

# 2

## Basic operations with AT90S1200 and TINY12

---

The best way to learn is through example and by doing things yourself. For the rest of the book we will cover example projects, many of which will be largely written by you. For this to work most effectively, it helps if you actually try these programs, writing them out as you go along in Notepad, or whatever development environment you're using. If you don't have any special AVR software at the moment, you can still write the programs out in Notepad and test them later.

First of all, copy out the program template covered in the previous chapter, adjusting it as you see fit, and save it as **template.asm**. If you are using Notepad, make sure you select File Type as *Any File*. The **.asm** file extension refers to *assembly source*, i.e. that which will be assembled.

### Program A: LEDon

- Controlling outputs

Our first few programs will use the 1200 chip. Load up the template, **Save As** to keep the original template unchanged, and call the file **ledon.asm**. Make the appropriate adjustments to the headers etc. relevant to the 1200 chip (header, **.device**, and **.include**). This first program is simply going to turn on an LED (and keep it on). The first step is to assign inputs and outputs. For this project we will need only one output, and will connect it to RB0. The second step in the design is the flowchart. This is shown in Figure 2.1. From this we can now write our program. The first box (Set-up) is performed in the **Init** routine. You should be able to complete this section yourself (remember, if a pin is not connected, make it an output).

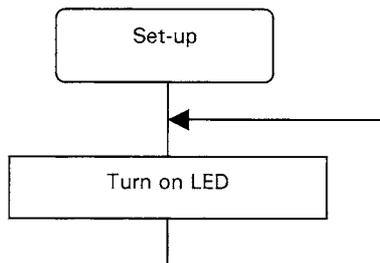


Figure 2.1

The second box involves turning on the LED, which means making RB0 high, which means setting bit 0 on PORTB to 1. To do this we could move a number into **temp**, and then move that number into **PortB**; however, there is a shortcut. We can use the following instruction:

```
sbi    ioreg, bit    ;
```

This sets a bit in an I/O register. Although you cannot move a number directly into an I/O register, you can set and clear the bits in *some* of them individually. You *cannot* set and clear individual bits in I/O registers 32–63 (\$20–\$3F in hex). Fortunately, PortB (\$18) and indeed all the PORTx and PINx registers can be controlled in this fashion. The equivalent instruction for clearing the bit is:

```
cbi    ioreg, bit    ;
```

This clears a bit in an I/O register, though remember this only works for I/O registers 0–31. For our particular application, we will want to set **PortB, 0** and so will use the following instruction at the point labelled **Start**:

```
Start: sbi    PortB, 0    ; turns on the LED
```

The next line is:

```
rjmp   Start    ; loops back to Start
```

This means the chip will be in an indefinite loop, turning on the LED. The program is now ready to be assembled. You can check that you've done everything right by looking at the complete program in Appendix J under *Program A*. All subsequent programs will be printed in the back in the same way. We will now assemble the program, but if you do not have the relevant software just read through the next section. You can download AVR Studio from Atmel's website ([www.atmel.com](http://www.atmel.com)) for free (last time I checked). This assembles, simulates and (with the right hardware) allows you to program the AVR chip.

### *AVR Studio – assembling*

First of all load AVR Studio. Select **Project** → **New Project** and give it a name (e.g. LEDon), pick a suitable location, and choose AVR Assembler in the bottom box. In your project you can have assembly files, and other files. The program you have just written is an assembly file (.asm) and so you will have to add it to the project. Right click on **Assembly Files** in the Project Window and choose **Add File**. Find your original saved LEDon.asm and select it. You should now see your file in the Project Window. Now press **F7** or go to **Project** → **Assemble** and your file will be assembled. Hopefully your file should

assemble with no errors. If errors are produced, you will find it helpful to examine the *List File* (\*.lst). Load this up in Notepad, or some other text editor and scan the document for errors. In this simple program, it is probably nothing more than a spelling mistake. Correct any problems and then move on to testing.

## Testing

There are three main ways to test your program:

1. Simulating
2. Emulating
3. Programming an actual AVR and putting it in a circuit

The first of these, *simulating*, is entirely software based. A piece of software pretends it's an AVR and shows you how it thinks the program would run, showing you how the registers are changing etc. You can also pretend to give it inputs by manually changing the numbers in PINB etc. You can get a good idea of whether or not the key concepts behind your program will work with this kind of testing, but other real-world factors such as button-bounce cannot be tested. Atmel's AVR Simulator comes with AVR Studio.

## AVR Studio – simulating

We will now have a go at simulating the LEDon program. After you assemble your .asm file, double click on it in the Project Window to open it. Some of the buttons at the top of the screen should now become active. There are three key buttons involved in stepping through your program. The most useful one of these, , is called Trace Into or Step Into. This runs the current line of your program. Pressing this once will begin the simulation and should highlight the first line of your program (**rjmp Init**). You can use this button (or its shortcut **F11**) to step through your program. We will see the importance of the other stepping buttons when we look at subroutines later on in the book. In order for this simulation to tell us anything useful, we need to look at how the I/O registers are changing (in particular bit 0 of PortB). This can be done by going to **View** → **New IO View**. You can see that the I/O registers have been grouped into categories. Expand the PortB category and this shows you the PortB, DDRB and PinB registers. You can also view the working registers by going to **View** → **Registers**. We will be watching R16 in particular, as this is **temp**. Another useful shortcut is the reset button,  (**Shift + F5**).

Continue stepping through your program. Notice how **temp** gets cleared to 00, PortB and PortD are also cleared to 00, then **temp** is loaded with 0xFF (0b1111111), which is then loaded in DDRB and DDRD. Then (crucially) PortB, bit 0 is set, as shown by the tick in the appropriate box. You may notice

how this will automatically set PinB, bit 0 as well. Remember the difference between PortB and PinB – PortB is a register representing what you wish to output through the port, and PinB represents the actual, physical state of those pins. For example, you could try to make an input high when the pin is accidentally shorted to ground – PortB would have that bit high whilst PinB would show the bit low, as the pin was being pulled low.

## Emulating

Emulating can be far more helpful in pinning down bugs, and gives you a much more visual indication of the working of the program. This allows you to connect a probe with an end that looks like an AVR chip to your computer. The emulator software then makes the probe behave exactly like an AVR chip running your program. Putting this probe into your circuit should give you the same result as putting a real AVR in, the great difference being that you can step through the program slowly, and see the inner workings (registers etc.) changing. In this way you are testing the program and the circuit board, and the way they work together. Unfortunately, emulators can be expensive – a sample emulator is Atmel's ICE (In-Circuit Emulator).

If you don't have an emulator, or after you've finished emulating, you will have to program a real AVR chip and put it in your circuit or testing board. One of the great benefits of AVR's is the Flash memory which allows you to keep reprogramming the same chip, so you can quite happily program your AVR, see if it works, make some program adjustments, and then program it again with the new, improved code.

For these latter two testing methods you obviously need some sort of circuit or development board. If you are making your own circuit, you will need to ensure certain pins on the chip are wired up correctly. We will now examine how this is done.

## Hardware

Figure 2.2 shows the 1200 chip. You will already be familiar with the PBx and PDx pins; however, there are other pins with specific functions. VCC is the positive supply pin, and in the case of the 1200 chip needs between 2.7 and 6.0 V. The allowed voltage range depends on the chip, but a value between 4 and 5 V is generally safe. GND is the ground (0 V) pin. There is also a  $\overline{\text{Reset}}$  pin. The bar over the top means that it is *active low*, in other words to make the AVR reset you need to make this pin *low* (for at least 50 ns). Therefore, if we wanted a reset button, we could use an arrangement similar to that shown in Figure 2.3.

The power supply to the circuit is likely to take a short time to stabilize once first turned on, and crystal oscillators need a 'warm-up' time before they assume regular oscillations, and so it is necessary to make the AVR wait a short while after the power is turned on before running the program. Fortunately, this

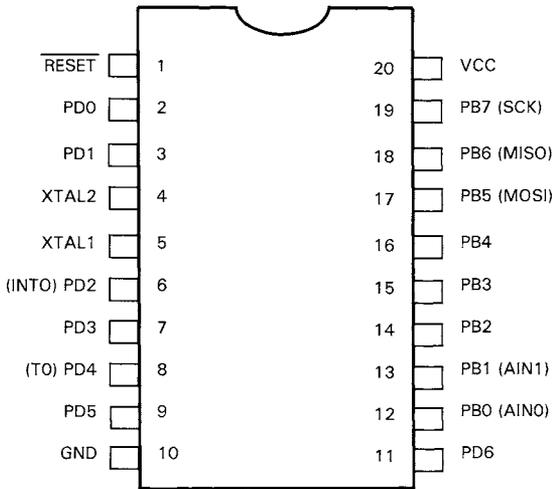


Figure 2.2

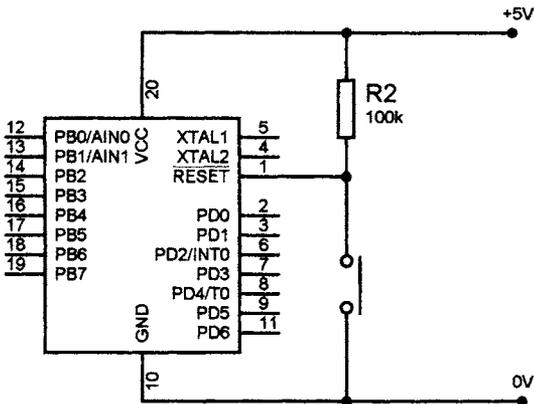


Figure 2.3

little delay is built into the AVR (lasting about 11 ms); however, if you have a particularly bad power supply or oscillator, and want to extend the length of this ‘groggy morning feeling’ delay you can do so with a circuit such as that shown in Figure 2.4. Increase the value of C1 to increase the delay.

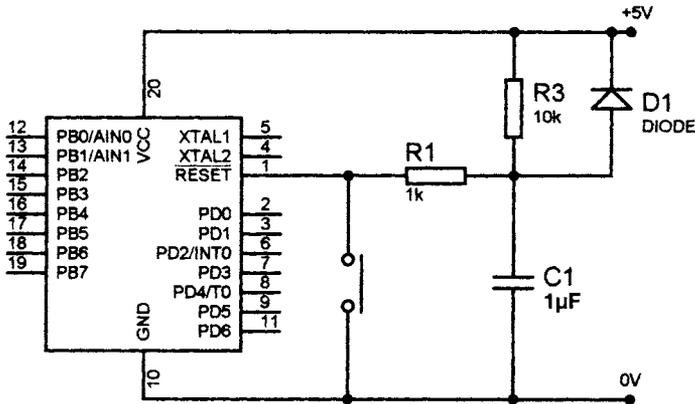


Figure 2.4

Finally, pins XTAL1 and XTAL2, as their names suggest, are wired to a crystal (or ceramic oscillator) which is going to provide the AVR with the steady pulse it needs in order to know when to move on to the next instruction. The faster the crystal, the faster the AVR will run through the program, though there are maximum frequencies for different models. This maximum is generally between 4 and 8 MHz, though the 1200 we are using in this chapter can run at speeds up to 12 MHz! Note that on some AVR's (in particular the Tiny AVR's and the 1200), there is a built-in oscillator of 1 MHz, which means you don't need a crystal. This internal oscillator is based on a resistor–capacitor arrangement, and is therefore less accurate and more susceptible to temperature variations etc.; however, if timing accuracy isn't an issue, it is handy to free up space on the circuit board and just use the internal oscillator. Figure 2.5 shows how you would wire up a crystal (or ceramic oscillator) to the two XTAL pins.

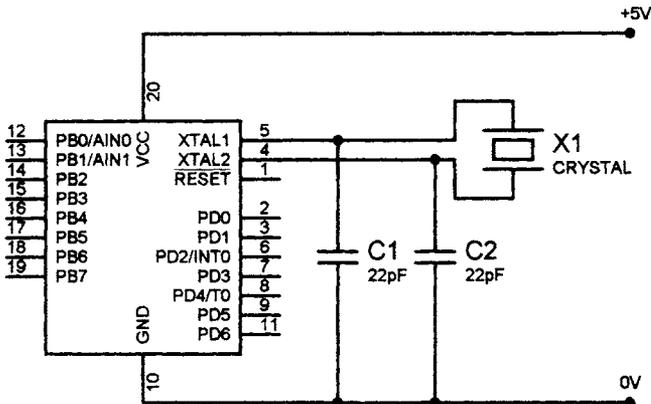


Figure 2.5

If you would like to synchronize your AVR with another device, or already have a clock line with high-speed oscillations on it, you may want to simply feed the AVR with an external oscillator signal. To do this, connect the oscillator signal to XTAL1, and leave XTAL2 unconnected. Figure 2.6 shows how using an HC (high-speed CMOS) buffer you can synchronize two AVR chips.

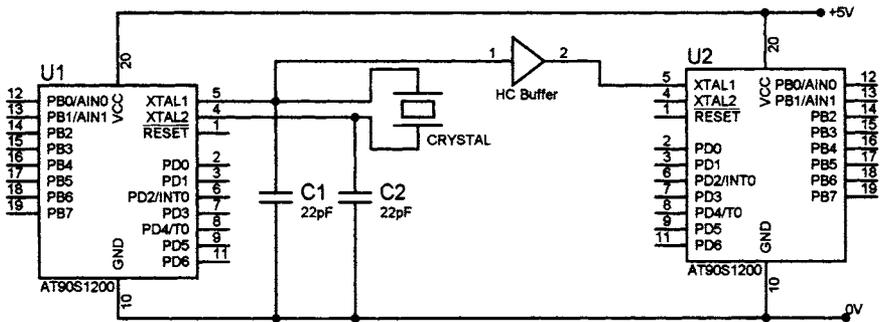


Figure 2.6

### AVR Studio – programming

In order to test a programmed AVR, you will need a circuit board or development board. The simplest solution is to make up the circuit boards as you need them, but you may find it quicker to construct your own development board to cover a number of the projects covered in this book. The required circuit diagram for the LEDon program is shown in Figure 2.7.

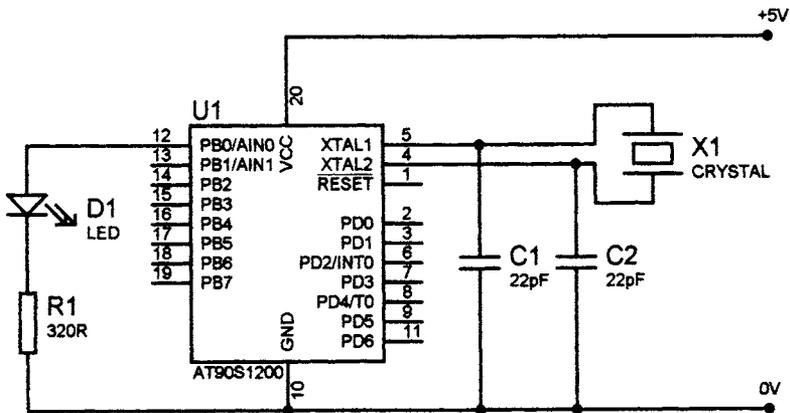


Figure 2.7

If you have a development board, you may need to check how the LEDs are wired up. We have been assuming the pins will *source* the LED's current (i.e. turn the pin high to turn on the LED). If your circuit board is configured such that the pin is *sinking* the LED's current, you will have to make changes to the software. In this case, a *0* will *turn on* the LED and a *1* will *turn off* the LED. Therefore, instead of starting with all of PortB set to 0 at the start of the Init section, you will want to move 0b11111111 into PortB (to turn off all the LEDs). You will also have to clear PortB, bit 0 rather than set it, in order to turn on the LED. This can be done using the **cbi** instruction in place of **sbi**.

Also note that although the program has been written with the 1200 in mind, by choosing the simplest model AVR we have made the program compatible with all other models (assuming they have sufficient I/O pins). Therefore if you have an 8515 (which comes with some development kits), simply change the **.device** and **.include** lines in your program and it should work.

We will now program the device using the STK500 Starter Kit. The steps required with the other types of programmer should not vary too much from these. To program your device, place the chip into the appropriate socket in the programming board. You may need to change the jumper cables to select the correct chip. In AVR Studio select **Tools** → **STK500**, and choose the relevant device (at90s1200). You will be programming the Flash Program memory. If you've just been simulating and your program is still in the simulator memory, you can tick the box labelled **Use Current Simulator/Emulator Flash Memory**, and then hit **Program**. If the program isn't in the Simulator/Emulator Memory, just load the program, assemble it, start the simulator, and it will be.

### *Fuse bits*

You may notice some other tabs in the programming window. The one labelled fuses enables you to control some of the hardware characteristics of the AVR. These fuses vary between different models. For the 1200 we have two fuses available. **RCEN** should be set if you are using the internal RC oscillator as your clock. If you are using an external clock such as a crystal (as indeed we are in this project), this fuse bit should be clear. The other fuse is **SPIEN**, *Serial Program Downloading*, which allows you to read the program back off the chip. If you want to keep your program to yourself and don't want others to be able to read it off the chip, make sure this fuse bit is clear.

All this just to see an LED turn on may seem a bit of an anticlimax, but there are greater things to come!

## **Programs B and C: push button**

- Testing inputs
- Controlling outputs

We will now examine how to test inputs and use this to control an output. Again, the project will be quite simple – a push button and an LED which turns on when the button is pressed, and turns off when it is released. There are two main ways in which we can test an input:

1. Test a particular bit in PINx using the **sbi** or **sbis** instructions
2. Read the entire number from PINx into a register using the **in** instruction

The push button will be connected between PD0 and 0V, and the LED to PB0. The flowchart is shown in Figure 1.3, and the circuit diagram in Figure 2.8.

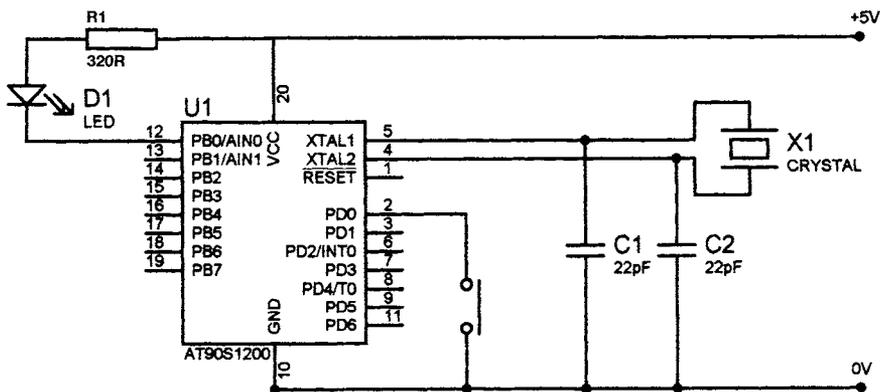


Figure 2.8

You should be able to write the **Init** section yourself, noting that as there is no external pull-up resistor shown in the circuit diagram, we need to enable the internal pull-up for PD0. The beginning of the program will look at testing to see if the push button has been pressed. We have two instructions at our disposal:

```
sbi    ioreg, bit    ;
```

This tests a bit in a I/O register and **skips** the following line if the **bit** is **clear**. Similarly

```
sbis  ioreg, bit    ;
```

tests a bit in a I/O register and **skips** the following line if the **bit** is **set**. Note that like **sbi** and **cbi**, these two instructions operate only on I/O registers numbered between 0 and 31 (\$0-\$1F). Fortunately, PIND, the register we will be testing,

is one of these registers (number \$10). So to test our push button (which makes pin PD0 high when it is pressed), we write:

```
sbis    PinD, 0          ; tests the push button
```

This instruction will make the AVR skip the next instruction if PD0 is high. Therefore the line below this one is only executed if the button is *not* pressed. This line should then turn off the LED, and so we will make the AVR *jump* to a section labelled **LEDOff**:

```
rjmp   LEDOff          ; jumps to the section labelled LEDOff
```

After this line is an instruction which is executed only when the button is pressed. This line should therefore turn the LED on, and we can use the same instruction as last time.

**EXERCISE 2.1** Write the *two* instructions which turn the LED on, and then loop back to **Start** to test the button again.

This leaves us with the section labelled LEDOff.

**EXERCISE 2.2** Write the *two* instructions which turn the LED off, and then loop back to **Start**.

You have now finished writing the program, and can double check you have everything correct by looking at Program B in Appendix J. You can then go through the steps given for testing and programming Program A. While you are doing your simulation, you can simulate the button being pressed by simply checking the box for **PIND, bit 0** in the I/O registers window.

Sometimes it helps to step back from the problem and look at it in a different light. Instead of looking at the button and LED as separate bits in the two ports, let's look at them with respect to how they affect the entire number in the ports. When the push button is pressed, the number in PinD is **0b00000000**, and in this case we want the LED to turn on (i.e. make the number in PortB **0b00000000**). When the push button isn't pressed, PinD is **0b00000001** and thus we want PortB to be **0b00000001**. So instead of testing using the individual bits we are going to use the entire number held in the file register. The entire program merely involves moving the number that is in PinD into PortB. This cannot be done directly, and so we will first have to read the number out of PinD using the following instruction:

```
in     register, ioreg  ;
```

This copies the number from an I/O register into a working register. To move

the number from a working register back out to an I/O register, we use the **out** instruction. The entire program can therefore consist of:

```

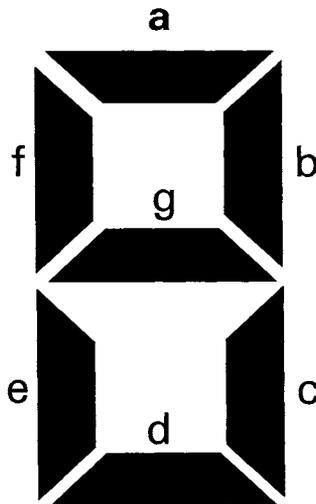
Start:      in      temp, PinD      ; reads button
              out      PortB, temp    ; controls LED
              rjmp     Start         ; loops back
  
```

This shorter program is shown as *Program C*.

### *Seven segment displays and indirect addressing*

Using an AVR to control seven segment displays rather than using a separate decoder chip allows you to display whatever you want on them. Obviously all the numbers can be displayed, but also most letters: A, b, c, C, d, E, F, G, h, H, i, l, J, l, L, n, o, O, P, r, S, t, u, U, y and Z.

The pins of the seven segment display should all be connected to the same port, in any order (this may make PCB design easier). The spare bit may be used for the dot on the display. Make a note of which segments (a, b, c etc.) are connected to which bits. The segments on a seven segment display are labelled as shown in Figure 2.9.



**Figure 2.9**

*Example 2.1* Port B Bit 7 = d, Bit 6 = a, Bit 5 = c, Bit 4 = g, Bit 3 = b, Bit 2 = f, and Bit 1 = e. I have assigned the letters to bits in a random order to illustrate it doesn't matter how you wire them up. Sometimes you will find that due to physical PCB restrictions there are some configurations that are easier or

more compact than others. The software is easy to change – the hardware normally less so.

If the display is wired up as described in Example 2.1, the number to be moved into Port B when something is to be displayed should be in the format **dacgbfe-** (it doesn't matter what bit 0 is as it isn't connected to the display), where the value associated with each letter corresponds to the required state of the pin going to that particular segment.

So if you are using a common cathode display (i.e. make the segments high for them to turn on – see Figure 2.10), and you want to display (for example) the letter A, you would turn on segments: a, b, c, e, f and g.

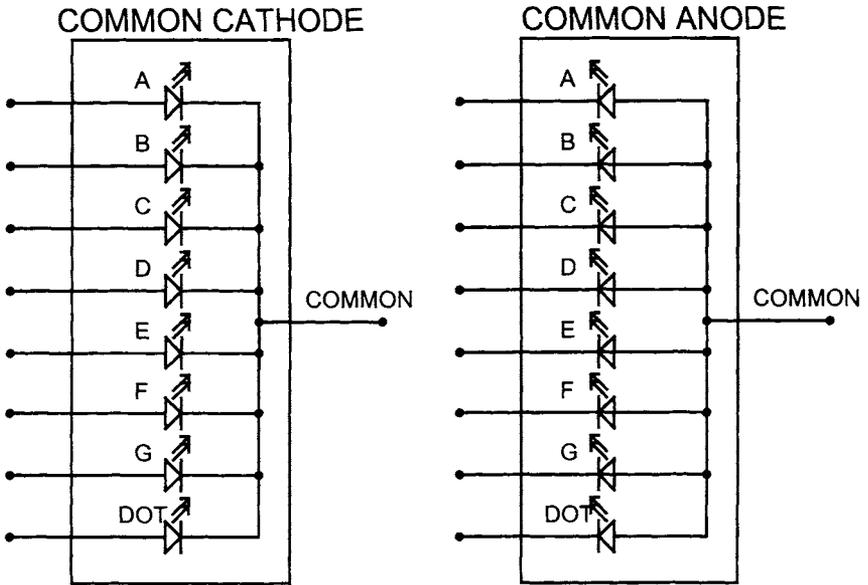


Figure 2.10

Given the situation in Example 2.1, where the segments are arranged **dacgbfe-** along Port B, the number to be moved into **PortB** to display an **A** would be **0b01111110**. Bit 0 has been made 0, as it is not connected to the display.

*Example 2.2* If the segments of a common cathode display are arranged **dacgbfe-** along Port B, what number should be moved into **PortB**, to display the letter C, and the letter E?

The letter **C** requires segments a, d, e and f, so the number to be moved into Port B would be **0b11000110**. The letter **E** requires segments a, d, e, f and g so the number to be moved into Port B would be **0b11010110**.

**EXERCISE 2.3** If the segments are arranged **abcdefg-** along Port B, what number should be moved into **PortB** to display the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, b, c, d, E and F.

The process of converting a number into a seven segment code can be carried out in various ways, but by far the simplest involves using a *look-up table*. The key idea behind a look-up table is *indirect addressing*. So far we have been dealing with *direct addressing*, i.e. if we want to read a number from register number 4, we simply read register number 4. Indirect addressing involves reading a number from register number X, where X is given in a different register, called **Z** (the 2-byte register spread over R30 and R31).

It's a bit like sending a letter, where the letter is the contents of a working register (R0–R31), and the address is given by the number in **Z**.

*Example 2.3* Move the number **00** into working registers numbers **R0** to **R29**.

Rather than writing:

```
clr    R0           ; clears R0
clr    R1           ; clears R1
clr    R2           ; clears R2
etc.
clr    R29          ; clears R29
```

we can use indirect addressing to complete the job in fewer lines. The first address we want to write to is **R0** (address 0), so we should move **00** into **Z** (making 0 the address on the letter). **Z**, remember, is spread over both **ZL** and **ZH** (the higher and lower bytes of **Z**), so we need to clear them both:

```
clr    ZL           ; clears ZL
clr    ZH           ; clears ZH
```

We then need to set up a register with the number 0 so we can send it 'by post' to the other registers. We already have a register with a 0 (**ZH**), so we will use that.

```
st     register, Z   ;
```

This indirectly stores (sends) the value in **register** to the address pointed to by **Z**. Therefore the instruction:

```
st    ZH, Z    ;
```

sends the number in **ZH** (0) to the address given by **Z** (also 0), and so effectively clears **R0**. We now want to clear **R1**, and so we simply increment **Z** to point to address 01 (i.e. R1). The program then loops back to cycle through all the registers, clearing them all in far fewer lines than if we were using direct addressing. All we need to do is test to see when **ZL** reaches 30, as this is past the highest address we wish to clear.

How do we tell when **ZL** reaches 30? We subtract 30 from it and see whether or not the result is zero. If **ZL** is 30, then when we subtract 30 from it the result will be 0. We don't want to *actually* subtract 30 from **ZL**, or it will start going backwards fast! Instead we use one of the compare instructions:

```
cp    register, register ;
```

This 'compares' the number in one register with that in another (actually subtracts one register from the other whilst leaving both unchanged). We then need to see if the result is zero. We can do this by looking at the *zero flag*. There are a number of flags held in the **SREG** register (\$3F), these are automatically set and cleared depending on the result of certain operations. The zero flag is set when the result of an operation is zero. There are two ways to test the zero flag:

```
brbs  label, bit    ;
```

This **branches** to another part of the program if a **bit** in **SREG** is **set** (the zero flag is bit 1, and so **bit** would have to be a 1). Note that the label has to be within 63 instructions of the original instruction. Similarly,

```
brbc  label, bit    ;
```

This **branches** to another part of the program if a **bit** in **SREG** is **clear**. Here is where some of the instruction redundancy comes in, because as well as this general instruction for testing a bit in **SREG**, each bit has *its own particular instruction*. In this case, for the zero flag:

```
breq  label    ;
```

which stands for **branch if equal** (more specifically, branch if the zero flag is set). The opposite of this is:

```
brne  label    ;
```

which stands for **branch if not equal** (more specifically, branch if the zero flag

is clear). The complete set of redundant/non-critical instructions is shown in Appendix C, along with their equivalent instructions. To compare a register with a number (rather than another register), we use the instruction:

```
cpi    register, number    ;
```

*Please note that this only works on registers R16–R31, but as ZL is R30 we are all right. The complete set of instructions to clear registers R0 to R29 is therefore:*

```
clr    ZL                ; clears ZL  
clr    ZH                ; clears ZH  
ClearLoop: st    ZH, Z        ; clears indirect address  
inc    ZL                ; moves on to next address  
cpi    ZL, 30            ; compares ZL with 30  
brne   ClearLoop        ; branches to ClearLoop if ZL ≠ 30
```

This six line instruction set is useful to put in the **Init** subroutine to systematically clear a large number of file registers. You can adjust the starting and finishing addresses by changing the initial value of ZL and the final value you are testing for; note, however, that you don't want to clear ZL in the loop (i.e. don't go past 30) because otherwise you will be stuck in an endless loop (think about it).

**EXERCISE 2.4** *Challenge!* What six lines will write a 0 to R0, a 1 to R1, a 2 to R2 etc. all the way to a 15 to R15?

As well as writing indirectly, we can also read indirectly:

```
ld     register, Z        ;
```

This indirectly **loads** into **register** the value at the address pointed to by **Z**. We therefore have a *table* of numbers kept in a set of consecutive memory addresses, and by varying **Z** we can read off different values. Say, for example, we keep the codes for the seven segment digits 0–9 in working registers R20–R29. We then move 20 into **Z** (to 'zero' it to point at the bottom of the *table*) and then add the number we wish to convert to **Z**. Reading indirectly into **temp** we then get the seven segment code for that number:

```
ldi    ZL, 20            ; zeros ZL to R20  
add    ZL, digit        ; adds digit to ZL  
ld     temp, Z          ; reads Rx into temp  
out    PortB, temp      ; outputs temp to Port B
```

The above code translates the number in **digit** into a seven segment code which

is then outputted through Port B. Note that you will have to write the code to the registers in the first place:

```
ldi    R20, 0b11111100    ; code for 0
ldi    R21, 0b01100000    ; code for 1
etc.
ldi    R29, 0b11110110    ; code for 9
```

Note that using working registers for this purpose is unusual and indeed wasteful, but as there is no other SRAM on the 1200 we have no choice. On other chips that do have SRAM, we can use that for look-up tables. Furthermore, on other chips there is also an instruction **lpm**, which allows you to use the Program Memory for look-up tables as well. More on this in the Logic Gate Simulator project on page 67.

## Programs D and E: counter

- Testing inputs
- Seven segment displays

Our next project will be a counter. It will count the number of times a push button is pressed, from 0 to 9. After 10 counts (when it passes 9), the counter should reset. The seven segment display will be connected to pins PB0 to PB6, and the push button will go to PD0. Figure 2.11 shows the circuit diagram, pay particular attention to how the outputs to the seven segment display are arranged. The flowchart is shown in Figure 2.12.

You can write the Init section yourself, remembering the pull-up on the push button. Start **PortB** with the code for a 0 on the display. We will be using a register called **Counter** to keep track of the counts, you should define this in the declarations section as **R17**. The reason we have assigned it R17 is that, as you may remember, registers R16–R31 are the ‘executive assistants’ – more powerful registers capable of a wider range of operations. We therefore tend to fill up registers from R16 upwards, and then use R0–R15 if we run out. In the Init section, set up registers **R20** to **R29** to hold the seven segment code for numbers 0 to 9. (HINT: If you do this before setting up **PortB**, you can move **R20** straight into **PortB** to initialize it. Also remember to clear **Counter** in the Init section.)

**EXERCISE 2.5** What *three* lines will test the push button, loop back and test it again if it isn’t pressed? If it is pressed it should jump out of the loop and add one to **Counter**?

Then we need to see whether **Counter** has exceeded 9. We use **cpi** to compare, and **brne** to skip if they are not equal. If they are equal, **Counter** must be reset

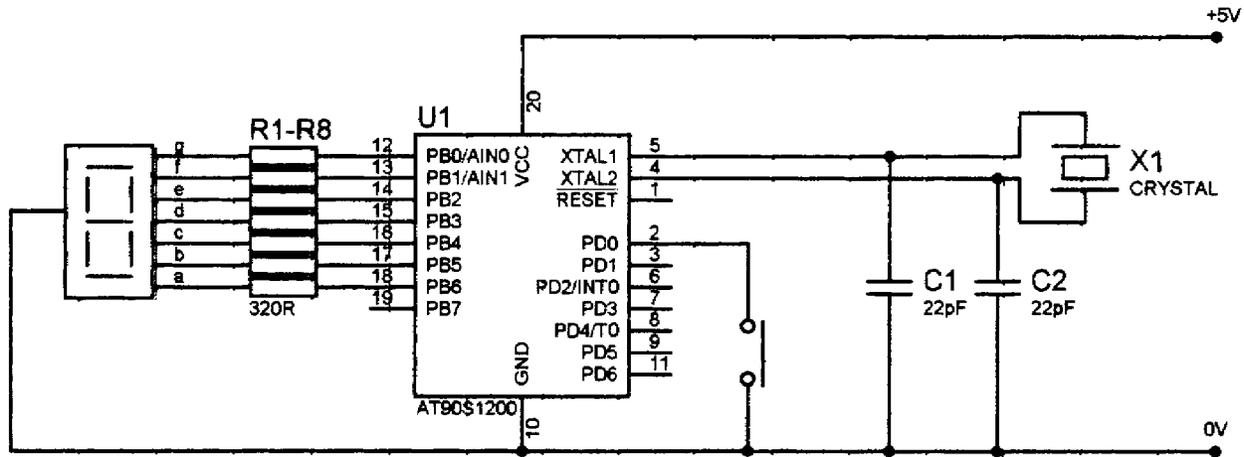
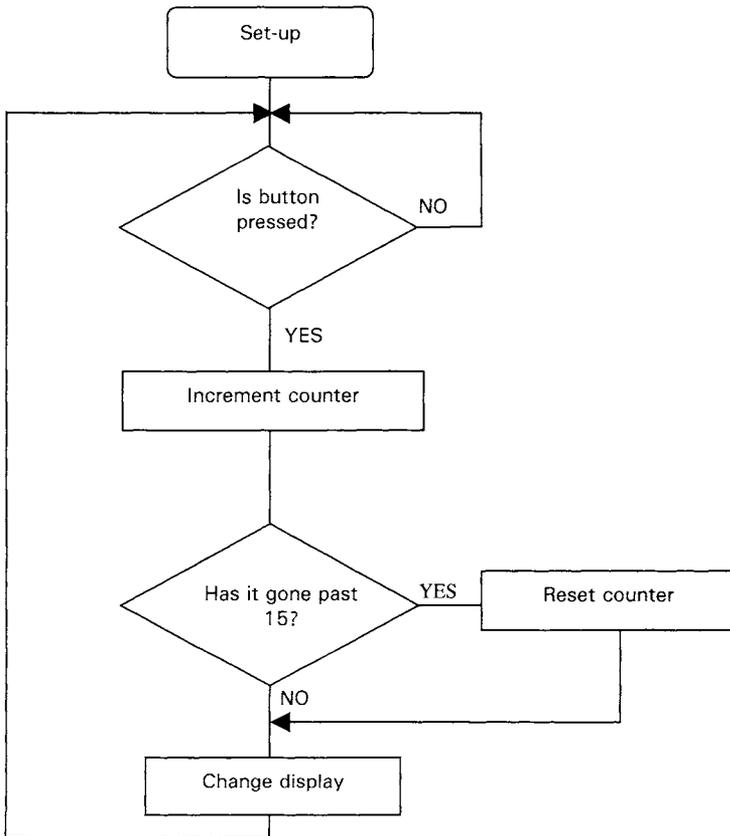


Figure 2.11



**Figure 2.12**

to 0. A useful trick with **brne** and similar instructions: it is often the case that rather than jumping somewhere exotic when the results aren't equal, we simply want to skip the next instruction (as we do with the **sbis** and **sbic** instructions). To do this with branch instructions, write **PC+2** instead of a label – this skips 1 instruction (i.e. jumps forward 2 instructions). PC stands for Program Counter which is described in more detail on page 54.

**EXERCISE 2.6** What *three* lines will test if **Counter** is equal to 10 and reset it if it is? You may want to use the PC+2 trick.

Now we need to display the value in **Counter**. Do this by setting ZL to point to R20 and adding **Counter** to it, as described already.

**EXERCISE 2.7** What *five* lines will display the value in **Counter** through Port B, and then loop back to **Start**?

The program so far is shown as Program D. It is recommended that you actually build this project. Try it out and you will spot the major flaw in the project.

The basic problem is that we are not waiting for the button to be released. This means that **Counter** is being incremented for the entire duration of the button being pressed. If we imagine that the button is held down for 0.1 s, and the crystal frequency is 4 MHz, one trip around the program takes about 14 clock cycles, and so **Counter** is incremented about  $4\,000\,000 / (14 \times 10) = 28\,600$  times for every press of the button! Effectively what we have is a pretty good random number generator (as an aside, random number generators are quite hard to make without some form of human input – computers are not good at being random). You could make this into an electronic dice project, but we will return to our original aim of a reliable counter.

Figure 2.13 shows the new flowchart. The necessary adjustment can be made at the end to wait for the button to be released before looping back to start.

**EXERCISE 2.8** Write the *two* new lines needed to solve the problem, and show where they are to be added. (HINT: you will need to give this loop a name.)

Try out this new program (Program E), and you may notice a lingering problem, depending on the quality of your push button. You should see that the counter counts up in jumps when the push button is pressed (e.g. jumping up from 1 to 4). This is due to a problem called *button bounce*. The contacts of a push button actually bounce together when the push button is pressed or released, as shown in Figure 2.14.

In order to avoid counting one press as many, we will have to introduce a short delay after the button has been released before testing again. This affects the minimum time between counts, but a compromise must be reached.

*Example 2.4* To avoid button bounce we could wait 5 seconds after the button has been released before we test it again. This would mean that if we pressed the button 3 seconds after having pressed it before, the signal wouldn't register. This would stop any bounce, but means the minimum time between signals is excessively large.

*Example 2.5* Alternatively, to attempt to stop button bounce we could wait a hundred thousandth of a second after the button release before testing it again. The button bounce might well last longer than a hundred thousandth of a second so this delay would be ineffective.

A suitable compromise might be around a tenth of a second but this will vary from one type of button to the next and you will have to experiment a little. In order to implement this technique, we will have to learn about timing, which brings us to the next section.

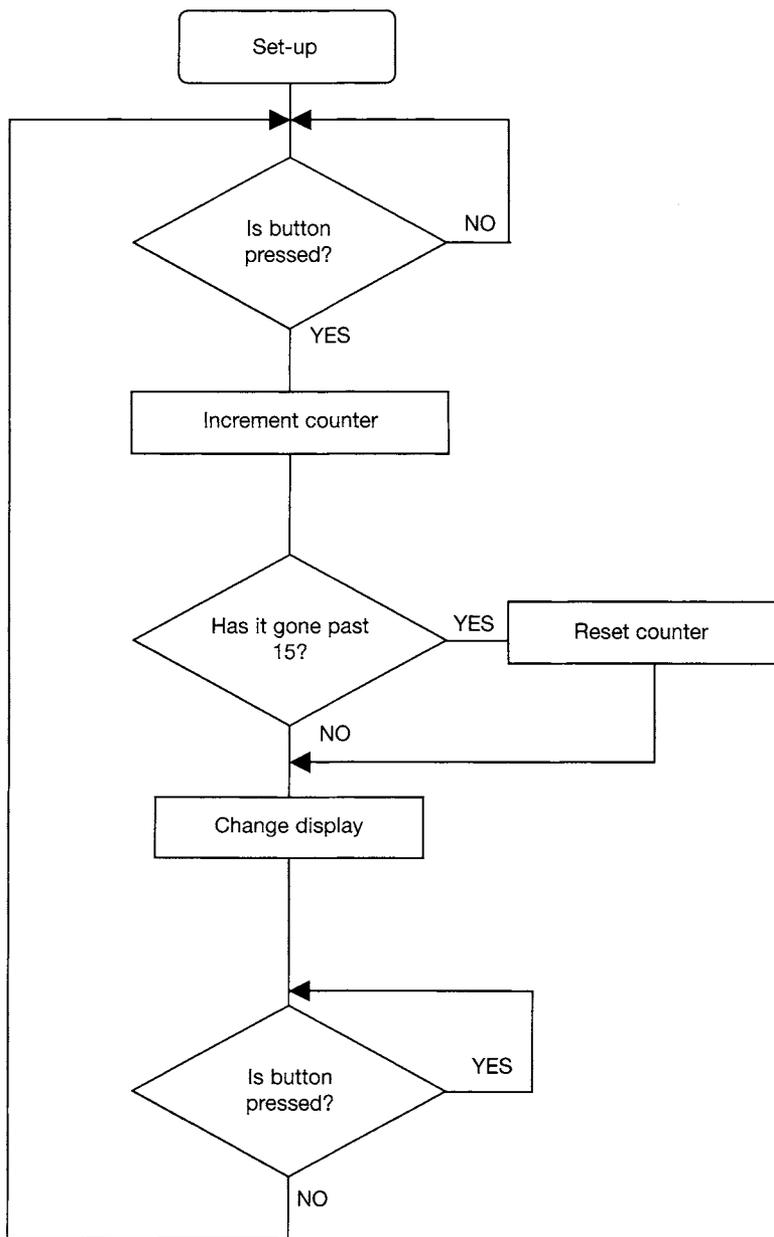


Figure 2.13

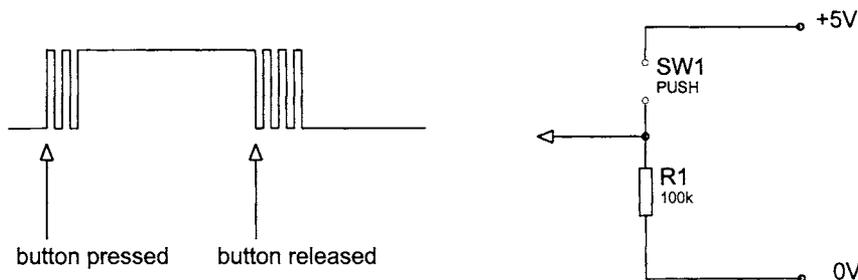


Figure 2.14

### Timing

If you cast your mind back to the list of I/O registers (it may help if you glance back at page 14), you will notice a register called **TCNT0** (\$32), or **Timer Counter 0**. This is an on-board timer, and will automatically count up at a specified rate, resetting to 0 when it passes 255. We can use this to perform timing functions (e.g. one second delays etc.). In more advanced chips there are several timers, some of which are 16 bits long. The reason it is also called a ‘Counter’ is that it can also be made to count the number of signals on a specific input pin (PD4 – pin 8 in the case of the 1200). For the purposes of the immediate discussion, we will be using **TCNT0** as a timer, and so I will be referring to it as **Timer 0**, or **T/C0** for the sake of brevity.

Before we can use Timer 0, we will have to configure it properly (e.g. tell it to time and not count). We do this with the **T/C0 Configuration Register: TCCR0** (\$33). In this register, each bit controls a certain aspect of the functioning of T/C0. In the case of the 1200, only bits 0–2 are used:

#### **TCCR0** – T/C0 Control Register (\$33)

bit no.	7	6	5	4	3	2	1	0
bit name	-	-	-	-	-	CS02	CS01	CS00

000	STOP! T/C0 is stopped
001	T/C0 counts at the clock speed (CK)
010	T/C0 counts at CK/8
011	T/C0 counts at CK/64
100	T/C0 counts at CK/256
101	T/C0 counts at CK/1024
110	T/C0 counts on falling edge of T0 pin
111	T/C0 counts on rising edge of T0 pin

Bits 3–7 have no purpose, but by setting bits 0–2 in a certain way, we can make T/C0 behave in the way we wish. If we don't wish to use T/C0 at all, all three bits should be 0. If we wish to use it as a timer, we select one of the next five options. Finally, if we want it to count external signals (on PD4), we can choose one of the last two options. The options available to us when using T/C0 for timing are to do with the speed at which it counts up. The clock speed (CK) is going to be very fast indeed (a few MHz) – this is the speed of the crystal which you connect to the AVR – and so in order to time lengths of the order of seconds we are going to have to slow things down considerably. The maximum factor by which we can slow down Timer 0 is 1024. Therefore if I connect a crystal with frequency 2.4576 MHz to the chip (this is actually a popular value crystal), Timer 0 will count up at a frequency of  $2\,457\,600/1024 = 2400$  Hz. So even if we slow it down by the maximum amount, Timer 0 is still counting up 2400 times a second.

*Example 2.6* What number should be moved into the TCCR0 register in order to be able to use the T/CO efficiently to eventually count the number of seconds which have passed?

Bits 3 to 7 are always 0.

Timer 0 is counting *internally*, at its slowest rate =  $CK/1024$

Hence the number to be moved into the TCCR0 register is **0b00000101**.

**EXERCISE 2.9** What number should be moved into the TCCR0 register when a button is connected between PD4 and +5 V, and TCNT0 is to count when the button is pressed.

In order to move a number into TCCR0, we have to load it into temp, and then use the **out** instruction, as with the other I/O registers. As you are unlikely to want to keep changing the Timer 0 settings it is a good idea to do this in the **Init** subroutine, to keep it out of the way.

In order to time seconds and minutes, you need to perform some further frequency dividing yourself. We do this with what I call a *marker* and then any number of *counter* registers. These are working registers we use to help us with the timing. The basic idea is to count the number of times the value in Timer 0 reaches a certain number. For example, in order to wait one second, we need to wait for Timer 0 to count up 2400 times. This is equivalent to waiting for Timer 0 to reach 80, for a total of 30 times, because  $30 \times 80 = 2400$ . We could do this with any other factors of 2400 that are both less than 256.

To test if the number in Timer 0 is 80, we use the following lines:

```

out    TCNT0, temp    ; copies TCNT0 to temp
cpi    temp, 80       ; compares temp with 80
breq   Equal         ; branches to Equal if temp = 80

```

This tests to see if Timer 0 is 80, and branches to Equal if it is. The problem is we're not always testing to see if Timer 0 is 80. The first time we are, but then next time round we're testing to see if Timer 0 is 160, and then 240 etc. We therefore have a register (which I call a marker) which we start off at 80, and then every time Timer 0 reaches the marker, we add another 80 to it. There isn't an instruction to add a number to a register, but there is one to subtract a number, and of course subtracting a negative number is the same as adding it.

```
subi    register, number    ;
```

This **subtracts** the immediate number from a register. *Note the register must be one of R16–R31.* So far, we have managed to work out when the Timer 0 advances by 80. We need this to happen 30 times for one second to pass. We take a register, move 30 into it to start with, and then subtract one from it every time Timer 0 reaches 80.

```
dec     register           ;
```

This **decrements** (subtracts one from) a register. When the register reaches 0 we know this has all happened 30 times. This all comes together below, showing the set of instructions required for a one second delay.

```
ldi    Count30, 30      ; starts up the counter with 30
```

```
ldi    Mark80, 80      ; starts up the marker with 80
```

```
TimeLoop: out    TCNT0, temp    ; reads Timer 0 into temp  
cp     temp, Mark80    ; compares temp with Mark80  
brne   TimeLoop      ; if not equal keeps looping
```

```
subi    Mark80, -80    ; adds 80 to Mark80
```

```
dec     Count30        ; subtracts one from Count30
```

```
brne   TimeLoop      ; if not zero keeps looping
```

The first two instructions load up the counter and marker registers with the correct values. Then TCNT0 is copied into temp, this is then compared with the marker. If they are not equal, the program keeps looping back to TimeLoop. If they are equal it then adds 80 to the marker, subtracts one from the counter, looping back to TimeLoop if it isn't zero. Note that you will have to define Mark80 and Count30 in the declarations section, and that they will have to be one of R16–R31.

## Program F: chaser

- Timing
- Reading inputs
- Controlling outputs

The next example project will be a ‘chaser’ which consists of a row of LEDs. The LEDs are turned on in turn to give a chasing pattern. The speed of this chase will be controlled by two buttons – one to speed it up, the other to slow it down. The default speed will be 0.5 second per LED, going down to 0.1 second and up to 1 second.

The LEDs will be connected to Port B, and the buttons to PD0 and PD1. The flowchart and circuit diagram are shown in Figures 2.15 and 2.16 respectively.

The set-up box of the flowchart should be fairly straightforward, though remember that you may want to configure TCCR0 in the Init section, and that as we are timing the order of a second, we will want to use TCNT0 as a timer, slowed down by its maximum. Note also that PD0 and PD1 will require pull-ups, and that PortB should be initialized with one LED on (say, for example, PB0).

It is now worth giving a little thought to how we are going to have a time delay which can vary between 0.1 second and 1 second. The shortest time delay, 0.1 second, can be timed using a marker of 240 ( $2400/240 = 10$  Hz), assuming the Timer 0 is counting at CK/1024 and a 2.4576 MHz crystal is being used. Then the counter can be varied between 1 and 10 to vary the overall time between 0.1 and 1 second. You may want to think about this a little. We will therefore have a marker register **Mark240**, and a variable counter register called **Counter**. **Counter** will be normally reset to 5 (for 0.5 second), but can be reset to other values given by **Speed**. Don’t forget to define these registers at the declarations section at the top of the program).

Looking back at our flowchart, the first box after the set-up looks at the ‘slow-down button’. We shall make the button at PD0 the ‘slow-down button’, and test this using the **sbic** instruction. If the button is not pressed (i.e. the pin is *high*), the next instruction will be executed, and this skips to a section where we test the ‘speed-up button’ button (call this **UpTest**).

If the button *is* pressed, we want to add one to **Speed** (slow down the chase). This can be done using the following instruction:

```
inc    register    ;
```

This **increments** (adds one to) a register. We don’t want the delay to grow longer than 1 second, and so we must check that **Speed** has not exceeded 10 (i.e. if it is 11 it has gone too far). We do this with the compare immediate instruction already introduced, **cpi**. If **Speed** is *not* equal to 11, we can then branch to **ReleaseDown** and wait for the button to be released. If it is equal to 11 we have

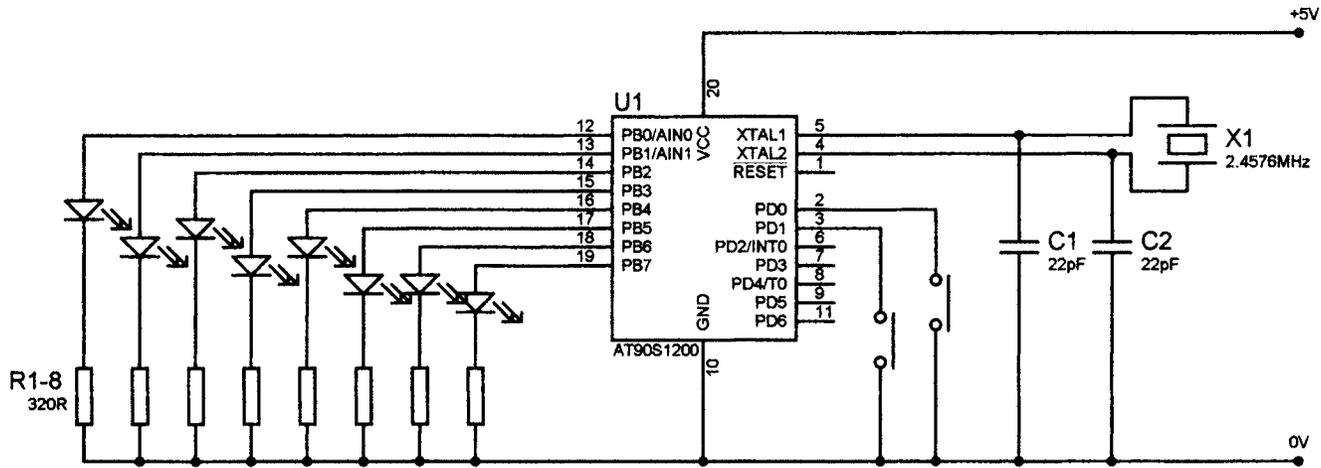


Figure 2.15

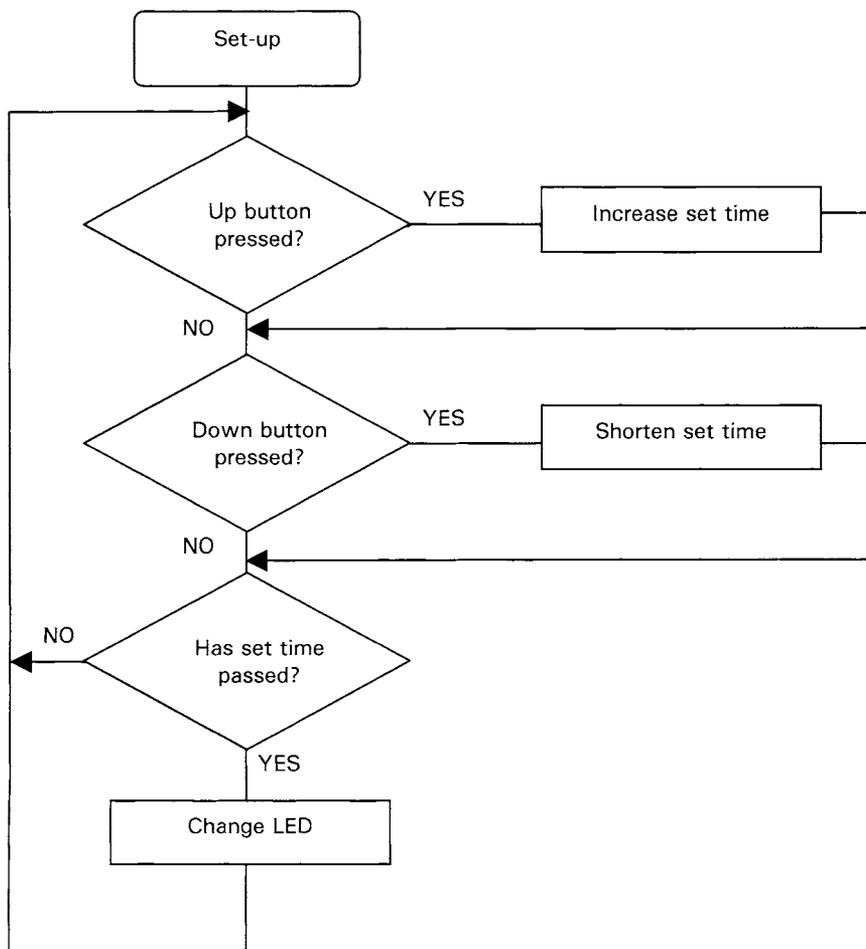


Figure 2.16

to subtract one from it (using the **dec** instruction). The first few lines of the program are therefore:

```

Start:      sbic   PinD, 0      ; checks slow-down button
            rjmp  UpTest    ; not pressed, jumps

            inc   Speed     ; slows down time
            cpi   Speed, 11 ; has Speed reached 11?
            brne  ReleaseDown ; jumps to ReleaseDown if not equal
            dec   Speed     ; subtracts one from Speed
  
```

**ReleaseDown:**

```

sbis   PinD, 0      ; waits for button to be released
rjmp  ReleaseDown: ;

```

In **UpTest**, we do the same with the ‘speed-up button’, PD1, and instead of jumping to **UpTest**, we jump to the next section which we will call **Timer**. If the speed-up button is pressed we need to decrement **Speed**, and instead of testing to see if it has reached 11, we test to see if it has reached 0 (and increment it if it has). We could use **cp** **Speed, 0**, but this line is unnecessary as the zero flag will be triggered by the result of the **dec** instruction, and so if we decrement **Speed** and the result is zero, we can use the **brne** in the same way as before.

EXERCISE 2.10 Write the *seven* lines which follow those given above.

The next section, called **Timer**, has to *check* to see if the set time has passed, and *return to the beginning if the time hasn't passed*. This means the timing routine must loop back to **Start** rather than stay in its own loop.

We will also put in the lines which set up the marker and counter registers in the **Init** section. **Mark240** should initially be loaded with 240; **Speed** and **Counter** should be loaded with 5. This means we can go straight into the counting loop.

```

Timer:   in      temp, TCNT0   ; reads Timer 0 into temp
         cp      temp, Mark240 ; compares temp with Mark240
         brne   Start         ; if not equal loops back to Start

         subi   Mark240, -240 ; adds 240 to Mark240

         dec    Counter       ; subtracts one from Counter
         brne   Start         ; if not zero loops back to Start

```

This should be familiar from the last section on timing. Note that instead of looping back to **Timer**, it loops back to **Start**. You may find, however, that you can reduce button bounce by looping back to **Timer** rather than **Start** in the 0.1 second loop. This means the buttons will only be tested once every 0.1 second, which means that a button will have to be pressed for at least 0.1 second. After the total time has passed, we need to chase the LEDs (i.e. rotate the pattern), and also reset the **Counter** register with the value in **Speed**. To do this we use:

```

mov    reg1, reg2 ;

```

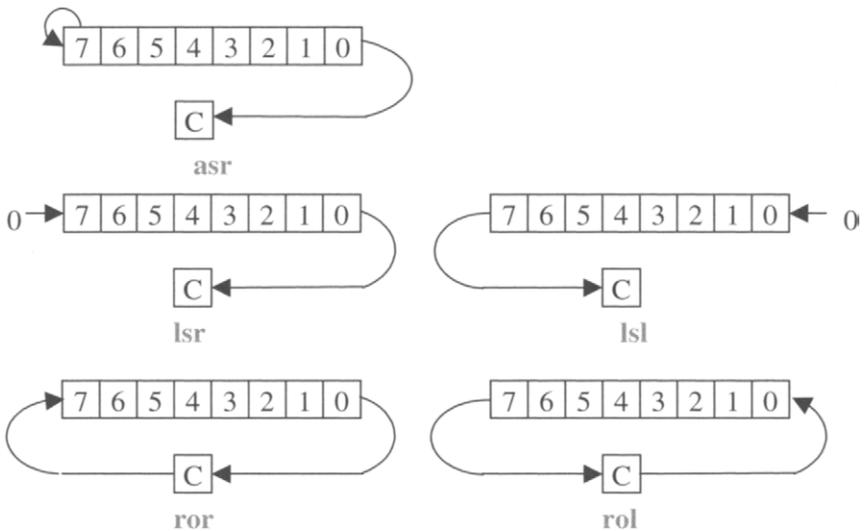
This **moves** (copies) the number from reg2 into reg1.

EXERCISE 2.11 What *one* line resets **Counter** with the value in **Speed**?

To rotate the pattern of LEDs we have a number of rotating instructions at our disposal:

<b>asr</b>	<b>register</b>	<b>; arithmetic shift right</b>
<b>lsr</b>	<b>register</b>	<b>; logical shift right</b>
<b>lsl</b>	<b>register</b>	<b>; logical shift left</b>
<b>ror</b>	<b>register</b>	<b>; rotate right</b>
<b>rol</b>	<b>register</b>	<b>; rotate left</b>

The *arithmetic* shift right involves shifting all the bits to the right, whilst keeping bit 7 the same and pushing bit 0 into the *carry flag*. The carry flag is a flag in **SREG** like the zero flag. The *logical* shift right shifts all the bits to the right, and moves 0 into bit 7. The *rotate* right rotates through the carry flag (i.e. bit 7 is loaded with the carry flag, and bit 0 is loaded into the carry flag). This is summarized in Figure 2.17.



**Figure 2.17**

As we rotate the pattern along, we don't want any 1s appearing at the ends, because this would turn on edge LEDs out of turn, which would then propagate down the row and ruin the pattern. It would therefore seem that **lsl** or **lsr** is appropriate. For the sake of argument, we will pick **lsl**, to rotate the pattern to the left. We cannot apply these rotating instructions directly to **PortB**, so we have to read in the pattern to **temp**, rotate **temp**, and then output back to **PortB**. Before we output it to **PortB**, we have to see whether or not we've gone too far

(rotated eight times), in which case we need to reset PortB back to its initial value (all off except PB0). We can do this by monitoring the carry flag, which will be high if we rotate a high bit off the end (a quick glance at Figure 2.17 should confirm this). The instruction for this is:

```
brcc label ;
```

This **branches to label** if the carry flag is clear. Therefore the lines we need are:

```
in temp, PortB ; reads in current state  
lsl temp ; rotates to the left  
brcc PC+2 ; checks Carry, skip if clear  
ldi temp, 0b0000001 ; resets to PB0 on, others off  
  
out PortB, temp ; outputs to PortB  
rjmp Start ; loops back to Start
```

You will notice that if the carry flag is clear, we skip the next instruction using the **PC+2** trick. The program is shown in its entirety as Program F in Appendix J.

You can go through and assemble this, and simulate it. For the simulation, you will notice that stepping through the entire program waiting for Timer 0 to count up will take a long time. For this reason, ways to run through parts of the program at high speed are on offer. For example, if you right click on a line in the program (when in simulation mode), you are given the option to ‘Run to Cursor’ (**Ctrl + F10**). This will run to where you have clicked at high speed (not quite real time, but close).

So far we have covered quite a few instructions; it is important to keep track of all of them, so you have them at your fingertips. Even if you can’t remember the exact instruction name (you can look these up in Appendix C), you should be familiar with what instructions are available.

**REVISION EXERCISE** What do the following do: **sbi, cbi, sbic, sbis, rjmp, ldi, st, ld, clr, ser, in, out, cp, cpi, brbs, brbc, breq, brne, brcc, subi, dec, inc, mov, asr, lsr, lsl, ror** and **rol**? (Answers in Appendix D.)

### *Timing without a timer?*

Sometimes we will want to use the **TCNT0** for other purposes (such as counting signals on **T0/PD4**), and so we will now look at timing without the use of this timer. Each instruction takes a specific amount of time, so through the use of carefully constructed loops we can insert delays which are just as accurate as with Timer 0. The only drawback of this is that the loop cannot be interrupted (say, if a button is pressed), unlike the Timer 0, which will keep counting regardless.

The overall idea is to find the number of clock cycles we need to waste and count down from this value to 0. The problem lies when the number is greater than 255 (which is the case almost all the time). In this case we need to somehow split the number over a number of registers, and then cascade them. We decrement the lowest byte until it goes from 00 to FF (setting the carry flag as it does so), and then decrement the next highest byte etc.

<i>Example 2.7</i>	Higher byte	Lower byte	Carry flag?
	<b>0x1A</b>	<b>0x04</b>	<b>no</b>
	<b>0x1A</b>	<b>0x03</b>	<b>no</b>
	<b>0x1A</b>	<b>0x02</b>	<b>no</b>
	<b>0x1A</b>	<b>0x01</b>	<b>no</b>
	<b>0x1A</b>	<b>0x00</b>	<b>no</b>
	 <b>0x1A</b>	 <b>0xFF</b>	<b>YES (so decrements upper byte)</b>
	<b>0x19</b>	<b>0xFF</b>	<b>no</b>
	<b>0x19</b>	<b>0xFE</b>	<b>etc.</b>

The first step is to work out how many instruction cycles the time delay requires. For example, to wait one second with a 4 MHz crystal, we need to ‘kill’ 4 million clock cycles. The loop we will write will take ‘*x*’ instruction cycles, where *x* is given in Table 2.1.

**Table 2.1**

<i>x</i>	Length of time with 4 MHz clock	With 2.4576 MHz clock
3	0–63 $\mu$ s	0–102 $\mu$ s
4	64 $\mu$ s–16 ms	102 $\mu$ s–26 ms
5	16 ms–4.1 seconds	26 ms–6.7 seconds
6	4.2 seconds–17 minutes	6.7 seconds–27 minutes
7	17 minutes–74 hours	27 minutes–120 hours

We are timing one second, which means  $x = 5$ . We therefore divide 4 000 000 by 5, getting in this case 800 000. We convert this number to hexadecimal, getting 0xC3500. Write this number with an even number of digits (i.e. add a leading 0 if there are an odd number of digits), and then split it up into groups of two digits. For example, our values are 0x00, 0x35 and 0x0C.

At the start of the delay in the program we put these numbers into file registers, note the order.

```
ldi    Delay1, 0x00    ;
ldi    Delay2, 0x35    ;
ldi    Delay3, 0x0C    ;
```

The delay itself consists of just one line per delay register plus one at the end (i.e. in our case four lines). To help us achieve such a short loop we need to use a new instruction:

```
sbc    reg, number    ;
```

Subtract the immediate number from a register, and also subtract 1 if the carry flag is set. For example:

```
sbc    Delay2, 0      ;
```

This effectively subtracts 1 from Delay 2 if the carry flag is set, and subtracts 0 otherwise. Our delay loop is as follows:

```
Loop:   subi    Delay1, 1 ; subtracts 1 from Delay1  
         sbc    Delay2, 0 ; subtracts 1 from Delay2 if Carry is set  
         sbc    Delay3, 0 ; subtracts 1 from Delay3 if Carry is set  
         brcc   Loop      ; loops back if Carry is clear
```

When it finally skips out of the loop, one second will have passed. The first thing to note is that the length of the loop is five clock cycles (the branching instruction takes *two* clock cycles). You can now see where the numbers in Table 2.1 come from – for every extra delay register you add there is an extra cycle in the loop. The reason we have used **subi** to subtract 1 instead of **dec** is that unlike **subi**, **dec** doesn't affect the carry flag. We clearly rely on the carry flag in order to know when to subtract from the higher bytes, and when to skip out of the loop.

### *The program counter and subroutines*

There is an inbuilt counter, called the *program counter*, which tells the AVR what instruction to execute next. For normal instructions, the program counter (or PC for short) is simply incremented to point to the next instruction in the program. For an **rjmp** or **brne** type instruction, the number in the PC is changed so that the AVR will skip to somewhere else in the program.

#### *Example 2.8*

```
Start:  
039    sbi    PortB, 0    ; turns on LED  
03A    sbic   PinD, 0    ; tests push button  
03B    cbi    PortB, 0    ; turns off LED
```

**Loop:**

```

03C      dec    Counter      ;
03D      breq   PC+2          ; skips next line if 0
03E      rjmp  Start         ;
03F      rjmp  Loop         ;

```

The above example segment has the program memory addresses for each instruction on the left-hand side in hexadecimal. Note that blank lines aren't given addresses, nor are labels, for they are actually labelling the address that follows. Looking at the behaviour of the PC in the above, it starts at 039 and upon completion of the **sbi** instruction gets incremented to 03A. Then **PinD, 0** is tested. If it is high, the PC is simply incremented to 03B, but if it is low, the program skips, i.e. the PC is incremented twice to 03C. The **rjmp Start** instruction moves 039 into the PC, making the program skip back to **Start**. This also sheds some light on the PC+2 trick we've used a few times already, if the result is 'not equal' (i.e. zero flag clear), the program adds 2 to the PC rather than 1, thus skipping one instruction.

**EXERCISE 2.12** In the example above, what is the effect of the instruction **rjmp Loop** on the PC?

This now brings us to the topic of *subroutines*. A subroutine is a set of instructions within the program which you can access from anywhere in the program. When the subroutine is finished, the program returns and carries on where it left off. The key feature here is the fact that the chip has to *remember* where it was when it called the subroutine so that it can know where to carry on from when it returns from the subroutine. This memory is kept in what is known as a *stack*. You can think of the stack as a stack of papers, so when the subroutine is called, the number in the program counter is placed on top of the stack. When a returning instruction is reached, the top number on the stack is placed back in the program counter, thus the AVR returns to execute the instruction after the one that called the subroutine. The 1200 has a *three level* stack. When a subroutine is called within a subroutine, the number in the PC is placed on top of the stack, *pushing* the previous number to the level below. The subsequent returning instruction will, as always, select the number on the top of the stack and put it into the PC. A three level stack means you can call a subroutine within a subroutine within a subroutine, but not a subroutine within a subroutine within a subroutine within a subroutine. This is because once you've pushed three values on to the stack, and you call another subroutine, hence pushing another value on to the stack, the bottom of the stack is lost permanently. The example in Figure 2.18 illustrates this problem.

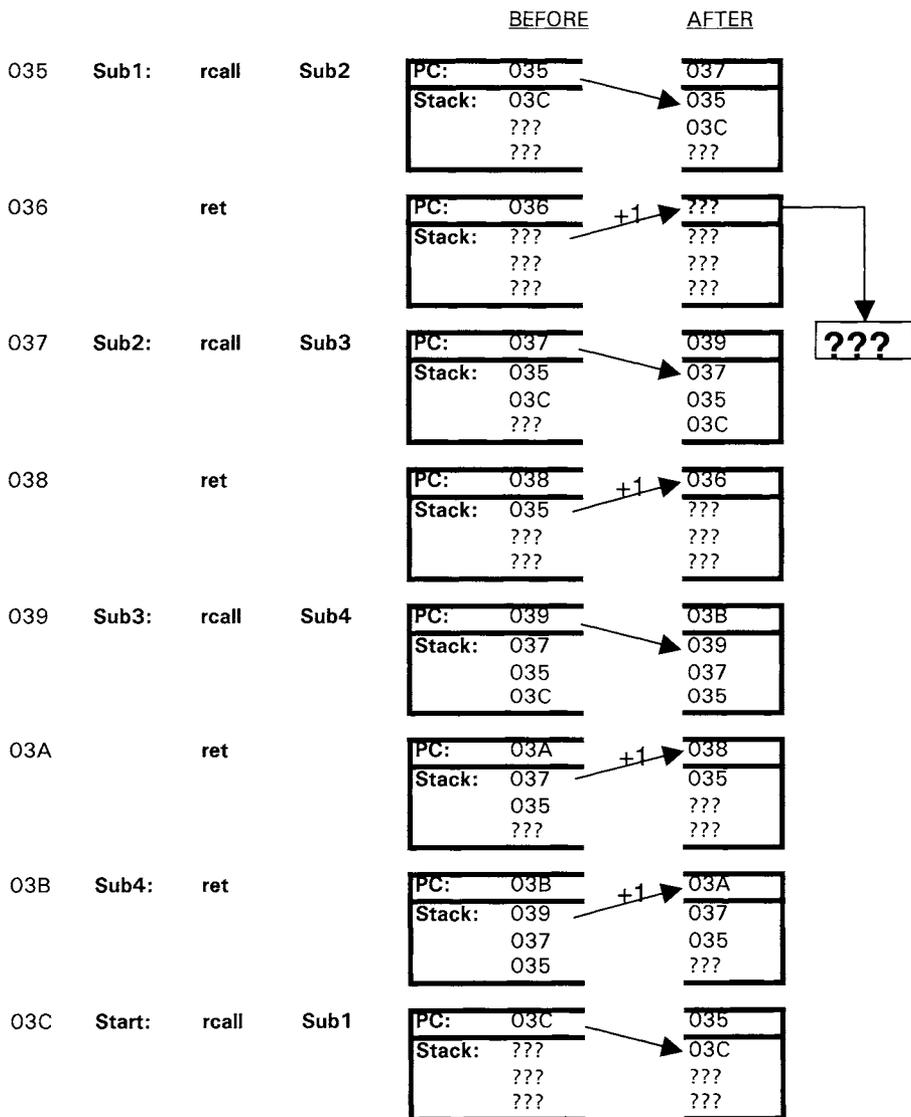


Figure 2.18

The instruction to call a subroutine is:

**rcall    label            ;**

Which is a **relative call**, and so the subroutine needs to be within 2048 instructions of the **rcall** instruction. To **return** from a subroutine use:

```
ret ;
```

Of course, you can call as many subroutines as you like within the same subroutine like so:

```
Sub1: rcall Sub2 ;
      rcall Sub3 ;
      rcall Sub4 ;
      ret ;
```

```
Start: rcall Sub1 ;
```

Note that the programs so far have been upwardly compatible (this means they would work on more advanced types of AVR). This ceases to be strictly true with subroutines, and if you are developing these programs on a chip other than the 1200 or Tiny AVRs you will have to add the following four lines to the **Init** section – Chapter 3 explains why:

```
ldi temp, LOW(RAMEND) ; stack pointer points to
out SPL, temp ; last RAM address
ldi temp, HIGH(RAMEND) ;
out SPH, temp ;
```

The simulator button  is used to *step over* a subroutine – i.e. it runs through the subroutine at high speed and then moves on to the next line. The *step out* button, , is used when the simulator pointer is in a subroutine and will make the simulator run until the return instruction is reached.

## Program G: counter v. 3.0

- Debouncing inputs
- Seven segment display

Now that we know how to implement a timer, we can look back to improving the counter project to include debouncing features to counteract the effect of button bounce. The new flowchart is shown in Figure 2.19.

We can see from the flowchart that we need to insert two identical delays before and after the **ReleaseWait** section in the program. Rather than duplicating two delays, we can have a delay *subroutine* that we call twice. For example, if we call our delay subroutine **Debounce**, the following would be the last few lines of the new program:

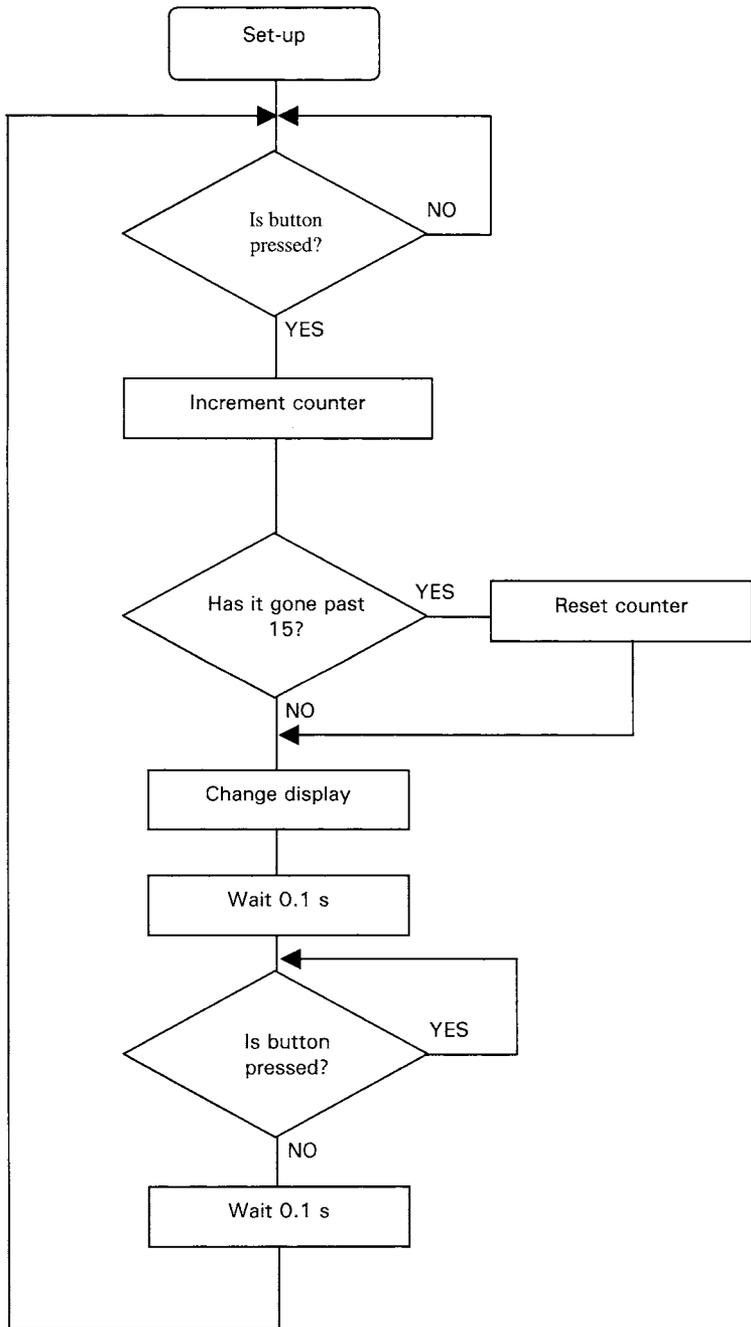


Figure 2.19

	<b>rcall</b>	<b>Debounce</b>	<b>; inserts required delay</b>
<b>ReleaseWait:</b>	<b>sbis</b>	<b>PinD, 0</b>	<b>; button released?</b>
	<b>rjmp</b>	<b>ReleaseWait</b>	<b>; no, so keeps looping</b>
	<b>rcall</b>	<b>Debounce</b>	<b>; inserts required delay</b>
	<b>rjmp</b>	<b>Start</b>	<b>; yes, so loops back to start</b>

Finally we can write the **Debounce** subroutine. I like to keep my subroutines in the top half of the page to keep things tidy, after the **rjmp Init** line, but before the **Init** section itself. In this case we will use the delay without Timer 0.

EXERCISE 2.13 How many clock cycles will it take to create a 0.1 second delay, given a 4 MHz crystal? Convert this number into hexadecimal, and split it up over a number of bytes. What should the initial values of the delay registers be?

EXERCISE 2.14 *Challenge!* Write the *eight* lines that make up the **Debounce** subroutine.

You must also remember to define the three new registers you have added. With R20–R29 taken up by the seven segment code registers, and R30,31 belonging to ZL and ZH, you may think you’ve run out of useful room, and may have to use the less versatile R0–R15. However, notice that while in the **Debounce** subroutine, you are not using the **temp** register. You could therefore use **temp** instead of **Delay1**. Either define **Delay1** as R16 (there is nothing strictly wrong with giving a register two different names), or as this is potentially confusing you may prefer to scrap the name **Delay1** and use **temp** instead in the **Debounce** subroutine. Try this program out and see if you’ve eliminated the effect of the button bounce. Can you make the time delay smaller? What is the minimum time delay needed for reliable performance?

## Program H: traffic lights

- Timing without Timer 0
- Toggling outputs

Our next project will be a traffic lights controller. There will be a set of traffic lights for motorists (green, amber and red), and a set of lights for pedestrians (red and green) with a yellow WAIT light as well. There will also be a button for pedestrians to press when they wish to cross the road. There will be two timing operations needed for the traffic lights. We will be monitoring the time between button presses as there will be a minimum time allowed between each time the traffic can be stopped (as is the case with real pedestrian crossings). As well as this, we will need to measure the length of time the lights stay on, and blinking. We will use the Timer 0 to control the minimum time between button presses (which we’ll set to 25 seconds), and use the ‘Timerless’ method just introduced for all other timing. The circuit diagram is shown in Figure 2.20, and the flowchart in Figure 2.21.

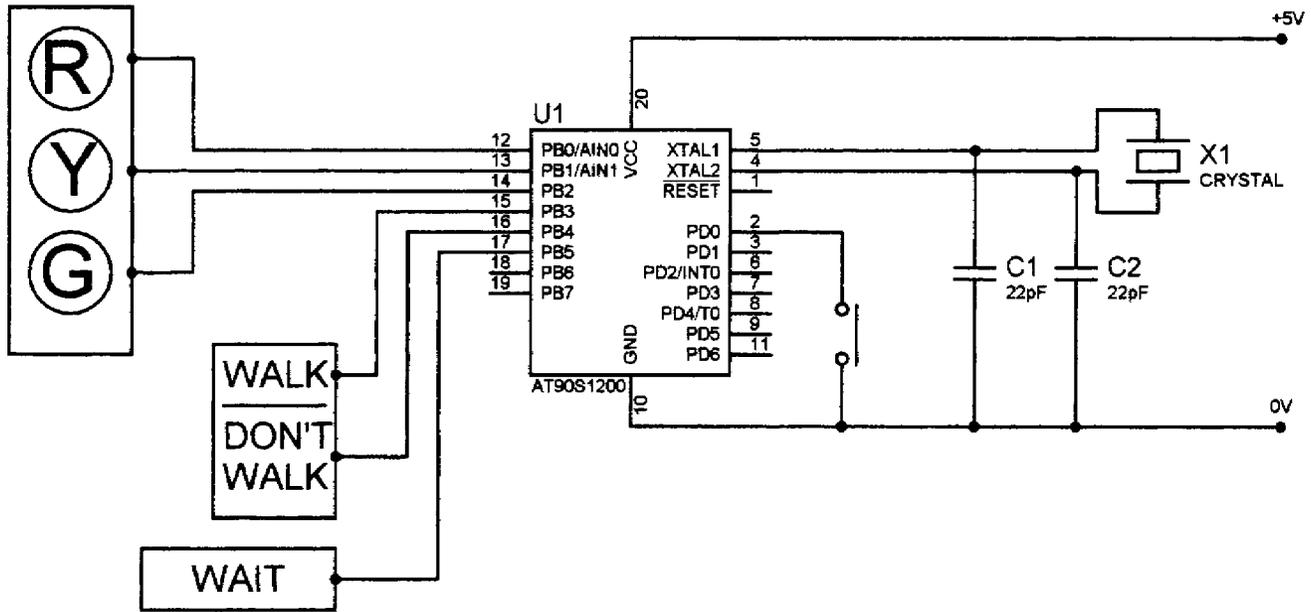


Figure 2.20

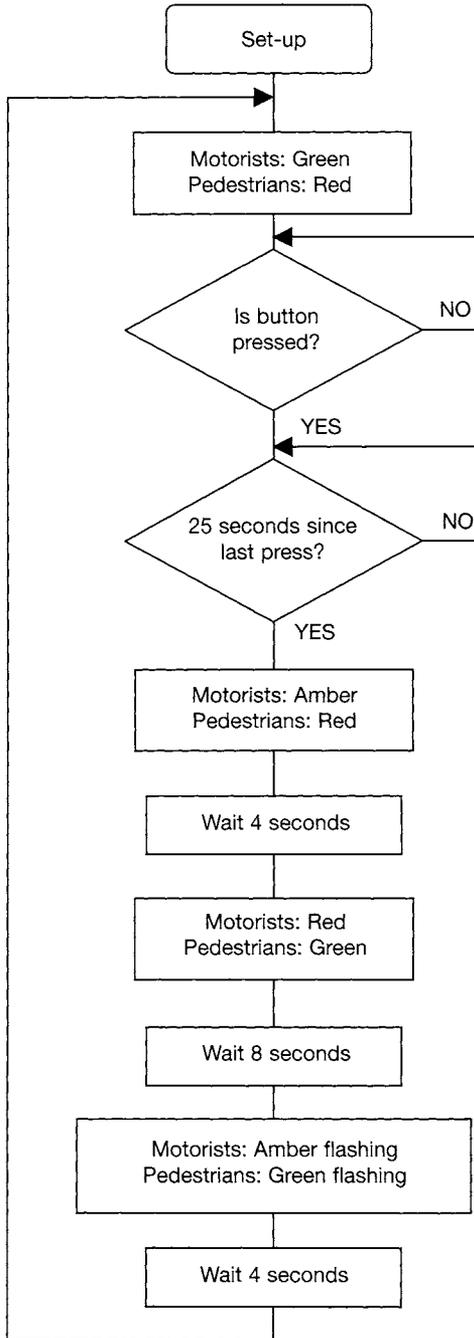


Figure 2.21

You can write the **Init** section yourself, noting that PD0 requires an internal pull-up. Set up **TCNT0** to count at CK/1024.

The first two lines get the LEDs in the correct state with the red pedestrian light on, as well as the motorists' green.

EXERCISE 2.15 What *two* lines will do this?

We need to perform some sort of timing during this initial loop so that while it is waiting for the button, it can also be timing out the necessary 25 seconds. This will be taken care of by a subroutine called **Timer** which we will write later. So after these two first lines insert:

```
rcall Timer ; keeps timing
```

In this subroutine we will use the **T** bit in SREG, a temporary bit you can use for your own purposes. We will use it to signal to the rest of the program whether or not the required 25 seconds have passed. It will initially be off, but after the traffic is stopped, and the people cross etc., it is set. When it is set and **Timer** is called, it will count down, but rather than staying in a loop until the time has passed it returns (using **ret**) if the required time hasn't passed. When the required time does pass, the **T** bit is cleared again, and the rest of the program knows it's OK to stop the traffic again. After this instruction we test the button.

EXERCISE 2.16 What *two* lines will then test the push button and loop back to **Start** if it isn't pressed?

EXERCISE 2.17 If the button is pressed the pedestrian's WAIT light should be turned on, what *one* line does this?

To test the **T** bit, you can use one of the following instructions:

```
brts label ; branches if the T bit is set  
brtc label ; branches if the T bit is clear
```

EXERCISE 2.18 What *two* lines form a new loop which calls **Timer**, and tests the **T** bit in SREG, staying in the loop until the **T** bit is clear.

After the required time has passed, we can start slowing the traffic down. Turn the green motorists' light off, and the amber one on. Keep all other lights unchanged.

EXERCISE 2.19 What *two* lines achieve this?

As the flowchart shows, there are quite a few time delays required, and rather

than copy the same thing over and over, it makes sense to use a time delay subroutine. If we look at the minimum delay we will be timing (which is 0.5 second for the flashing), we can write a delay for this length and then just call it several times to create longer delays. The delay will be called **HalfSecond**, and so to wait 4 seconds we call this subroutine 8 times. We could simply write **rcall HalfSecond** eight times, but a shorter way would be the following:

```

        ldi    temp, 8        ;
FourSeconds:
        rcall  HalfSecond    ;
        dec   temp           ;
        brne  FourSeconds    ;

```

**temp** is loaded with 8, and then each time it is decremented, **HalfSecond** is called. After doing this eight times it skips out of the loop.

After this 4 second delay the red motorists' light must be turned on, and the amber one off. The red pedestrian light must be turned off, and the green one on. The pedestrian's WAIT light must also be turned off.

EXERCISE 2.20 Which *two* lines will make the required output changes?

EXERCISE 2.21 Which *four* lines make up an 8 second delay?

After the 8 seconds, the red motorists' light turns off, and the motorists' amber and pedestrians' green lights must flash. Start by turning the flashing lights on, and then we will look at how to make them flash.

EXERCISE 2.22 Which *two* lines make the required output changes?

To toggle the required two lights, we need to invert the states of the bits. There are two ways to invert bits. We could take the *one's complement* of a register, using:

```

com    register           ;

```

This inverts the states of all of the bits in a register (0 becomes 1, 1 becomes 0).

EXERCISE 2.23 If the number in **temp** is **0b10110011**, what is its resulting value after **com temp**?

However, we want to *selectively* invert the bits. This is done using the *exclusive OR* logic command. A logic command looks at one or more bits (as its inputs) and depending on their states produces an output bit (the result of the logic operation). The table showing the effect of the more common *inclusive OR* command on 2 bits (known as a *truth table*) is shown below:

inputs		result
0	0	0
0	1	1
1	0	1
1	1	1

The output bit (**result**) is high if either the first **or** the second input bit is high (or if both are high). The exclusive OR is different in that if *both* inputs are high, the output is low:

inputs		result
0	0	0
0	1	1
1	0	1
1	1	0

One of the useful effects is that if the second bit is 1, the first bit is toggled, and if the second bit is 0, the first bit isn't toggled (see for yourself in the table). In this way certain bits can be selectively toggled. If we just wanted to toggle bit 0 of a file register, we would exclusive OR the file register with the number **00000001**.

The exclusive OR instruction is:

```
eor    reg1, reg2    ;
```

This **exclusive ORs** the number in **reg2** with the number in **reg1**, leaving the result in **reg1**.

EXERCISE 2.24 What *four* lines will read state of the lights into **temp**, selectively toggle bits 1 and 3, and then output **temp** back to **PortB**. (Hint: You will need a new register, call it **tog**.)

EXERCISE 2.25 *Challenge!* Incorporate the previous answer into a loop that waits half a second, selectively toggles the correct lights, and repeats eight times. You will need a new register to count the number of times round the loop; call this **Counter**, and call the loop **FlashLoop**. This should take *eight* lines.

The traffic lights can now return to their original states, but before looping back to **Start**, remember to *set* the **T** bit. You can do this directly using the following instruction:

```
set                ; sets the T bit
```

EXERCISE 2.26 Write the final *two* lines of the program.

What remains for us now are the two subroutines, **HalfSecond** and **Timer**. We will tackle **HalfSecond** first as it should be the more straightforward.

EXERCISE 2.27 Without using the Timer 0, create a half second delay, and use this to write the *eight* lines of the **HalfSecond** subroutine. A 2.4576 MHz crystal is being used.

For **Timer**, we first test the **T** bit. If it is clear we can simply return.

EXERCISE 2.28 Write the first *two* lines of the **Timer** subroutine.

We can then use the same method we used before in timing loops; however, instead of looping to the top of the section, return from the subroutine. The required time is 25 seconds, which on a 2.4576 MHz crystal with Timer 0 running at CK/1024 corresponds to a marker of 240 and a counter of 250 (work it out!).

EXERCISE 2.29 *Challenge!* Write the remaining *ten* lines of the **Timer** subroutine. Assume your counter and marker registers have been set up in the **Init** section (do this!), and reset the counter register with its initial value at the *end* of the subroutine. Don't forget to *clear* the **T** bit at the end of the subroutine (use the **clt** instruction).

Congratulations! You have essentially written this whole program yourself. To check the entire program, look at Program H (Appendix J).

## Logic gates

We had a short look at the inclusive OR and exclusive OR logic gates, and now we'll look at other types: AND, NAND, NOR, ENOR, BUFFER, NOT. The truth tables are as follows:

### AND

inputs		result
0	0	0
0	1	0
1	0	0
1	1	1

This is useful for *masking* (ignoring certain bits). If the second bit is 0, the first bit is *masked* (made 0). If the second bit is 1, the first bit remains intact.

Therefore ANDing a register with 0b00001111 masks bits 4–7 of the register, and leaves bits 0–3 the same.

### NAND

inputs		result
0	0	1
0	1	1
1	0	1
1	1	0

*This is the opposite of an AND*

### NOR

inputs		result
0	0	1
0	1	0
1	0	0
1	1	0

*This is the opposite of an OR*

### ENOR

inputs		result
0	0	1
0	1	0
1	0	0
1	1	1

*This is the opposite of an EOR*

### NOT

input	result
0	1
1	0

*Only one input, output is opposite of input*

### Buffer

input	result
0	0
1	1

*Only one input, output copies input*

There aren't specific instructions for all these gates, but they can be implemented using a combination of available instructions.

## Program I: logic gate simulator

- Logic functions
- TinyAVR

Our next project will be a logic gate simulator which can be programmed to act as any of the eight gates given above. It will therefore require two inputs and one output, and three inputs will together select which gate it is to emulate. This makes a total of six I/O pins, which just fits on the Tiny AVR chips. We will be writing this program for the Tiny12 AVR in particular, but it can be adapted to most of the other types, including the 1200 that we have so far been writing for. Figure 2.22 shows the pin layouts of some of the members of the Tiny family.

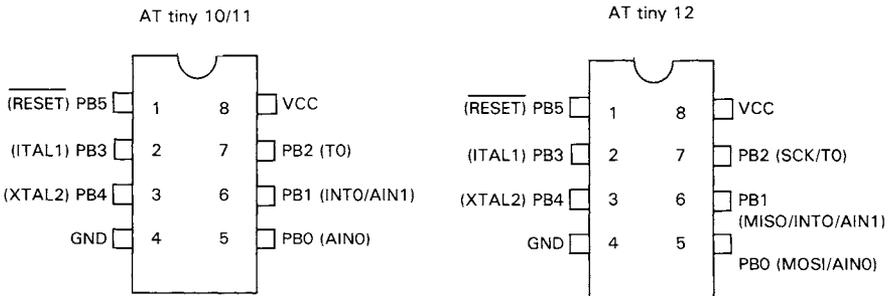


Figure 2.22

Basic features about this family include having a 6-bit Port B (PB0–PB5), but these six I/O pins are available only under certain circumstances. For example, you can see that PB3 and PB4 are also the oscillator inputs, and so to use these as I/O pins requires selection of the *internal* oscillator. Using a separate oscillator (and therefore only needing XTAL1 as a clock input) means PB4 is available, but PB3 isn't. Using the RESET pin as a reset pin means losing PB5. So you can see that having six I/O is very much a maximum. Also, take note that on the Tiny10 and Tiny11 PB5 is an *input only*. On the Tiny12, PB5 is an input or an *output drain* (this means you can make it an output, but only a low output – i.e. it can sink but not source current). This means that although **PinB** and **DDRB** are 6 bits long, **PortB** is only 5 bits long. PB5 therefore has no internal pull-up, and so needs an external resistor. An advantage of the Tiny AVR's over the 1200 model we have been using so far is the availability of the following instruction:

lpm ;

This loads the contents of the program memory pointed to by Z into register R0. This means we can use the program memory itself as a look-up table, as opposed to using up working registers. It is also more efficient on code, as each instruction in the program memory is 16 bits long, so we can store 2 bytes in place of an instruction. We will be needing this instruction in the example project.

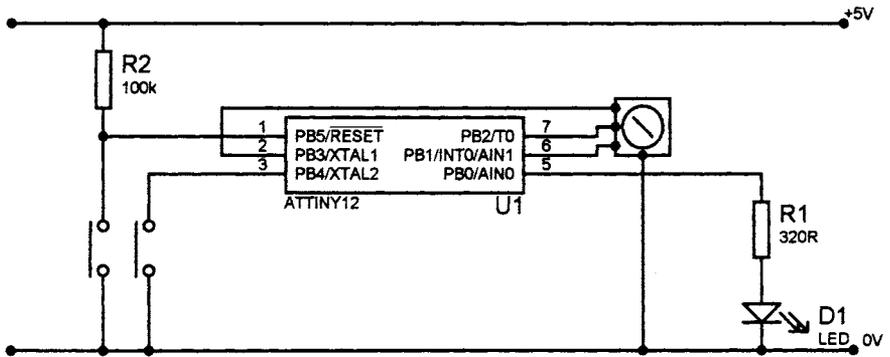


Figure 2.23

The circuit diagram for the logic gate project is shown in Figure 2.23. Note that the NOT and Buffer gates take only one input, and so we will be using PB1 as the input for these gates. Therefore, the effective two-input truth tables for the NOT and Buffer gates are:

**NOT**

inputs		result
0	0	1
0	1	1
1	0	0
1	1	0

**Buffer**

inputs		result
0	0	0
0	1	0
1	0	1
1	1	1

EXERCISE 2.30 Have a go yourself at constructing the flowchart, before looking at my version in the answer section. You need not make it more than three boxes in size, as we aren't yet concerned with sorting out how to manage the imitating of the individual gate types.

When writing the Init section the output, PB2, should initially be off. To choose which logic gate the AVR is to imitate, we have a binary switch which sets its outputs between (000) and (111) depending on the state of the switch. We therefore have to use this in the program to determine which section to jump to. Although the output from the switch is between 000 and 111, the resulting number in **PinB** is between xx000x and xx111x, where the states of bits 0, 4 and 5 must be ignored. We therefore take the number in **PinB** and mask bits 0, 4 and 5 using:

```
andi    reg, number    ;
```

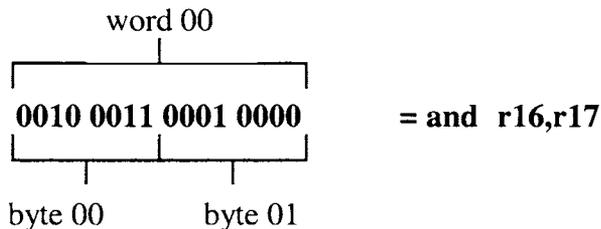
This **ANDs** the number in a register with the immediate **number** (*only for registers R16–R31*). To mask bits 0, 4 and 5, but keep bits 1–3 intact, we AND the register with 0b001110. We then rotate it once to the right, making sure that only zeros appear in bit 5 during the rotation.

EXERCISE 2.31 What is the appropriate rotation instruction to use?

The result is a number between 0 and 7 which we shall use to access a location in the program memory, and so we should load **PinB** into the **ZL** register as this will be used to point to a specific address.

EXERCISE 2.32 Write the *three* lines which read **PinB** into **ZL**, mask bits 0, 4 and 5, and then rotate it to the right as required.

Our look-up table will begin after the **rjmp Init** instruction. This instruction is at address 000 of the program memory (which is why it is the first one executed). Instructions are 16 bits long, and so take up 2 bytes (or one *word*). Program memory addresses are therefore *word addresses*, and the *byte address* is 2 times the word address. Figure 2.24 illustrates this.



**Figure 2.24**

Our look-up table will therefore start at word address 001 which is equivalent to byte address 002. **ZL** points to the byte address, so we will have to add 2 to **ZL** to start it pointing to the bottom of the look-up table.

**EXERCISE 2.33** Which *two* lines will add 2 to **ZL** and then use **ZL** to read a value from the program memory into R0?

Now the real question is what to have in the look-up table that is going to tell the program how to act like a particular logic gate. After some thought, I have found that using a split form of the truth table for each gate gives us the most straightforward solution. What we are about to do now may appear far from obvious, but hopefully after some thought you will see that ultimately it works rather neatly.

We are going to have a byte for each logic gate. For each gate, we take the truth table and look at the set of output states (e.g. 0001 for an AND gate, and 0111 for an inclusive OR). We then split these nibbles into two sets of 2 bits, and make these bits 4 and 5 and 0 and 1 of a byte. For example, AND: 0001 splits into 00 and 01, and then becomes 00000001. Inclusive OR: 0111 splits into 01 and 11, and the becomes 00010011.

**EXERCISE 2.34** What are the relevant bytes for the NAND, NOR, ENOR, EOR, NOT and Buffer gates?

We then list these in the look-up table in any order we choose (noting that their position in the table defines how the code in PB1, 2 and 3 refers to a particular gate). The assembler has *directives* (instructions for the assembler) which tell it to place the following word or byte into the program memory. These directives are **.dw** (**d**efine **w**ord) and **.db** (**d**efine **b**yte). If using **.dw**, you will have to group the bytes derived above into pairs (arbitrarily if you wish), e.g.:

**.dw            0b0000000100010011      ; code for AND and IOR**

OR

```
.db      0b00000001, 0b00010011 ; code for AND, code for IOR
```

There is one important difference between the two lines above. When using **.dw**, the lower byte of the word has the lower byte address. For example, if the two lines above were both written at *word address* 00, the code for the IOR would be at byte address 00 in the **.dw** example, and at byte address 01 in the **.db** example. As long as you take note of the correct byte addresses, it doesn't matter which way you do it.

EXERCISE 2.35 Complete the other *three* lines of the look-up table using **.dw** or **.db**.

Therefore, using the **lpm** instruction we have obtained a form of the truth table for each gate at **R0**. We will then test Input A of the gate (PB4). If it is low we *swap* the nibbles of **R0** (e.g. 00000001 becomes 00010000). What this does is select which half of the truth table we wish to access (remember we split it up into two halves). The swap instruction is:

```
swap    reg          ;
```

and **swaps** upper and lower nibbles of a register. We then test Input B of the gate (PB5). If it is low we *rotate* the number in **R0** to the right. What this does is select which of the two outputs remaining in the truth table is the right one. The four lines we need are therefore:

```
sbis    PinB, 4      ; tests Input A  
swap    R0           ; swaps nibbles if low  
sbis    PinB, 5      ; tests Input B  
ror     R0           ; rotates right if low
```

The state of **R0, bit 0** now holds the output we wish to produce in PB0. However, we don't want to change the states of the pull-ups on the inputs, so we want to move a number into PortB that is all 1s for PB1–4, and PB0 equal to bit 0 of **R0**. Just like ANDing is a way to force certain bits low (masking), inclusive ORing is a way to force certain bits high. For example, in this case if we IOR **R0** with 0b11110 we will get a number that is all 1s except PB0 whose state is intact. We can then move the result of this into PortB safe in the knowledge that the pull-ups will remain. The inclusive OR instruction is:

```
ori     reg, number  ;
```

This inclusive **ORs** a register with the immediate **number**, but only works on registers R16–R31. We therefore have to move **R0** into **temp** using the **mov** instruction.

EXERCISE 2.36 What *four* lines take the number in R0, move it to **temp**, force bits 1–4 high and then output it to **PortB** before looping back to **Start**.

This finishes off the program, it is shown in its complete form in Appendix J.

### *SREG – the status register*

We have seen some of the bits of SREG (zero flag, carry flag and T bit), and we will now look at the remaining five. They can all be individually tested, set or cleared using general SREG instructions: **brbc** and **brbs** which we have already met, and:

<b>bset</b>	<b>bit</b>	<b>; sets a bit in SREG</b>
<b>bclr</b>	<b>bit</b>	<b>; clears a bit in SREG</b>

Each bit also has its own personalized instructions (such as **breq** and **brcc**) which are listed in Appendix C. The bits in SREG are:

**SREG – STATUS Register (\$3F)**

<b>bit no.</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>
<b>bit name</b>	<b>I</b>	<b>T</b>	<b>H</b>	<b>S</b>	<b>V</b>	<b>N</b>	<b>Z</b>	<b>C</b>

**Carry flag:**  
 Reacts to carrying with arithmetic operations, and to the **ror** and **rol** instructions.

**Zero flag:**  
 0: The result wasn't 0  
 1: The result was 0

**Negative flag:**  
 0: MSB of result is 0  
 1: MSB of result is 1

**Two's complement overflow flag:**  
 0: No two's complement overflow  
 1: Two's complement overflow occurred

**Sign flag:** (XOR of V and N bits)  
 0: Result is positive  
 1: Result is negative

**Half carry flag:**  
 Like the carry flag, except for the lower nibble (i.e. 4 lsbs)

**T bit:**  
 A temporary bit

**Global interrupt enable:**  
 Master switch for the interrupts  
 (cleared when an interrupt occurs)

If you want to check whether a particular instruction affects a certain flag, check out the Instruction Overview (Appendix D). The purposes of the negative, two's complement overflow, and sign flags should be clear if you cast your

mind back to the section on negative binary numbers. The half carry flag behaves in exactly the same way as the carry flag, except for the lower nibble. For example:

$$\begin{array}{r} \phantom{+} 1111 \\ 01011010 = 90 \\ + 00001111 = 15 \\ \hline 01101001 = 105 \end{array}$$

This operation would set the half carry flag, as there was a carry on the bit 3 pair. The global interrupt enable will be introduced in the section on interrupts in Chapter 4.

### *Watchdog timer*

A potentially useful feature of AVR chips is the *watchdog timer*: a 1 MHz internal timer, independent of outside components, which resets the AVR at regular intervals. In order to stop the AVR resetting, the watchdog timer must be cleared at regular intervals (i.e. before it has time to reset the chip). It is chiefly used as a safety feature, for if the program crashes the watchdog timer will shortly kick in and reset the chip, hopefully restoring normal operation. The watchdog timer is cleared using:

```
wdr ;
```

This resets the **watchdog** timer (often called ‘patting the dog’). The watchdog timer (WDT for short) is controlled by the WDTCR register:

**WDTCR** – Watchdog Timer Control Register (\$21)

bit no.	7	6	5	4	3	2	1	0
bit name	-	-	-	-	WDE	WDP2	WDP1	WDP0

000	15 ms
001	30 ms
010	60 ms
011	0.12 second
100	0.24 second
101	0.49 second
110	0.97 second
111	1.9 seconds

**Watchdog enable:**

0: Watchdog Timer disabled

1: Watchdog Timer enabled

WDE controls whether or not the WDT is enabled, and WDP0-2 controls the length of time before the chip is reset. Note that the times given in the table are susceptible to temperature effects and are also a function of the supply voltage. The values in the table are for a supply of 5.0 V. For a 3.0 V supply the times are approximately three times longer.

**Sleep**

There are often applications where you wish the chip to be idle for a while until something happens. In such cases it is handy to be able to send the AVR to a low power mode called *sleep*. The AVR can be woken up from sleep by an external reset, a WDT reset, or by an interrupt (these are discussed in Chapter 4). The instruction to send the AVR to sleep is simply:

```
sleep          ;
```

There are two types of sleep: a light snooze and a deep sleep. The light snooze (called *idle mode*) halts the program but keeps the timers (such as Timer 0) running. The deep sleep (called *power-down mode*) shuts down everything such that only the WDT,  $\overline{\text{Reset}}$  pin, and INT0 interrupt can wake it up.

For example, to design a device that turns on when moved, we could do the following. Test the vibration switch and go to (deep) sleep if it is off. The WDT will then wake up the AVR and reset it. Testing the vibration switch will take a few microseconds, and the WDT could be set to time out every 60 ms, meaning

the AVR is only on for about a thousandth of the time. When the vibration switch does eventually trigger the AVR will skip the sleep instruction and continue with normal operation. The WDT could then be disabled or reset at regular intervals using **wdr**.

To control the sleep properties of the AVR, we use an I/O register called **MCUCR (\$35)**. Bit 5 of the **MCUCR** is the sleep enable, and this bit must be set if you wish to use the **sleep** instruction. Bit 4 selects which type of sleep you require (0 for idle mode and 1 for power-down mode).

### *More instructions – loose ends*

Through the example projects we have encountered the majority of the instructions for the 1200 and Tiny AVR's. Here is the remainder:

```
neg    reg          ;
```

This instruction makes the number in a register **negative** (i.e. takes the *two's complement*).

```
nop          ;
```

This stands for **no operation**, in other words do nothing. This essentially wastes one clock cycle, and can be quite useful. There are further instructions which perform logic and arithmetic operations on pairs of registers:

```
and  reg1, reg2 ; ANDs reg1 and reg2, leaving result in reg1  
or   reg1, reg2 ; ORs reg1 and reg2, leaving result in reg1  
add  reg1, reg2 ; adds reg1 and reg2, leaving result in reg1  
adc  reg1, reg2 ; as add, but adds an extra 1 if the Carry flag is set  
sub  reg1, reg2 ; subtracts reg2 from reg1, leaving result in reg1  
sbc  reg1, reg2 ; as sub, but subtracts a further 1 if the Carry flag  
; is set
```

There are also instructions to load a specific bit in a register into the **T** bit of SREG:

```
bst  reg, bit   ; stores a bit in a register into the T bit  
bld  reg, bit   ; loads a bit in a register into the T bit
```

There are two more comparing instructions:

```
cpse reg1, reg2 ;
```

This **compares** two registers and **skips** the next instruction if they are equal. In

the same way that the **cp** instruction effectively performs a **sub** between two registers without actually changing them, the instruction **cpc** effectively performs an **sbc** between two registers without actually changing them. The SREG flags (e.g. carry and zero flag etc.) are affected in exactly the same way as with the **sub** and **sbc** instructions:

```
cpc  reg1, reg2 ; compares two registers taking the Carry flag into
                ; account
```

Finally there are two instructions for testing the state of a bit in a working register:

```
srbc reg, bit   ; tests a bit in a register and skips next instruction if
                ; clear
srbs reg, bit   ; tests a bit in a register and skips next instruction if
                ; set
```

## Major program J: frequency counter

- Multiple seven segment display
- Timing + counting
- Watchdog timer

We will end the chapter with a large project covering the key issues raised. We will design a frequency counter with a range 1 Hz–999 kHz. The frequency will be displayed on three seven segment displays, giving the frequency in Hz if it is less than 1 kHz, and in kHz otherwise. An LED will indicate the units. As an added feature, the device will stay on only when a signal greater than 1 Hz is fed into the input, and it will go to sleep when such a signal disappears. The circuit diagram is shown in Figure 2.25.

Notice that as we will be strobing the seven segment displays, each display will be on for only one-third of the time. In order to give each LED the same average current as it would be getting if it were being driven continuously, we need to divide the LEDs' series resistors by 3. Assuming a 5 V supply and a 2 V drop across the LED, there will be 3 V across the resistor. To supply a current of 10 mA to the LED if it were driven continuously, we would therefore choose a resistor value of 300 ohms. For this case I have therefore gone for a value of 100 ohms.

There are two ways to measure frequency. For high frequency signals it is best to take a fixed amount of time and count the number of oscillations on the input during that time. This can then be scaled up to represent a frequency. For lower frequency signals this method becomes too inaccurate, and so instead we measure the length of time between rising edges on the input. The program will have to work out whether the input frequency is high or low, and therefore which method it should use.

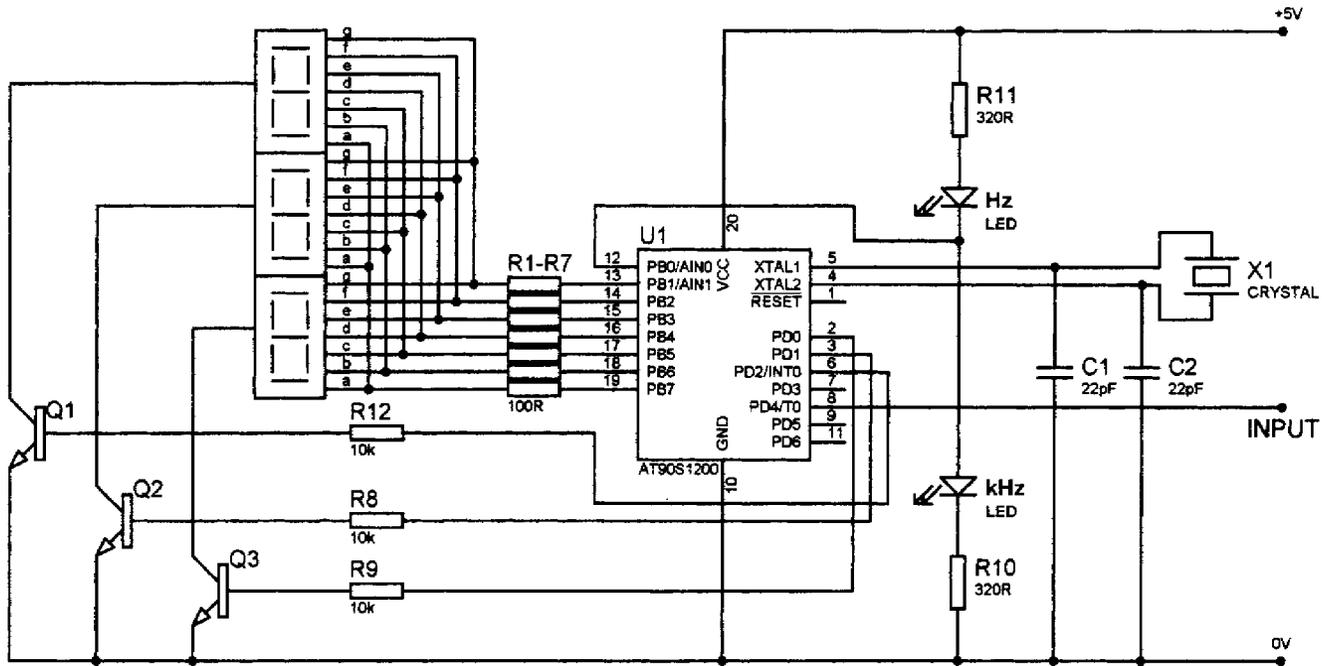


Figure 2.25

We have only one timer/counter at our disposal, which is an inconvenience, but something we can live with. For high frequency signals it is necessary to use T/C0 to count the input signal, as it will be difficult to test the input reliably. For lower frequency signals it will be easier to test the input directly, and more importantly to measure time accurately. This will be a long program, so it is all the more important to have a clear flowchart, shown in Figure 2.26.

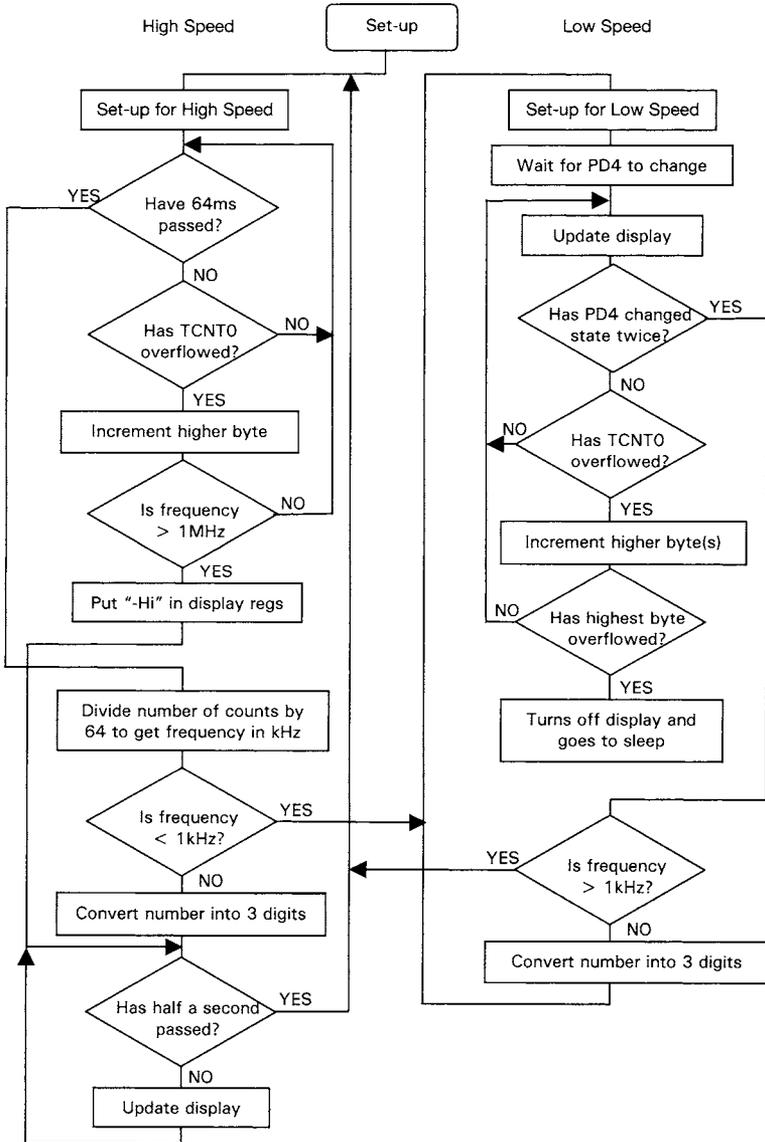


Figure 2.26

The test for high frequency signals takes the shortest time (64 ms), so the program will run this first. If the frequency measured is less than 1 kHz, the program will jump to the low-speed testing. The idea behind the high-speed testing is to time 64 ms by counting clock cycles (i.e. without T/C0), and count signals on T/C0. The only problem is that for timing up to 1 MHz, we would expect 64 000 cycles, i.e. well above 256. We therefore need to be monitoring T/C0 to see when it overflows, and increment a counter which would act as a higher byte for T/C0. You can now see why I chose 64 ms. The maximum number which can be stored over two registers is  $0xFFFF = 65\,536$ , so 64 000 is close to the maximum. Furthermore to convert the number of counts into a frequency in kHz, we need only to divide the number of counts by 64. Dividing a number by  $2^n$  is easy – we simply rotate the number to the right  $n$  times (you may want to try this out on paper). This makes 64 ms an ideal choice.

For the low-speed test, we change T/C0 to count internally. We wait for the input to change and then start timing, waiting until the input changes a further two times before stopping again (this times the length of one cycle). Again, if we look at 1 Hz, with T/C0 counting at 4 MHz, this represents 4 million cycles, and we will need *three* registers to hold the entire number. If the time is greater than these three registers can hold, we know the time is less than 1 Hz, and so send the AVR to sleep. The WDT will be set to wake up the AVR every 1024 ms (i.e. about once a second), though note that in normal operation the WDT will have to be cleared regularly.

For the Init section, set up the ports with no pull-up on the input signal pin. Also, set up the WDTCR to enable the WDT to reset every 1024 ms, and configure MCUCR to enable *deep* (power-down) sleep.

We now need to carefully construct the main loop in which the timing will be carried out – this is the most important part of the program. We can guess that the loop is going to take somewhere between 4 and 10 cycles, so for 64 ms = 256 000 clock cycles, we are going to have to count down between 64 000 and 25 600 times, we can therefore make a guess that *two* counting registers (Delay1 and Delay2) can be used to count the time, but we will have to actually write the loop before we can be sure. Before we enter the loop we will have to set up the delay registers (we don't know what we will have to move into them as this depends on the loop length), set up how T/C0 is going to count, and reset T/C0 to 0. We will also use the move  $0b10000000$  into Port B to turn on the kHz LED and reset the display. You will notice there is also a line clearing a register called **upperbyte**, we will see the significance of this register shortly.

```
ldi    Delay1, ??      ;
ldi    Delay2, ??      ;

ldi    temp, 0b0000111 ; sets T/C0 to count rising edge
out    TCCR0, temp     ; on T0 (PD4)
ldi    temp, 0b1000000 ; turns off all displays and turns on
```

```

out   PortB, temp      ; kHz LED

clr   upperbyte       ; clears a counting register
clr   temp             ; resets Timer 0
out   TCNT0, temp     ;

```

The loop itself starts with the standard decrementing of the 2-byte number spread over the delay registers, skipping out of the loop if the time has passed:

### HighSpeed:

```

subi   Delay1, 1      ; decrements Delay1
sbci   Delay2, 0      ; decrements Delay2 if carry high
brcs   DoneHi         ; jumps out of loop if time passed

```

We then need some way of testing to see if T/C0 has overflowed. There are two ways of doing this. The simplest is to test the *timer overflow flag*, which, unlike the other flags we've met so far, is stored in the **TIFR** I/O register. Unfortunately, we cannot test this flag directly with the **sbic** or **sbis** instructions, as it is number 0x38 which is greater than 0x1F. We would therefore have to read TIFR into a working register, then test the bit. More irritating is the fact that we need to reset it by writing a one to it. Again, we cannot use the **sbi** instruction, and instead have to do it through a working register. This overall process takes five instructions, but there is an alternative method which only uses four. The concept behind this method is to store the current value of T/C0 and compare it with the value that was in T/C0 the previous time in the loop. We would expect the current value to always be greater than the previous value, *except when it overflows*. By comparing the old and new values, and branching if the new is less than the old, we therefore detect an overflow, and no resetting of flags is needed. In the code below, we use register **temp** to store the new value, and **temp2** to store the old value:

```

mov    temp2, temp    ; copies temp into temp2 (old value)
in     temp, TCNT0    ; reads new value into temp
cp     temp, temp2    ; compares old and new
brsh   HighSpeed     ; loops back if new is 'same or higher'

```

If you count through the total **HighSpeed** loop of seven instructions, you will see it takes eight clock cycles if T/C0 doesn't overflow (remember a branching instruction takes two clock cycles). What we need to do now is construct a similar loop that will increment the higher byte, see if it's too high, decrement our counting registers, skip out if they've reached zero, and loop back to **HighSpeed**, all *in the same number of clock cycles*. This final part is crucial to ensure the timing is perfect. Fortunately we can do it all, with a clock cycle to spare! We therefore use **nop** to waste one cycle. The maximum

number of counts we are allowing on the input is 63 999 in the 64 ms (i.e. 1 MHz is just too high, and so 64 000 is just too high – 64 000 translates as 0xFA00, which is handy as we can simply test if the upper byte has reached 0xFA). If it has we know how to skip out of the loop:

```

inc   upperbyte      ; increments higher byte
cpi   upperbyte, 0xFA ; too high?
breq  TooHigh       ; skips out of loop if too high
subi  Delay1, 1     ; decrements counting registers
sbci  Delay2, 0     ;
brcs  DoneHi        ; skips out of loop if done counting
nop   ; wastes one cycle
rjmp  HighSpeed     ; loops back

```

Now you may be thinking ‘hang on, there are *nine* cycles in the above segment, not eight!’. You are right, of course, but think about the number of cycles in the previous section if the program does *not* loop back to **HighSpeed**. If the program does *not* loop back, it does *not* branch, and so takes one less clock cycle. We make up for this one less clock cycle in the loop above with one more in this loop. Thus in the running of this whole section, the counting registers will either decrement once every eight clock cycles or twice every 16 clock cycles. You may want to write the whole loop down and work through it to convince yourself of this. Now that we know the delay registers decrement every eight clock cycles, we can work out what to initialize them to in order to create a 64 ms delay.

EXERCISE 2.37 What should Delay1 and Delay2 be initialized to?

That’s the hardest part done! We now need to immediately store the current value of T/C0. The only problem is, what if T/C0 has overflowed in between the last test for overflowing and now? We need to use the same test as before.

EXERCISE 2.38 Write the *six* lines which make up the section called **DoneHi**, which stores T/C0 into **lowerbyte**, and compare this value with **temp** (which represents the old value of T/C0). If **lowerbyte** is ‘same or higher’ it skips to a section called **Divide64**, if it isn’t, it increments **upperbyte**, tests to see if it has reached 0xFA, and jumps to **TooHigh** if it has.

The next section needs to divide the 2-byte number split up over **lowerbyte** and **upperbyte** by  $64 = 2^6$ . We do this by rotating the whole number six times; to rotate the upper byte into the lower byte, we rotate the upper byte right with zeros filling bit 7, and then rotate the lower byte right with the carry flag filling bit 7.

EXERCISE 2.39 What *two* lines divide the 2-byte number by 2?

The **Divide64** loop does this six times. First we set up **temp** with the number 6, then divide by 2 as we've done above. Then decrement **temp**, looping back if it does not equal zero. We don't want to reset **temp** with 6, so we really want to jump to **Divide64** and then skip one instruction. This can be done using the following trick:

**rjmp Divide64+1 ; jumps to Divide64 and then skips one**

This works with any jumping/branching instruction, and for any number of skips. Note that large skips (e.g. +8) lead to unwieldy programs which are hard to follow and easy to get wrong.

EXERCISE 2.40 What *five* lines make up the **Divide64** section?

We test to see if the number is too low. The 2-byte word holds the frequency in kHz, so if this number is less than 1 (i.e. 0) we know how to change to the low-speed testing method.

EXERCISE 2.41 What *four* lines test to see if both bytes are 0, and skips to **LowSpeed** if they are.

We then need to convert this number split over 2 bytes into a number of hundreds, tens and ones so that they can be displayed easily. This will be done in a subroutine, as we will have to do it in the **LowSpeed** section as well. To do the conversion we will call **DigitConvert**. As the displays are being strobed, we need to be calling a display subroutine at regular intervals. Unfortunately, our carefully constructed timing loop above cannot accommodate the calling of a display subroutine, as this would insert large numbers of clock cycles and disrupt the timing. The timing routine only takes 64 ms, so the idea here is to leave the displays idle for 64 ms, and then let them run for half a second.

We stick in a simple half second delay using counting registers, making sure we call the **Display** subroutine during the loop.

EXERCISE 2.42 Write the *nine* instructions which set up the three delay registers, and then create a half second delay loop which also calls **Display**. When the required time has passed, the program should jump back to **Start**. You will have to take the length of the **Display** subroutine into account when doing your calculations. The **rcall** instruction actually takes three cycles, and the **ret** instruction takes four. On average, the subroutine itself will take two instructions, so assume the whole subroutine action adds nine clock cycles to the loop. Call the delay loop **HalfSecond**.

All that remains in the high-speed timing method is to deal with the **TooHigh** section, which simply has to make the display registers show -HI. The numbers

to be displayed will be stored in registers called **Hundreds**, **Tens** and **Ones**. There will be a look-up table as before, except in this table 10 will be translated as the symbol for an 'H', and 11 as the symbol for a hyphen '-'. A 12 will be translated as a blank space (i.e. no segments on), and so you should set all digits to 12 in the Init section. We therefore need to move 11 into **Hundreds**, 10 into **Tens** and a 1 into **Ones** (as a 1 will look like an l), and the **Display** subroutine will do the rest. After this we jump to *three lines before* the start **HalfSecond** section (these three lines previously set up the **HalfSecond** counting registers).

EXERCISE 2.43 What *four* lines make up the **TooHigh** section?

This marks the end of the high-speed timing method, and therefore the halfway point in the program.

Let's have a look at the **DigitConvert** subroutine. This takes a number split over **upperbyte** and **lowerbyte**, and converts it into a number of hundreds, tens and ones. This is done by repeatedly subtracting 100 from the 2-byte number until there is a carry. 100 is then added back, and the process is repeated with 10. The number left in the lower byte after this is simply the number of ones, so we can just move the number across. Once we have extracted the number of hundreds, we no longer need to involve the upper byte, as we know the number is now entirely contained in the lower byte (if the number is less than 100 it fits in one byte).

#### **DigitConvert:**

```
clr    Hundreds    ; resets registers
clr    Ones        ;
clr    Tens        ;
```

#### **FindHundreds:**

```
subi   lowerbyte, 100 ; subtracts 100 from lower byte
sbci   upperbyte, 0   ; subtracts 1 if carry
brcs   FindTens      ; does 10's if carry
inc    Hundreds      ; increment number of hundreds
rjmp   FindHundreds ; repeats
```

#### **FindTens:**

```
subi   lowerbyte, -100 ; adds back the last 100
subi   lowerbyte, 10   ; subtracts 10 from lower byte
brcs   FindOnes       ; does 1's if carry
inc    Tens            ; increments number of tens
rjmp   FindTens+1     ; repeats, but doesn't add 100 again
```

#### **FindOnes:**

```
subi   lowerbyte, -10 ; adds back the last 10
```

```

mov    ones, lowerbyte ; number left in lowerbyte = ones
ret    ; finished

```

You may want to work your way through this program with a sample number (e.g. convince yourself that 329 gets reduced to 3 hundreds, 2 tens and 9 ones).

The other subroutine is **Display**. This has to choose which of the three displays to activate, find the appropriate number in **Hundreds**, **Tens** or **Ones**, and then display it. In the half second loop we've written, the subroutine is called about once every 4 ms. We can't make the displays change this often as the LEDs won't have time to turn fully on and the display will be faint with shadows (numbers on other displays appearing on the wrong display). We therefore build in an automatic scaling of 50 – i.e. the subroutine returns immediately having done nothing 49 times, and then on the 50th time it's called, it performs the display routine, and then repeats. This means the displays are changing every 0.2 ms which is far better; however, should you experience any of the effects described above, you may wish to increase 50 to a higher value.

We will use a register called **DisplayCounter**. This will be set up in the Init section with the value 50. The beginning of **Display** therefore decrements **DisplayCounter**, and returns if the result is not 0. If it is 0, **DisplayCounter** should be reloaded with 50. Furthermore, we can take this opportunity to clear watchdog timer. This must be done regularly, and the **Display** subroutine is called regularly in whichever part of the program it happens to be (by regularly I mean at least once a second). A simple solution is therefore to reset the WDT when the **Display** subroutine continues.

EXERCISE 2.44 Write the *five* lines at the start of the **Display** subroutine.

We need some way to know which display we will be displaying, and will store this as a number between 0 and 2 in a register called **DisplayNumber**. Therefore, the first thing we do is increment **DisplayNumber** and reset it to 0 if it has reached 3 (you will also have to clear **DisplayNumber** in the Init section).

EXERCISE 2.45 Write the subsequent *four* lines of the subroutine which perform this.

Now we need to do some serious indirect addressing! First, we extract the right number to be displayed from **Hundreds**, **Tens** or **Ones**. You will have to define these at the top of the program, I defined mine as R26, R27 and R28 respectively. We therefore set up ZL to point to R26 (move 26 into ZL), and then add the number in **DisplayNumber**. This will point to one of the three numbers we want to display. Using the **ld** instruction we load this value into **temp**. The seven segment display codes are stored in registers R0–R12, and so we now zero ZL to R0 (move 0 into it). Adding to R0 the number read into **temp** should point to

the seven segment code of the number to be displayed. Again, load this value into **temp**. We mustn't clear bit 7 of PortB if it is on (indicating kHz). Therefore, test bit 7 of Port B, if it is on, OR the number in **temp** with 0b10000000, and then in either case move **temp** into Port B.

EXERCISE 2.46 Write the *nine* lines which output the correct seven segment code to Port B.

The remainder of the subroutine must turn on the correct seven segment display. Remember the essence of strobing: the number you have just outputted to Port B is going to *all* of the displays, but by turning only one of them on, the number only appears in one of the displays. We basically want to turn on PortD bit 0, then bit 1, then bit 2 and then back to bit 0. The easiest way to do this is to read PinD into **temp**, rotate it left without letting any 1s creep in (i.e. use **lsl**), test to see if bit 3 is high (i.e. gone too far), and reset the value to 0b00000001 if it is.

EXERCISE 2.47 What *six* lines turn on the correct display and then return from the subroutine?

Now all that is left is the low-speed testing section. We need to set up T/C0 to count up every clock cycle (this gives us maximum resolution). We also need to (reset) clear the delay registers **Delay2** and **Delay3**, and clear PB7 to turn on the Hz LED.

EXERCISE 2.48 What *five* lines will start off the **LowSpeed** section?

We need a way to see when PD4 changes (remember now T/C0 is counting internally we need to test the input pin manually). There are a few methods at our disposal, the one I suggest is as follows. Store the initial value in PinD, and then enter a loop which reads in the current value of PinD, and *exclusive OR* it with the initial value. The effect of the EOR is to highlight which bits are different.

*Example 2.9*            0b00011001  
                           EOR 0b10001001  
                           **0b10010000** ← shows that bits 7 and 4 were different

We are interested only in bit 4 (PD4) which is connected to the input, and so after performing the EOR we can test bit 4 of the answer and keep looping until it is high. When in any loop that lasts a long time (as this one might), we must also keep calling the **Display** routine.

```

                in      store, PinD    ; stores initial value
FirstChange:  recall  Display        ; keeps displays going

```

```

in      store2, PinD   ; reads in current value
eor     store2, store  ; EORs current and initial values
sbrs    store2, 4     ; skips out of loop if PD4 changed
rjmp    FirstChange  ; keeps looping until PD4 changes

```

The main loop of the low-speed testing section consists of repeating the above test for *two* changes (i.e. wait for one complete period of the input's oscillation), and incrementing the higher bytes when T/C0 overflows. We deal with the T/C0 overflow in the same way as before, with one important difference. We cannot use **temp** to store the old value because **temp** is used repeatedly in the **Display** subroutine we have just written. It is very important you look out for these kinds of traps as they can be a source of many problems – try to keep your use of working registers local (i.e. don't expect them to hold a number for too long), in this way you can use a register like **temp** all over the program. We can use **Delay1** instead of **temp**, as at the end of the looping, we want **Delay1** to hold the current value in T/C0.

Before we enter the low-speed loop we need to clear **Delay1** and T/C0. We will also need some sort of counter to count the number of times the input changes. We need it to change only twice, so set up a register called **Counter** and load 2 into it.

EXERCISE 2.49 Write the *three* pre-loop instructions.

Now the loop looks for a change in the input in the same way as before, and jumps to a section called **Change** if there is a change.

EXERCISE 2.50 Write the *five* lines which perform this test. (HINT: One of them is before the start of the loop, call the loop **LowLoop**.)

We then call the **Display** subroutine, as we have to do this regularly, then test to see if the T/C0 has overflowed. If it hasn't overflowed, loop back to **LowLoop**. If it has overflowed, increment **Delay2**, and if this overflows increment **Delay3**. The minimum frequency is 1 Hz, and hence the maximum amount of time is about 4 000 000 counts, which in hexadecimal is 0x3D0900. Therefore if **Delay3** reaches 0x3E we know the input frequency is too slow and will jump to a section called **TooSlow**.

EXERCISE 2.51 *Challenge!* What *11* lines form the rest of the low-speed section.

The **Change** section should decrement **Counter**, and loop back to **LowLoop** if it isn't zero. On the second change, it doesn't loop back but instead checks to see if the stored number is low enough to deserve high-speed testing. The maximum frequency measured with this method is 999 Hz, which corresponds

to 4004 clock cycles, hence if the result is 4000 (0xFA0) or less we should branch to **Start** and perform the high-speed testing. It may not be entirely clear how we test to see if the number spread over three registers is less than 0x000FA0. For a start, we cannot subtract the number, as this would change the number in the delay registers. Instead, we use the compare instructions as we would if we were just testing one byte, but also make use of the **cpc** instruction, which compares two registers and also takes the carry flag into account. It is simply analogous to subtracting with the carry (e.g. **sbc**) but without actually changing the registers). The only problem with **cpc** is that it only works between two file registers, not a file register and a number, so we have to load the numbers into temporary working registers. The necessary lines for **Change** are therefore:

```
Change:  in      store, PortB      ; updates new value of PortB
          dec     Counter          ; waits for second change
          brne   LowLoop          ; not second change so loops

          ldi    temp, 0x0F        ; sets up temporary registers
          ldi    temp2, 0x00       ;
          cpi    Delay1, 0xA0     ; compares three-byte number with
          cpc    Delay2, temp     ; 0x000FA0
          cpc    Delay3, temp2    ;
          brcc   PC+2             ; less than FA0 so goes to high-speed
          rjmp   Start           ;
```

You will notice that instead of the expected line (**brcc Start**) – i.e. branch to **Start** if the carry flag is set, we choose to skip the (**rjmp Start**) line if the carry flag is clear. These two methods are clearly identical in their end result, but why introduce an extra line? The reason lies in the fact that the **brcc** can only branch to lines which are 64 instructions away. The **Start** line is, in fact, further away than this, and so must be branched to using the **rjmp** instruction. Points like this will be picked up when you try to assemble the program and are generally missed at the writing stage – so you don't have to start counting 60 odd lines whenever you introduce a **brcc** or similar instruction.

We then convert the time period of the oscillation into a frequency. To do this we need to take 4 000 000 and divide it by the length of time (in clock cycles) we have just measured. If we measured 40 000 clock cycles over one period, this will correspond to 100 Hz. There is a way to perform binary long division, but by far the simplest method of dividing **x** by **y** is to see how many times you can subtract **y** from **x**. This does take fewer instructions, but will take longer to run. We set up 4 000 000 = **0x3D0900**, spread over three temporary registers (**temp**, **temp2** and **temp3**). Every time we successfully subtract the number spread over **Delay1**, **Delay2** and **Delay3**, we increment a lower byte of the answer. When this overflows, we increment the higher byte. The answer will be

between 1 and 1000 so we need only two bytes for the answer. The following lines set up the division:

```
ldi    temp, 0x00    ; moves 4 000 000 spread over 3
ldi    temp2, 0x09   ; temporary registers
ldi    temp3, 0x3D   ;
clr    lowerbyte     ; resets the answer registers
clr    upperbyte     ;
```

EXERCISE 2.52 Write the *eight* lines of the loop called **Divide** which divides 4 000 000 by the number in the delay registers. (Hint: Call the next section **DoneDividing** and jump to this section when a subtraction was unsuccessful (carry flag was set).)

As with the high-speed section, we then convert the number in **lowerbyte** and **upperbyte** into hundreds, tens and ones. We can use the **DigitConvert** subroutine we have already written. The program then loops back to **LowSpeed**.

EXERCISE 2.53 What *two* lines wrap up the low-speed testing loop?

All that remains is the section called **TooSlow** which is branched to when the period of oscillations is more than one second. In this case we want to turn the displays off and send the AVR to sleep.

EXERCISE 2.54 Write the *three* lines which make up the **TooSlow** section.

You will have to remember to set up registers R0 to R11 with the correct seven segment code in the Init section. As you can use only the **ldi** instruction on registers R16–R31 you will have to move the numbers first into **temp**, and then move them into R0 to R11 using the **mov** instruction. Also, remember to set up PortD with one of the displays selected (e.g. 0b00000001), and define all your registers at the top of the program. It should now be ready for testing with the simulator. This may be worth building as it performs a useful function; however, you will notice that its resolution isn't great as you get only three-figure resolution between 100 Hz–999 Hz and 100 kHz–999 kHz. You may want to think about ways to improve the program to give three-figure resolution for all frequencies in the given range. In the coming chapters we will learn methods that will allow us to simplify this program hugely, and it will be worth coming back to this at the end and gleefully hack bits off to trim down the program.

Working on this larger program also introduces the importance of taking breaks. Even when you are 'in the zone' it is always a good idea to step back for a few minutes and relax. You will find you return looking at the bigger picture and may find you are overlooking something. Good planning and flowcharts

help reduce such oversights. Another good piece of advice is to talk to people about decisions you have to make, or problems when you get stuck. Even if they don't know the first thing about microcontrollers, simply asking the question will surprisingly often reveal the answer.