRS.BIT0 = RightIR
    IRS.BIT1 = LeftIR
    IF IRS <> OldIRS THEN
      dA = 3
      SELECT OldIRS
        CASE 0
           IF IRS = 2 THEN dA = 1
           IF IRS = 1 THEN dA = 2
        CASE 1
           IF IRS = 0 THEN dA = 1
           IF IRS = 3 THEN dA = 2
        CASE 2
           IF IRS = 3 THEN dA = 1
           IF IRS = 0 THEN dA = 2
        CASE 3
           IF IRS = 1 THEN dA = 1
           IF IRS = 2 THEN dA = 2
      ENDSELECT
      OldIRS = IRS
    ENDIF
    SELECT dA
       CASE 1
```

```
          dAngle = dAngle-1

       CASE 2
          dAngle = dAngle+1
       CASE  3
          Error = 1
          IF Direction = 1 THEN dAngle = dAngle-1
          IF Direction = 2 THEN dAngle = dAngle+1
    ENDSELECT
    IF dA <> 3 THEN Direction = dA
RETURN
'=============================================================
```

Figure 12: The Quadrature Encoder Program in PBasic for the BS2.