3

Gaining Experience

These initial remarks are directed to the lecturer or examiner of a course on mechatronics. The experiment instructions that follow can be used as course notes for the students.

Practical laboratory experience is an essential ingredient for linking together the diverse aspects of mechatronics. But it is necessary to choose among a wealth of alternatives when selecting and designing experiments.

Numerous experiments are available on the market, but they are usually very costly. Much worse, many of them require very little creativity or understanding on the part of the student, consisting instead of a ritual of knob turning, measurement, and graph plotting.

In the experiments that follow, the students are required to create software from scratch, not just by dragging icons in a Graphic User Interface. They add hardware by connecting circuits that they could easily replicate from component level—indeed, an able technician should have no difficulty in fabricating these experiments from catalog components.

There is, however, the problem of the choice of computing platform.

When mechatronics ability comes to be applied by the graduated student, will it be to use a PC for control and coordination tasks, or will the objective be a mechatronic product with an embedded microcomputer? Should the software lean toward the latest version of the .NET environment, or should it be based on downloading the simplest code to a single chip?

An embeddable chip such as the HC12 certainly has all the input–output capability and the computing power to control any of the experiments here.

It has numerous ADCs and output pins, with an abundance of program memory. But its "user interface" is limited, to say the least. It will need to be linked via its RS232 serial capability to a PC (or its equivalent) on which the student can write, edit, save and then compile or assemble the software. The link will also be needed to upload data for display or plotting.

Instead, a simple PC can in principle perform all the tasks of development interface, performance monitor, and real-time controller, with no additional circuitry beyond an output driver to power any motors and a simple analog-to-digital converter for reading sensor signals. The fly in the ointment is that simple PCs and ADCs are becoming increasingly hard to find!

If you do not have a suitable ADC card, you will find the software in Chapter 5 for using an MCP3204 four-channel 12-bit ADC chip, costing only a few dollars. Apart from a connector for the printer port and a length of ribbon cable, no other components are needed for the interface if the signal voltages are of low impedance. The chip can even be powered from one of the output pins. One more chip such as the TL074 can provide four buffer amplifiers for higher-impedance signals.

Each elaboration of the Windows operating system seems designed to put more distance between the user and the actual operation of the hardware. Vendors have dropped simple ADC cards from their lists in favor of FIFO-buffered elaborations that require a monstrous software driver library to use. However there are "workarounds" to enable you to use even these, although life is much easier if you have a board of traditional design. Also, many problems can be avoided if you have retained a copy of your old Windows 98.

An alternative is to use the HC12 or a similar microprocessor as a "slave" interface. With a simple protocol, it can be asked to reply with the values on any of the ADC inputs, to latch output values and apply them to output bits or mark–space registers. Using the serial interface of the PC, rather than a proprietary plugged-in card, it should be proof for some years from the efforts of the operating system writers. The effective conversion time will be greatly increased by the serial communication, but when set to a high baud rate, this should still be acceptable.

This might at first threaten to present the same dangers as the Labview approach, in which the interfacing is regarded as a piece of magic into which the student is not supposed to delve. But the student can certainly unravel the simple code of the HC12 to see how it ticks. What is more, a second level of the experiments can see the students writing HC12 code to achieve the control objectives without the intervention of the PC, once the code has been downloaded.

A further resource, to be found on the accompanying Website at http://www.essmech.com/3/vb.htm, is an example of the use of Visual Basic rather than QBasic for constructing code for these experiments. The conversion is a very simple one.

The first experiment, however, requires no input at all. It uses the printer port to drive two stepper motors that will form the essence of a "mobile robot." There is little theory involved, but it offers the great satisfaction of seeing a computer moving a "robot" about with homegrown software. But first it is necessary to settle on the software environment to use. The following discussion is aimed at both instructors and students.

3.1 COMING TO GRIPS WITH QBasic

The choice of computer language is always a thorny issue. Partisan support can be as ardent as that of any football supporter, and any choice is going to upset somebody. It could be tempting to select the "newest and the best," perhaps a version of C with the greatest number of pluses after it, but for getting started the greatest simplicity will give the greatest advantage.

An early mainstream language was FORTRAN, but it was hidebound with conventions designed to fit in with the use of punchcards! Some time later it was followed by Algol, a language that treated line endings with disdain and abounded in semicolons.

When "personal" microcomputers began to be sold, the advantage of simplicity had market value. Two varieties of Basic, BasicA, and GWBasic had simple syntax and were well within the capabilities of a generation of schoolchildren. However, they had their roots in "line at a time" program entry and editing and depended heavily on line numbers, even more so than FORTRAN.

Meanwhile Algol had evolved into Pascal, taking its semicolons with it. Algol and Pascal both use labels, rather than numbers, to tag special points in their code.

Soon on-screen editing was the only way to go, and line numbers could be dropped. Quick Basic borrowed some of the best features of Pascal and combined them with the simplicity of Basic. Version 4.5 included a compiler that could efficiently reduce your code to an .EXE file and this was soon followed by Visual Basic for DOS.

Quick Basic caught the attention of Microsoft and a stripped-down version, QBasic, was included in DOS operating system disks and in installation disks for all versions of Windows before XP. Meanwhile, the same syntax was used for Visual Basic, both as part of Visual Studio and as a scripting language in most, if not all, Microsoft Office applications.

It is probable that any serious real-time programming will be performed in some version of C. It is second cousin to assembly language, the lowest level at which it is convenient to program a microcomputer, and as such it has access to processes at a fundamental level. However, C has quirks of cryptic syntax that make reading the software of even the most careful programmer an arduous exercise in concentration.

QBasic code can be read like a novel, as I hope that you will soon agree. Just as in C, there are ways to perform fundamental operations such as writing and reading bytes directly to or from peripheral interfaces.

3.1.1 A Simple Start

Launch QBasic, and the introductory page will appear. You can take a quick tour of the Help files or press ⟨escape⟩ to go straight in. Type in the program line

```
play "cdeccdecefgn0efg"
```

As you press ⟨return⟩ at the end, you will see the word PLAY change into uppercase. It means that the syntax has been checked and the keyword recognized.

Now run the program—yes, it really is a program. There are two ways. You can click on RUN in the menu bar and then START in the submenu that drops down, or else you can press ⟨shift–F5⟩.

What did it do? You should have heard a simple tune.

If you want to know more about the PLAY routine, put the cursor on the word and press ⟨F1⟩—there's everything you could ever ask for. Press ⟨escape⟩ to clear the Help page.

So, what has this got to do with control?

Here we have a very simple way to measure out time, because the music plays the notes at a speed that does not depend on the speed of the processor. We can easily control the speed at which it plays—put another line at the start of the program, to get

```
PLAY "L8"
PLAY "cdeccdecefgn0efg"
```

Run it again. What is the difference? Now change `L8` to `L16`, run again. Try `L64`. Then try `"L64T255"`.

So, a PLAY statement can measure out an interval of time that is independent of the speed of the computer. In order to perform real-time digital filtering, we have to have good control of timing.

3.1.2 Using Graphics

Clear your program and start again by selecting NEW. (To find it, click on FILE.) Do not save.

The line

```
SCREEN 12
```

will set a graphics mode. A second line

```
WINDOW (0, -1.1) - (1000, 1.1)
```

sets the screen coordinates to a range of 0–1000 across and –1.1 to +1.1 from bottom to top. Then

```
LINE (0, 0) - (1000, 0), 9
```

It will draw a blue line across the screen. The 9 specifies "bright blue." Put the cursor on the word LINE and press ⟨F1⟩ to see all the details.

Put in these three lines and run the program. Well, it's a start! Now let's draw something. Enter the following lines:

```
FOR i = 1 TO 1000
  PSET (i, SIN(I / 100)
NEXT
```

The PSET (point set) routine puts a dot at the coordinates in the brackets.

Run the program, and a sine wave will flash onto the screen. So how do we slow it down? Put the line

```
PLAY "L64T255"
```

at the start of the program and

```
   PLAY "n0"
```

just before the NEXT line. (n0 is the code for silence!)

Now you see the sine wave crawl across the screen, taking 12 seconds to run.

3.1.3 A Real-Time Model

Let us now try to model a lowpass system—an example of such a system would be a resistor–capacitor *lag*. It is really not difficult, although you will find much more information on the theory later in the book.

First we need an input signal. Let us make a square wave. A neat way to do it is with a logic operation.

In numbers ranging from 128 to 255, from 384 to 511, and two more ranges below 1000, the "128-bit" of the binary value is set. So the logic function (i AND 128) will cycle from 0 to 128 four times.

```
IF (i AND 128)>0 THEN
  u = 1
ELSE
  u = -1
END IF
```

Put these lines just before the PSET line and change the PSET to

```
PSET(i, u)
```

This time you should see four cycles of square-wave—lines of dots at the top and the bottom of the screen.

So, how do we make the lowpass filter? Try adding the lines

```
x = x + (u - x) / 20
PSET (i, x), 14
```

after the `END IF` line. There in yellow is the waveform that you would expect from an *RC* circuit driven by a square wave.

So, how does it work? The differential equation corresponding to a lowpass filter with time constant *T* and input *u* is

$$T\,dx/dt = u - x$$

So to a first (and pretty good) approximation, the change in *x* over a small time *dt* is

$$(u - x)\,dt/T$$

or in code terms

```
x = x + (u - x) * dt / T
```

Instead of *dt/T*, we have used the numeric value 1/20 in the code above—in other words, *T* has the value 20*dt*. Now in our real-time plot, we have *dt* = 12 ms, so the time constant *T* is 240 ms or around a quarter of a second.

Later we will see how this sort of filter can be useful.

You might like to save this as SIM1.BAS before you choose NEW.

3.1.4 SUBs and FUNCTIONs

Clear the decks again with NEW (first click on FILE).

By defining a function, a single word can be put in your program to represent a whole operation such as reading an input device. Type the line

```
function twice(n)
```

On pressing ⟨return⟩ on your keyboard, you will see this change to

```
FUNCTION twice(n)
END FUNCTION
```

and now you can type in your function between these lines. Put in the line

```
twice = 2 * n
```

Now we are on a special page just devoted to this function. To get up to the main program, press ⟨F2⟩—where you will see the top line with UNTITLED and a second line with TWICE. Click on UNTITLED. You will see a blank page. Now enter the line

```
PRINT twice(7)
```

and run the program. You should not be too surprised to see the number 14 appear.

Functions can be called recursively. Try changing the line to

```
PRINT twice(twice(7))
```

and run the program.

SUBs are similar, except that they do not return a value. Clear the decks again and type

```
SUB treble(n)
n = 3 * n
```

(The computer will have added an END SUB.)

Press ⟨F2⟩ to go to the main program page (UNTITLED). Now enter the lines

```
CLS    'Clear the screen
x = 5
treble x
PRINT "The answer is "; x
```

There are a few points to notice. In the SUB, `n` is a "dummy variable." The SUB does its task on whatever variables are passed to it, in this case the variable `x`. The value of `x` is changed inside the routine. (For C buffs, the default is that `x` is passed as a pointer)

The single quote after `CLS` means that the rest of the line is a "remark" and is not treated as code.

3.2 THE SIMPLEST MOBILE ROBOT

The "turtle" was popular in the mid-1980s for teaching children the rudiments of programming. It accepted combinations of commands telling it to turn or advance, then it trundled across the floor.

In essence, it consisted of two stepper motors, one driving the left wheel and the other, the right. It steered like a wheelchair, by turning the driving wheels by differing amounts, while skids in front and behind stopped it toppling.

3.2.1 Driving a single Stepper

A starting point for this project is to drive just a single stepper motor. Raid the junk heap for a discarded $5\frac{1}{4}$-in. floppy disk drive. In addition to the electronics and a very interesting motor that rotates the disk, it has a square, chunky stepper motor that drives the head in and out.

This stepper has the advantage that it will operate on a small current, and if you are prepared to take the risk (use a "thirdhand" computer), you can drive it directly from the printer port. It has the disadvantage that when operated in this way it has very low torque, and serves merely to demonstrate how a stepper steps. Its two windings have no center tap, so to use it in the "turtle," you would have to use an H-bridge driver rather than the simpler Darlington driver. But more of that later.

Use a test meter to determine how the four leads are connected to the two windings—pair those leads between which you find some conduction as NS and EW.

To use the printer port, you will need a printer cable. Crimp a pair of 25-way connectors onto a ribbon cable, so that you have plugs of opposite "genders" at the ends, wired pin to corresponding pin. This will act as a printer port extender cable. When connected, the "pins" at the free end will take the form of hollow sockets.

For this elementary test, you can push wires into the connector sockets to attach the motor—connect N, S, E, and W to sockets 2, 3, 4, and 5, respectively.

You will need to know the address of the printer port—it is probably at `&H378` as listed in the software below, but might instead be at `&H278`. The `&H` means that the rest of the number is in the hexadecimal scale of 16. So `&H10` is the decimal value 16, `&H100` = 256, and `&HFF` = 255.

You can use Windows to find the address of the port. Follow the trail START, SETTINGS, CONTROL PANEL, SYSTEM, DEVICE MANAGER, PORTS, LPT1, PROPERTIES, and RESOURCES—and there at long last you will see the address range of the port. The first address is the one you want. Substitute your correct address in all the statements below.

Launch QBasic. Then twiddle the motor shaft with your finger and thumb. It should turn freely.

Enter the following line of code, and then run it:

```
OUT &H378, 1
```

Feel the motor again—it should feel lumpy when you turn it.

You have just applied 5V (or a little less) across the NS winding. Try

```
OUT &H378, 5
```

and run it. The lumpiness should be somewhat greater—you have applied 5 V across both the NS and EW windings.

For neatness, run

```
OUT &H378, 0
```

to ease the load on the printer port.

Stick and fold a label over the motor shaft, to form a pointer so that you can see any movement more clearly.

Now is a good time to start to give some structure to the code. Let us first define the port address as a constant, in the form

```
CONST port = &H378
```

Now we can arrange the main part of the program as a loop, as follows:

```
FOR i = 1 TO 50
   stepto 5
   stepto 6
   stepto 10
   stepto 9
NEXT
```

So, what does `stepto` do? It does nothing until we write it!

```
SUB stepto(n)
   OUT port, n
   PLAY n0
END SUB
```

There's another use for the `PLAY` routine for timing.

Run the program, and you should see the pointer stepping sedately round, making one complete revolution if the motor has 200 steps per revolution. Feel the torque that is required to stop it.

Now speed things up by adding the line

```
PLAY "L64T255"
```

at the top of the program.

Rotation should now be much more brisk. But what has happened to the torque, when you grasp the shaft?

The significance of the numbers 5, 6, 10, and 9 is that they represent the binary numbers

```
0101
0110
1010
1001
```

The 4-bit code can be regarded as representing WENS (the most significant bit comes first, so N = 1, S = 2, E = 4, and W = 8) and these codes will give us NE, SE, SW and NW.

For more precise control, you can employ *half-step mode* using N, NE, E, SE, S, SW, W, and NW, taking eight half-steps for each electrical cycle. The corresponding numbers are 1, 5, 4, 6, 2, 10, 8, 9.

Try it!

3.2.2 Driving More Powerful Stepper Motors

Now we are ready to move on to the stepper motors that you will use for the trolley. These should have six wires, so that in addition to the N, S, E, and W connections, there are center taps to the two coils. Now these center taps can be connected to +12 V, and the drive circuit will pull one or two of the NSEW connections to ground.

A drive circuit that can do the job is a single-chip "octal Darlington driver" such as the ULN2803A, which can be connected directly to the output pins of the printer port. It has eight outputs that can be connected to the NSEW pins of two stepper motors. It even contains diodes to suppress the "spikes" when the inductive loads are turned off.

Connect the circuit as shown in Figure 3.1. Switch on and run the same software again, to make sure that the connections are in order. (When troubleshooting mechatronics, try to make only one change at a time, inching your way from one working system to the next.)

It is now time to test the second motor. Change a line of your `stepto` routine to

```
    OUT port, n * 16
```

which will control the most significant 4 bits of the port.

The new motor should move in the same way as the other. Try

```
    OUT port, n * 17
```

and both motors should move together.

3.2.3 The Mobile Robot

Mount the motors and wheels as shown in Figure 3.2, and you have the rudiments of a mobile robot. Now, however, we must write some much better structured software.

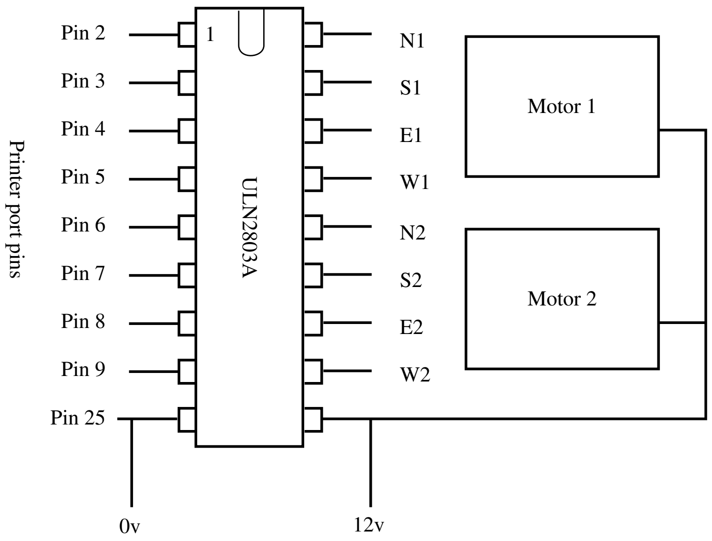

Figure 3.1 *Printer port connections and driver chip.*

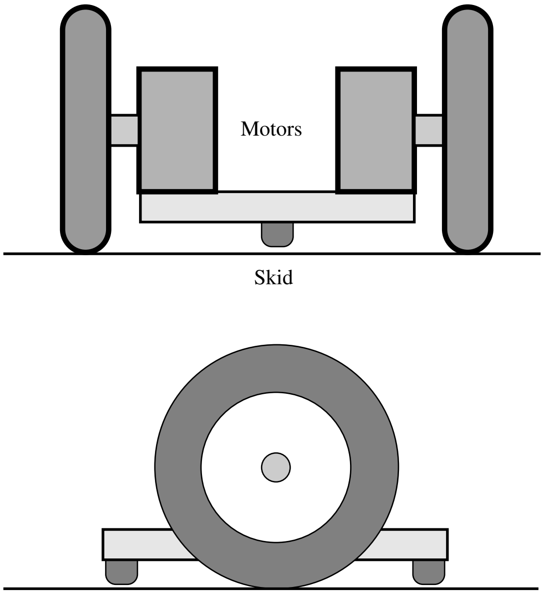

Figure 3.2 *Sketch of a mobile robot.*

Let us define variables Rpos and Lpos for the number of steps that the right and left wheels will have made.

Let us also fill two tables with codes for the right and left wheels, respectively:

```
CONST port = &H378
DIM SHARED Rpos, Lpos
DIM SHARED Rtable(7) AS INTEGER, Ltable(7) AS INTEGER
PLAY "T255L16"
FOR i = 0 to 7
   READ j
   Rtable = j
   Ltable = 16 * j
NEXT

DATA 1, 5, 4, 6, 2, 10, 8, 9
```

This might need some explanation. The SHARED means that the variables defined in the DIM statement will exist in all subroutines—otherwise subroutines can have their own private variables of the same name. READ picks up values one by one from a DATA statement.

Now we need a "user interface" to provide manual control of the robot. There is a function INKEY$ that grabs a value from the keyboard. If no key is pressed it is the "null string." Let us start simply by defining five keys, f and b to run forward or backward, l and r to spin left or right and the space bar to stop. We will use variables dl and dr to hold the value by which to change Lpos and Rpos at each timestep:

```
DO
   a$ = INKEY$
   SELECT CASE a$
      CASE "f"
         dr = 1
         dl = -1
      CASE "b
         dr = -1
         dl = 1
      CASE "l"
         dr = 1
         dl = 1
      CASE "r"
         dr = -1
         dl = -1
      CASE " "
         dr = 0
         dl = -0
```

```
    END SELECT
    Rpos = Rpos + dr
    Lpos = Lpos + dl
    DoMotors
LOOP UNTIL a$ = "q"
OUT port, 0
END
```

Note that when both wheels run forward, one motor must turn clockwise and the other one anticlockwise!

Now we need the `DoMotors SUB`. We can use the `MOD` operator to find `Rpos` modulo 8. That means that as `Rpos` increases, `Rpos MOD 8` cycles repeatedly through the numbers 0 to 7, just the thing we need to look up the drive value in the table:

```
SUB DoMotors
OUT port, Rtable(Rpos MOD 8)+Ltable(Lpos MOD 8)
PLAY "n0"
END SUB
```

Enter and run the program—do not forget to save it first. The trolley should obediently respond as you tap keys, stopping when you press the spacebar and the program ending when you press ⟨q⟩.

But that is just the start. From `Lpos` and `Rpos` you can calculate how far you have gone (you will have to include a variable representing the circumference of the wheels), and you can calculate your heading (you will have to include another variable to represent the separation of the wheels). Indeed, you can keep an estimate of your current position and change your program to accept target coordinates.

You can even add simple sensors. The printer port has input bits for "online," "out-of-paper," and several other conditions. You can read these bits with

```
inp(port+2)
```

You can include the following definition in your software:

```
pin15 = 8       'True if high,  used for error
pin13 = &H10    'True if high,  printer present
pin12 = &H20    'True if high,  out of paper
pin10 = &H40    'True if high,  -ack
pin11 = &H80    'True if low,   -busy
```

If you now add a contact that connects pin 12 to ground if the robot touches a wall, then

```
inp(port+2) AND pin12
```

will have the value zero if the wall is touched and 32 otherwise.

There are even some additional output bits at address `port+1`. These are not buffered and cannot drive a load, however. Be careful. Other bits on this port control the mode of the printer port, determining whether it is buffered or can act also as eight input bits.

```
pin1  = 1     'True gives low,  Strobe
pin14 = 2     'True gives low,  Auto linefeed
pin16 = 4     'True gives high, initialise
pin17 = 8     'True gives low,  select
```

3.3 BALL AND BEAM

In the mid-1960s the focus of my research was "Fast model predictive control for higher order systems." I needed a "higher-order system" on which to demonstrate its effectiveness and devised the ball-and-beam experiment, based on a childhood memory of a game in a seaside amusement arcade.

One outcome was that I discovered that it could be controlled just as efficiently by much simpler pragmatic methods. My research was a success, but my belief in the usefulness of "interesting" academic solutions was seriously undermined.

You might like to see an article linked at http://www.essmech.com/3/3.

3.3.1 Construction

A ball rolls in a grooved plank that is hinged at a central pivot. The motor is driven to tilt the plank, and the task is to control the position at which the ball comes to rest. In this stepper motor version, the ball will oscillate gently close to the target.

When manual control is provided as an alternative, it can be seen that the computer performs much better than a human being. The mechanical construction is clear from Figure 3.3.

There remains the problem of sensing the ball position. One variation is to mount a Webcam above the track and deduce the ball position from the "white blob" in the image. However, we will try a solution that uses much simpler technology.

The method of making the original sensor, and one that is still acceptable, is to stretch a resistance wire along the track. The steel ball makes contact between the wire and another wire on the opposite face of the "V," thereby forming a potentiometer.

The resistance of the wire is likely to be rather small, so a series resistor will have to be added to limit the current and avoid overheating. This means

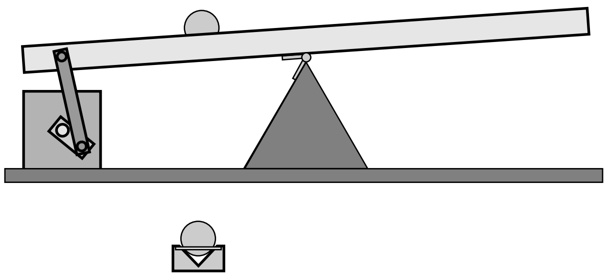

Figure 3.3 *Mechanics of ball-and-beam experiment.*

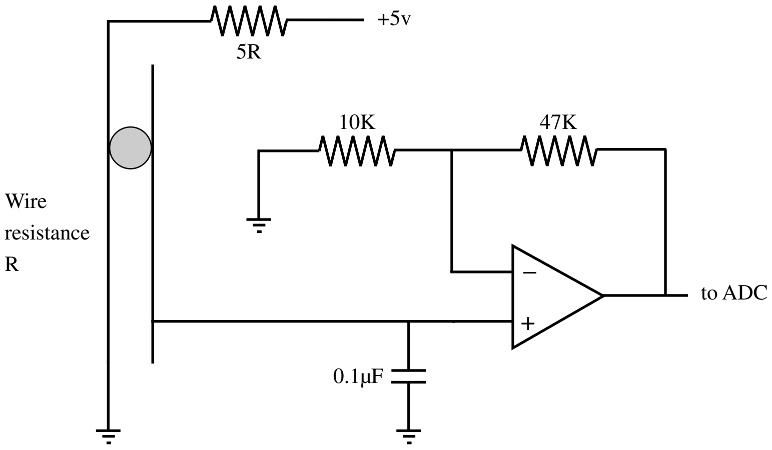

Figure 3.4 *ADC input circuitry, including operational amplifier.*

that the voltages will also be small and there may be a need to add an amplifier.

As the ball rolls, the contact will almost certainly break at times. This would cause the output voltage to suffer from steps to zero and would introduce a large noise signal. With the addition of a capacitor, maybe 0.1 μF (e.g., see Fig. 3.4), the output voltage stays constant when the circuit is broken and the noise consists simply of the step to the new value when contact is resumed.

3.3.2 A Control Strategy

Now we have several new points to establish before the task is complete:

- We must be able to read the ball position into the computer, by writing a routine to drive the ADC.
- We must be able to estimate the ball velocity.

- We must be able to control the plank tilt in response to the ball position and velocity.

The control strategy is of a simple "nested loops" format:

For each value of position error, we will define a demanded velocity. This function will be nonlinear, in that there is a top demanded speed above which we do not wish to go. The difference between the demanded velocity and the estimated velocity is the velocity error.

For each value of velocity error, we will demand a tilt angle. This function will be nonlinear, in that there is a maximum tilt beyond which we do not wish to go.

The stepper motor will be driven in the direction that reduces the tilt error at the maximum stepping rate.

3.3.3 Software

We will start with the version of software that gives manual control, with code very similar to that for the mobile trolley's stepper motor:

```
CONST port = &H378
DIM SHARED Tilt, Demand
DIM SHARED Table(7) AS INTEGER
PLAY "T255L64"
FOR i = 0 to 7
   READ Table(i)
NEXT

DATA 1, 5, 4, 6, 2, 10, 8, 9

DO
   a$ = INKEY$
   IF VAL(a$)>0 then
      Demand = (VAL(a$) - 5)/4   'a fraction -1 to 1
   END IF
   DoTilt
LOOP UNTIL a$ = "q"
OUT port, 0
END

SUB DoTilt
Tilt = Tilt + Demand
OUT port, Table(Tilt MOD 8)
PLAY "n0"
END SUB
```

You will need to use ⟨F1⟩ to discover what the function VAL does.

The number key that you hit will set the speed of the tilt motor, ⟨1⟩ tilting one way, ⟨9⟩ tilting the other way, and ⟨5⟩ causing it to stop.

3.3.4 The ADC Routine

This is a good time to add the ADC routine. It will differ for each interface technique, but the rest of the code can remain the same.

In Section 5.3.5, the code is given for interfacing a chip, the MCP3204, directly to the printer port. This is certainly the most economical and "future-proof" way to go if you do not already have an ADC card.

For most ADC cards, the principles are the same. The card has a base address, just as the printer port has address &H378. It will in fact use a range of 8 or maybe 16 addresses from the base upward. Writing to one of these addresses will set the channel number. Another will start the conversion.

Reading from one of the addresses will give access to a "busy" or "data ready" bit. The simplest code will enter a loop, repeatedly reading this bit until the answer is ready. It wastes a few microseconds, but is less prone to error than relying on interrupts.

If the converter is a 12-bit one, the lower 4 bits of the byte that contains the data-ready bit will contain the most significant 4 bits of the result. Yet another register will contain the lower 8 bits of the result, so we need only combine these and return to the program.

A little extra finesse is added if we scale the answer to lie in the range –1 to +1, so that the rest of the code is not changed if our precision is different. Here is an example. It is for a Contec Series 100 ADC12-16M board, so ancient that it will no longer fit into the PCI slot that is provided on current PCs:

```
CONST b = &H220             'board base address
CONST ADlo = b + 4
CONST ADhi = b + 5
CONST chan = b + 10         'multiplexer channel
CONST start = b + 12
CONST busy = 16             'busy bit in ADhi value

FUNCTION adc (c%)
DIM v%
OUT chan, c%                'set channel number
OUT start, 255              'tell ADC to start conversion
DO                          'wait until ready
   v% = INP(ADhi)
LOOP UNTIL (v% AND busy) = 0
v% = (v% AND 15) * 256 + INP(ADlo)
```

```
                              'combine high nibble
                              'with low byte
adc = (v% - 2048) / 4096      'change range to +/- 1
END FUNCTION
```

Here is the code for a different brand of board, an Advantech PCL-818L, also made to fit the older card slots:

```
CONST adstart = &H2A6
CONST adhi = &H2A7
CONST adlo = &H2A6
CONST readybit = &H40

FUNCTION adc(chan%)
DIM hibyte AS INTEGER
OUT adstart, &H80+chan%
DO
   hibyte = INP(adhi)
LOOP UNTIL hibyte AND readybit
adc = (INP(adlo) + 256*(hibyte AND 15) - 2048)/2048
END FUNCTION
```

A more recent Advantech card that does fit the newer slots is the PCI-1710. It provides the data in a 16-bit word, and to use it, it is necessary to extend QBasic with a library function that uses the INW function of the PC's micro. To add this library when you run, simply launch QBasic with the line `qbasic /linw`, either by launching it from the RUN control in the START bar or by making a shortcut on the desktop. You can edit the shortcut to add the extra `/linw`.

The ADC code and the library function have been supplied by Rodney Elliott of the University of Canterbury, New Zealand. You can find both at http://www.essmech.com/3/3/4.htm.

Two other alternatives are to use a "satellite" microcontroller attached to the serial port to perform the ADC conversion, or to construct the simple ramp-based converter described in Section 5.3.5.

So, having added the appropriate constants and ADC routine to our code, we can add some graphical output.

3.3.5 Graphics

After the constants have been defined, add

```
CONST Tmax = 4, dt=.012
SCREEN 12
WINDOW(-.1, -1.1)-(Tmax, 1.1)
LINE (0,0)-(Tmax,0),9
```

Now we plot the ball position by adding

```
t=t+dt
if t>Tmax then t=0
Ball = Adc(0)
PSET (t, Ball)
```

immediately after the `DoTilt` line.

This will give us a dot that crosses the screen in 4 seconds, since the note length that we have set is equivalent to 0.012 s.

You might need to make adjustments to the amplifier, if you use one, to get the value of the ball position to change over the range of –1 to 1. However, provided you have a reasonable range of change, you can replace the `Ball =` line with

```
Ball =  2 * (Adc(0) - Ballmin)/(Ballmax - Ballmin) -1
```

where `Ballmin` and `Ballmax` are the ADC values you have read at the two extremes of the plank. That will check out the hardware, but we still need to estimate the ball velocity.

Do you still remember the simulation of the lowpass filter in Section 3.1.3? It is no harder to simulate a highpass filter to make an estimate of the ball's speed. The whole thing reduces to two lines of code:

```
Ballvel = (Ball - Ballslow) * 10
Ballslow = Ballslow + Ballvel*dt
```

The argument is the same as before. `Ballslow` is a lowpass-filtered version of `Ball`, `just` as `x` was a low-pass-filtered version of `u` in Section 3.1.3. In this case, the *time-constant* smoothing the differentiation will be $\frac{1}{10}$s. Do not worry. There will be ample theory on this later in the book.

Add these lines, together with

```
PSET (t, Ballvel),14
```

after the existing `PSET` line in your code, and you should be able to see the movement of the ball and its velocity on your screen, as you command the plank to tilt.

3.3.6 The Strategy in Software

Now we just need to automate the process. But let us do it in stages.

First, let us make the demand input control the tilt target, rather than the tilt rate. Add a line at the top

```
DIM SHARED TiltDemand
```

and change the first line of code in `DoTilt` to

```
Tilt = Tilt + SGN(TiltDemand - Tilt)
```

(Use the ⟨F1⟩ key to see what `SGN` does.) Then add the line

```
TiltDemand = 20 * Demand
```

just before the `DoTilt` line in the main program, and you should find that the ball is a little easier to control. You might find that you need to change the number 20 to some other value that gives a useful range of tilt angles.

Of course, the computer does not "know" when the plank is level, so you must hold it level when you start to run the program.

For the next stage, let us make `Demand` control the velocity of the ball. Now we need something like

```
TiltDemand = kt * (0.2 * Demand - BallVel)
```

where `kt` is a *gain constant* that we would like to be large, to correct velocity errors quickly.

We do not want the tilt to be too great, because it takes time to drive the plank back to a level position. Let us define `SUB Limit`.

```
SUB Limit(x, lim)

IF x > lim THEN
   x = lim
ELSEIF x < -lim THEN
   x = -lim
ENDIF
```

Now the line

```
Limit TiltDemand, 20
```

after `TiltDemand` is calculated, will limit the tilt to 20 steps either way.

Make the necessary changes to your code, and experiment with values of the tilt gain, `kt`. Maybe you need to reduce the value of the tilt limit.

For the final step, make `Demand` set the target position of the ball.

In the final program, we will have the following constants and definitions at the top, to which must be added the constants needed for the ADC routine

```
CONST port = &H378
DIM SHARED Tilt, TiltDemand
DIM SHARED Table(7) AS INTEGER
PLAY "T255L64"
FOR i = 0 to 7
   READ Table(i)
NEXT

DATA 1, 5, 4, 6, 2, 10, 8, 9
CONST Tmax = 4, dt=.012
SCREEN 12
WINDOW(-.1, -1.1)-(Tmax, 1.1)
LINE (0,0)-(Tmax,0),9
```

while the main loop of our code will become

```
DO
   a$ = INKEY$
   IF VAL(a$)>0 then
      Demand = (VAL(a$) - 5)/4
   END IF
   Ball =  2 * (Adc(0) - Ballmin)/(Ballmax - Ballmin) -1
   BallVel= (Ball - Ballslow) * 10
   Ballslow = Ballslow + BallVel * dt
   t = t + dt
   IF t > Tmax then t = 0
   PSET (t, Ball)
   PSET (t, BallVel), 14
   VelDemand = kv * (Demand - Ball)
   Limit VelDemand, Velmax
   TiltDemand = kt * (VelDemand - BallVel)
   Limit TiltDemand, TiltMax
   DoTilt
LOOP UNTIL a$ = "q"
OUT port, 0
END
```

You will have to add lines at the top to set the values that you choose for `kt`, `kv`, `VelMax` and `TiltMax`. You might also wish to change the differentiator time constant, which is at present set to 10.

As well as the code above, you will have an appropriate ADC routine and

```
SUB DoTilt
Tilt = Tilt + SGN(TiltDemand - Tilt)
OUT port, Table(Tilt MOD 8)
```

```
PLAY "n0"
END SUB

SUB Limit(x, lim)
IF x > lim THEN
   x = lim
ELSEIF x < -lim THEN
   x = -lim
ENDIF
```

You still have to hold the plank level when the program starts to run.

3.3.7 The Next Step

Now that your system has sensor inputs, the system can discover many of its own parameters. The demonstration version can include switching between control modes and can perform its own calibration at startup. This procedure is as follows:

1. Tilt the plank left 100 steps. This will ensure that the plank hits its limit stop, after which the motor will just "shudder" as more steps are commanded.
2. Wait 3 s for the ball to run down the plank. Measure the value of `Adc(0)` and store it as `BallMin`.
3. Tilt the plank slowly to the right, until the value of `Adc(0)` starts to change. Here the plank will be level, so we would like to set `Tilt` to zero. However, we want to output that particular NSEW combination to the stepper motor to level the plank. So first we set a variable `Tilt0` to `(Tilt MOD 8)`, then set `Tilt` to zero and afterward use (`(Tilt0 + Tilt) MOD 8`) for looking up values from the table.
4. Tilt the plank to `Tilt=20` and wait 3 s, or until the ball stops moving.
5. Measure the value of `Adc(0)` and save it as `BallMax`.

Now you can enter the loop for ball position control.

3.4 "PROFESSIONAL" POSITION CONTROL

An axis of an industrial robot is a far cry from the usual laboratory position control experiment. Most students will be content to derive a response curve that matches the classical "damped second-order response" found in books on linear systems. The professional designer of a motion controller will reject this out of hand. He or she will expect the system to stop at the target position as though hitting a brick wall and to resist all deflecting forces with only the slightest perturbation.

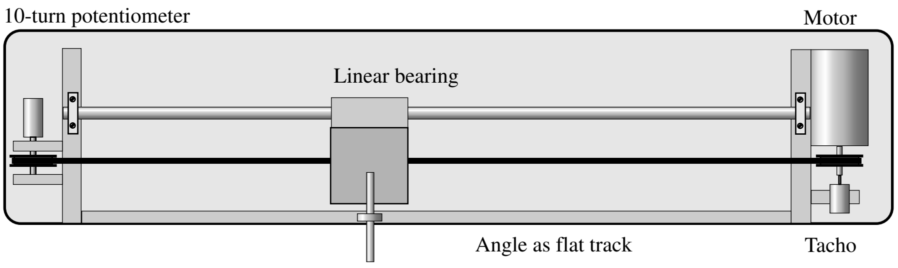

Figure 3.5 *Position control hardware.*

Far too many laboratory experiments are protected by a transparent cover. This prevents the student from feeling the stiffness of the output. The present experiment is meant to be poked and prodded unmercifully.

The hardware (see, e.g., layout in Fig. 3.5) forms three-quarters of an inverted-pendulum experiment that follows, although the control strategy is quite different.

There are many ways to construct this experiment. The machine in our mechatronics laboratory has survived several generations of students, although the new version illustrated here has just been completed. The original motor was marketed as a component for a battery lawnmower, but many suitable motors are now sold for adding electric drive to bicycles.

The 100 W DC motor drives a pulley at one end of a toothed belt. The belt pulls a trolley weighing half a kilogram along a track, constrained on one side by a linear bearing and on the other by a ball race running on a flat track formed by a length of angle.

The belt is joined to the trolley on one side, passes around the motor pulley, back under the trolley, around a second pulley at the other end of the track, and back to the other side of the trolley. The second pulley turns a 10-turn potentiometer. Also connected to the motor shaft is a small motor that acts as the tachometer.

Care must be taken to avoid the rotating parts binding, when bearings are aligned. The potentiometer and the tacho are mounted in sprung clips, so that they can float to accommodate any misalignment.

Two separate power supplies should be used. The sensors and ADC circuitry will use one supply while a second power supply drives the motor. If sensors and motor were to share the same supply, there would be a risk of high-frequency oscillation as the motor drive affected the sensor supply voltage. Another advantage of separating them is that the motor supply can be kept at zero when the program starts, increasing toward 12 V only as the student's full attention is on any oscillations or excursions of the system.

In the original experiment, the potentiometer is supplied by +15 and –15 V supply lines, so that zero volts will represent the center of the track. In the new version, using the single-chip converter, it is supplied from 0 and 5 V,

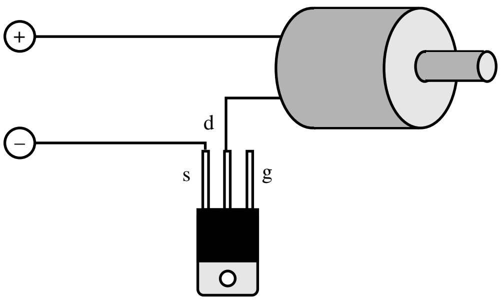

Figure 3.6 *Sketch of motor, FET, supply.*

with the ADC software arranged so that zero is read in the middle of the range. In this case the tacho "ground" should be connected to the midpoint of two 1 kΩ resistors connected across the 5-V supply.

There are a number of steps that we need to accomplish on the way. First we must control a motor of up to 100 W, driving it bidirectionally. We must also consider the use of mark–space control, so that the motor is not continuously under full power.

We also look at the effect of "tacho feedback" for reducing the effect of disturbances.

3.4.1 Simple Motor Control

The experiment can start with a variable DC power supply, a small DC motor, and an N-channel power field effect transistor (see Fig. 3.6):

1. Connect the motor to the power supply. Increase the voltage—probably to around 10 V—so that the motor starts readily and runs at a reasonable speed.
2. Switch off and disconnect the negative connection. Connect the field effect transistor in series with the motor. The *source* is connected to the negative lead of the power supply, while the motor is connected to the *drain*. The *gate* is left unconnected.
3. Switch on again. Hold the negative power supply contact and touch the gate with your finger. The motor will not run. Hold the positive lead instead and touch the gate. The motor should run as it did before connecting the transistor.

A suitable power transistor for this part is a BUK553 N-channel FET. It is designed to be controlled by TTL (transistor–transistor logic) voltages, so you can connect a computer output line directly to the gate.

This next part of the experiment again uses the parallel printer port of the computer. The pins of interest are 2 to 9 for the output bits and pin 25 to serve as a ground pin. As before, the port address will either be `&H278` or `&H378`.

Run QBASIC.EXE, then press the function key ⟨F6⟩ to select the IMMEDIATE WINDOW. This allows you to run code at once, rather than saving it in a program. Enter the line

```
OUT &H378, 255
```

followed by RETURN. You should use a meter or an oscilloscope to check the voltage of pin 2—it should be at 5 V. In fact, this command should have set all eight lines from pin 2 to pin 9 to 5 V.

Now enter

```
OUT &H378,0
```

followed by RETURN. The voltage should fall to zero.

If these actions do not work, repeat them using &H278 instead. In that case also use 278 instead of 378 in the program below. Enter the following program to make changing the outputs neater and more convenient:

```
port = &H378

DO
   a$=INKEY$
   IF a$<>"" THEN
      OUT port, VAL(a$)
   END IF
LOOP UNTIL a$="q"

OUT port, 0
```

If you press the ⟨1⟩ key, pin 2 should be at 5 V and pin 3 at zero. Press the ⟨2⟩ key, and pin 3 should be at 5 V and pin 2 at zero. Press the ⟨0⟩ key, and both should become zero. Press ⟨q⟩ and the program should end, tidying up by setting the port to zero. Save it as P2.BAS.

Now switch off the motor power supply. Connect the computer ground (pin 25) to the negative line and pin 2 to the gate of the transistor. Switch on again. By tapping the ⟨1⟩ and the ⟨0⟩ keys, you should be able to start and stop the motor at will.

Of course, the motor will also run if you tap ⟨3⟩, ⟨5⟩, ⟨7⟩ or ⟨9⟩ and stop if you choose an even number.

3.4.2 Unidirectional Speed Control

The "safe" version of hardware for this part of the experiment uses two miniature DC motors (see Fig. 3.7) that require only a few watts to drive them. Later the much larger motor of the position control experiment can be used,

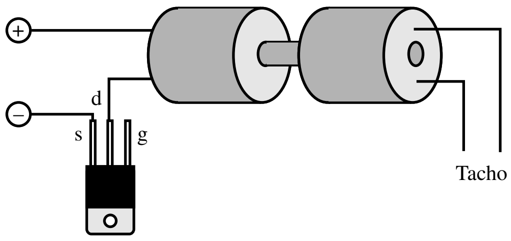

Figure 3.7 *Two small motors with FET.*

with the belt disconnected, so that the trolley does not move. This motor uses much greater power, and care must be taken to avoid overheating the field effect transistors used in the experiment.

You should first run through these experiments with the miniature motors, then repeat them with the larger motor. In each case the driven motor is connected mechanically to a second DC motor. Both turn together. When they run, a voltage is generated on the second motor that is proportional to the speed. By connecting this voltage to an analog input of the computer, you can use the speed as a feedback signal. This second motor is being used as a tachometer.

As in previous experiments, the ADC routine will first read in an integer and then scale it to a floating-point value in the range –1 to 1. You need to find the code that is appropriate for your ADC and enter both the FUNCTION and the constants that define the interface.

With this ADC routine and with the velocity voltage connected to channel 0, the start of a program to view the tacho signal can be as follows:

```
CONST tmax=4
SCREEN 12

WINDOW(0,-1)-(tmax,1)

dt = .01

DO

   a$ = INKEY$
   v = ADC(0)
   t = t + dt
   IF t > tmax THEN t = 0
   PSET (t, v)
LOOP UNTIL a$ = "q"
```

This will read the voltage and display it on the screen as a sort of *oscilloscope display*. Spin the motors by hand, then run the driven motor by "dabbing" a wire across source and drain of the control transistor.

One of the most important tasks in feedback control is to ensure that the signals are of the correct polarity. If they are not, negative feedback can become positive feedback with disastrous results.

When you run the motor, make sure that the resulting velocity is shown on the screen as positive. If it is not, reverse the connections of the tacho.

Pressing the ⟨q⟩ key will end the program.

When you are sure that this part of the program is working correctly, you can expand it to obtain closed-loop control. You need to include a command for turning the motor on and off and another for comparing the speed with some demanded value. In turn, something is needed for changing the demand value at the press of a key:

```
CONST port= &H378
CONST tmax=4

SCREEN 12
WINDOW (0,-1) - (tmax,1)

dt = .01

vdemand = 0

DO
   a$ = INKEY$
   if a$ = ">" THEN vdemand = vdemand + 0.1
   if a$ = "<" THEN vdemand = vdemand - 0.1
   v = ADC(0)
   IF vdemand > v THEN OUT port, 1
   IF vdemand < v THEN OUT port, 0
   t = t + dt
   IF t > tmax THEN t = 0

   PSET (t, v)
   PSET (t, vdemand), 12 'red

LOOP UNTIL a$ = "q"

OUT port, 0
```

Run the program. The motor should remain still. Now tap the ">" key once or twice (remember to press ⟨shift⟩), and the motor should turn. Hold the motor shaft loosely to try to slow it down. You should see the motor current increase as indicated on the power supply meter, with very little drop in speed.

In practice it is preferable to use "." in place of ">" and "," in place of "<" so that there is no need to press the ⟨shift⟩ key.

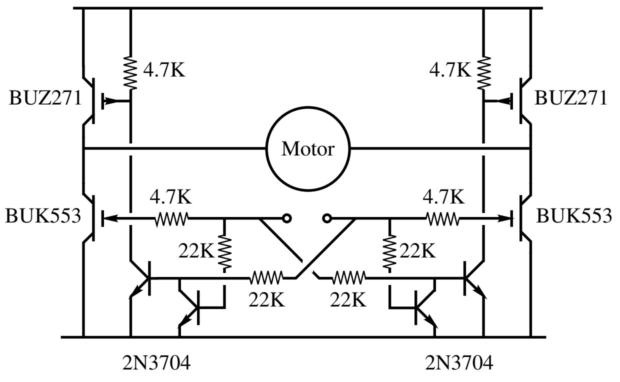

Figure 3.8 *H-bridge schematic.*

3.4.3 Bidirectional Speed Control

When the motor turns in just one direction, the speed can be controlled simply by turning the motor on and off. If the motor is to be capable of turning both ways, we must have some means of driving it with either a positive or negative voltage. We could use two power supplies to give positive and negative voltages, or we can use a single power supply with an *H-bridge* (see Fig. 3.8).

As we have seen, this enables each wire of the motor to be connected either to the positive supply or to ground. The schematic has the appearance of the letter "H," giving it its name. The circuitry must at all costs prevent one side being connected to both positive and ground at the same time!

The circuit uses two bits of the parallel printer port, bit 0 and bit 1 on pins 2 and 3. When the output value is 1, the motor runs in one direction; when it is 2, it runs in the opposite direction; and if it is zero, the motor free-wheels until it stops. (Remember that the binary value of bit 0 is 1; of bit 1, is 2; of bit 2, is 4; and so on.)

Start off by using the program of the last section. You will have changed the connections to the motor in order to replace the single transistor by a full H-bridge drive. In the process, the sense of the motor drive could have changed.

When you tap the ">" key, the motor should start to run. If the velocity trace on the screen is positive, all is well. If not, you must reverse the motor leads or the tacho leads. But which?

If you are using the position control rig with the belt removed, check that the motor runs in a sense that would carry the trolley to the right. If not, reverse it. Now check the tacho voltage and if necessary reverse the tacho connections to ensure that the trace is positive.

You should now get the same velocity control as before—simply running in the positive direction. Now is the time to change the software to take advantage of the two-way drive.

The previous program needs only slight modification to give "bang-bang" control. If the velocity error (`v` – `vdemand`) is positive, full negative drive is applied. If the error is negative, the drive is positive.

```
CONST port= &H378
CONST tmax=4

SCREEN 12

WINDOW (0,-1) - (tmax,1)
dt = .01

vdemand = 0

DO
   a$ = INKEY$
   IF a$ = "." THEN vdemand = vdemand + 0.1
   IF a$ = "," THEN vdemand = vdemand - 0.1
   v = ADC(0)
   olddrive=drive
   IF vdemand > v THEN
      drive = 1
   ELSEIF vdemand < v THEN
      drive = 2
   ELSE
      drive = 0
   END IF
   IF drive = olddrive THEN
      OUT port, drive
   ELSE
      OUT port, 0
   END IF
   t = t + dt
   IF t > tmax THEN t = 0
   PSET (t, v)
   PSET (t, vdemand), 12 'red
LOOP UNTIL a$ = "q"
OUT port, 0
```

So, why is the code complicated with `drive` and `olddrive`? If we switch repeatedly from forward to reverse drives, there is a tendency for the H-bridge transistors to overheat. Here a zero is output between changes of sign.

Nevertheless, there is the disadvantage that for most of the time maximum drive is applied. Is there a way to reduce the drive when full drive is not needed?

3.4.4 The Proportional Band

The previous strategy resulted in the motor being connected to the power supply at all times, resulting in a substantial power drain even when there is no disturbing torque and the demanded speed is zero.

The current drain could be reduced if the motor were switched off when “near” the target. There would, in effect, be a “gap” where there is zero drive.

Add a line to the program above:

```
CONST gap = .05
```

Then change the “middle” lines to

```
IF vdemand > v + gap THEN
   drive = 1
ELSEIF vdemand < v - gap THEN
   drive = 2
ELSE
   drive = 0
END IF
```

This will reduce the current drain—indeed, it will be zero when zero speed is demanded—but the velocity can be in error by an amount “gap” with no corrective action. With an on off controller, how can we get proportional action to fill in the gap? Many commercial power amplifiers provide a proportional mark-space output—at great expense. We can construct a mark–space controller in software.

We set up a variable `g` that will shuttle to and fro across the gap in a triangular wave, as follows:

```
g = g + dg
IF g > gap THEN dg = -.1 * gap
if g <= 0 THEN dg = .1 * gap
```

Now we change the “engine room” lines again, to

```
IF vdemand > v + g THEN
   drive = 1
ELSEIF vdemand < v + g - gap THEN
   drive = 2
ELSE
   drive = 0
END IF
```

dg must be set to 0.1 * gap at the top of the program, after gap is defined. After 20 times round the loop, g cycles through its range of values. If the velocity error is exactly zero, the drive will be set to zero all the time. If vdemand - v = gap/2, the drive will be positive for half the time. If it is greater than gap, the drive will be positive all the time.

We have a mark–space ratio that can increase the drive in steps of 10%, which is dg's proportion of gap. If dg is made much smaller, the cycle time is longer and the motor buzzes accordingly.

3.4.5 Position Control

By now you will have applied velocity control to the motor and tacho of the position control experiment, having carefully removed the belt from contact with the motor pulley.

Instead of using the keyboard to set the velocity demand, we can use the 10-turn potentiometer mounted on the second pulley.

Connect the potentiometer output to the input of ADC channel 1. The software function should now return a value that varies from –1 to 1 as the potentiometer is turned from end to end:

```
CONST port= &H378
CONST tmax=4
CONST gap=.05

dg = .1 * gap
k = 1

SCREEN 12
WINDOW (0,-1) - (tmax,1)

dt = .001 'or a smaller value to suit the display rate
DO
   a$ = INKEY$
   v = 5 * ADC(0) 'The tacho voltage is rather small
   x = ADC(1)
   vdemand = -k * x
   g = g + dg
   IF g > gap THEN dg = -.1 * gap
   IF g <=0 THEN dg = .1 * gap
   verror = vdemand - v 'more about this later
   IF verror > g THEN
      OUT port, 1
   ELSEIF verror < g -gap THEN
      OUT port, 2
```

```
    ELSE
       OUT port, 0
    END IF
    t = t + dt
    IF t > tmax THEN t = 0
    PSET (t, v)
    PSET (t, x), 14 'yellow
LOOP UNTIL a$ = "q"
OUT port, 0
```

As a consequence of the way we have used `g` and `gap`, we should always have one or more periods of zero drive before a drive reversal, so we can dispense with `drive` and `olddrive`.

Run the program. The potentiometer will control the speed. If the polarities are correct, the motor should spin in the opposite sense to the way you turn the potentiometer shaft. If this is not the case, reverse the supply connections to the potentiometer.

Now switch off the power and reconnect the drive belt. Beforehand, be sure that the speed control is "good".

Cautiously increase the voltage of the power supply that runs the motor. As the motor starts to move, it backs off the potentiometer shaft to zero—you have achieved position control.

The next step is to put back a demand signal, this time a position demand:

```
if a$ = "." THEN demand = demand + 0.1
if a$ = "," THEN demand = demand - 0.1
if a$ = "0" THEN demand = 0
```

The `vdemand` line is now

```
vdemand = k * (demand - x)
```

Tapping a key will step the demand along the potentiometer travel. Pressing the ⟨0⟩ key will return the demand to zero, so that a larger step response can be seen.

Experiment with various values of `k` and also `gap`, trying to get a fast response without overshoot.

Now put the control system to its real test. How far can the trolley's position be pushed away from the target before full corrective drive is applied? With linear feedback tuned to avoid overshoot, the control might be rather "soggy." With a nonlinear strategy, something much "crisper" can be achieved.

Try setting `gap` to zero—but make the test very brief, since the power transistors will be getting hot. This reverts to bang-bang control, and the system will be very stiff indeed, although the motor will buzz and the current will be high. Increasing `gap` will make the performance "quieter" at the expense of a softer response to disturbances.

3.4.6 Nonlinear Correction

An overshoot occurs if the motor approaches the target "too fast to stop." Drive saturation plays an important part in the performance. Even when the linear parameters have been tuned for a heavily damped response for small deflections, a larger disturbance can cause it to overshoot badly.

An easy answer is to put a limit on the demanded velocity. Now the system will approach the target at constant speed, however large the deflection. By setting this speed limit lower, the time constant of the final settling response can be made faster while still avoiding an overshoot. There is always a compromise to be made.

Add the line at the top of the program

```
vmax = 0.1
```

and after `vdemand` is defined, add the lines

```
IF vdemand > vmax THEN vdemand = vmax
IF vdemand < - vmax THEN vdemand = -vmax
```

Now experiment by varying the values of `k`, `vmax`, and `gap`.

3.4.7 Estimating Velocity

Suppose that we have no tacho signal. How can we stabilize the system?

In the ball-and-beam experiment, you produced an estimate of the ball's velocity in real time. The play routine was used to control a precise time increment, a necessary part of the process.

We used the property that a highpass filter can be expressed as the difference between the original signal and the output of a lowpass filter as follows:

$$Ts/(1+Ts) = 1 - 1/(1+Ts)$$

In other words, to estimate the velocity, we construct a "lagged" version of the position and subtract it from the position. We can set up a "chicken and egg" situation where we use `vest` to update the lagged version and in turn calculate `vest` from the lagged version

```
vest = (x - xslow) * kt
```

while the lagged position is updated by

```
xslow = xslow + vest * dt
```

If the variables are updated at intervals `dt`, the time constant of the lag will be `1/kt` seconds, so if `kt` = 20, it is 50 ms.

Add a line at the top of the program:

```
CONST kt = 20
```

Now the scale factor of `vest` is unity, so that a value of 1 indicates that `x` is changing at 1 unit per second. In contrast, `v` was scaled by an arbitrary factor determined by the tacho.

The gain and the value of `vmax` you must use with `vest` will differ considerably from the values you chose when using `v`. You will find that `vest` would go off screen if displayed at the same scale as `v`; so display `vest/20` instead.

Our previous way of getting a timed response was by inserting a `PLAY` command into the program loop, slowing down the whole process. We would instead like to run the mark–space calculation as fast a possible. QBasic allows us to set up an “interrupt” so that `vest` will be updated in time “stolen” from the main loop every 10 msec.

Since the earliest days of computers, interrupts have been a fundamental part of the system. When your output device was a teletype, tapping away at 10 characters per second, you did not want to waste time waiting for it and instead preferred to get some more computing done between taps.

So, every time the interface is ready for another character, it interrupts the computer. The values in the registers used for the task in hand are tucked away safely, then the machine attends to loading the code of the next character to be printed and outputting it. Then it must retrieve and restore the values of the registers and perform an “interrupt return.” Modern peripherals may be much faster, but they still operate at a snail’s pace compared with computing speeds.

In just the same way, QBasic allows us to play music in the background, interrupting us for some more notes when the tune is coming to an end. If we set the play rate to the highest speed and supply just one note at a time, we will receive interrupts at intervals of 0.012 s.

Near the top of the program, the command

```
PLAY "mbl64t255"
```

sets the fastest playing speed and also tells the music to play in “background mode.”

Add some further instructions just before the loop begins:

```
ON PLAY(1) GOSUB rates
PLAY ON

PLAY "cde"
```

These tell the software that when the number of notes queued for playing falls to 1, there should be an interrupt causing a subroutine call to the label `rates`. The third line "sets the alarm clock" with three notes to play.

Now at the end of the program, add a line

```
END
```

so that program execution cannot "fall through" to the subroutine.

Now for the subroutine itself. This is added after the `END`:

```
rates:
vest = (x - xslow) * kt
xslow = xslow + vest * dt
PLAY "n0"
RETURN
```

Add one last line in the heart of the control loop to display the estimated velocity in red, among the other `PSET` lines:

```
PSET (t, vest / 20), 12
```

Now run the program again. Remember that the actual control program is exactly as before—we are using "real" velocity and not `vest`.

As you demand steps of movement, the tacho velocity is shown in white. The estimated velocity is shown in red. If you have "got it right," the values in the constant speed section will be the same size. Try values other than 20 to scale `vest` to match the traces.

When you are at last satisfied with your estimate of the velocity, you can try it in the control loop. In the `verror =` line, replace `v` with `vest/20` (or the value you found for the best match). Now your control depends on estimated velocity, not on the tacho.

Control will be a bit more wobbly, overshoots will be harder to avoid, and the control may have to be softer. Once more, experiment with `k`, `vmax`, and `kt` to see what you can achieve.

3.4.8 Discrete-Time Control

There is just one more step to try. We will move the entire control loop into the interrupt routine. Now the top-level program merely "twiddles its thumbs" in a loop. Every 10 ms, it is interrupted to allow the ADCs to be measured, the feedback to be calculated, and the drive to be output.

The mark–space drive behavior given by `gap` would become a nuisance if `g` were changed only every 10 ms. Its cycle through 20 steps would take 200 ms. The system would vibrate at five cycles per second.

We can instead calculate a drive signal `u`

```
u = (vdemand - v) * kv
```

inside the interrupt routine, but let the mark–space gap part of the routine run much faster in what is left of the main program. With a value of 1.0 for gap, the drive will be positive all the time if u is greater than 1, will be negative all the time if u is less than –1, and will give a proportional mark–space ratio in between. The result is like this:

```
'CONSTants required for the ADC routine go here

CONST port= &H378        '(might be &H278)
CONST tmax= 4
CONST gap= 1
dg = .1 * gap
CONST vmax = 0.1

k = 1
kv = 10
kt = 10

SCREEN 12
WINDOW (0,-1) - (tmax,1)

PLAY "mbl64t255"
dt = .01                 'Interval for this playing rate

ON PLAY(1) GOSUB rates
PLAY ON
PLAY "cde"

DO
   a$ = INKEY$
   if a$ = "." THEN xdemand = xdemand + 0.1
   if a$ = "," THEN xdemand = xdemand - 0.1
   if a$ = "0" THEN xdemand = 0
   g = g + dg
   IF g > gap THEN dg = -gap/10
   if g < 0 THEN dg = gap/10
   IF u > g THEN
      OUT port, 1
   ELSEIF u < g - gap THEN
      OUT port, 2
   ELSE
      OUT port, 0
   END IF
```

```
LOOP UNTIL a$ = "q"
PLAY OFF
OUT port, 0
END

rates:
v = ADC(0)
x = ADC(1)
vest = (x - xlsow) * kt
xslow = xslow + vest * dt
vdemand = k * (xdemand - x)
IF vdemand > vmax THEN vdemand = vmax
IF vdemand < - vmax THEN vdemand = -vmax
u = (vdemand - v) * kv '(or later try vest)
t = t + dt
IF t > tmax THEN t = 0
PSET (t, v)
PSET (t, x), 14         'yellow
PSET (t, vest / 20), 12 'red

PLAY "n0"
RETURN

FUNCTION adc(chan%)

' The code for the ADC routine goes here

END FUNCTION
```

Experiment with various values of `k`, `kv`, and `vmax`.

Now try using `vest` instead of `v` for the control by changing the line suggested.

3.4.9 Summary

Now you have seen how a motor can be switched by one computer output line if it runs in just one direction, or by two lines with the aid of an H-bridge if it is to be bidirectional.

You have seen that a tacho velocity sensor enables the position control to be very stiff, as would be required by a machine tool positioner. You have seen that without such a tacho, a softer control is achievable with the use of a digital filter to estimate the velocity.

You have seen that the gain can be expressed in terms of a *proportional band* and that this proportional drive can be achieved as mark–space modulation by means of further software.

In short, you have seen that a motor, an amplifier, and two transducers can be turned into an *industrial-grade* position control system with a few simple lines of software.

3.5 AN INVERTED PENDULUM

In the previous experiment, a position control loop has been closed after some cautious tests. A variety of nonlinear strategies have been investigated to obtain the performance expected of an actuator such as a robot axis.

Most of the feedback decisions were made empirically, experimenting to find values that would give a swift response with no overshoot. For the control of an inverted pendulum, we can also follow an empirical approach. However, we must avoid the mistake of thinking that we can add the pendulum control on top of the algorithm that we have found for position control. You may find some of the feedback coefficients surprising.

3.5.1 Skeleton Software

We can strip out the control algorithm from the position control software, to leave the following skeleton:

```
'Constants required for the ADC routine go here

CONST port= &H378       '(might be 278)
CONST tmax= 4
CONST gap= 1
dg = .1 * gap
CONST vmax = 0.1

k = 1
kv = 10
kt = 10
SCREEN 12
WINDOW (0,-1) - (tmax,1)

PLAY "mbl64t255"
dt = .01            'Interval for this playing rate

ON PLAY(1) GOSUB rates
PLAY ON
PLAY "cde"

DO
   a$ = INKEY$
   if a$ = "." THEN xdemand = xdemand + 0.1
   if a$ = "," THEN xdemand = xdemand - 0.1
   if a$ = "0" THEN xdemand = 0
   g = g + dg
   IF g > gap THEN dg = -gap/10
   if g < 0 THEN dg = gap/10
   IF u > g THEN
      OUT port, 1
   ELSEIF u < g - gap THEN
      OUT port, 2
   ELSE
      OUT port, 0
   END IF
LOOP UNTIL a$ = "q"
PLAY OFF
OUT port, 0
END

rates:
v = ADC(0)
x = ADC(1)
vest = (x - xlsow) * kt
xslow = xslow + vest * dt
'The control goes here, with a line u = . . .

t = t + dt
IF t > tmax THEN t = 0
PSET (t, v)                'white
PSET (t, x), 14            'position in yellow

PLAY "n0"
RETURN

FUNCTION adc(chan%)

' The code for the ADC routine goes here

END FUNCTION
```

3.5.2 The Pendulum and Tilt Sensor

You might have noticed the tubular rod projecting from the front of the new position control experiment. This is a tube in which a pair of crossed Hall effect sensors are mounted, forming the pivot for the pendulum. Simple rubber O-rings restrain the pendulum mounting and prevent it dropping off.

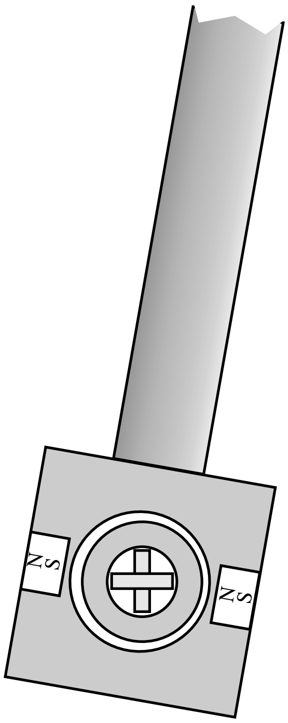

Figure 3.9 *Pivot and magnets.*

The pivot of the pendulum takes the form of a mounting for a pair of magnets, as shown in Figure 3.9. This slides onto the sensor tube. Into it can be screwed a variety of lengths of lightweight aluminum tubing.

The sensors are linear Hall effect sensors, chips the size of a small transistor with three connections. The UGN3504 is now obsolete, but equivalents such as the A1302 are available from Allegro. Supply lines of 5 and 0V are connected to two of these, while the third delivers an output voltage that varies either side of 2.5V in proportion to the normal component of the magnetic field.

With two sensors, both sine and cosine signals are available. For calculation and simple control, we can take the sine to be the same as the angle in radians over the small relevant range. However, we also have the information we need it we wish to swing the pendulum from hanging down to standing up.

The new version of the experiment uses the single-chip ADC described in detail in Section 5.3.4. Both input and output are handled by the printer port connection, while the signals from the Hall effect sensors are connected to inputs ADC2 and ADC3, pins 3 and 4 of the MCP3204 chip (see schematic in Fig. 3.10).

Add the following lines in the skeleton program above, in the subroutine `rates`:

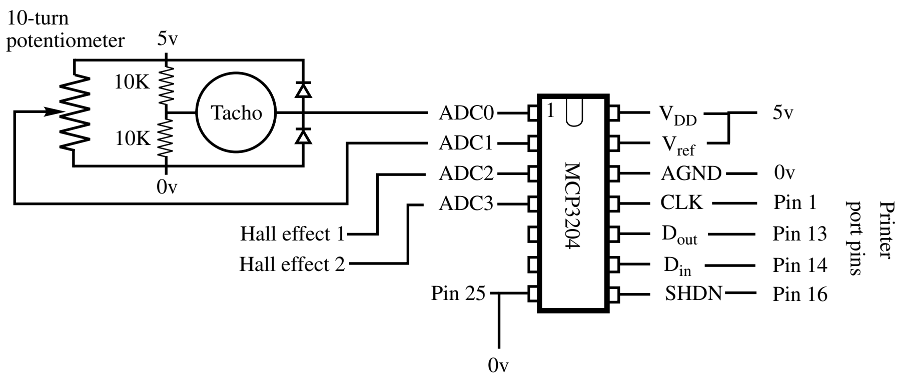

Figure 3.10 *Circuit and printer port connections.*

```
tilt = ADC(2)
PSET (t, tilt), 10 'Plot tilt in green
```

Make sure that the motor drive supply is switched off, and run the program. Rotate the sensor—swing the pendulum about—and note that the sensor voltage (shown in green) varies over a reasonable range. You may need to swap the sensor connections, so that `ADC(2)` represents the sensor that gives a value near zero when the pendulum is upright. The signal should move positive as the pendulum is tilted to the right. If it does not, slide the mounting off the pivot, reverse it front to back and replace it.

To centralize the tilt reading, we have to subtract the value given when the pendulum is straight up. Change the `tilt=` line to

```
tilt = (ADC(2) - tilt0)
```

But how do we get the value of `tilt0`? After the `dt=.01` line at the top of the code, add

```
tilt0 = adc(2)
```

and make sure that you balance the pendulum before running the program.

Rotate the pendulum and check that the value swings equally positive and negative.

3.5.3 Finding the Tilt Rate

Now we use the interrupt routine again to estimate the tilt rate from the tilt signal. Since we are using a substantial gain already, this estimate might be rather noisy:

```
tiltrate = (tilt - slowtilt) * 20
slowtilt = slowtilt + tiltrate * dt
```

Once again the constant `20` defines a time constant of 50ms.
Add these lines to the `rates` routine, and also add

```
PSET (t, tiltrate), 12 'red
```

among the other `PSET`s to display it.
Now we have enough variables to hand to try to control the system.

3.5.4 Building a Strategy

Start with the skeleton program with the modifications of the last section. Before going any further, we should make sure that the polarities are correct. Run the program, but be sure to keep the motor power supply turned off.

Move the trolley to the right by hand, and make sure that the yellow trace rises on the screen. Move it swiftly a little way to the right and make sure that the white velocity trace also rises. Lean the pendulum to the right—the green trace should rise. Hold the pendulum straight up, and the green trace should be central. Wave the pendulum to the right, and the red `tiltrate` trace should be positive for a short burst.

If all is well, we are ready to begin.

The whole feedback exercise comes down to finding a suitable expression for the `u =` line. We can start with

```
u = 10 * tilt
```

If the pendulum tilts to the right, the trolley moves to the right. This implies that the trolley will move to try to hold the pendulum upright.

Hold the pendulum high up, with the trolley in the center of the track. Being careful to keep your hands clear of the trolley, turn the motor power supply voltage setting down to zero, switch on, and increase the volts steadily to 10 or so.

You will see that this feedback arrangement acts as a sort of position control. As you move the top of the pendulum to the left, the trolley moves to follow it. There is no damping, however, so the response will be fairly oscillatory. We should add some damping. Do this by changing the vital line to

```
u = 10 * tilt + 10 * tiltrate
```

By adjusting the two coefficients, you can obtain a swift and agile response. Try releasing the pendulum briefly—it should remain upright, but the trolley will drift to the left or right until the pendulum hits the stops.

It is time to add feedback to keep the trolley in the center of the track. But should we add positive or negative position feedback? Change the line again to

```
u = a * tilt + b * tiltrate + 10 * x
```

where `a` and `b` are the numbers you have chosen by trial and error.

Hold the pendulum again and power up. Now you will see that as you move the pendulum left and right, the trolley follows as before—but with a difference. It moves rather more than you move the pendulum, so that the pendulum leans inward—or at least it should if you have the coefficients right. Adjust the coefficient of `x` so that the pendulum rotates about a point roughly twice as high as the length of the pendulum.

Now give the pendulum another solo run, starting near the center. The trolley will "swing" to and fro with increasing amplitude. Catch it before it hits the ends.

Now add yet another term, some constant times `v`. When you have the right coefficient, the "swing" will be damped and the pendulum will "rest" near the centre. "Rest" might not really be the right word, since the trolley will jitter to and fro.

You can instead use a multiple of `vest`. It will actually give a smoother response. So now the only sensors used are the potentiometer measuring `x` and the sensor measuring `tilt`. The other variables are deduced from these.

We can go a little further by replacing `x` in the `u=` line with `(x – xdemand)`. As the keys are tapped, the trolley will obediently wobble to the left or right as commanded.

You have followed a systematic but empirical process to arrive at feedback coefficients that will stabilize the pendulum. It might have been surprising at first that positive, rather than negative feedback had to be used for the trolley position. In a later chapter, the equations of the systems will be analyzed and your results will be explained.

Another exercise will involve simulating the system. At present we know very few of its parameters, so you should take the opportunity now to measure some.

3.5.5 Measuring the System Parameters

To make an accurate mathematical model of the system, we need to measure a number of parameters. By modifying some of the earlier programs, we can let the computer do most of the hard work for us.

The key parameter is the acceleration of the trolley under full drive. Another is the effect of the velocity on the acceleration.

Which units should we use to measure the position with? The most convenient measure is in terms of the potentiometer voltage. Since we have scaled the ADC output to the range –1 to 1, we will take this range to be two units of position.

Next we need to have some absolute scale of time. We must use the interrupt method in preference to one that relies on the speed of a program loop. The following program will calibrate the interrupt process to give the correct value to use for dt:

```
CLS
PRINT "Calibrating dt - please wait five seconds"

PLAY "mbl64t255"
ON PLAY(1) GOSUB rates
PLAY ON
PLAY "cde"
t = TIMER
n=0
DO
LOOP UNTIL n>=500
PLAY OFF
PRINT "Use dt =";(TIMER - t)/n
END
rates:
n = n + 1
PLAY "n0"
RETURN
```

It simply uses the TIMER function (press ⟨F1⟩ to check it out) to measure the duration of 500 interrupts, which we know to be in the region of five seconds.

Now we need a program to measure acceleration. To test the acceleration, we first ask for the trolley to be moved by hand to the left hand end of the track. When we press ⟨space⟩ for "go", the trolley accelerates under full drive. When it reaches the center, the drive is removed and friction brings the trolley to a halt.

Load in the skeleton you used in Section 3.5.1 and delete even more, then enter the new code. Edit the program that follows to use the dt value that you have just found:

```
'CONSTants required for the ADC routine go here

CONST port= &H378       '(might be 278)
CONST tmax= 1 'Note the change of value
SCREEN 12
WINDOW (0,-1) - (tmax,1)
PLAY "mbl64t255"
dt = .01     'Change this to the value you have found
```

```
ON PLAY(1) GOSUB rates
PLAY ON
PLAY "cde"
'------New code from here on
OUT port, 0
PRINT "Move the carriage to the left end and press
<space>"
DO
LOOP UNTIL INKEY$ = " "      'Wait for key
x0 = ADC(1)
t=0
OUT port, 1                  'Off we go
DO
   PSET (t, x), 14
LOOP UNTIL x>=0  'When we get to the middle
t1 = t           'make a note of the time
x1 = x           'and check the position
OUT port, 0   'switch off and coast to a halt
t=0
DO
   PSET (t, x), 14
LOOP UNTIL t >.9 'Should stop before 0.9 seconds
x2 = x           'How far did it coast?
a = 2 * (x1 - x0) / (t1^2) 'using s=1/2 a t^2
b = a * (x1 - x0) / (x2 - x1) 'v^2 = 2 a s1 = 2 b s2
PRINT "Acceleration = "; a
PRINT "Deceleration = "; b
'----End of new code
DO            'Leave the information on the screen
   a$ = INKEY$
LOOP UNTIL a$ = "q"
PLAY OFF
OUT port, 0
END

rates:
x = ADC(1)
t = t + dt
'Point A
PLAY "n0"
RETURN

FUNCTION adc(chan%)
' The code for the ADC routine goes here
END FUNCTION
```

So, that gives us some numbers. What do they mean?

The fact that the stopping distance is of the same order as the run up suggests that the major slowing effect is not viscous drag but friction. On this assumption we can model the trolley with two equations:

$$dx/dt = v$$

$$dv/dt = cu - \text{friction}$$

where u is +1, –1 or zero according to the drive setting. Friction will be of constant magnitude, multiplied by the sign (±1) of the velocity. Its magnitude will be given by the deceleration that we have just found.

Since the drive has to act on top of the friction, the constant c will be the sum of the acceleration and the deceleration. We are now in a position to make a mathematical model for the trolley movement.

At the point marked `A` in the `rates` subroutine, add

```
vm = vm + (c * u - friction *SGN(vm)) * dt
xm = xm + vm * dt
```

substituting the values you have found for `c` and `friction`.

Now we have to introduce `u` and set the model initial conditions.

At the command `OUT port, 1`, insert

```
u = 1
xm = x
```

and at the `OUT port, 0` command, insert

```
u = 0
xm = x
```

Finally, we must display the model position by adding

```
PSET (t, xm), 12
```

next to the existing `PSET` command. As a result, the new program will show the actual and modeled positions superimposed.

For a more complete model, we need to know more about the pendulum. We can measure the vital parameter with minimal technology. Hang the pendulum upside-down. Give it a swing and measure the period of oscillation.

3.5.6 Final Touches

The program can be polished up for demonstration purposes. The datum value can be written to a file on disk, so that the pole does not have to be balanced at the start.

The original version of the experiment has brackets at the end, forming stops near the midheight of the pendulum, while there are limit stops on the angle to which it can topple. The program can start by applying velocity control to run the trolley gently toward the end. As the pendulum is pushed past the upright position by the stop, the mode switches to balance control and the pendulum is balanced. You can see this in action in a video file to be found at http://www.essmech.com/3/5/6.htm.

A simple test on the tilt angle will detect whether the pendulum has toppled and enable the routine to erect it and start again.

On the new version that is just being commissioned (in 2005), the pendulum can swing freely in a complete circle. The trick now is to erect it from a hanging position by building up oscillations until it can be "caught" at the top.