

Lottery Random Number Generator

John Morton

This is an excerpt from the first and second editions of the book “PIC: Your Personal Introductory Course”. It works from basic concepts introduced elsewhere in the book, and so does not function as a standalone tutorial. As this has proved a popular program, but was removed from the latest, third edition version, I have included it here. I would like to thank Max Horsey who came up with the idea and designed the circuit.

This project will be a multipurpose random number generator. Its five modes will produce random numbers between 1 and 6, 1 and 12, 1 and 12 (with probabilities such that it is comparable to using two dice), and 1 and 99. There will also be a National Lottery number generator, producing six different numbers between 1 and 49. It will automatically switch off if not used after 3.5 minutes. It will have four seven-segment displays, to show the number, a push button to select the mode and two contacts to be pressed when a number is required. The two contacts are in effect a ‘vibe’ sensor. By holding your finger over the two contacts you make the connection using the resistance of your skin. This resistance is measured (using the analogue input) and used to generate the random number. In this way it produces personal random numbers (particularly important when playing the National Lottery).

The four seven-segment displays can be strobed, so that they only require 11 outputs, the push button will require one input, and the contacts an analogue input, creating a total of 13 I/O pins. It will therefore fit on the PIC71. The push button will be connected to RB0 as an external interrupt, the controlling pins of the seven-segment displays to RA1 to RA4, and the seven display pins from both of the displays to RB1 to RB7. The contacts will use the AN0 (RA0). All this is shown in the circuit diagram (Figure 3.8).

The next step is to construct a flowchart. This will be a very complex program, so a good flowchart is really vital. We won’t be able to foresee all the complications from this early stage, so as with most complicated programs, we will have to add some things as we go along. The basic flowchart is shown in Figure 3.9. One of the first things the PIC will need to do is work out which mode it should be in, and display this on the seven-segment displays. The mode will be shown as: **1-6_**, **1-12**, **-6-6**, **1-99** and **Lott** (‘_’denotes a blank space). When the device first starts up, it should be in the 1-6 mode. Lots of different things will be going in the program and the display subroutine will have to be regularly called, if the strobing is to work properly. It will need to display the mode which the device is in, and then later the random numbers etc., so our conventional display method may have to be changed slightly. We will have four file registers called **dig1**, **dig2**, **dig3** and **dig4**. The display subroutine will display whatever code is in these through the correct seven segment display, so no seven-segment encoding takes place in the display subroutine. Those four file registers act as a transporter for displaying code. In a section of the program, a number may be moved into one of them, and then when that particular display is next turned on, the effect will be noticed (this happens effectively instantly due to the PIC’s high processing speed). Figure 3.10 shows how the four file registers are allocated.

When the mode push button is pressed it should trigger the external interrupt and cause the PIC to enter the isr. Once in the isr we can make the PIC branch to a mode changing section. This section will allow the device to scroll through the different operating modes, displaying everything on the displays. We need to debounce the button as well as waiting for it to be released (without stopping the displays from working).

Exercise 3.8 Write the **Init** subroutine, don’t forget to set up TMR0, prescaled at 256, and enable the TMR0, external, and global interrupts.

Our next step will be writing the external interrupt part of the **isr**. We will need some sort of general purpose file register to keep track on which mode the PIC is in. Each mode is thus assigned a number between 0 and

4. This file register (let's call it **Chooser**) will therefore need to be incremented when the external interrupt is triggered, thus selecting the next mode. If the number in **Chooser** gets to 5, we know that it has gone too far and must be reset to 0.

Exercise 3.9 Write the five lines which will achieve this.

The PIC will need to change the display every time a new mode is selected, so the PIC will then need to use the number in **Chooser** to move the correct codes into dig1 to dig4. This can be done using the program counter.

Exercise 3.10 Write the seven lines that use the program counter and the number in **Chooser** to jump to one of five different sections labeled **Dis6**, **Dis66**, **Dis12**, **Dis99** and **DisLot**.

As three out of the five modes have the same first two symbols, they can all go to the same place when setting those first two displays up. In **Dis6**, **Dis12** and **Dis99** the PIC need only set up **dig3** and **dig4** individually. The arrangement of the seven segments on Port B is simply: a, b, c, d, e, f, g, -. To display the number six, the following number is therefore used: 10111110 (work it out!).

Exercise 3.11 Write the five lines in **Dis6** to move the correct numbers into **dig3** and **dig4**, and then jump to a section called **DisCommon**. Then do the same for **Dis12** and **Dis99**.

When returning from the **isr**, we do not want to enable the global interrupt bit, because we will need to debounce the mode button. This is done by leaving the interrupt disabled until it has been released for a tenth of a second.

Exercise 3.12 Write the five lines in **DisCommon**, which set up displays 1 and 2, and then returns from the **isr** (not enabling the global interrupt).

DisLot and **Dis66** will move the correct numbers into **dig1-4**, and then return.

Exercise 3.13 Write the nine lines which make up **DisLot**, and the seven which make up **Dis66** (think carefully on how to get it down to seven lines).

Thus the entire mode changing section is complete, however, as the PIC needs to start with 1-6 on the displays, we need to fake an external interrupt when it first starts up. This is simply done by setting the external interrupt flag and calling the **isr** (with the **call** instruction). Setting the flag tells the **isr** to go to the external interrupt section (rather than the **TMR0** section). If the PIC is to start in 1-6 mode, the number 4 should be moved into **Chooser** in the **Init** subroutine. Thus when **Chooser** is incremented it will become 5, which is too big, and thus reset to 0 which is the mode number for 1-6.

Exercise 3.14 Write the three lines which should be executed before the PIC enters the main loop. These will set things up, and cause a fake external interrupt. Add the extra two lines to your **Init** subroutine which set **Chooser** up correctly (not shown in Answer section).

The PIC should then test the contacts, while still displaying everything. As they are connected to an A/D converter we need to see whether the A/D conversion result is over a certain minimum value. This has been found, through experimentation, to be 10. The PIC should first start an A/D conversion, and then call the **display** subroutine which handles the strobing of the four displays. The A/D clock is set to one eighth the operating frequency. An instruction is executed in one quarter of the operating frequency, so after two instructions the A/D conversion will have taken place. The **display** subroutine will have taken well over two instructions, and so by the time it returns the A/D conversion will have finished. There appears, therefore, to be little need for an A/D interrupt in this case. After calling the **display** subroutine the PIC, in order to debounce the mode button, should test to see whether it has been released. If it has been released the PIC

should enable a TMR0 interrupt (and the global interrupt). When the TMR0 interrupt later occurs, the external interrupt can then be re-enabled, and the mode button will again affect the program.

Exercise 3.15 What should the first six lines of the **Main** loop be, if they are to start an A/D conversion, call the **display** subroutine, and then handle the debouncing of the mode push button. (*Note:* If the push button is still pressed, the PIC should jump to a section labeled **ADTest**.)

The PIC should now double check that the A/D conversion is now complete by testing bit 2 of **ADCON0**, going back to **Main** and skipping one instruction if it is still in progress. Then we need to see whether the A/D conversion result is 10 or greater. If the result is less than 10, the PIC should go back to main.

Exercise 3.16 Which six lines will achieve this?

Once the contacts are pressed, the PIC should reset the registers it is using to time the 3.5 minutes (until automatic switch off), so that it starts counting afresh. Before we can reset the timing registers, we need to know the exact nature of the timing. We will therefore now write the section which will send the PIC to sleep after 3.5 minutes of the contacts not being pressed. Rather than constantly calling a subroutine and checking how much time has passed, we can use the TMR0 interrupt to make the PIC automatically jump to a certain section after a certain time has passed. The TMR0 interrupt is enabled whenever the mode changing push button isn't pressed (which is most of the time), so can be used for this purpose. Using a 2.47 MHz oscillator, and prescaling TMR0 by 256, the TMR0 interrupt occurs every 0.106 seconds. To wait 3.5 minutes we need the TMR0 interrupt to occur 1981 times. We can do this using two prescalers – one at 256 (the maximum), and another at 7. Though this doesn't produce a time of exactly 3.5 minutes, it doesn't matter for our purposes. This first thing we will need to do in the isr is test the external interrupt flag to see what event caused an interrupt. If the external interrupt flag is clear, the PIC should jump to **TMRInt**, otherwise it should continue with the mode changing section that you have already written.

Exercise 3.17 Write the two lines which will do this.

The first thing to happen in **TMRInt** is the enabling of the external interrupt (as by this time it will have been released for long enough to overcome button bounce). We can then begin to time 3.5 minutes – first we check whether this interrupt has occurred 256 times (by decrementing a file register which initially was clear, and testing to see if the result is 0). If it hasn't happened 256 times, the PIC should return while enabling the global interrupt. If it has happened, the PIC needs to test to see whether the TMR0 interrupt has occurred 256 times, 7 times. A similar technique is used. If the full 3.5 minutes have passed, the PIC should reset the second prescaler, clear both ports (so everything turns off), and go to sleep. Because we have enabled the external interrupt, pressing the mode changing push button will wake the PIC from sleep, but as the global interrupt isn't enabled the isr will (fortunately) not be called and the mode will not be changed. The PIC will simply carry on where it left off, so the instruction after sleep should make the PIC return from the isr, while enabling the global interrupt.

Exercise 3.18 Which eleven lines make up the **TMRInt** section? You will need two general purpose file registers as postscalers (call them Post256 and Post7 if you want).

Going back to the main loop, we now know how to reset the timing registers when the contacts are pressed. We should clear the first postscaler, and move seven into the second one. The PIC should also ignore the mode changing button (i.e. all interrupts) from then on.

Exercise 3.19 Which four lines will perform these tasks? These are to be performed once the contacts have been touched.

The PIC should then wait a very short while for the circuit created through the skin to stabilize, before storing the A/D conversion result. This can be done by performing the A/D conversion 256 times. The following

section (you can call it **Stabilizer**) will start A/D conversion, and keep looping until it has finished. It should then wait for this to happen 256 times (you should know how to do this), looping back until it has. The PIC should then store the A/D result in a file register (call it **Skin** if you want).

Exercise 3.20 Write the seven lines which will achieve this (I called the file register which counts 256 times **ADCount**).

The PIC now needs to wait for the finger to be removed so as to break the circuit until this happens, the displays need to be chasing. We should first start an A/D conversion, and then sort out the chasing of the display (by calling a special subroutine – let’s call it **Chaser**). This routine will change the numbers in **dig1** to **dig4**, and so we then need to call the **display** subroutine to keep the displays on while the contacts are pressed. After this, the PIC should check that the A/D conversion has finished, looping back to the beginning of this section (**Main2**), skipping the next instruction which started the conversion.

Exercise 3.21 Which five lines will achieve this? (Do not worry about the **display** or **Chaser** subroutines.)

Once it has been confirmed that the A/D conversion is finished, the result should be compared with 10, to see whether or not the contacts have been released. If they are pressed, the PIC should loop back to **Main2** and start another A/D conversion. Alternatively the PIC should wait for the result to become suitably low, 200 consecutive times (the resistance between the contacts is very variable and this confirms that the finger has really been removed). After the result has been confirmed, the general purpose file register used to count 200 times needs to be reset.

Exercise 3.22 Which eight lines will perform these tasks?

The PIC now needs some way of generating a random number (this is one of the hardest things for machines to do). Humans are much more random than PICs, so the best random number generator is linked to the human user. TMR0 can be made to count up 614400 times a second, so if we look at the number in TMR0 at the moment the contacts are released, it will be random. In this way the human user dictates the number in TMR0, but has no real control over it. This random number will be between 0 and 255, and we need to squeeze this down to between 1 and 6 or 1 and 99, etc., but we will tackle this at a later stage. The PIC nevertheless needs to store the number currently in TMR0 in a general purpose file register.

Exercise 3.23 Which two lines will store the value? You could call the GPF **TouchTime**.

The PIC should now use the value in **Skin** to choose a suitable message (this can be done in a subroutine), and then wait 3 seconds. We could copy out the routine to create a 3 second delay, or could try and use the TMR0 interrupt section. The first prescaler in the **TMRInt** section extends the time length to about 27 seconds, and then the second one to the full 3.5 seconds. By changing the values of these prescalers we can still create the same delay in the **TMRInt** section, using different numbers. If we make the first prescaler create a delay of 3 seconds, we can then use a larger second prescaler to bring that length up to 3.5 minutes. $3/0.106 = 28.3$, so if the first prescaler is 28, it will create a roughly 3 second delay. 3.5 minutes is 210 seconds, so the second prescaler should be 70. We will need to now go back and make these changes throughout the program it is normally useful to use Notepad to Search for the words **Post256** and **Post7** to make sure nothing is left out. You should naturally change the names of the postscalers as well.

To now use the TMR0 interrupt section to create a 3 second delay, we first need to reset the first postscaler, and then enable the TMR0 (and global) interrupts. We should then set a general purpose bit (e.g. **sec3**) which will be cleared in **TMRInt** when the first prescaler has reached 0, and keep testing it until it is cleared. Assign the **sec3** to a file register you laid aside for general purpose bits using the **#define** instruction in the declarations section. Don’t forget to insert the line to clear **sec3** in the appropriate place in **TMRInt**.

Exercise 3.24 Write the nine lines which call the message choosing subroutine and then wait 3 seconds (while constantly calling the **display** subroutine). The loop involved may be called **Loop3Sec**.

Exercise 3.25 Then, in the TMRInt section, after the following pair of lines:

```
defsz      Post28
retfie
```

... three lines must be added. The first two would reset the first postscaler, and the third should clear **sec3**. Write these three lines.

To get our random number we need to use the value in **TouchTime** (completely variable), together with the value in **Skin** (quite variable), to create a random number. We will therefore add the two together and store the result in a general purpose file register (I called it **Random**). The PIC then needs to use the value in **Chooser** (which tells it which mode it's in), to jump to one of five different places, where it can convert the number between 0 and 255 into something more suitable for that particular mode. These five sections could be called: **Ran6**, **Ran66**, **Ran12**, **Ran99** and **RanLot**.

Exercise 3.26 Write the tenlines that will achieve all this.

To change a number between 0 and 255 into one between 1 and 6, we need to repeatedly add the number 6 to the number between 0 and 255. Take the number 156 for example – add 6 to it and you get 162. Do this repeatedly until the number overflows past 255 and you get a number between 1 and 6: $162 + 6 + 6 \dots + 6 = 252$. $252 + 6 = 258 = 2$. In this way we have converted the number 156 into 2. We can do the same for the other modes by adding 12, 99 or 49 repeatedly. One of the problems we face with this method is that adding 6 to 250 sets the carry flag, as the result is 0, however 0 is of no use to us. It would be far easier if we were looking for a number between 0 and 5, so we will look for a number between 0 and 5, and then add one to it afterwards. As the 1–6, 1–12, and 1–99 modes are very similar, we can use the same section for all three. For this to be possible we should add the number in a file register (**Scaler**), rather than the number itself. In this way we can move the appropriate number into the file register and then go to a section common to all three.

Exercise 3.27 Write the **Ran6**, **Ran12** and **Ran99** sections which move the appropriate number into the working register, and then branch to the common section (you may call it **Adder**). They should therefore consist of two lines each.

The section **Adder** should firstly store the number in the working register (I called the GPF **Scaler**). The number in **Scaler** should then be added to **Random**, and then the carry flag should be tested. If set, the number has grown past 255 and can be used, however if clear the PIC should loop back to **Adder**, skipping the instruction which moves the working register into **Scaler**. Finally, once out of the loop the PIC should add 1 to **Random** so that it becomes a number between 1 and 6 or 1 and 12, etc.

Exercise 3.28 Which six lines will perform these tasks?

We have completed three out of the five different random number generators. The double 1–6 generator should be completed by having two 1–6 generators. We can simply use the one we used for the 1–6 mode to begin with, so change the line:

```
          b      Ran66
... to:   b      Ran6
```

At the end of the **Adder** section you should test bit 2 of the **Chooser** register which will be set if in **-6-6** mode. If clear the PIC should go to some other place – just write the goto (or b) instruction with no destination for the moment. Otherwise the PIC should skip that line and move **Random** into a general purpose file register which we shall call **Tens**. It should also get a new random number (by adding **TouchTime** to **Ran-**

dom), and convert it into something between 1 and 6. After this the PIC should go to somewhere which we shall, for the moment, call **Continue66**, with the number in **Random** in the working register. Make the necessary changes to the **Adder** section.

Exercise 3.29 Which ten lines make up the section after **Adder**? (You will need a loop in these lines, I have called mine **Ran66**.)

We have left the most complicated number generator to the end. We must get a number between 1 and 49, however we cannot have the same number more than once in every consecutive group of six. It is therefore necessary to store the previous five lottery numbers to compare the current number with, but let us first get our number between 1 and 49.

Exercise 3.30 Which five lines will get a random number between 1 and 49?

When you are doing the same thing to a string of file registers, it is convenient to use indirect addressing. We will therefore make the file registers which store the previous lottery numbers (I have called them **Lott1** to **Lott5**), consecutive file registers. It is easier to test to see whether the number in a file register has reached a certain value by simply testing one bit, rather than going through the business of subtracting the value and testing the zero flag, etc. If **Lott5** is file register number 1F, going one too far as the PIC scrolls through the five file registers could be tested by looking at bit number 5 in the **FSR**. **Lott1** to **Lott5** should therefore specifically be file registers numbers 1B to 1F. The **FSR** should first be loaded with the number of the first file register (1B). The number in **Random** should then be compared with that in the Indirect Address (**INDF**). If the two are the same, the PIC should jump to a section to change the random number (you could call it **Changer**), otherwise it should continue.

Exercise 3.31 *Challenge!* Which six lines will achieve this? You could call this section **CompareLott**.

How can we get a new lottery number? One of the easiest solutions is adding the value in **Skin** to the number in **Random**, and then going through the 1–49 conversion process all over again.

Exercise 3.32 Write the three lines which make up **Changer**. (At the end it should go to **RanLot**.) It might be advisable to put **Changer** after the subroutines and before **Start**.

We should now move on to the next lottery value by incrementing the **FSR**. We need to check whether we have used all five registers; this can be done by testing bit 5 of the **FSR**. If we haven't compared the number with all five, the PIC should loop back to **CompareLott**, skipping the first two instructions.

Exercise 3.33 Write the three lines which will perform this task.

Now that it has been confirmed that the new number is different from any of the others, we need to store it for comparison with later numbers. We will need a file register to keep track of which lottery number we are on (1–6). The device should display this number along with the one between 1 and 49, e.g. **2-37** shows that this is the second number of six, and the number is 37. I've called this register **LottCount**. It is reset in the **Init** subroutine, and then it is incremented after each lottery number (and reset after the sixth). To work out which file register (**Lott1** to **Lott5**) to store the number in, we simply add the hexadecimal number **1B** to the number in **LottCount** (a number between 0 and 5), and move that number into the **FSR**, thus selecting one of **Lott1** to **Lott5**. The random number is then moved into the **INDF** and sent to the appropriate location.

Exercise 3.34 Which six lines will store the random number correctly, and then increment **LottCount**.

Now that we have got our random number, we can begin to display it. Our problem is converting a number (e.g. 37) into its tens and ones digits (3 and 7). The tens digit is the number of times we can subtract ten until the result is negative, minus one. The ones digit is number left when we subtract ten enough times. The loop

increments a file register holding the tens value, then subtracts the number 10 from **Random**; if the result is negative it stops (skips out of the loop), otherwise it loops back and does it again. We don't want the PIC to increment the tens digit the first time round, so when we jump to this loop from the ends of **Adder** and **RanLot**, we should skip the first instruction. The file register holding the tens value could be called **Tens**, and the loop **TensLoop**.

Exercise 3.35 Write the instruction to jump to **TensLoop** and skip one instruction, at the end of the **Adder** and **RanLot** sections. Then write the first five lines of **TensLoop**.

The number in the working register now is 10 less than the units digit of the random number.

Example 3.4 The number was 26, 10 was subtracted three times, and the file register **Tens** was incremented twice. The number in **Tens** is now 2 (the tens digit), and the number left in **Random** is -4 (252 because there are no negative numbers).

We simply need to add 10 to the number in **Random** to get our units digit. In all modes, the units digit is displayed on the display furthest to the right (**dig4**), so we can then convert the units digit into a seven-segment code (using a decoding subroutine) and then move it into **dig4**. We can also clear **dig1**, **dig2** and **dig3** leaving the three left hand digits blank. It is the line which calls the decoding subroutine which the PIC should jump to after the **Ran66** section, so this line should be labeled **Continue66**.

Exercise 3.36 Write the seven lines to do all this. Then write the eleven line subroutine which converts the units digit into seven-segment code.

We could now simply use the number in **Tens**, turn it into seven-segment code, and then move it into **dig3**. However, it would look quite nice if we got rid of leading zeroes (displaying **_4** instead of **04**). Thus we move the number from **Tens** into the working register, and then before calling the **Decoder** subroutine, we should test the zero flag, and skip to the next section (called **LottCounter** if it is set). Otherwise, the PIC should skip that line and call the **Decoder** subroutine. If the device is in any mode except the **-6-6**, the tens digit is displayed on **dig3**, however when in the **-6-6** mode, the tens digit is the other number, and so should be displayed as far away as possible (**dig1**). The mode number for **-6-6** is 4, so we need to see whether or not the number in **Chooser** is 4 (i.e. bit 2 of **Chooser** is set). We should then move the tens digit into **dig3** or **dig1** depending on the result of the test.

Exercise 3.37 Write the nine lines to complete these tasks. At the end of the section the PIC can jump to a section called **LottCounter**. Afterwards you may add a section used specifically to move the seven-segment code into **dig3**, which then leads on to **LottCounter**.

LottCounter is a section in which **Chooser** is checked to see whether or not the device is in the lottery mode. If so, the actual lottery number (1 to 6) should also be displayed in the following format: **n-** (e.g. **3-39** means that the third lottery number is 39). Conveniently, when in the lottery mode, **Chooser** holds the number 1, so we can simply apply the instruction **decfsz, w** to **Chooser**, to see if the device is in lottery mode. If it isn't, the PIC can return to **Main** (the numbers have been chosen and displayed), otherwise we will have to put something in **dig1** and **dig2**. Before we return to **Main**, we must reset the file register **Tens**, which is incremented to find the tens digit. This can be done at the beginning of the **LottCounter** section.

Exercise 3.38 What three lines will reset the **Tens** file register, and then make the PIC jump back to **Main** if the device is not in lottery mode?

It is now necessary to move the code into **dig2** that will display a dash (turn on the g segment). We can also use the number in **LottCount**, to display the correct number through **dig1**. **LottCount** was 0 before the first digit was chosen, after which it was incremented to 1. We can therefore use the number in **LottCount** directly, turn it into seven-segment code using the **Decoder** subroutine, and then move it into **dig1**.

Exercise 3.39 Write the five lines which set up **dig1** and **dig2** with the correct values.

After this the PIC should return to **Main**, clearing the lottery file registers if necessary.

We have (at last) completed the main body of the program, along with various subroutines, but there remain some to be done: **display**, **Chaser** and **MessageChooser**.

The **display** subroutine should use the two least significant bits in TMR0 to jump to one of four sections. These would turn on a particular display and move the relevant number (from **dig1** - **dig4**) into Port B. After each section the PIC should return, enabling the global interrupt.

Exercise 3.40 What 27 instructions complete the entire display subroutine?

The second subroutine we need to write is the one which causes the segments to chase while the contacts are being pressed. This subroutine is constantly being called during this time, but we want the segments to change every tenth of a second. It therefore needs some sort of timing element, a postscaler (**Mark240**) which allows the PIC to return if a tenth of a second hasn't passed, and continue otherwise. A 2.47 MHz crystal is being used, so a value around 240 would be a suitable postscaler.

Exercise 3.41 Which six lines will perform this task (don't forget to update the marker after the correct time has passed)?

We can create the impression of a chase using only three different positions, as illustrated in Figure 3.11.

Exercise 3.42 Write the codes which need to be in the four displays for each of the three different cases (a total of twelve numbers), and clearly label them.

We will need a file register (**ChaseCount**) to keep track of which version is being displayed (this should be incremented every time a tenth of a second passes), and then used to jump to one of **Chase1**, **Chase2** or **Chase3**. In **Chase3**, the number 255 should be moved into **ChaseCount** to reset it.

Exercise 3.43 Write the 35 instructions which make the PIC go to one of **Chase1** to **Chase3**, and which make up the three sections in which numbers are moved into file registers **dig1-dig4**.

At the end of each section, the PIC should return. Finally, we need to complete the **MessageChooser** subroutine. This uses the number in **Skin** to select a message (maximum of four letters) which is displayed for 3 seconds after the contacts have been released. You can have as many messages as you want, and can adapt the ones suggested in this example.

I have chosen my critical values to be 11–12, 13–15, 16–20, 21–25, 26–35, 36–50 and 50+ (a total of 7). The respective messages will be: sad, bad, cool, john, hot, tops, and ace. We know that the value must be greater than 10, because otherwise the PIC would believe that the contacts aren't pressed, and not call **MessageChooser**.

Exercise 3.44 What four lines will jump to a section called **sad** if the number in **Skin** is less than 12 (to be put at the start of the **MessageChooser** subroutine)?

Exercise 3.45 Write the other 21 instructions which will allow the PIC to go to **bad**, **cool**, **john**, **hot**, **tops** or **ace** depending on the value in **Skin**.

Each of the sections **sad** to **ace** must then move the correct numbers in **dig1** to **dig4**. You are best off using a look-up table to find out the codes for the letters.

Each letter will have its own subroutine which returns with the appropriate seven-segment code.

Example 3.5 The subroutine `_E` would consist of:

```
_E    retlw    b'10011110'    ; returns with correct seven-segment code
```

Exercise 3.46 Write the letters' look-up table. Include upper and lower case letters and don't forget to start the name of each subroutine with an underscore (`_`), so that letters such as `b`, and `c` etc., aren't confused by the assembler as the `branch` instruction, or the `carry` flag. My version contained 26 instructions, but yours may differ (don't forget to have a blank).

The sections `sad` to `ace`, now need only call the appropriate subroutine and then move the working register in one of `dig1` to `dig4`.

Exercise 3.47 Write the seven sections `sad` to `ace` which set up `dig1` etc., and then return. They should each consist of nine instructions.

All that remains is to define all the general purpose file registers (unless you have been doing so all along), and to set them up correctly in the `Init` subroutine. You may find you have to move some sections to the end of your program due to problems with adding to the program counter. For example, if we add the number in `Chooser` to the program counter in the upper half of the page (instruction addresses 100–1FF), we will come across problems. If you want to check your answers to the exercises, they're shown below.

There are various ways in which you can personalize this device. You could for example change the mode that it first started up in (not a huge difference as the device only 'starts up' when you first connect the batteries). This is done by changing the number you move into `Chooser` in the `Init` subroutine. You could also change the number and content of messages displayed.

Answers to Exercises

3.8

```

Init      clrf      porta      ; resets I/O ports
          clrf      portb
          b         Init+4    ; leaves address 0004 for isr
          b         isr
          bsf      STATUS,5   ; selects bank 1
          movlw    b'00001'   ; RA0:contacts,RA1- RA3:7
          movwf    TRISA      ; seg.controlling pins
          movlw    b'00000001' ; RB0:mode button,
          movwf    TRISB      ; RB1-RB7:7 seg.code
          movlw    b'00000111' ; sets up TMR0,prescaled at
          movwf    OPTION     ; 256
          movlw    b'00000010' ; RA0 and RA1 are analogue,
          movwf    ADCON1     ; Vref is supply voltage
          bcf      STATUS,5   ; returns to bank 0
          movlw    b'00110000' ; sets up interrupts:TMR0 on,
          movwf    INTCON     ; external on,global off
          movlw    b'01000001' ; sets up A/D conversion:
          movwf    ADCON0     ; clock:Fosc/8,channel:AN0,
                               ; converter is on
          return              ; returns from subroutine
    
```

3.9

```

          incf      Chooser    ; goes on to next mode
          movlw    d'5'       ; has it gone through all five
          subwf    Chooser,w   ; modes?
          btfsc   STATUS,Z    ;
          clrf     Chooser     ; yes,so reset back to mode 1
                               ; (1-6)
    
```

3.10

```

          movfw    Chooser     ; takes the number out of Chooser
          addwf    PCL,f      ; skips that many instructions
          b         Dis6      ; 1-6_
          b         DisLot    ; Lott
          b         Dis12     ; 1-12
          b         Dis99     ; 1-99
          b         Dis66     ; -6-6
    
```

3.11

```

Dis6      movlw    b'10111110' ; 6
          movwf    dig3
          movlw    b'00000000' ; blank
          movwf    dig4
          b         DisCommon ; goes to common place to set up 1 -
    
```

Dis12

```

          movlw    b'11011010' ; 2
          movwf    dig4
          movlw    b'01100000' ; 1
          movwf    dig3
          b         DisCommon ;
    
```

Dis99

```

          movlw    b'11100110' ; 9
    
```

```

movwf    dig3
movlw    b'11100110' ; 9
movwf    dig4
b        DisCommon ;

```

3.12

DisCommon

```

movlw    b'01100000' ; 1
movwf    dig1
movlw    b'00000010' ; -
movwf    dig2
return   ; returns, without re-enabling the interrupts

```

3.13

DisLot

```

movlw    b'00011100' ; L
movwf    dig1
movlw    b'00111010' ; o
movwf    dig2
movlw    b'00001110' ; t
movwf    dig3
movlw    b'00001110' ; t
movwf    dig4
return

```

Dis66

```

movlw    b'00000010' ; sets a dash
movwf    dig1
movwf    dig3
movlw    b'10111110' ; sets a number 6
movwf    dig2
movwf    dig4
return

```

3.14

Start

```

call     Init        ; set everything up
bsf     INTCON,1    ; tell the isr to go to mode changing section
call     isr         ; call isr

```

3.15

Main

```

bsf     ADCON0,2    ; starts A/D conversion
call    display     ; sorts out displays
btfsc  portb,0      ; tests for up button
b       ADTest      ; still pressed,so changes
                    ; nothing
movlw   b'10100000' ; released,so enables TMR0
movwf   INTCON      ; and global interrupt

```

3.16

ADTest

```

btfsc  ADCON0,2    ; conversion is in progress
b       Main+1     ; goes back to main,and skips one line
movlw   d'10'      ; is result less than 10? (are
subwf   ADRES,w    ; contacts pressed?)
btfss  STATUS,C    ;
b       Main       ; no,so loops back

```

```

etc. ; yes,so continues

3.17 btfss INTCON,1 ; has external interrupt occurred?
      b TMRInt ; no,so goes to timer interrupt
      etc. ; yes,so continues

3.18 TMRInt bsf INTCON,4 ; enables external interrupt
      decfsz Post256 ; first postscaled by 256
      retfie ;
      decfsz Post7 ; secondly postscaled by 7 (3.5 minutes passed?)
      retfie ; no
      movlw d'7' ; yes,so resets second
      movwf Post7 ; postscaler
      clrf portb ; clears outputs
      clrf porta ;
      sleep ; sleeps (low power mode)
      retfie ; returns,enabling the global interrupt

3.19 clrf Post256 ; resets sleeping postscalers
      movlw d'7'
      movwf Post7
      bcf INTCON,7 ; no more interrupts from now on

3.20 Stabilizer bsf ADCON0,2 ; starts conversion
      btfsc ADCON0,2 ; has it finished?
      b Stabilizer ; no,so keeps looping
      decfsz ADCCount ; has this happened 256 times
      b Stabilizer ; no,so loops back
      movfw ADRES ; yes,so stores A/D result
      movwf Skin

3.21 Main2 bsf ADCON0,2 ; starts A/D conversion (are contacts released?)
      call Chaser ; sorts out chasing while contacts are pressed
      call display ;
      btfsc ADCON0,2 ; is conversion finished?
      b Main2+1 ; no,so loops back

3.22 movlw d'10' ; is A/D low enough to suggest
      subwf ADRES,w ; finger has been removed?
      btfsc STATUS,C ;
      b Main2 ; no,so loops back and starts another conversion
      decfsz LetGoCount ; yes,confirms low result 200 times
      b Main2 ;
      movlw d'200' ; resets the counting GPF
      movwf LetGoCount ; register

3.23 movfw TMR0 ; takes the number out of TMR0
      movwf TouchTime ; stores it in TouchTime

3.24 call MessageChooser ; choose message using number in Skin

```

```

Loop3Sec      movlw    d'28'      ; resets first postscaler,so that
              movwf   Post28    ; the PIC times the full 3 seconds
              bsf     sec3      ;
              movlw   b'10100000' ; enables TMR interrupt
              movwf   INTCON    ;
              call    display   ; keeps displays going
              btfsc  sec3      ;
              b       Loop3Sec  ; keeps looping until 3 seconds have passed

3.25         movlw    d'28'      ; resets postscaler
              movwf   Post28    ;
              bcf     sec3      ;

3.26         movfw    TouchTime ; adds TouchTime and Skin together,storing the
              addwf   Skin,w    ; result in the GPF,Random
              movwf   Random    ;
              movfw   Chooser   ; use the mode to jump to the
              addwf   PCL      ; correct section
              b       Ran6
              b       RanLot
              b       Ran12
              b       Ran99
              b       Ran66

3.27         Ran6     movlw d'6'
              b       Adder
              Ran12   movlw d'12'
              b       Adder
              Ran99   movlw d'99'
              b       Adder

3.28
Adder        movwf   Scaler     ; stores the number from Ran6,Ran12,or Ran99
              movwf   Scaler     ; repeatedly adds the number
              addwf   Random    ; until it passes 255
              btfss  STATUS,C   ; is the value suitable?
              b       Adder+1   ; no,so loops back
              incf   Random    ; adds one to Random

3.29
Ran66       movfw    Random    ;
              movfw   Tens      ; stores as first 1-6 value
              addwf   TouchTime ; gets new random number
              movwf   Random    ;
              movlw   d'6'      ; converts it into 1-6
              addwf   Random    ;
              btfss  STATUS,C   ;
              b       Ran66     ;
              incf   Random,w   ;
              b       Continue66 ; jumps to correct place in TensLoop

3.30
RanLot      movlw    d'49'      ; repeatedly adds 49 until it
              addwf   Random    ; goes past 255

```

	btfs	STATUS, C	; is number suitable?
	b	RanLot	; no,so loops back
	incf	Random	; adds on,so it is between 1 and 49
3.31 <i>Challenge!</i>			
CompareLott	movlw	1Bh	; selects Lott1 first
	movwf	FSR	
	movwf	INDF	; compares with previous
	subwf	Random, w	; lottery values
	btfs	STATUS, Z	;
	b	Changer	; the two are the same,so gets new number
3.32			
Changer	movfw	Skin	; gets new random number
	addwf	Random	;
	b	RanLot	; converts it to a number between 1 and 49
3.33			
	incf	FSR	; moves on to next lottery number
	btfs	FSR, 5	; has it gone too far?
	b	CompareLott+2	; no,so loops back etc.
			; yes, so continues
3.34			
	movfw	LottCount	; use number from LottCount,
	addlw	1B	; and add 1B to it in order to select the correct GPF
	movwf	FSR	;
	movfw	Random	;
	movwf	INDF	;
	incf	LottCount	; moves on to next lottery number
3.35			
TensLoop	incf	Tens	; works the tens digit of the number
	movlw	d'10'	; (keeps subtracting 10 until result is negative)
	subwf	Random	;
	btfs	STATUS,C	;
	b	TensLoop	;
3.36			
	movlw	d'10'	; gets units value by adding 10
	addwf	Random,w	; to Random and leaving the result in the working register.
Continue66	call	Decoder	; converts units value into 7 seg.code
	movwf	dig4	; moves code into appropriate digit
	clrf	dig3	; blanks out left three digits
	clrf	dig2	;
	clrf	dig1	;
Decoder	addwf	PCL	; converts number into 7 seg. code
	retlw	b'11111100'	; number 0
	retlw	b'01100000'	; number 1
	retlw	b'11011010'	; number 2
	retlw	b'11110010'	; etc.
	retlw	b'01100110'	;
	retlw	b'10110110'	;
	retlw	b'10111110'	;

```

retlw    b'11100000' ;
retlw    b'11111110' ;
retlw    b'11100110' ;

3.37     movfw    Tens
          btfsz   STATUS,Z ; checks to see if tens is zero
          b       LottCounter ; yes,so displays nothing on dig3
          call    Decoder ; converts result into 7 seg. code
          btfsz   Chooser,2 ; is the device in -6-6 mode?
          b       Not66 ; goes to a place where the tens is shown on dig3
          movwf   dig1 ; in -6-6 mode,so moves code into dig1
          b       LottCounter ; now goes to label the lottery number correctly
Not66    movwf   dig3 ; moves code into dig 3 LottCounter etc.

3.38     LottCounter
          clrf    Tens ; resets Tens register
          decfsz  Chooser,w ; is the device in lottery mode?
          b       Main ; no,so loops back to main
          etc. ; yes,so continues

3.39     movlw    b'00000010' ; displays a dash on digit 2
          movwf   dig2 ;
          movfw   LottCount ; takes the number out of
          call    Decoder ; LottCount and uses it in dig1
          movwf   dig1 ;

3.40     display
          movfw   TMR0 ; use TMR0 to select one of
          andlw   b'00000011' ; four sections
          addwf   PCL ;
          b       digit1
          b       digit2
          b       digit3
          b       digit4

digit1    movlw    b'10000' ; turns on relevant display
          movwf   porta ;
          movfw   dig1
          movwf   portb ; moves relevant code into port
          retfie ; b

digit2    movlw    b'00010' ; turns on relevant display
          movwf   porta ;
          movfw   dig2
          movwf   portb ; moves relevant code into port
          retfie ; b

digit3    movlw    b'00100' ; turns on relevant display
          movwf   porta ;
          movfw   dig3
          movwf   portb ; moves relevant code into port
          retfie ; b

digit4    movlw    b'01000' ; turns on relevant display

```

```

movwf porta ;
movfw dig4

movwf portb ; moves relevant code into port
retfie ; b

```

3.41

```

Chaser    movfw Mark240 ; has a tenth of a second
          subfw TMR0,w ; passed?
          btfss STATUS,Z ;
          return ; no,so returns
          movlw d'240' ; yes,so continues and resets
          addwf Mark240 ; marker

```

3.42

	First	Second	Third
dig1:	00000100	10010000	00001000
dig2:	00010000	00000000	10000000
dig3:	10000000	00000000	00010000
dig4:	00100000	10010000	01000000

3.43

```

incf ChaseCount ; scrolls through the 3 different displays
movfw ChaseCount ; codes,creating a chase
addwf PCL
b Chase1
b Chase2
b Chase3

```

Chase1

```

movlw b'00000100' ; sets up first chase pattern
movwf dig1 ;
movlw b'00010000'
movwf dig2
movlw b'10000000'
movwf dig3
movlw b'00100000'
movwf dig4
return

```

Chase2

```

movlw b'10010000' ; sets up second chase
movwf dig1 ; pattern
movlw b'00000000'
movwf dig2
movlw b'00000000'
movwf dig3
movlw b'10010000'
movwf dig4
return

```

Chase3

```

movlw b'00001000' ; sets up third chase
movwf dig1 ; pattern
movlw b'10000000'
movwf dig2
movlw b'00010000'
movwf dig3

```

```

movlw    b'01000000'
movwf    dig4
movlw    d'255'      ; resets ChaseCount GPF
movwf    ChaseCount ; register
return

```

3.44

```

MessageChooser  movlw    d'12'      ; is Skin between 11 and 12?
                 subwf    Skin,w    ;
                 btfss   STATUS, C  ;
                 b        sad      ; yes, so displays SAd

```

3.45

```

                 movlw    d'15'      ; is Skin between 13 and 15?
                 subwf    Skin,w    ;
                 btfss   STATUS, C  ;
                 b        bad      ; yes, so displays bAd
                 movlw    d'20'      ; is Skin between 16 and 20?
                 subwf    Skin,w    ;
                 btfss   STATUS, C  ;
                 b        cool     ; yes, so displays cool
                 movlw    d'25'      ; is Skin between 21 and 25?
                 subwf    Skin,w    ;
                 btfss   STATUS, C  ;
                 b        john     ; yes, so displays John
                 movlw    d'35'      ; is Skin between 26 and 35?
                 subwf    Skin,w    ;
                 btfss   STATUS, C  ;
                 b        hot      ; yes, so displays hot
                 movlw    d'50'      ; is Skin between 36 and 50?
                 subwf    Skin,w    ;
                 btfss   STATUS, C  ;
                 b        tops     ; yes, so displays to PS
                 b        ace      ; no, so displays ACE (above 50)

```

3.46

```

_A          retlw    b'11101110' ; letter A
_b          retlw    b'00111110' ; letter b
_C          retlw    b'10011100' ; letter C
_c          retlw    b'00011010' ; letter c
_d          retlw    b'01111010' ; letter d
_E          retlw    b'10011110' ; E
_F          retlw    b'10001110' ; F
_g          retlw    b'11110110' ; g
_H          retlw    b'01101110' ; H
_h          retlw    b'00101110' ; h
_I          retlw    b'00001100' ; I
_i          retlw    b'00001000' ; i
_j          retlw    b'01110000' ; j
_L          retlw    b'00011100' ; L
_n          retlw    b'00101010' ; n
_O          retlw    b'11111100' ; O
_o          retlw    b'00111010' ; o
_P          retlw    b'11001110' ; P
_q          retlw    b'11100110' ; q

```

```

_r      retlw    b'00001010' ; r
_S      retlw    b'10110110' ; S
_t      retlw    b'00001110' ; t
_U      retlw    b'01111100' ; U
_u      retlw    b'00111000' ; u
_y      retlw    b'01110110' ; y
blank   retlw    b'00000000' ; blank

```

3.47

```

sad      call     _S
          movwf  dig1
          call   _A
          movwf  dig2
          call   _d
          movwf  dig3
          call   blank
          movwf  dig4
          return

```

```

bad      call     _b
          movwf  dig1
          call   _A
          movwf  dig2
          call   _d
          movwf  dig3
          call   blank
          movwf  dig4
          return

```

```

cool     call     _C
          movwf  dig1
          call   _O
          movwf  dig2
          call   _O
          movwf  dig3
          call   _L
          movwf  dig4
          return

```

```

john     call     _j
          movwf  dig1
          call   _o
          movwf  dig2
          call   _h
          movwf  dig3
          call   _n
          movwf  dig4
          return

```

```

hot      call     _h
          movwf  dig1
          call   _o
          movwf  dig2
          call   _t

```

	movwf	dig3
	call	blank
	movwf	dig4
	return	
tops	call	_t
	movwf	dig1
	call	_o
	movwf	dig2
	call	_P
	movwf	dig3
	call	_S
	movwf	dig4
	return	
ace	call	_A
	movwf	dig1
	call	_C
	movwf	dig2
	call	_E
	movwf	dig3
	call	blank
	movwf	dig4
	return	