CHAPTER 13

An Introduction to Control Systems

Chapter Outline

13.1 Control Systems

We have already implemented a number of *electromechanical control systems*, which can be broadly defined as electronic and/or software systems that ar e designed to control a designated physical variable. For example, in Chapter 4 we discussed a system that moved a servo to a desired position by applying the correct pulse width modulation (PWM) waveform. Electromechanical control systems are very common, with applications ranging from home and office equipment, to industrial machinery, automotive vehicles, robotics and aerospace. Think of how much control there must be in a humble inkjet printer: drawing in and guiding the paper, shooting the print cartridges backwards and forwards with extraordinary precision, and squirting just the right amount of ink, of the right color, in just the right place. The accuracy of a control system is a very important aspect, as errors and inaccuracy can have significant relevance to performance, wear and safety.

The whole topic of control systems is a very detailed and broad field, which draws on advanced mathematical theory as well as electronics, digital communications, applied

Fast and Effective Embedded Systems Design. DOI: 10.1016/B978-0-08-097768-3.00013-1

mechanics, and sensor and actuator design. It is not possible to give a thorough insight into control systems in this book, but the mbed is a valuable device which can be used to implement some simple control algorithms and communication methods. This chapter therefore acts as a starting point for the reader who may wish to go on to study or implement control systems at a later date.

13.1.1 Closed and Open Loop Control Systems

Closed loop control systems employ internal error analysis to achieve accurate control of actuators, given continuous sensor input readings. For example, this means that the system measures the position of an actuator, compares it with a signal representing the position it is meant to be at (called the *setpoint*), and from this information tries to move the actuator to the correct position. The fundamental components of a closed loop system are therefore an actuator, which causes some form of action; a sensor, which measures the effect of the action; and a controller, to compute the difference between setpoint and actual position. Although we say position, any physical variable can be controlled. For example, a heater element might be used to heat an oven to a specified temperature and a temperature sensor used to measure the actual heat of the oven. If the closed loop controller tells the oven to be at 200°C then the heater will provide heat until the sensor signals the controller that 200°C has been achieved. If the sensor measurement shows that the actuator has achieved its desired action (i.e. 200°C is achieved), then the error between the desired setpoint and the actual reading is zero and no further action is required. If, however, the sensor reads that the desired action has not been achieved (e.g. the temperature is only 190°C), then the control system knows that more action is required to meet the desired setpoint.

An *open loop control system* is one that employs no analysis sensor and relies on a calibrated actuator setting to achieve its desired action. For example, a 5 V electric motor might rotate at different speeds over the voltage input range, say 0 V gives 0 revolutions per second (rps) and 5 V gives 100 rps. If we want to rotate the motor at 50 rps we might simply assume that the response characteristic of the motor is linear (or have a calibrated look-up table for its response profile) and provide 2.5 V. However, we will not actually know if the motor is rotating at the desired speed; for different operating temperatures, with different friction loading and with component drift over time, the speed achieved given 2.5 V input cannot be accurately predicted. Here, an improved closed loop system with a speed sensor could be used to modify the input voltage and ensure that an accurate 50 rps is achieved.

Advantages of closed loop systems are improved accuracy of a controlled variable and the ability to automatically make continuous adjustment. In fact, many actuators are quite erratic and non-linear in terms of performance, so in many cases closed loop control is a necessity rather than a desire. Closed loop systems built around fast microcontrollers can also allow advanced control of systems that were previously thought uncontrollable. For example, the

Figure 13.1:
The Segway personal transporter (www.segway.com). *(Image reproduced with permission of Segway Inc.)*

Segway personal transportation systems (Figure 13.1, Reference 13.1) use sensitive gyroscopes to ensure that the standing platform always remains horizontal. If weight shifts forward (and the gyroscope shows an imbalance) then the motor in the wheels moves forwards marginally to compensate and stops moving when a horizontal position has been achieved again. The microcontroller, sensors and actuators inside read and compute so rapidly that this process can be performed faster than a human's motion can displace their weight, so the controller can always ensure that the platform stays stable.

13.1.2 Closed Loop Cruise Control Example

Another example of closed loop control is *cruise control* in many automotive vehicles. Here, the driver chooses a desired speed and the vehicle automatically maintains that speed until the driver disables the mode, either by switching it off or by applying the brake.

A key point to note is that the actuator performs a different operation from the specific sensor reading that is being used for comparison with the setpoint. For example, the setpoint might be a vehicle speed of 80 kph, but the actuator is actually the electronic throttle valve, which determines the amount of air and hence fuel that is drawn into the engine on each piston cycle. So a complex chain of events occurs, from automatically moving the throttle, to sucking air and fuel into the engine, engine combustion, torque being applied to the wheels, the vehicle moving, and the vehicle motion finally being measured and converted to a horizontal speed value in kph. Many factors affect the accuracy of this process, for example changes in air pressure, fuel type, engine wear, tire size and tire pressure, as well as road friction and

environmental factors such as drag and gravity. An open loop system could never be possible for cruise control.

A closed loop control system can be quite simple. All we need to do, given a microprocessor-based system, is to write a control algorithm which takes a setpoint (desired speed chosen by the driver) and calculates the error between the setpoint and the actual speed. The algorithm then uses this error to calculate the required action. For example, if the error is negative (actual speed is less than the desired setpoint) then the action must be positive to increase the amount of fuel to the engine, increase the torque to the wheels and increase the car speed. If the error is positive, then the action must be negative, to reduce the car speed to the setpoint. The use of *negative feedback*, where the measured position is subtracted from the setpoint (in order to calculate the error), is therefore found in most closed loop control algorithms.

The mechanical system, between the actuator and the sensor, involves a chain of events with an inherent response time. For a cruise control system, the response time is the time it takes for an increase in fuel injected to the engine to cause an increase in speed of the vehicle – in some cases this might be of the order of seconds, or at least hundreds of milliseconds. The slow response time can cause problems for an automatic control system, because once the desired setpoint is achieved, it is likely to overshoot. With cruise control, once the desired speed has been achieved it will take a small amount of time for the engine to reduce its torque output (as fuel will already have been injected into the engine), so the controller could overshoot and cause an error in the opposite direction. The controller will automatically then attempt to reverse the action and reduce the error back towards zero again, but may overshoot in the opposite direction too. In a poorly designed cruise control system the controller may never even achieve a satisfactory and steady 80 kph as desired by the driver. Indeed, the system may oscillate continuously between, say, 78 and 82 kph, which would be an unpleasant driving experience.

It follows that the performance of the control system depends on the effectiveness of the control algorithm employed. Design and tuning of the control algorithm is a delicate process, requiring knowledge of the system characteristics and usually experimental testing to understand response times. When final calibration is performed effectively, however, very stable and responsive control systems can be developed.

13.1.3 Proportional Control

The simplest form of closed loop control uses a proportional algorithm to form a linear relationship between the measured error and the actuator control. Figure 13.2 describes the electromechanical process of a closed loop cruise control system. The driver tells the microcontroller the speed at which they would like the vehicle to cruise. At the same time, the actual vehicle speed is calculated from a wheel speed sensor reading. The speed

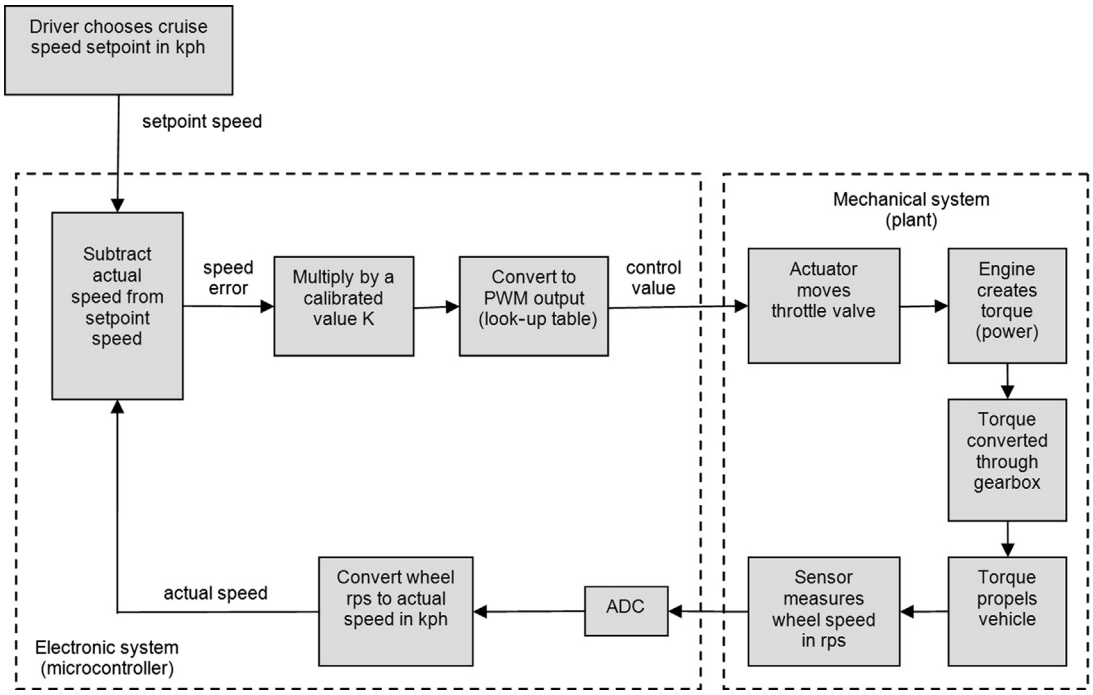

Figure 13.2:
Closed loop cruise control example. PWM: pulse width modulation; ADC: analog-to-digital converter

setpoint is compared to the actual vehicle speed to calculate the error, which is then multiplied by a calibrated value *K*. The calculated value is then used to set a PWM output of the microcontroller, and there may be a conversion look-up table here to ensure that a suitable PWM output is set. The PWM output controls a solenoid, which can move the engine's throttle position. The position of the throttle valve determines how much air and fuel are drawn into the engine, and hence the amount of torque or power generated. The engine torque, via the vehicle gearbox, determines how much power is delivered to the wheels, which subsequently dictates the speed of the vehicle. If a setpoint is chosen then, with a continuously sampling microcontroller, the control loop will automatically self-adjust so that a speed error of zero is achieved. In reality, a zero error is not quite achieved, but the control loop will get as close as possible. If suddenly the setpoint is changed, for example from 80 kph to 90 kph, there is as an immediate *step change* in the setpoint. It will take some time for the vehicle to speed up to the new setpoint, and the time taken for the *step response* is dependent on a number of factors. It will depend predominantly on the value of *K*, the physical response time of the mechanical system and the speed of the microprocessor software loop.

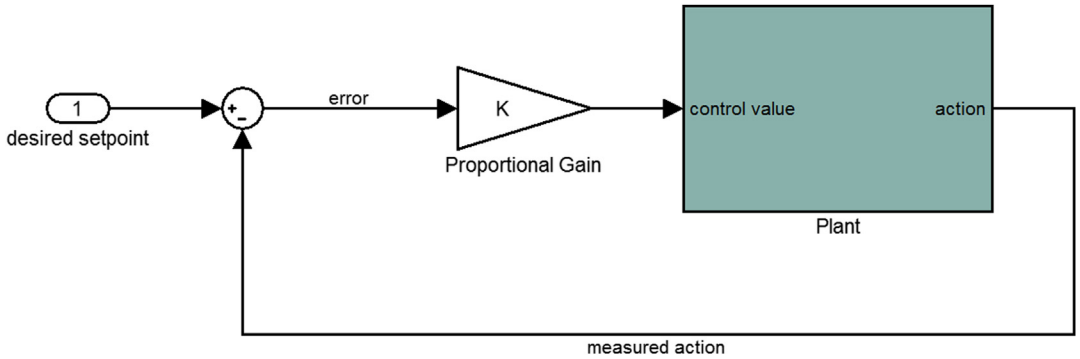

Figure 13.3:
Proportional control system

Figure 13.2 also shows the elements of the electronic control system (i.e. the microcontroller) and the mechanical system, which is sometimes referred to as a *plant*. In many cases the plant itself is modeled mathematically and implemented in another microcontroller, so that engineers can test their control algorithm in simulation before implementing it on a real vehicle.

A more general representation of the closed loop control system is shown in Figure 13.3. We can see that when the error is positive (i.e. the measured value is above the setpoint), the control value is reduced (owing to negative feedback) and vice versa for negative errors. Furthermore, when the error is large, the control value is changed by a large amount; when the error is small, the control value is adjusted by only a small amount. The value of the gain determines how quickly the system responds to errors.

The fact that the control value is altered proportionally is a good thing when the error is large; however, this causes problems when the error is small. As the error gets smaller, the control value adjustment get smaller too, which results in the desired setpoint never actually being achieved. With proportional only control there is always therefore a *steady-state error*. The steady-state error can be reduced by making the gain larger, but unfortunately this could easily result in overshoot and a longer settling period, and even continuous instability if the gain is too high. Figure 13.4 shows an example step response of a proportional control system with high and low gain values.

To overcome this error with controlled overshoot and stability it is necessary to add integral and derivative terms to the control algorithm.

13.1.4 Proportional Integral Derivative Control

Proportional integral derivative (PID) control is similar to proportional control, but with the addition of algorithm components relating to the integral and derivative values of the error

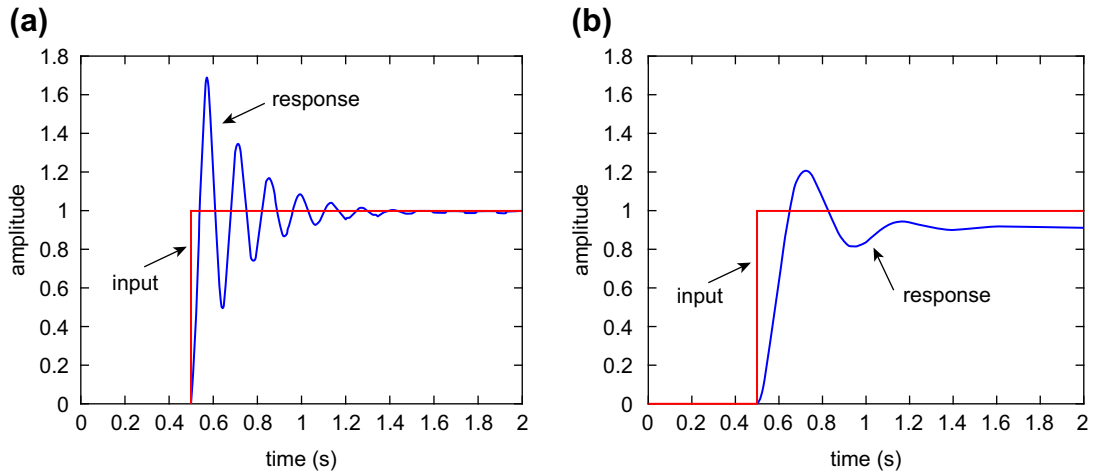

Figure 13.4:
Step response with (a) high proportional gain; (b) low proportional gain

data. This adds an element of history to the algorithm, rather than it being responsive to the current error value alone. A PID control system is shown in Figure 13.5.

Figure 13.6 shows that the error is integrated and multiplied by an integral gain value as well as differentiated and multiplied by a differential gain. These are then summed together with the proportional gain term to give a control value which takes into account not only

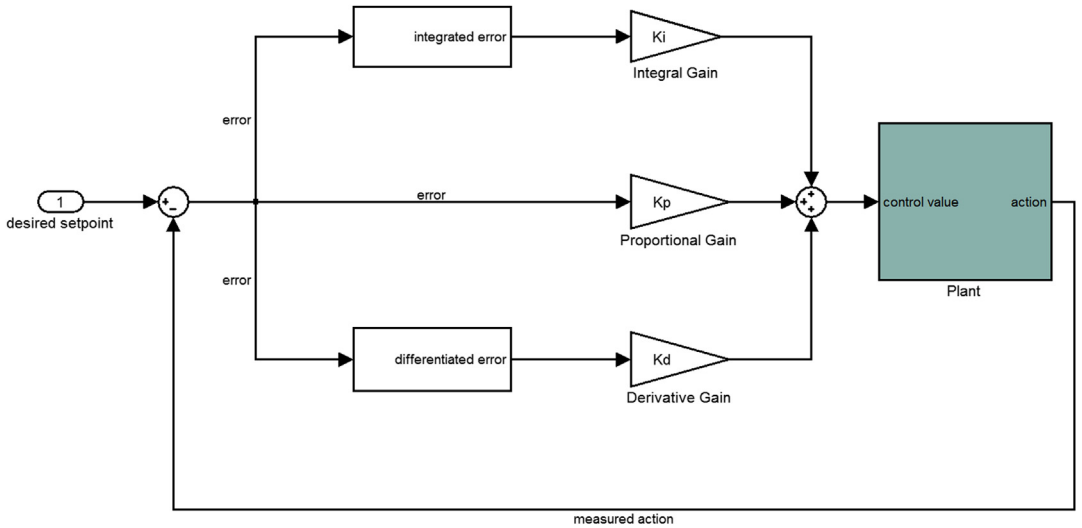

Figure 13.5:
PID control system

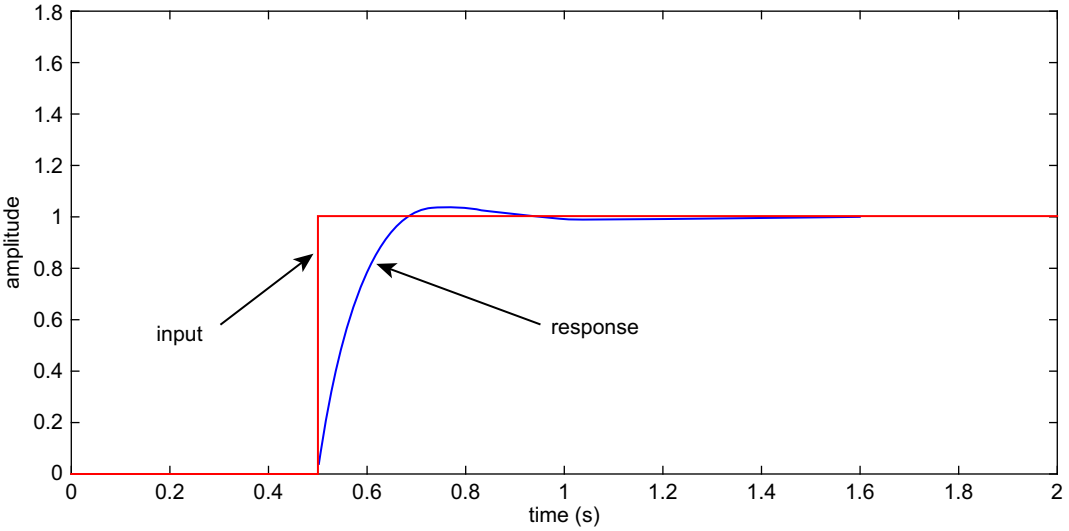

Figure 13.6:
PID controller tuned to give step response with low overshoot and low steady-state error

current position, but also the rate of change of the position and the longer term history of change. Now the tuning of the system becomes even more complex, but the results can be an accurate control system with low steady error and low overshoot. The gain terms can be calculated by experimentation and by employing software tools. Complex mathematical methods can help to determine the best PID gain values (see Reference 13.2, for example), although some fine manual tuning is often required also. Figure 13.6 shows the step response of a PID control system which has its gain values tuned to give a fast response time with low overshoot and low steady-state error.

13.2 Closed Loop Digital Compass Example

A closed loop system requires a sensor and an actuator, whereby the sensor's position or state is altered by the actuator's action. A simple practical example is that of a digital compass integrated circuit chip, positioned on a servo. As the servo rotates, the compass readings will alter accordingly. We can design a closed loop system to automatically adjust the servo position to a desired compass reading and use a simple algorithm to ensure accurate positioning and smooth movement. Implementing a full PID closed loop controller is rather complex and outside the scope of this book. However, it is possible to implement a simple proportional control algorithm for the purpose of introducing closed loop control on the mbed.

In this example, we will use the mbed with an HMC6352 digital compass module with a 360 degree rotation servo. You will also need a 6 V battery supply for the servo. It is important to

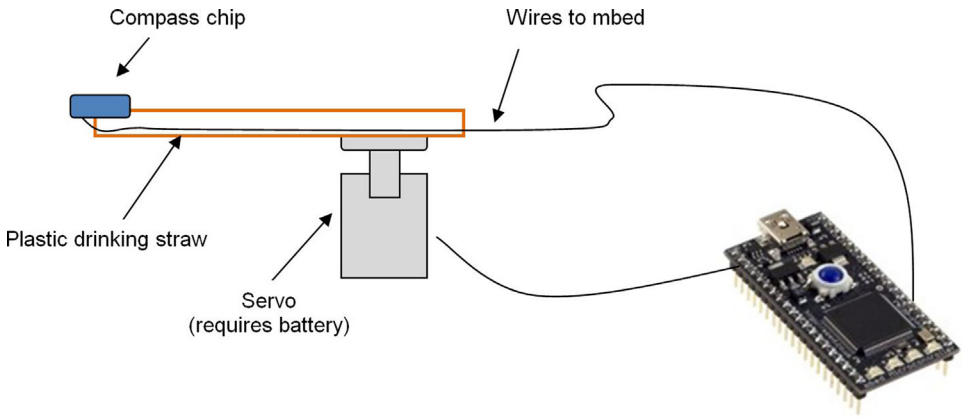

Figure 13.7:
Closed loop compass system

think carefully about connections and wiring to ensure that the wires do not become tangled as the servo rotates. The compass sensor also needs to be kept as far away as possible from any unwanted magnetic fields. An example test setup is shown in Figure 13.7.

13.2.1 Using the HMC6352 Digital Compass

Honeywell's HMC6352 compass is another highly integrated sensor, similar in concept to the ADXL345 accelerometer and TMP102 temperature sensor introduced in Chapter 7. Fully integrated onto a single integrated circuit (IC), it contains sensors, signal conditioning, microprocessor and inter-integrated circuit (I^2C) serial interface. The two sensors, mounted at right angles to sense the components of the surrounding magnetic field, are based on the magnetoresistive principle. This relies on the fact that the apparent resistance of a conductor can be seen to change in the presence of a magnetic field. We use it on a tiny breakout board, as shown in Figure 13.8. Full information is given in Reference 13.3.

Figure 13.8:
HMC6352 mounted on a breakout board. *(Image reproduced with permission of Sparkfun Electronics)*

Table 13.1: Pin connections for HMC6352 digital compass

HMC6352 pin	mbed pin
SCL (I^2C)	27
SDA (I^2C)	28
VCC (3.3 V)	40
GND (0 V)	1

The HMC6352 compass can be connected to the mbed via an I^2C interface, as described by the pin connections in Table 13.1. Before building a complete system, we will first write a program to ensure that the compass initializes and reads data correctly.

As detailed in the HMC6352 datasheet (Reference 13.3), the base address for the compass is 0x42. The datasheet also describes that in order to initialize the IC we write an ASCII 'G' (0x47) command to inform the device that we wish to write to a particular RAM address. This is followed by the operation mode register address (0x74), followed by the operation mode control byte. We will initialize the compass to update in continuous mode at a rate of 20 Hz, and we also want to enable the Set/Reset function. This gives a data byte value of 0x72. The compass setup can therefore be executed by an I^2C write message containing the three command bytes as a single array, i.e:

```
cmd[0] = 0x47;        // 'G' write to RAM address
cmd[1] = 0x74;        // Operation mode register address
cmd[2] = 0x72;        // Op mode = 20H, no S/R, continuous
```

A test program to set up and read the HMC6352 is shown in Program Example 13.1.

```
/* Program Example 13.1: HMC6352 Compass Setup and Read
*/
#include "mbed.h"
// mbed objects
I2C compass(p28, p27);      // sda, scl
Serial pc(USBTX, USBRX);    // tx, rx
// variables
const int addr = 0x42;      // define the I2C write Address
char cmd[3];                // command array for read and write
float pos;                  // measured position

// main code
int main() {
  cmd[0] = 0x47;        // 'G' write to RAM address
  cmd[1] = 0x74;        // Operation mode register address
  cmd[2] = 0x72;        // Op mode = 20H, no S/R, continuous
  compass.write(addr,cmd, 3);  // Send operation
  while (1) {
    compass.read(addr, cmd, 2);  // read the two-byte echo result
```

```
    pos = 0.1 * ((cmd[0] << 8) + cmd[1]);  //convert to degrees
    if (pos>180){
       pos=pos-360;
    }
    pc.printf("deg = %.1f\n", pos);
    wait(0.3);
  }
}
```

Program Example 13.1 HMC6352 compass setup and read

To read data from the compass we use an infinite loop and output data values to a host PC terminal application. Inside the infinite loop we simply perform a 2-byte read which returns the 16-bit data value (in tenths of a degree). To form a single number from these 2 bytes we shift the first byte left by 8 (as it is the more significant byte) and add the second byte; this forms a 16-bit number. In the same line of code this is then multiplied by 0.1, to give a result in degrees. It is actually more useful to have degrees in the range -180 to $+180$ (rather than 0 to 360), so values greater than 180 have the 360 subtracted.

With the compass IC connected to mbed pins 27 and 28, Program 13.1 should initialize and run, allowing you to verify that correct compass readings are shown on the PC terminal. Zero degrees indicates north and 180 degrees (or -180 degrees) indicates south, and hence 90 degrees represents due east and -90 degrees represents due west. Be careful to ensure that large magnets or power sources do not interfere with the compass's sensitivity. If you are in a steel-framed building or a well-equipped laboratory, you may find that the earth's field is totally overwhelmed by what is around you!

13.2.2 Implementing a 360 Degree Rotation Servo

We can implement an open loop position controller with a 360 degree servo and a potentiometer to control. Servos with 360 degree motion can be sourced (as detailed in Appendix D), or a standard 180 degree motion servo can be modified to allow complete rotation, as described by Reference 13.4. The 360 degree servo requires the same mbed connections as the standard 180 degree servo (as shown in Figure 4.10). However, the 360 degree version is a more erratic device and more challenging to control. When connecting a PWM signal to the device a low duty cycle signal will cause the servo to continuously turn anti-clockwise, whereas a higher duty cycle will cause the servo to rotate continuously clockwise. There will be a single PWM duty value which is a 'zero' point at which the servo holds stationary.

We will now implement a scheduled program to allow different rate functions. In particular, we will use a fast rate function to control the servo positioning and to read the sensor data (100 Hz), but a slower rate function to send data to the host terminal application (5 Hz). We will therefore use two Ticker objects and two task functions to control and implement the program timing. In this program we will use a potentiometer to

allow open loop control of the servo; we will also read and display compass data to the screen to quantify the direction in which the compass is pointing. We therefore need to define an analog input (to read the potentiometer) and a PWM output (to actuate the servo). We will also need to implement the same HMC6352 initialization and read commands as used in Program Example 13.1.

Start a new project and enter the code shown in Program Example 13.2.

```
/* Program Example: 13.2 Open Loop Compass
*/
#include "mbed.h"
//***** mbed objects *****
I2C compass(p28, p27);      // sda, scl
PwmOut PWM(p25);
AnalogIn Ain(p20);
Serial pc(USBTX, USBRX);    // tx, rx
Ticker s100hz_tick;         // 100 Hz (10ms) ticker
Ticker s5hz_tick;           // 5 Hz (200ms) ticker
//***** variables *****
const int addr = 0x42;      // define the I2C write Address
char cmd[3];
float pos;                  // measured position
float ctrlval;              // PWM control value
//***** function prototypes *****
void s100hz_task(void);     // 100 Hz task
void s5hz_task(void);       // 5 Hz task
//***** main code *****
int main() {
 // initialize and setup data
  PWM.period(0.02);
  cmd[0] = 0x47;             // 'G' write to RAM address
  cmd[1] = 0x74;             // Operation mode register address
  cmd[2] = 0x72;             // Op mode = 20H, S/R, continuous
  compass.write(addr,cmd, 3);// Send operation
  s100hz_tick.attach(&s100hz_task,0.01);   // attach 100 Hz task
  s5hz_tick.attach(&s5hz_task,0.2);        // attach 5 Hz task
  while(1){
  // loop forever
    }
}

//***** function 100hz_task *****
void s100hz_task(void) {
  compass.read(addr, cmd, 2);   // read the two-byte compass data
  pos = 0.1 * ((cmd[0] << 8) + cmd[1]);    //convert to degrees
  if (pos>180)
  pos=pos-360;              convert to ±180deg
  ctrlval=Ain;     // set control value (also try ctrlval=Ain/4)
  PWM=ctrlval;     // output control value to PWM
}
```

```
//***** function 5hz_task *****
void s5hz_task(void) {
   pc.printf("deg = %.1f  PWM = %.4f\n", pos, ctrlval);
}
```

Program Example 13.2 Open loop compass

If a potentiometer input is now connected to pin 20 and the PWM output on pin 25 is connected to a powered servo, your compiled code should allow the potentiometer to move the servo clockwise and anti-clockwise. Position data will also be sent to a host PC running Tera Term.

You will notice that the motion of the servo is quite sensitive and erratic; it is actually very difficult to achieve an exact stationary position. Indeed, the analog input gives values between 0.0 and 1.0, but we know from our experience of servos in Chapter 4 that the servo only uses the PWM duty cycle range of about 5 to 20%. It is therefore possible to decrease the open loop sensitivity by implementing either a scaling factor or some hard limits to the possible PWM control values.

Exercise 13.1

1. Modify Program Example 13.2 so that the control value **ctrlval** is only a fraction of **Ain** by dividing by a fixed number. Try different values and see whether the control and stability can be improved.
2. Looking at the Tera Term output, find the PWM value (**ctrlval**) that causes the servo to hold stationary. This is a constant for the servo you are using. Make a note of this 'zero' value, as we will need it later on, at which time we will call it **PWM_zero.**

■

13.2.3 Implementing a Closed Loop Control Algorithm

We can use a simple proportional control algorithm to implement a closed loop system. To demonstrate this we will set our desired compass position to be 0 degrees, i.e. the compass will always point north (much like a real compass). The servo will move the position of the compass automatically to ensure that it does this. First, we will need to define a few variables for calculating error, the desired setpoint (= 0) and associated PWM control value, as follows:

```
float setpos=0; // desired setpoint position = zero (North)
float error;    // calculated error
float ctrlval;  // PWM control value
float kp=0.0002;        // proportional gain (to be tuned)
float PWM_zero=0.075;   // zero value (as deduced in Exercise 13.1)
```

Program Example 13.3 implements the closed loop algorithm, which retains much of the code from Program Example 13.2.

```
/* Program Example 13.3: Closed loop compass program
*/
#include "mbed.h"

// mbed objects
  I2C compass(p28, p27);    // sda, scl
  PwmOut PWM(p25);
  AnalogIn Ain(p20);
  Serial pc(USBTX, USBRX);     // tx, rx
  Ticker s100hz_tick;          // 100 Hz (10ms) ticker
  Ticker s5hz_tick;            // 5 Hz (200ms) ticker

// variables
  const int addr = 0x42; // define the I2C write Address
  char cmd[3];
  float pos;             // measured position
  float setpos=0;        // setpoint position = zero (North)
  float error;           // calculated error
  float ctrlval;         // PWM control value
  float kp=0.0002;         // proportional gain
  float PWM_zero=0.075;    // zero value

// function prototypes
  void s100hz_task(void);  // 100 Hz task
  void s5hz_task(void);    // 5 Hz task

// main code
  int main() {
 // initialize and setup data
   PWM.period(0.02);
   cmd[0] = 0x47;               // 'G' write to RAM address
   cmd[1] = 0x74;               // Operation mode register address
   cmd[2] = 0x72;               // Op mode = 20H, S/R, continuous
   compass.write(addr,cmd, 3);  // Send operation
 // assign timers
   s100hz_tick.attach(&s100hz_task,0.01); //attach 100 Hz task to 10ms tick
   s5hz_tick.attach(&s5hz_task,0.2);      //attach 5Hz task to 200ms tick
   while(1){
 // loop forever
   }
}

// function 100hz_task
  void s100hz_task(void) {
   compass.read(addr, cmd, 2); // read the two-byte echo result
 //convert data to degrees
   pos = 0.1 * ((cmd[0] << 8) + cmd[1]);
   if (pos>180)
    pos=pos-360;
   error = setpos - pos;          // get error
   ctrlval = (kp * error);        // calculate ctrlval (proportional)
   ctrlval = ctrlval + PWM_zero;  // add control value to zero position
   PWM = ctrlval;                 // output to PWM
}
```

```
// function 5hz_task
  void s5hz_task(void) {
   pc.printf("deg = %.1f error=%.1f ctrlval=%.4f\n",pos,error,ctrlval);
  }
```

Program Example 13.3 Closed loop compass program

The 100 Hz task includes the closed loop control algorithm. Immediately after the **pos** variable has been calculated, the error between the actual and desired setpoint positions, i.e. the proportional control value **ctrlval**, is calculated as follows:

```
error = setpos - pos;       // get error
ctrlval = (kp * error);     // calculate ctrlval (proportional)
```

The control value is a calculated position that automatically adjusts to make up for the difference between the desired and actual in a smooth and controlled manner. However, we first need to add the zero position to the control value in the following line:

```
ctrlval = ctrlval + PWM_zero;  // add control value to zero position
```

The zero position, **PWM_zero**, is a simple calibratable value which we deduced in the second part of Exercise 13.1. It defines the PWM duty cycle that holds the servo in a stationary position. It now acts as an offset to the control value, so a negative error causes the servo to rotate to one side of the zero position, and a positive value causes it to rotate to the other side. When there is zero error in the control algorithm the servo stays still. In this example the zero offset value is calibrated as **PWM_zero = 0.075**, although your own servo may require a different value.

Exercise 13.2

1. Modify the value of **Kp** and see how this affects the system stability. In general, if **Kp** is too large then overshoot and instability are observed, whereas if **Kp** is too small, then the servo response is slow and inaccurate.
2. Investigate the required accuracy of the **PWM_zero** value. Are three decimal places sufficient or does refining this value to four or five decimal places enhance the controller's performance?

■

Exercise 13.3

1. **LCD output** – Add a liquid crystal display (LCD) to your system to display position, error and PWM data. You can also use the 6 V servo battery pack as a portable supply voltage for the mbed. This will allow you to disconnect the mbed from the host PC and have a fully mobile digital compass.
2. **Position control** – Develop your system to allow a user to input a desired position (in degrees) in a terminal application, and the servo will immediately move to that position.

■

13.3 Communicating Control Data over the Controller Area Network

13.3.1 The Controller Area Network

The Controller Area Network (CAN) is another type of serial communications protocol that was developed within the automotive industry to allow a number of electronic units on a single vehicle to share essential control data. A vehicle nowadays uses many microcontrollers for autonomous control systems. Each microcontroller system is referred to as an electronic control unit (ECU) and these include the engine management ECU, an anti-braking system (ABS) ECU, a dashboard ECU, active suspension and the radio/CD player, for example. Each of these ECUs manages its own control strategies, but they all need to access information relevant to their own operation, for example drawn from engine speed, throttle position, brake pedal position and engine temperature. However, an automotive vehicle generates a high level of electromagnetic interference and a wide temperature and humidity range; this is a hostile environment for any signal and indeed for any electronic device. Moreover, very high reliability is essential as vehicles are safety-critical systems. The serial standards developed for the benign environment of home or office, such as UART, SPI and I^2C, are completely inappropriate for this hostile environment, and a new standard was therefore needed. Initially, CAN was developed by the German company Bosch. They published Version 2.0 of the standard in 1991, and in 1993 it was adopted by the International Organization for Standardization (ISO), as ISO 11898. At the time of writing, CAN specification Version 2.0B can be downloaded from the Bosch website (Reference 13.5). The CAN standard has a high level of data security, it is inevitably complex and just the briefest overview is given here. The main features are listed below.

- Communication is asynchronous, half duplex, with (for a given system) a fixed bit rate. The maximum for this is 1 Mbit/s.
- The configuration is ‘peer to peer’, i.e. all nodes are viewed as equals. There is, however, a mechanism for prioritization. Master and slave designation is not used.
- Logic values on the bus are defined as ‘dominant’ or ‘recessive’, where dominant overrides recessive. Physical interconnect is not otherwise defined.
- The bus access is flexible. With all nodes being peers, any can start a message. An ingenious arbitration process is applied in the case of simultaneous access, which does not lead to loss of time or data. The arbitration process recognizes prioritization.
- There is an unlimited number of nodes.
- Bus nodes do not have addresses, but apply ‘message filtering’ to determine whether data on the bus is relevant to them.
- Data is transferred in frames, which have a complex format. This starts with identifier bits, during which arbitration can take place. Eight data bytes are allowed per frame.
- There is an exceptionally high level of data security, with exhaustive error checking. A node that recognizes that it is faulty can disconnect itself from the bus.

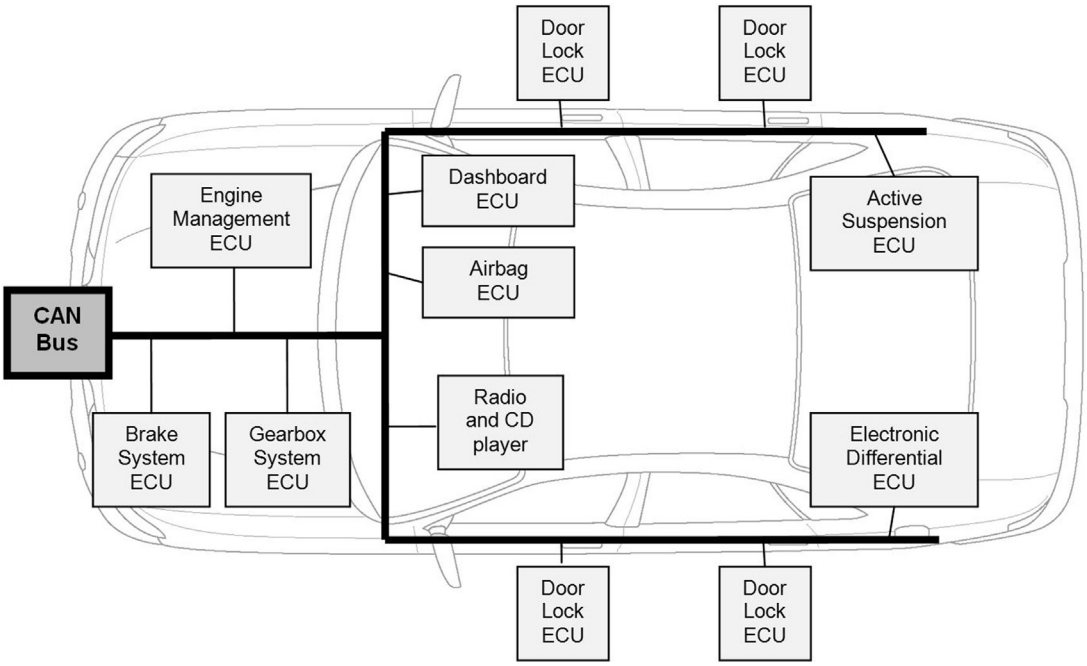

Figure 13.9:
An example vehicle CAN bus network

Figure 13.9 shows an example vehicle CAN bus network. The types of control data that need sharing along the bus can vary. For example, the dashboard ECU will need to access data including engine speed, engine temperature, vehicle speed and diagnostic information, which will be predominantly provided by sensors attached to the engine management and gearbox ECUs, as well as door open/closed data from the door ECUs. Furthermore, if this vehicle is to implement an automatic door lock strategy once a threshold vehicle speed has been achieved, then the door lock ECUs will need to access vehicle speed data from the CAN bus.

So it can be seen that a very reliable and thorough communications protocol for sharing control data around an automotive vehicle is required, and CAN provides this. CAN is also used in industrial and factory applications where noise issues and reliability requirements are similar to those of the automotive industry.

13.3.2 CAN on the mbed

The mbed has two CAN controller interfaces on pins 9−10 and 30−29; note that only the latter two pins appear as a CAN interface in Figure 2.1. The pins refer to receive (RD or RXD) and transmit (TD or TXD) signals, respectively. The CAN interface can be used to write data words out of a CAN port and will return the data received from another CAN device. The

Table 13.2: Selected CAN API functions

Function	Usage
`CAN CANMessage`	Creates a CAN interface connected to specific pins
	Creates an empty CAN message (for reading) or a CAN message with specific content (for writing):
	`id` the message ID
	`data` space for 8-byte payload
	`len` length of data in bytes
	`format` defines if the message has standard or extended format
	`type` defines the type of a message
`frequency write read`	Set the frequency of the CAN interface
	Write a CANMessage to the bus. Returns 0 if write failed, 1 if write was successful
	Read a CANMessage from the bus. Returns 0 if no message arrived, 1 if message arrived

CAN clock frequency can also be configured. A selection of the CAN application programming interface (API) functions is shown in Table 13.2.

When using the mbed CAN API, we first have to define a **CAN** object and then define a message (**CANMessage**) to be written to or read from the CAN bus. The mbed CAN controllers alone, however, cannot be used to communicate directly with a CAN network. To do this we need to use the mbed CAN interface to control a specific *CAN transceiver* IC such as the Microchip MCP2551. This is shown in Figure 13.10 and described in Reference 13.6.

The CAN bus implementation used is differential, and connects to the **CANH** and **CANL** pins. An external resistor connected at **Rs** controls the slew rate of the data signal, with a slower rate minimizing electromagnetic interference, although for simple testing this can be connected directly to ground. The mbed connects from its CAN controllers to the transceiver's **RXD** and **TXD** pins.

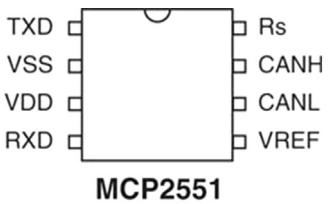

Figure 13.10:
Microchip MCP2551 CAN transceiver IC

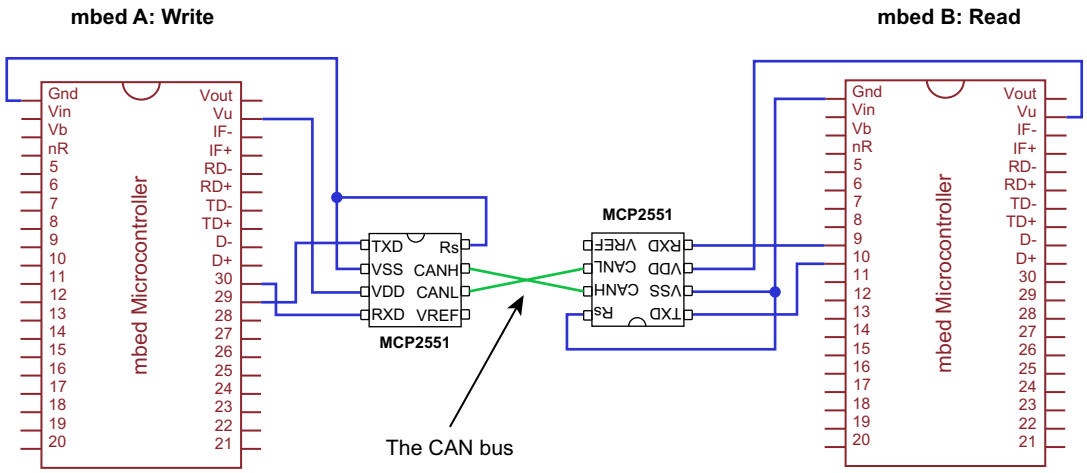

Figure 13.11:
Hardware build for mbed-to-mbed CAN communication

To demonstrate CAN with the mbed we will communicate some data between two mbeds over a CAN network. One mbed will send data to the CAN bus while the other will receive data from the CAN bus. The hardware connections required to create a simple CAN network test are shown in Figure 13.11 and Table 13.3. Notice that the 'CAN bus' is defined as the wires CANH and CANL, which sometimes require a termination resistance of 100–200 Ω between them over long wires. In this simple, noise-free example the short connections mean that a termination resistor is not necessary. Note also that we use the mbed CAN controller on pins 30 and 29 for the send system and the CAN controller uses pins 9 and 10 – these are arbitrarily chosen, predominantly to make the hardware connection diagram neat!

Having built the CAN test system described in Figure 13.11 and Table 13.3, Program Example 13.4 can be implemented to regularly send data values to the CAN bus.

```
/* Program Example 13.4: CAN data write – sends an incrementing count value to the CAN
bus every second
*/
#include "mbed.h"
Serial pc(USBTX, USBRX);      // tx, rx for Tera Term output

DigitalOut led1(LED1);        // status LED
CAN can1(p30, p29);           // CAN interface
char counter = 0;
int main() {
  printf("send... ");
  while (1) {
    // send value to CAN bus and monitor return value to check if CAN
    // message was sent successfully. If so display, increment and toggle
```

```
    if (can1.write(CANMessage(1, &counter, 1))) {
       pc.printf("Message sent: %d\n", counter);    // display
    counter++;                                     // increment
       led1 = !led1;                               // toggle status LED
    }else{
    can1.reset();
     }
     wait(1);
  }
}
```

Program Example 13.4 CAN data write

Here, one very important line of code is the **if** statement, which performs a number of actions:

```
if (can1.write(CANMessage(1, &counter, 1))) {
```

First, this line creates a **CANMessage** with **id = 1**, holding the data from the address of variable **counter** and of length 1 byte. The message is then written to the CAN bus and the

Table 13.3: Pin connections for mbed-to-mbed CAN communications

MCP2551 pin	Description	Pin connection
	Write system	
1 TXD	Transmit Data Input	mbed p29 CAN TD
2 VSS	Ground	mbed p1 GND
3 VDD	Supply Voltage	mbed p39 Vu (5 V)
4 RXD	Receive Data Output	mbed p30 CAN RD
5 Rs	Reference Output Voltage	mbed p1 GND
6 CANL	CAN Low-Level Voltage I/O	CAN bus low
7 CANH	CAN High-Level Voltage I/O	CAN bus high
8 VREF	Slope-Control Input	—
	Read system	
1 TXD	Transmit Data Input	mbed p10 CAN TD
2 VSS	Ground	mbed p1 GND
3 VDD	Supply Voltage	mbed p39 Vu (5 V)
4 RXD	Receive Data Output	mbed p9 CAN RD
5 Rs	Reference Output Voltage	mbed p1 GND
6 CANL	CAN Low-Level Voltage I/O	CAN bus low
7 CANH	CAN High-Level Voltage I/O	CAN bus high
8 VREF	Slope-Control Input	—

return value of the **write** operation is monitored. Only if the **write** operation returns a successful response (by returning a value 1) does the counter increment and light-emitting diode (LED) toggle.

Program Example 13.5 creates an empty **CANMessage**. It continuously monitors the CAN bus and displays any read CAN messages to Tera Term on a host PC.

```
/* Program Example 13.5: CAN data read – reads CAN messages from the CAN bus
*/
#include "mbed.h"
Serial pc(USBTX, USBRX);     // tx, rx for Tera Term output
DigitalOut led2(LED2);       // status LED
CAN can1(p9, p10);           // CAN interface
int main() {
  CANMessage msg;             // create empty CAN message
  printf("read...\n");
  while(1) {
    if(can1.read(msg)) {     // if message is available, read into msg
      printf("Message received: %d\n", msg.data[0]);  // display message data
      led2 = !led2;                                   // toggle status LED
    }
  }
}
```

Program Example 13.5 CAN data read

You can now implement the hardware connections shown and Program Examples 13.4 and 13.5 to show two mbeds communicating over CAN.

Exercise 13.4

1. Enhance the CAN write program to send a second data value to the CAN bus. Give the second data value a different ID value and make it two bytes in size and at a different message rate (say every 5 seconds). Evaluate how the CAN reader program interprets the CAN messages it receives.
2. Implement a procedure in the CAN reader program so that it can interpret the two different CAN messages based on the message ID, and uniquely display a text message in Tera Term based on the ID of the message received.

■

For a practical implementation of a CAN messaging system, such as for an automotive system, a *CAN specification* will define the messages that can be communicated between ECUs. Each message, for example vehicle speed, engine speed, engine temperature and door open/closed status, will have a unique ID as well as a definition of the size of the data and any conversion equation required in order to turn the raw data into a sensible value. Each ECU will be pre-programmed with the CAN specification so that when a message is received it can immediately know what the message is and how to interpret the data.

Several CAN interface devices have been developed and are available commercially for example to read diagnostic information from vehicles. It is also possible to use an mbed to read and write data messages to and from a vehicle CAN bus with the correct interface, and hence even control a vehicle this way. Of course, to do this you will need to know the CAN specification for the vehicle that is being interfaced with. A number of interesting examples and an mbed CAN interface board can be found in Reference 13.7.

Chapter Review

- Closed loop control systems use negative feedback of sensor data to ensure that a specific actuator position is accurate.
- When responding to a step change in desired setpoint, a closed loop control system may respond too slowly or overshoot, or have a steady-state error. Tuning the control algorithm can ensure a best-possible step response.
- The time taken for the step response is dependent on the control algorithm design, the physical response time of the mechanical system and the speed of the microprocessor software loop.
- Proportional, integral and derivative control can be tuned to give smooth control of many electromechanical systems.
- Control data can be shared on a network of electronic control units (ECUs) over the Controller Area Network (CAN) protocol.
- An automotive vehicle may have a number of ECUs for controlling, for example, engine management systems, brake systems, the dashboard, airbag control and door locks. These can all be connected by CAN.
- CAN is an extremely reliable communications protocol with high tolerance to noise and interference.
- To implement CAN on the mbed, additional CAN transceiver ICs must be used to create the CAN bus.

Quiz

1. What are the main differences between an open loop and a closed loop control system?
2. What do the terms setpoint, plant and negative feedback refer to in closed loop control?
3. Where and why might closed loop control be required in the design of a helicopter?
4. What does ‘the step response’ of a control system refer to?
5. What benefits does CAN communication have over other serial protocols such as I^2C and USB?
6. Give five example ECUs that may be found on a modern car.

References

13.1. The Segway website. http://www.segway.com
13.2. Copeland, B. R. (2008). The Design of PID Controllers using Ziegler−Nichols Tuning. http://www.eng.uwi.tt/depts/elec/staff/copeland/ee27B/Ziegler_Nichols.pdf
13.3. 2-Axis Compass with Algorithms. HMC6352. Document #900307. Rev. D. January 2006. http://www.magneticsensors.com/
13.4 Servo modification for 360 degree rotation. http://www.embeddedtronics.com/servo.html
13.5. The CAN section of the Bosch website. www.can.bosch.com
13.6. Microchip Technology (2003). MCP2551 High-Speed CAN Transceiver Data Sheet. Document DS21730E.
13.7. mbed CAN-Bus Demo Board. http://mbed.org/forum/news-announce

This page intentionally left blank