4
Intermediate operations

Interrupts

So far we have always had to test for certain events ourselves (e.g. test for a button to be pressed, test if T/C0 has overflowed etc.). Fortunately there are a number of events which can automatically alert us when they occur. They will, if correctly set up, interrupt the normal running of the program and jump to a specific part of the program. These events are called *interrupts*.

On the 1200, the following interrupts are available:

- Interrupt when the INT0 pin (PD2) is low
- Interrupt when there is a rising edge on INT0
- Interrupt when there is a falling edge on INT0
- Interrupt when T/C0 overflows
- Interrupt when the Analogue Comparator triggers a result

The first three constitute an *external interrupt* on INT0, and are mutually exclusive (i.e. you can enable only one of the three interrupts at any one time). The significance of the *Analogue Comparator* will be discussed later on in the chapter. When an interrupt occurs, the program will jump to one of the addresses at the start of the program. These addresses are given by what is known as the *interrupt vector table*. The interrupt vector table for the 1200 is shown in Table 4.1, the tables for the other AVR types are shown in Appendix E.

Table 4.1

Type of Interrupt/Reset	Program jumps to address ...
Power-on/Reset	0x000
External interrupt on INT0	0x001
T/C0 overflow interrupt	0x002
Analogue comparator interrupt	0x003

For example, when the T/C0 overflow interrupt is enabled, and T/C0 overflows, the program drops what it's doing and jumps to address 0x002 in the program memory. When using all three interrupts, the start of the program should look something like the following:

```
rjmp    Init            ; first line executed
rjmp    ExtInt          ; handles external interrupt
rjmp    OverflowInt     ; handles TCNT0 interrupt
rjmp    ACInt           ; handles A. C. interrupt
```

This will ensure the program branches to the correct section when a particular interrupt occurs (we will call these *interrupt handling routines*). We can enable individual interrupts using various registers. The enable bit for the External INT0 interrupt is bit 6 in an I/O register called **GIMSK** (**G**eneral **I**nterrupt **M**ask). Setting this bit enables the interrupt, clearing it disables it. The enable bit for the TCNT0 overflow bit is bit 1 in the **TIMSK** I/O register (**T**imer **I**nterrupt **M**ask). However, all of these interrupts are overridden by an interrupts 'master enable'. This is a master switch which will disable all interrupts when off, and when on it enables all *individually enabled* interrupts. This bit is the **I** bit in **SREG** (you may want to glance back to page 73).

The External INT0 interrupt can be set to trigger in one of three different circumstances, depending on the states of bits 0 and 1 of the **MCUCR** I/O register (the one that also holds the sleep settings). This relation is shown in Table 4.2.

Table 4.2

MCUCR Bit1	Bit 0	**Interrupt occurs when ...**
0	0	INT0 is low
0	1	*Invalid selection*
1	0	There is a falling edge on INT0
1	1	There is a rising edge on INT0

When an interrupt occurs, the value of the program counter is stored in the stack as with subroutines, so that the program can return to where it was when the interrupt handling is over. Furthermore, when the interrupt occurs, the master interrupt enable bit is *automatically cleared*. This is so that you don't have interrupts occurring inside the interrupt handling routine which would then lead to a mess of recursion. You will probably want to re-enable the master interrupt bit upon returning from the interrupt handling routine. Fortunately there is a purpose-built instruction:

```
reti                    ;
```

This **ret**urns from a subroutine and at the same time enables the master **i**nterrupt bit.

Each interrupt also has an *interrupt flag*. This is a flag (bit) that goes high when an interrupt *should* occur, even if the global interrupts have been disabled

and the appropriate interrupt service routine isn't called. If the global interrupts are disabled (for example, we are already in a different interrupt service routine) you can test the flag to see if any interrupts have occurred. Note that these flags stay high until reset, and an interrupt service routine will be called if the flag is high *and* the global interrupt bit is enabled. So you must reset all flags before enabling the global interrupt bit, just in case you have some interrupt flags lingering high from an event that occurred previously. Interrupt flags are reset by *setting* the appropriate bit – this sounds counterintuitive but it's just the way things are! The T/C0 Overflow interrupt flag is found in bit 1 of **TIFR** (**T**imer **I**nterrupt **F**lag **R**egister – I/O number $38), and the INT0 interrupt flag is in bit 6 of **GIFR** (**G**eneral **I**nterrupt **F**lag **R**egister – I/O number $3A).

Program K: reaction tester

- Interrupts
- Random number generation
- Seven segment displays

The next example program will be a reaction tester. A ready button is pressed, then an LED will turn on a random time later (roughly between 4 and 12 seconds). The user has to press a button when they see the LED turn on. The program will measure the reaction time of the user and display it in milliseconds on three seven segment displays. If the user presses the button before the LED turns on they will be caught cheating. The circuit diagram for the project is shown in Figure 4.1, and the flowchart in Figure 4.2.

We will be using the External INT0 and TCNT0 Overflow interrupts, so you will have to make the necessary changes to the top of the program. Note that as we will not be using the Analogue Comparator interrupt we don't need any particular instruction at address 0x003.

EXERCISE 4.1 What are the first *three* instructions of the program?

Write the Init section, setting T/C0 to count internally at CK/1024. You will have to enable the External INT0 and T/C0 Overflow interrupts, but don't set the master enable just yet. Set the External INT0 interrupt to occur when INT0 is low (i.e. when the button is pressed).

EXERCISE 4.2 What are the *six* lines which individually enable the interrupts?

At **Start** we first call the **Display** subroutine, and then test the 'Ready' button (PinD, 1). Keep looping until the Ready button is pressed.

EXERCISE 4.3 What *three* lines achieve this?

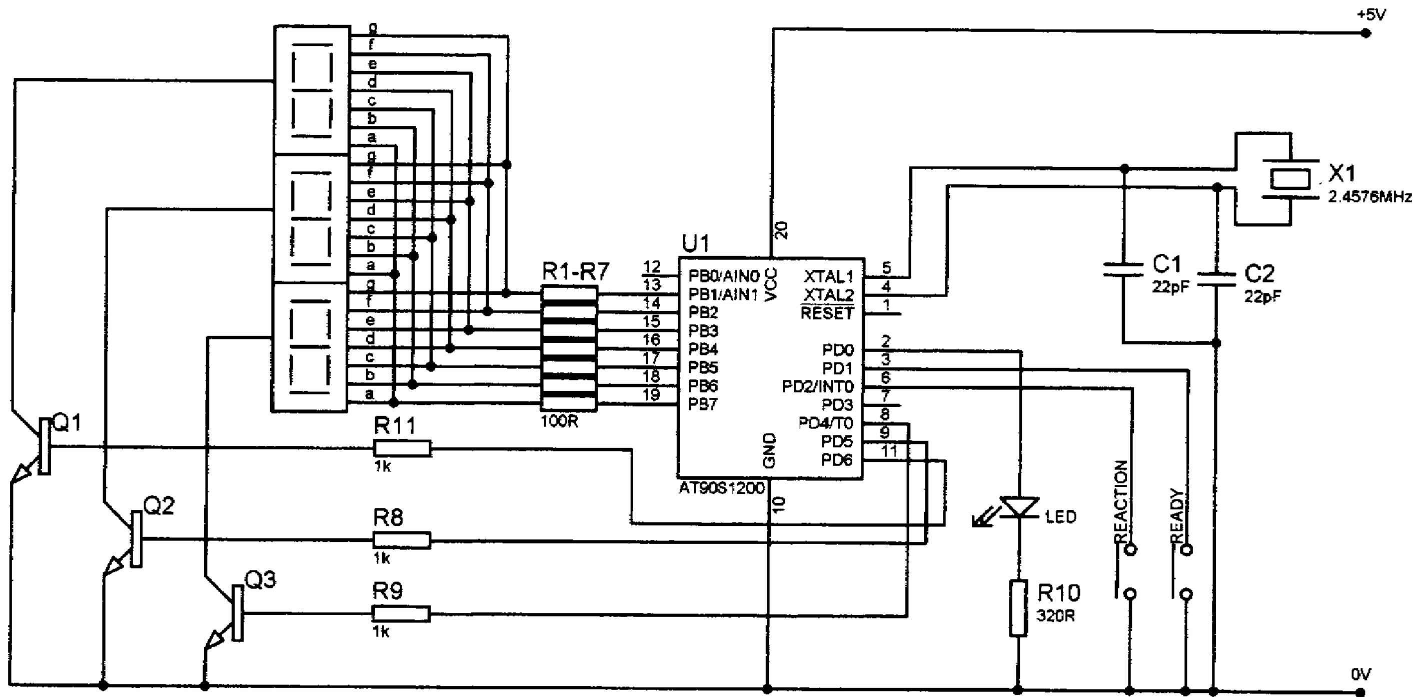

Figure 4.1

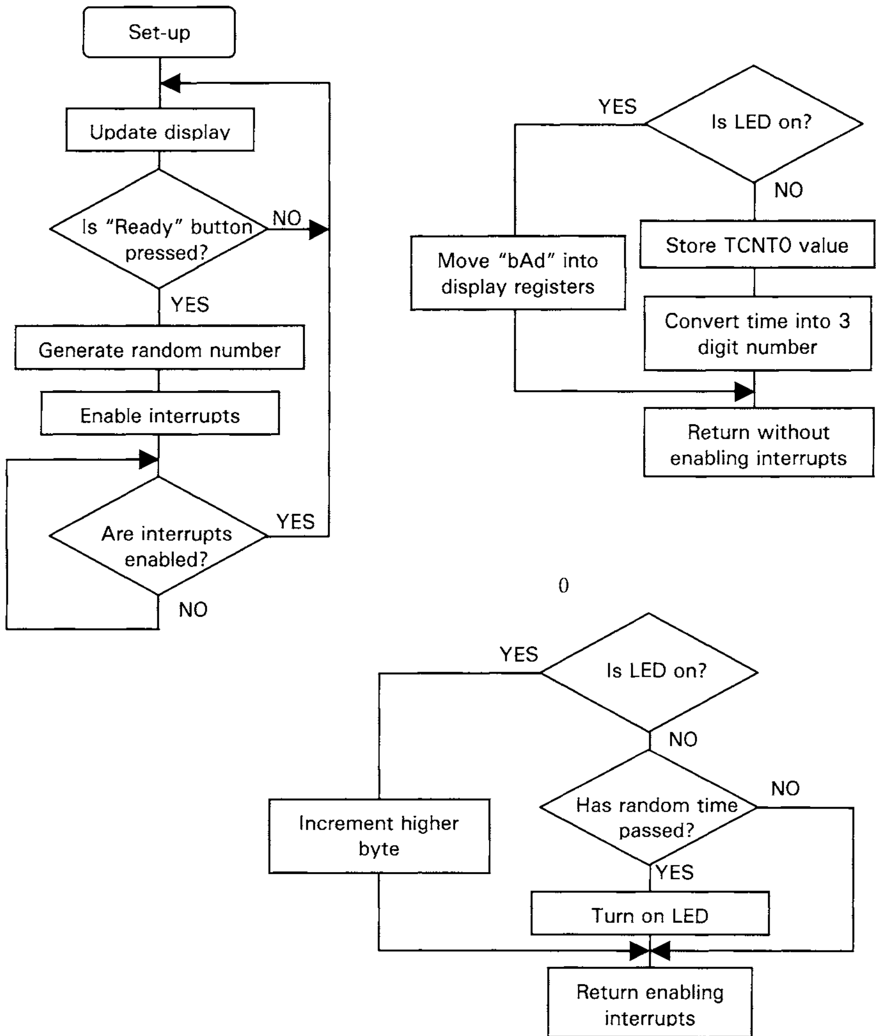

Figure 4.2

The **Display** subroutine will be almost exactly like the one in the frequency counter project. The only difference lies in the selection of the correct display. Instead of rotating between bit 0 and bit 2 of Port D, this part of the subroutine will have to rotate between bit 4 and bit 6, testing bit 7 to see when it has gone too far. Make the necessary changes to the subroutine and copy it in. We now need to create a random time delay.

Random digression

One of the interesting aspects of this program will be the generation of the random number to produce a time delay of random length. The most straightforward method for generating random numbers is to rely on some human input and convert this into a number. For example, we could look at the number in T/C0 when the 'Ready' button is pressed. T/C0, if counting internally, will be counting up and overflowing continuously, and so its value when the button is pressed is likely to be random. Very often, however, we don't have the luxury of a human input, and so we have to generate a string of random numbers. How is this done? There are a large number of algorithms available for generating random numbers, varying in complexity. We are restricted in the complexity of the functions we can straightforwardly apply using AVR assembly language, but fortunately one of the more simple algorithms relies purely on addition and multiplication. The *Linear Congruential Method* developed by Lehmer in 1948[1] has the following form:

$$I_{n+1} = \text{mod}_m(aI_n + c)$$

This generates the next number in the sequence by multiplying the previous number by a, adding c, and taking the result modulo m. $\text{mod}_m(x)$ is equal to the remainder left when you divide x by m. Conveniently, the result of every operation performed in an AVR program is effectively given in modulo 256. For example, we add 20 to 250. The 'real' answer is 270; however, the result given is 14. 14 *is* '270 modulo 256' or $\text{mod}_{256}(270)$. There are a number of restrictions on the choice of $\boldsymbol{a}$ and $\boldsymbol{c}$ in the above equation that maximize the randomness of the sequence (see the reference for more info). Given that the quickest algorithm is that with the smallest multiplier ($\boldsymbol{a}$), we will choose $\boldsymbol{a} = \mathbf{5}$ and $\boldsymbol{c} = \mathbf{1}$. You also have to pick a 'seed' – the first number in the sequence (I_0). You can set this model up on a spreadsheet and examine its *quasirandom* properties. First, you should notice that the randomness of the sequence does not appear sensitive to the seed; there is therefore no need to pick a particular one. You will also notice the sequence repeats itself every 256 numbers – this is an unfortunate property of the algorithm. Picking a larger modulus will increase the repetition period accordingly. We could use modulo 65 536 by using one of the 2-byte registers (X, Y or Z) and the **adiw** instruction. This would result in a sequence that repeats only every 65 536 numbers! For our purposes with the reaction tester, a period of 256 is quite acceptable.

To convert this random number into a random time we do the following. The maximum time is 10 seconds, and the T/C0 will overflow every 256 counts = 256/2400 = 0.066 second. We therefore would like a counter with a value roughly 61 and 183. You might notice the difference between these numbers is

[1] See reference on random numbers in Appendix I.

not far off 128 (it is in fact 122). Our life is made a lot easier if the difference is 128, so as the times needed are quoted only as approximate figures, we can use a counter that goes from 60 to 188 which will perform adequately. To convert our random number between 0 and 255 we first divide by two, then add 60.

Returning to the program, we will use register **Random** to hold the random number. We need then to multiply this by five (add it to itself four times), and then add one to it.

EXERCISE 4.4 What *six* lines will generate the next random number?

EXERCISE 4.5 What *three* lines will copy **Random** into **CountX**, divide **CountX** by two, and then add 60.

We then need to reset the higher byte of the timer (**TimeH**), turn off the displays (clear **PortB**), reset all the interrupt flags, and then set the master interrupt enable.

EXERCISE 4.6 Which *six* lines will reset **TimeH**, **PortB** and the interrupt flags?

There is a particular instruction for setting the master interrupt enable:

```
        sei                        ; Sets the interrupt enable bit.
```

The rest of the program is a loop which just tests the interrupt enable bit, and loops back to **Start** when it has been cleared. This is because after an External INT0 interrupt, the master interrupt bit will *not* re-enable interrupts and upon returning the program will loop back to **Start**. In contrast, after a T/C0 related interrupt the interrupts *will* be re-enabled so the program will stay in the loop.

EXERCISE 4.7 What *three* lines finish off the main body of the program?

Looking first at the T/C0 overflow interrupt handling routine (**TInt**), we see that the first test is to see whether or not the LED (PinD, 0) is on. If it is off we should be timing out the random time to see when to turn it on. If it is already on we should be incrementing the higher byte of our timing registers (**TimeH**). If the time exceeds the maximum that can be displayed on the scope, we should move '-HI' into the display registers and return without enabling interrupts.

The T/C0 is counting up 2400 times a second (with a register counting the higher byte as well). We need to convert this to milliseconds (i.e. something counting 1000 times a second). To do this we can multiply the 2-byte number by 5 and then divide by 12. Applying the reverse procedure to 999 (the maximum response time) we get 2397 = 95D. It would be much easier if we were testing only to see if the higher byte had reached a certain value (e.g. A00). This is easy to do by resetting T/C0 to 0xA2 when the LED is turned on, and

then subtract the 0xA2 back off the final answer at the end of the day:

```
TInt:       sbic    PinD, 0         ; tests LED
            rjmp    TInt_LEDon      ; jumps to different section if on

            dec     CountX          ; decrements random counter
            breq    PC+2            ; skips if clear
            reti                    ; returns otherwise

            sbi     PortD, 0        ; turns on LED when time passes
            ldi     temp, 0xA2      ; initializes TCNT0 to 0xA2
            out     TCNT0, temp     ; to facilitate testing for max
            reti                    ;

TInt_LEDon:
            inc     TimeH           ; increments higher byte
            cpi     TimeH, 0x0A     ; tests for maximum time
            breq    PC+2            ; skips if the user is too slow
            reti                    ;
            ldi     Hundreds, 13    ; -
            ldi     Tens, 14        ; H
            ldi     Ones, 1         ; I
            ret                     ; returns without setting I-bit
```

The External INT0 interrupt handling routine is more straightforward – we will call it **ExtInt**. This also involves testing the LED first. If it isn't on this means the user has cheated by pressing the button before the LED has turned on. In this case, we move numbers 10, 11 and 12 into **Hundreds**, **Tens** and **Ones** respectively in order to display 'bAd', and then return *without* re-enabling the master interrupt bit. If the LED is on, the press is valid, and so we have to halt the T/C0 and store the current time by moving T/C0 into **TimeL**. It is possible, however unlikely, that the T/C0 overflowed just after the INT0 interrupt occurred. We therefore need to test the T/C0 overflow interrupt flag, and increment **TimeH** if it is set. Then the total reaction time (split up over **TimeL** and **TimeH**) needs to have 0xA2 subtracted from it (as this was artificially added). It must then be multiplied by 5 and divided by 12.

EXERCISE 4.8 Which *12* lines test the LED at the start of **ExtInt**, test the LED, jump to a section called **Cheat** if it isn't on, and halt the T/C0 and store the current value, incrementing **TimeH** if necessary? 0xA2 should then be subtracted from the total reaction time, and T/C0 should be restarted at CK/1024.

EXERCISE 4.9 Which *four* lines form the **Cheat** section?

After subtracting 0xA2 we need to multiply the time by 5. As the time is split over two registers we need to use the **adc** to add a carry to the higher byte if and when there is a carry:

```
         ldi    Count4, 4       ; loads a counter with 4
         mov    temp, TimeL     ; stores time in temp and tempH
         mov    tempH, TimeH    ;
Times5:  add    temp, TimeL     ; adds TimeL to itself
         adc    tempH, TimeH    ; adds TimeH and Carry to itself
         dec    Count4          ; does this 4 times
         brne   Times5          ;
```

The product is now held over **temp** and **tempH**. We then divide the result by 12. The simplest way to do this is to see how many times we can subtract 12 from the total.

EXERCISE 4.10 *Challenge!* What *nine* lines will first clear **TimeL** and **TimeH**, and then enter a loop which divides the 2-byte number stored between **temp** and **tempH** by 12, leaving the result in **TimeL** and **TimeH**. (To skip out of the loop jump to the **DigitConvert** section.)

DigitConvert converts the 2-byte number into a three-digit number (this is copied from the frequency counter with the register names changed accordingly). Instead of the **ret** instruction at the end of the section, write **rjmp Start**.

You will have to set up all the registers (R0–R14) that hold the seven segment codes in the Init section. Registers R10, R11, R12, R13 and R14 hold the codes for a 'b', 'A', 'd', '-' and 'H' respectively. You can double check you've done everything correctly by looking at Program K in Appendix J. It should be quite fun to try this one out. Of course, the simplest way of using an AVR as a reaction tester is to get a friend to hold it between your fingers and drop it, and then see how far down the chip you caught it!

Analogue comparator

Another useful feature on most of the AVRs is an analogue comparator (AC) which compares the voltages on two pins (called **AIN0** and **AIN1** = PB0 and PB1 on the 1200) and changes the state of a bit depending on which voltage is greater. This is all controlled by the **ACSR** I/O register, whose bit assignments are shown in Figure 4.3.

Bit 7 is simply an on/off switch for the AC. You should disable the AC interrupt (clear bit 3) before disabling the AC otherwise an interrupt might occur when you try to switch it off. Bits 0 and 1 dictate what triggers an AC interrupt in terms of the AC result (i.e. interrupt when the AC result changes, when it rises, or when it falls). The remaining bits are self-explanatory.

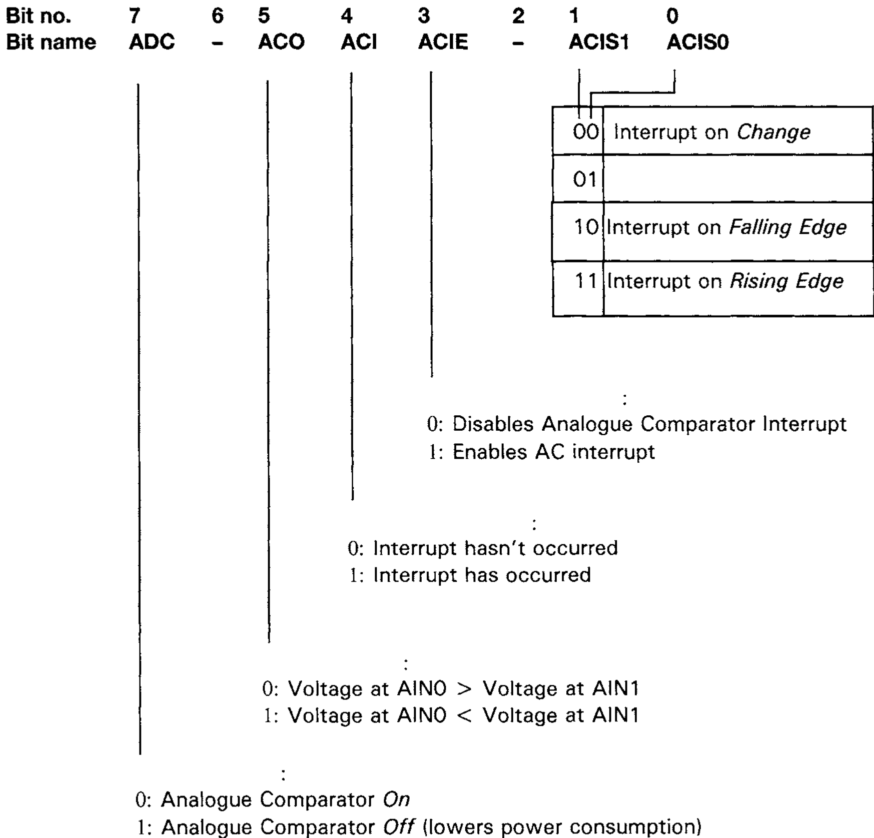

Figure 4.3

Program L: 4-bit analogue to digital converter

- Analogue comparator

This next project is very much a case of doing what you can with what you have. Some of the more advanced AVRs have full-blown 10-bit analogue to digital converters, and so with these the ability to create a 4-bit converter is clearly of limited value. However, many AVRs don't benefit from this luxury, being blessed with only a comparator, and in these cases the following program can be useful. The key to this project is using a summing amplifier to create one

of 16 possible reference voltages. By running through these reference voltages and comparing them with the input signal, we can determine the input voltage with 4-bit resolution and within four cycles of the loop. The circuit diagram is shown in Figure 4.4, pay particular attention to how the summing amplifier works. For more information on summing amplifiers, see the reference[2]. The straightforward flowchart is shown in Figure 4.5.

PD0 to PD3 control which reference voltage is being fed to the comparator, as summarized in Table 4.3.

Table 4.3

0000	0 V	1000	2.5 V
0001	0.312 V	1001	2.812 V
0010	0.625 V	1010	3.125 V
0011	0.937 V	1011	3.437 V
0100	1.25 V	1100	3.75 V
0101	1.562 V	1101	4.062 V
0110	1.875 V	1110	4.375 V
0111	2.187 V	1111	4.687 V

Write the Init section, remembering to turn on the analogue comparator by setting bit 7 of **ACSR**. Leave the AC interrupt off. At **Start** we first set up PortD with 0b00001000. This sets the most significant bit of the voltage selector and thus feeds 2.5 V into AIN0. This is then compared with the input at AIN1. If the input is higher than the reference, bit 5 of **ACSR** will be high, otherwise bit 5 will be low. If the input is higher than the reference, the answer is greater than 1000 and so we leave bit 3 of the reference high and set bit 2. If the input is lower than the reference, the answer is less than 1000 and so we clear bit 3, and then set bit 2.

EXERCISE 4.11 Write the *five* lines which set up PortD with the initial value and then test the AC result. If the AC result is low, clear bit 3 of PortD. In either case set bit 2 of PortD.

EXERCISE 4.12 Repeat the above for the remaining bits (*eight* more lines).

EXERCISE 4.13 *Challenge!* Write the *four* lines that transfer the resulting state of PD0-3 to the output bits (PB4-7), and then loop back to **Start**.

[2] See references: *Introducing Electronic Systems*, M. W. Brimicombe (1997) Nelson Thornes.

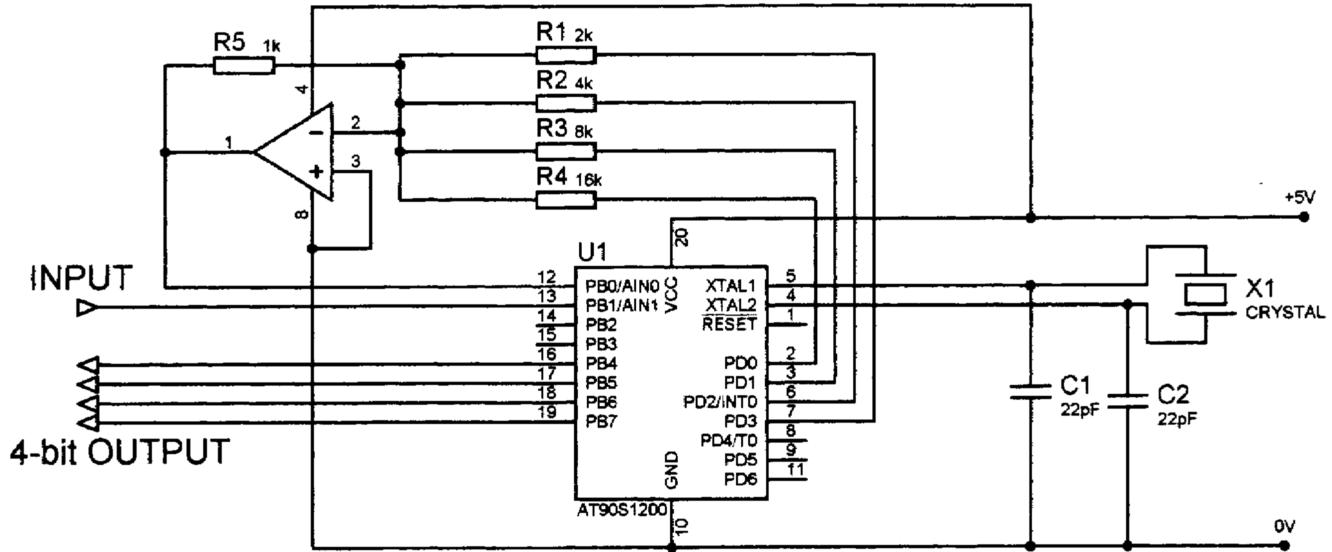

Figure 4.4

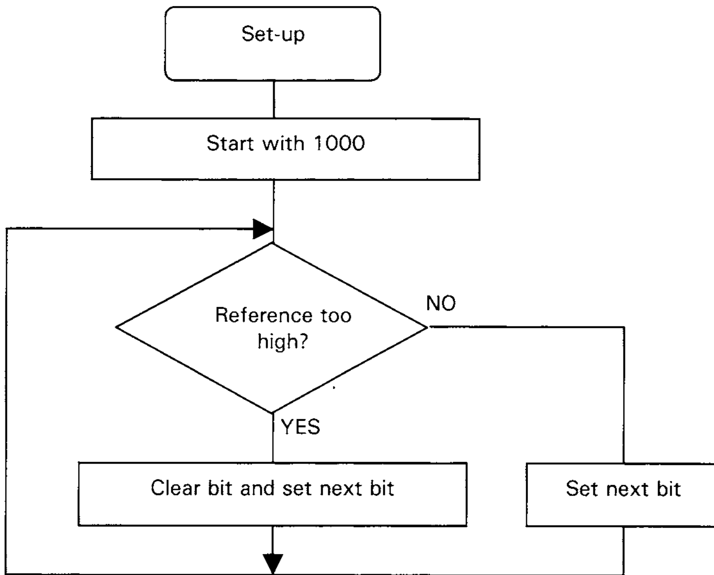

Figure 4.5

10-bit analogue to digital conversion (ADC)

Other AVR models such as the Tiny15, 4433 and 8535 have a built-in 10-bit A/D converter. This works in much the same way as the 4-bit converter we built in the previous section, except it is all done for us automatically and internally. The voltage on one of the analogue input channels is measured (with respect to the voltage on a reference pin AREF), converted into a 10-bit binary number, and stored over two I/O registers called **ADCL** and **ADCH** (which stand for **ADC** Result **L**ower byte and **ADC** Result **H**igher byte). There are two basic modes of operation: *Free Running* and *Single Conversion*. In 'Free Running' the ADC repeatedly measures the input signal and constantly updates **ADCL** and **ADCH**. In 'Single Conversion' the user must initiate every AD conversion themselves.

For the 4433 and 8535, the pin being read is selected using the I/O register called **ADMUX ($07)**. The bit assignment is shown in Table 4.4, all other bits are not used.

If you want to test a number of channels, you can change the **ADMUX** register, and the channel will be changed immediately, or, if an AD conversion is in progress, after the conversion completes. This means you can scan through channels in 'Free Running' mode more easily, as you can change the channel during one conversion, and the next conversion will be on the new channel.

The rest of the ADC settings are held in the **ADCSR** (**ADC** Status **R**egister), I/O register **$06**. The bit assignments are shown in Figure 4.6.

Table 4.4

ADMUX bits 2,1,0	Analogue input
000	Channel 0 (PA0)
001	Channel 1 (PA1)
010	Channel 2 (PA2)
011	Channel 3 (PA3)
100	Channel 4 (PA4)
101	Channel 5 (PA5)
110	Channel 6 (PA6)
111	Channel 7 (PA7)

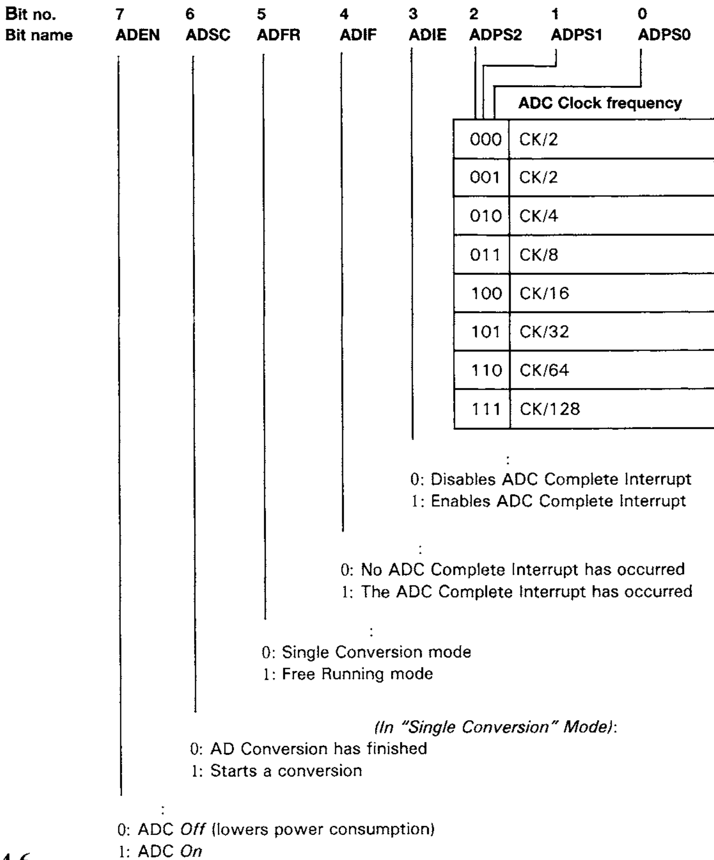

Figure 4.6

Bits 0 to 2 control the frequency of the ADC clock. This controls how long each conversion takes and also the accuracy of the conversion. A clock between 50 kHz and 200 kHz is recommended for full, 10-bit, accuracy. Frequencies above 200 kHz can be chosen if speed of conversion is more important than accuracy. For example, a frequency of 1 MHz gives 8-bit resolution, and 2 MHz gives 6-bit resolution. The ADC complete interrupt occurs (if enabled) when an ADC conversion has finished. The other bits are straightforward.

The ADC on the Tiny15 is slightly more advanced, offering features such as internal reference voltages and differential conversion (i.e. measuring the voltage difference between two pins). Moreover, in the case of the 4433 and 8535 the 10-bit ADC result is stored with the lower byte in **ADCL** and the remaining two msb's in **ADCH**. In the case of the Tiny15, you have the choice between this arrangement, and storing the upper byte in **ADCH** and the remaining two lsb's in **ADCL**. These changes all take place in the **ADMUX** register, shown in Figure 4.7.

Looking at bits 0 to 2 again, we see the option to look at the voltage difference between pins, namely ADC2 (PB3) and ADC3 (PB4). These inputs are put through a differential amplifier, and then measured using the ADC. The differential amplifier can either have a gain of x1 or x20. You will notice that two of the settings give the difference between ADC2 and itself! This is used for calibration purposes, as the differential amplifier used in the difference setting will have a small offset. By measuring this offset and subtracting from the answer of your real difference measurement, you will improve the accuracy of your result.

Another handy feature if you are interested in a high accuracy conversion is to send the chip to sleep and perform an AD conversion whilst in sleep. This helps eliminate noise from the CPU (central processing unit) of the chip. An ADC complete interrupt can then be used to wake up the chip from sleep. This method is demonstrated in Example 4.1.

Example 4.1

```
ldi    temp, 0b10001011  ; enables ADC, Single Conversion
out    ADCSR, temp       ; enables AD Complete Interrupt
ldi    temp, 0b00101000  ; enables sleep,
out    MCUCR             ; 'AD Low Noise mode'
sleep                    ; goes to sleep – this automatically
                         ;    starts AD Conversion
```

When the AD conversion completes, the AD conversion interrupt routine will be called (address $008 on the Tiny15, and address $00E on the 4433 or 8535), when the program returns from the routine it will carry on from the line after the sleep instruction.

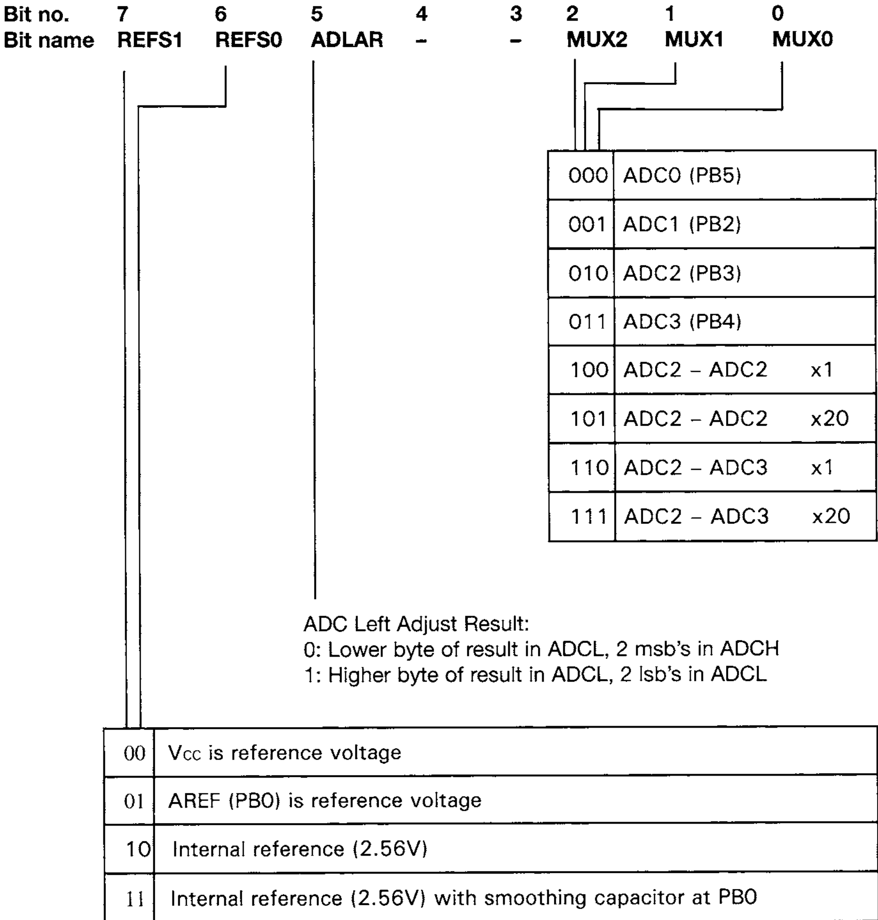

Figure 4.7

Program M: voltage inverter

- Analogue to digital conversion
- Digital to analogue conversion

We can use ADCs to make digital to analogue converters. The trick to this is to use the output to charge up a capacitor until it reaches the required output voltage. The AVR's output then goes open circuit (turns itself into an input). The capacitor will then slowly discharge through the input impedance of whatever is reading it, lowering the analogue output. Meanwhile another input is moni-

toring the voltage of the analogue output. If it falls below a certain mark, the AVR's output is turned on again to top up the analogue output. To lower the analogue voltage, the AVR output is cleared to 0 to discharge the capacitor quickly. Figure 4.8 illustrates this technique, though the jaggedness of the final output is exaggerated.

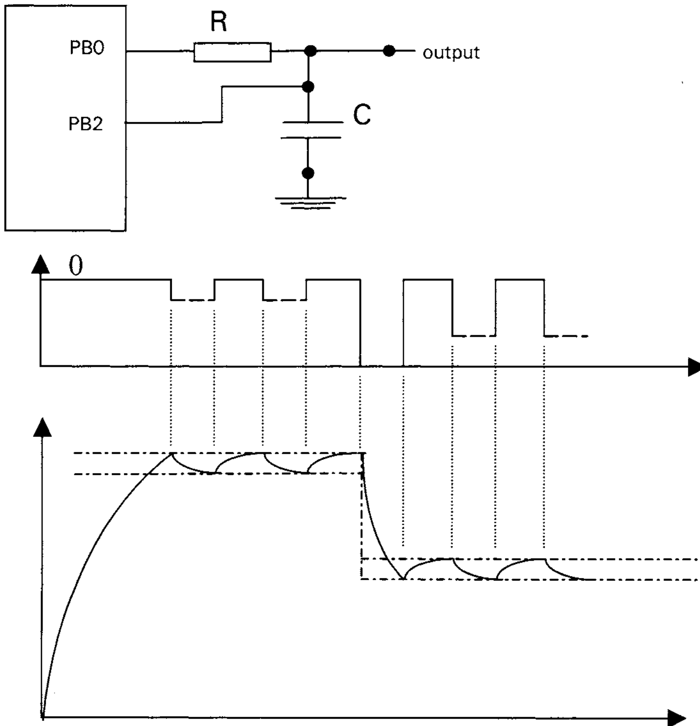

Figure 4.8

R should be made small enough to allow quick response time, and *C* high enough to give a smooth output. We will demonstrate this with a project that takes an input, *i*, between 0 and 5 V, and outputs (5 – *i*). For example, 2 V in becomes 3 V out. The circuit diagram is shown in Figure 4.9, and the flowchart in Figure 4.10.

In the Init section, we will have to enable A/D conversion, and select ADC0 to start with. We would like maximum accuracy, and so require a clock speed that is less than 200 kHz. We will be using the internal oscillator which runs at 1.6 MHz. This means that an ADC clock of CK/8 (200 kHz) will be acceptable. The ADC should be single mode, and set the 'Left Adjust' bit so that the upper byte of the ADC result is in ADCH and the two lsbs in ADCL. Finally, let V_{CC} be the reference voltage, and start an AD conversion.

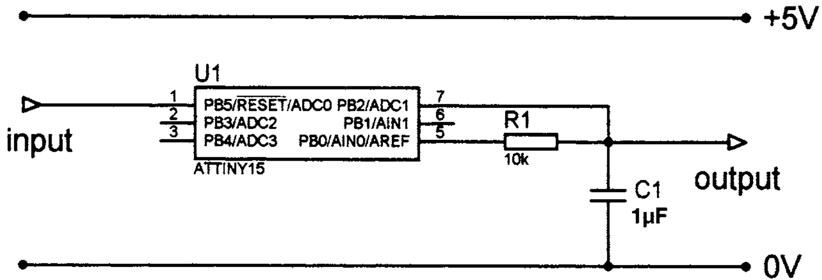

Figure 4.9

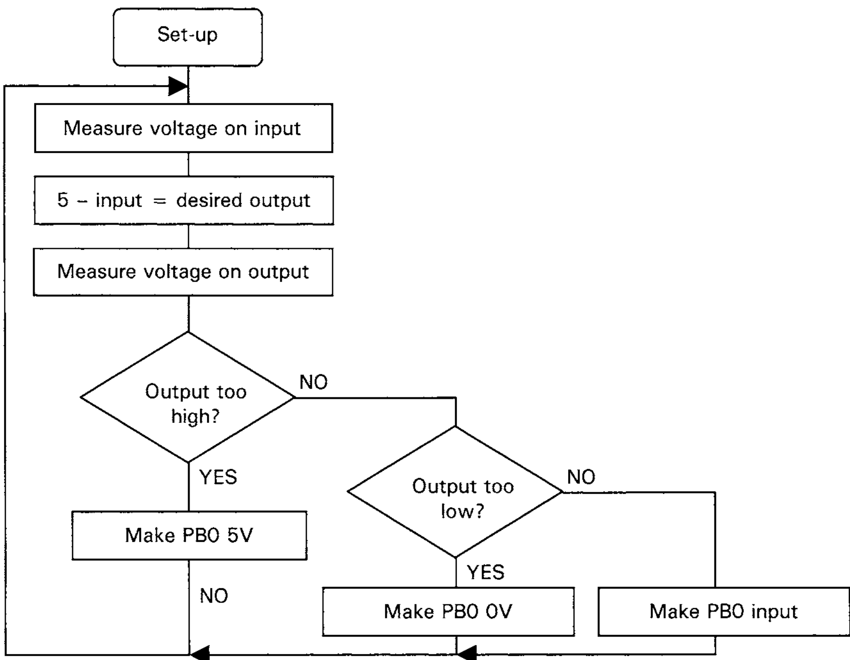

Figure 4.10

EXERCISE 4.14 What numbers should be moved into **ADCSR** and **ADMUX** in the Init section?

Write the whole of the Init section. Initially make PB0 an output. PB5 and PB2 should be inputs. Once the AVR reaches Start, the ADC0 channel should be

selected (by clearing ADMUX, bit 0), and an A/D conversion should be started (by setting ADCSR, bit 6). When the A/D conversion is over, this bit will be cleared, so we can test this bit and wait for it to set.

EXERCISE 4.15 What *four* instructions start an A/D conversion on ADC0 and wait for it to complete?

Once the conversion is complete, the input voltage will be stored in registers ADCL and ADCH. There is no need for the full 10-bit accuracy, and so we will simply use 8 bits. With Left Adjust enabled, this simply involves reading the number from ADCH. To perform the function (5 – input voltage) we simply invert the result (ones become zeros and vice versa). Invert the results using the **com** instruction, and store the result in a working register called **Desired** (this represents the voltage we *want* on the output).

EXERCISE 4.16 Which *six* instructions store and invert the measurement of the input voltage, change the input channel to select ADC1, and start a new conversion? It should also wait in a loop until the current conversion finishes.

Now the voltage on the output has been read and can be compared with the desired voltage. Save the measured voltage from ADCH into a working register called **Actual** (the *actual* voltage on the output). Then use the compare (**cp**) and branch-if-lower (**brlo**) instructions to jump to sections called **TooHigh** (the actual output is higher than the desired output), or **TooLow** (the actual output is less than the desired output).

EXERCISE 4.17 Which *seven* lines perform these tests and branch out as required? If the actual and desired voltages are equal, PB0 should be made an input (by clearing DDRB, bit 0) and then the program should jump back to **Start**.

The **TooHigh** section needs to lower the output, and so PB0 is made an output (by setting DDRB, bit 0) and then made low (0V) to discharge the capacitor and lower the output. **TooLow** needs to raise the output, and so PB0 is made an output and made high (5V) to charge up the capacitor.

EXERCISE 4.18 Write the *six* lines that make up the **TooHigh** and **TooLow** sections. The end of both sections should jump back to **Start**.

That wraps up Program M. You may want to experiment a little and make the device perform more complicated functions on the input, or perhaps on *two* inputs. Perhaps you can make some form of audio mixer by summing two input channels, or subtract the left and right channels of an audio signal to get a 'pseudo-surround sound' output. As you can see, there are a number of inter-

esting projects that can be based around the above, and all on the little Tiny15 chip!

EEPROM

In addition to the RAM and program memory that we have already seen, many AVRs have an additional memory store which combines the flexibility of RAM, with the permanence of program memory. Unlike the RAM, the EEPROM will keep its values when power is removed and unlike the program memory, the EEPROM can be read and written to while the program runs. EEPROM stands for **E**lectrically **E**rasable **R**ead-**O**nly **M**emory. There are three I/O registers associated with the EEPROM:

- **EEAR** – The **r**egister which holds the **a**ddress being written to/read from the **E**EPROM
- **EEDR** – The **r**egister which holds the **d**ata to be written to/read from the **E**EPROM
- **EECR** – The **r**egister which holds **c**ontrols the **EEPROM**
  – Set bit 0 of EECR to read from the EEPROM
  – Set bit 1 of EECR to write to the EEPROM

The 1200 has 64 bytes of EEPROM, though other AVRs can have much more (up to 512 bytes). The write operation takes a certain amount of time. To wait until the writing process is over, test bit 1 of **EECR** (the one you set to start the write) – when the writing finishes the bit is cleared automatically.

Example 4.2 To write the number 45 to EEPROM address 0x30, we would write the following:

```
        ldi     temp, 0x30      ; sets up address to write to
        out     EEAR, temp      ;
        ldi     temp, 45        ; sets up data to write
        out     EEDR, temp      ;
        sbi     EECR, 1         ; initiates write
EEWait: sbic    EECR, 1         ; waits for write to finish
        rjmp    EEWait          ; loops until EECR, 1 is cleared
```

Example 4.3 To read address 0x14 of the EEPROM we write the following. At the end of the segment of code, the data held in address 0x14 will be in **EEDR**.

```
        ldi     temp, 0x14      ; sets up address to read
        out     EEAR, temp      ;
        sbi     EECR, 0         ; initiates read
                                ; data now held in EEDR
```

EXERCISE 4.19 *Challenge!* Write a routine which sets up addresses 0x00 to 0x0F of the EEPROM to be an ASCII look-up table. This means address 'n' of the EEPROM holds the ASCII code for the 'n' character (i.e. the code for numbers 0–9, A, B, C, D, E and F). The ASCII codes for the relevant characters are given in Appendix G. The routine should be *14* lines long.

There are two ways to program the EEPROM when you are programming your chip. In AVR Studio, you can go to View → New Memory View (Alt + 4) and select EEPROM. This will give you a window with EEPROM memory locations. Simply type in the values you wish to program into the EEPROM, and when you select the programmer (e.g. STK500), select 'Program EEPROM' and choose 'Current Simulator/Emulator Memory'. This will load the contents of the EEPROM window onto the EEPROM of the chip. An easier way is to specify what you want to write to the EEPROM in your program itself. Use the **.eseg** directive (**E**EPROM **seg**ment) to define EEPROM memory. What you write after that will be written to the EEPROM. If you want to write normal code after this, you must write **.cseg** (**c**ode **seg**ment).

Example 4.4

```
.eseg                    ; writes what follows to the EEPROM
.db   0x04, 0x06, 0x07   ;
.db   0x50               ;

.cseg                    ; writes what follows to the program
                         ;   memory
ldi   temp, 45           ;
```

The **.db** directive stores the byte(s) which follow to memory. This particular code writes 0x04, 0x06, 0x07 and 0x50 to memory locations 00–03 in the EEPROM. Note that this is *not* a way to change the EEPROM during the running of the programming – it is only a way to tell the programmer what to write to the EEPROM *when you are programming the chip*. Directives such as **.org** can be used to select specific addresses in the EEPROM. On the 1200, which doesn't support the **lpm** instruction, it is a better use of resources to store the seven segment look-up table in the EEPROM, than in registers R0–R10, as previously done.

16-bit timer/counter 1

Some AVRs, such as the 2313, have a separate 16-bit timer/counter in addition to the 8-bit TCNT0. This is called Timer/Counter 1, and is quite useful as the need for markers and counters to time natural time lengths becomes greatly reduced. The number in Timer/Counter 1 (T/C1) is spread over two I/O registers: **TCNT1H** (higher byte) and **TCNT1L** (lower byte). The T/C1 can be

prescaled separately to T/C0 (i.e. it can be made to count at a different speed), and can also be made a counter of signals on its own input pin: T1 (as opposed to T0 which is the T/C0 counting pin). If the T/C1 is counting up at 2400 Hz, the 16 bits allow us to time up to 27 seconds without the need for any further counters. One very important point to note with this 2-byte timer/counter is that when you read T/C1, the 2 bytes must be read *at the same time*, otherwise there is a chance that in between storing the lower and higher bytes, the lower byte overflows, incrementing the higher byte, which lead to a large error in the stored answer. In order to do this you must therefore *read the lower byte first*. When you read in the **TCNT1L**, the number in **TCNT1H** is at the same time automatically stored in an internal TEMP register on board the AVR. When you then try to read in **TCNT1H**, the value read is taken from the TEMP register, and not from **TCNT1H**. Note that the internal TEMP register is completely separate to the working register R16 which we often call **temp**.

Example 4.5 Read Timer/Counter 1 into two working registers, **TimeL** and **TimeH**.

```
Value in T/C1
0x28FF    in    TimeL, TCNT1L   ; stores FF in TimeL, and stores 0x28
                                ;   into the internal TEMP reg.
0x2900    in    TimeH, TCNT1H   ; copies TEMP into TimeH
```

Therefore, even if T/C1 changes from 0x28FF to 0x2900 in between reading the bytes, the numbers written to **TimeL** and **TimeH** are still 0x28 and 0xFF, and *not* 0x28 and 0x00.

Similarly, when writing a number to both the higher and lower registers *you must write to the higher byte first*. When you try to write a number to **TCNT1H**, the AVR stores the byte in the internal TEMP register and then, when you write the lower byte, the AVR writes both bytes *at the same time*.

Example 4.6 Write 0x28F7 to the Timer/Counter 1.

```
ldi   TimeL, 0x28     ;
ldi   TimeH 0xF7      ;
out   TCNT1H, TimeH   ; writes 0x28 into internal TEMP reg.

out   TCNT1L, TimeL   ; writes 0xF7 to TCNT1L and 0x28 into
                      ;   TCNT1H at the same time
```

The T/C1 has some other 2-byte registers associated with it, such as **ICR1H, L** and **OCR1AH, L**, and they must be written to and read from in the same way as **TCNT1H, L**. The functions of these registers are discussed in the next two sections.

Input capture

Let's say, for example, that we wish to measure the time until an event occurs on a certain pin (as we had to do with the frequency counter project). We could just test the pin and then read the T/C1 as we did before, but in order to simplify the program and free up the processor on the chip, we can use a handy feature that captures the value in T/C1 for us. The *input capture* feature automatically stores the value in T/C1 into two I/O registers: **ICR1H** (Input **C**apture **R**egister for Timer/Counter **1**, **H**igher byte) and **ICR1L** (**L**ower byte) when an event occurs on the ICP (**I**nput **C**apture **P**in), which is PD6 on the 2313. This event can be a rising or falling edge. The input capture feature is controlled by an I/O register called **TCCR1B** (one of the two **T**imer **C**ounter **1** **C**ontrol **R**egisters) – the other control register for T/C1 is called **TCCR1A** and will be discussed in the next section.

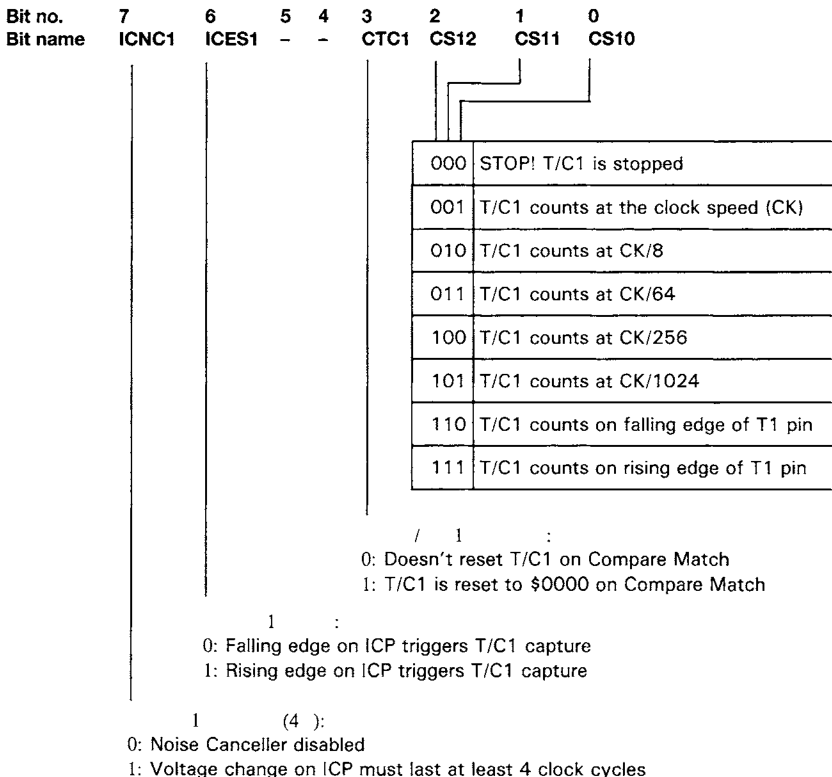

Figure 4.11

Bit 7 can be used to make the feature more robust to noise on the ICP pin. If this feature is enabled, the voltage must rise from logic 0 to 1, for example, and stay at logic 1 for at least four clock cycles. If the voltage drops back to logic 0 before the four clock cycles have passed, the signal is rejected as a glitch, and there is no input capture. If you are trying to read signals that will be less than four clock cycles, you will have to disable this noise cancelling feature (clear the bit). Bit 3 refers to the *output compare* function which is introduced in the next section. There is an input capture interrupt to let us know when an input capture has occurred. This calls address $003 (on the 2313). The enable bit is bit 3 of **TIMSK**.

Example 4.7 Input capturing could be used in a speedometer on a bicycle, where a magnet would pass by the sensor with every revolution of the wheel. The speed of the bike could be deduced as a function of the time between each revolution. The magnetic sensor could be attached to the ICP pin, which would go high every time the magnet passes over the sensor. We would want to be able to measure times up to about 1 second, which means prescaling of the CK/256 would be ideal. You may wish to remind yourself of the 2313 interrupt vector table in Appendix E. The skeleton of a speedometer program is shown below:

```
            rjmp   Init           ; address $000
            reti                  ; $001 – not using INT0 interrupt
            reti                  ; $002 – not using INT1 interrupt

IC_Int:                           ; $003 – the Input Capture interrupt
            in     temp, ICRL     ; stores captured value in working
            in     tempH, ICRH    ;    registers
            sub    temp, PrevL    ; finds different between old and new
            sbc    tempH, PrevH   ;    values
            mov    PrevL, ICRL    ; stores new values
            mov    PrevH, ICRH    ;
            rcall  DigConvert     ; converts two-byte time into digits
            reti                  ;

Display:    etc.                  ; left for you to write
            ret

DigConvert: etc.                  ; left for you to write
            ret

Init:       ldi    temp, 0b11000100    ; enables noise canceller
            out    TCCR1B, temp        ; T/C1 counts at CK/256
            ldi    temp, 0b00001000    ; enables TC interrupt
            out    TIMSK, temp         ;
```

```
          sei                           ; enables global interrupt
          etc.                          ; left for you to write

Start:    rcall   Display               ; keeps updating the displays
          rjmp    Start                 ; loops
```

The display and digit-convert subroutines are not included, but it is expected that you could write them based on the similar display routines in previous example projects. Note that the **DigConvert** subroutine should convert the number held over **temp** and **tempH** (i.e. the difference between the two times) into the digits to be displayed. The remainder of the **Init** section should also be completed – this sets up the inputs and outputs. Note that even though we are not using the interrupts that point to addresses $001 and $002, we still need instructions for those addresses. We could just use **nop** (**no** **op**eration, i.e. do nothing), but **reti** is safer. The idea is that if by some unforeseeable error an INT0 interrupt is triggered, the program will simply return, and no damage will be done. This is a basic example of *defensive programming* – i.e. expect the unexpected.

Output compare

In almost any application of the timer/counters, you are testing to see if the timer/counter has reached a certain value. Fortunately, all chips with a 'Timer/Counter 1' have a built-in feature which does this automatically. We can ask the AVR to continually compare the value in T/C1 with a certain 16-bit value. When T/C1 is equal to this value, an interrupt can occur, or we can change the state of one of the output pins, and we can also make the T/C1 reset (see bit 3 of the **TCCR1B** register shown on page 119). On the 2313, for example, the value that is to be compared with T/C1 is stored over two I/O registers: **OCR1AH** and **OCR1AL** (which stand for **O**utput **C**ompare **R**egister **A** for T/C1, **H**igher and **L**ower bytes respectively). The 'A' is to distinguish them from a second set of output compare registers (labelled 'B') that are found in other chips such as the 8515. The 8515, for example, can therefore constantly compare T/C1 with *two* different values. If we wish to use the output compare feature we will need to enable the Output Compare Interrupt, which occurs when **TCNT1H** = **OCR1AH** and **TCNT1L** = **OCR1AL**. The enable bit for this interrupt is in **TIMSK**, bit 6. The interrupt address varies between different models, but for the 2313 the output compare interrupt calls address $004. We will find the output compare feature very useful in the next project, and in the next chapter we will see how it can be used for PWM (pulse width modulation).

EXERCISE 4.20 *Challenge!* If we want an interrupt to occur every second, and we are using a 4 MHz oscillator, suggest numbers that should be moved into the following registers: **TCCR1B**, **TIMSK**, **OCR1AH**, **OCR1AL**.

Major program N: melody maker

- EEPROM
- Output compare
- Sounds

By driving a speaker at a certain frequency, we can use the AVR to create musical notes. In fact, using a square wave actually creates a more natural sound than a sine wave input. This end-of-chapter project will allow the user to program short melodies into the EEPROM of the chip, and then play them back through a speaker. The relation between some musical notes and frequencies is shown in Table 4.5.

Table 4.5

C	C#	D	D#	E	F
128 Hz	136 Hz	144 Hz	152 Hz	161 Hz	171 Hz
F#	**G**	**G#**	**A**	**A#**	**B**
181 Hz	192 Hz	203 Hz	215 Hz	228 Hz	242 Hz

The values for the next highest octave can be obtained by doubling the frequency. For example, the next 'C' will be at 256 Hz. Assuming we use four octaves, we can encode the note as the letter (which needs 4 bits) and the octave number (which needs 2 bits). The length of the note will be encoded in the remaining 2 bits. Each note in the melody will therefore take up 1 byte of EEPROM. The 2313 has 128 bytes of EEPROM, which means we can store a 128-note melody. If longer tunes are required, a chip with more EEPROM can be used instead, such as the 8515. The note will be encoded as shown in Figure 4.12.

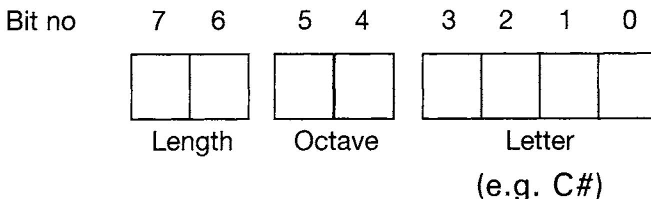

Figure 4.12

The circuit will simply consist of a speaker attached to PB0 (and the usual crystal oscillator on XTAL1 and XTAL2). The AVR can drive a speaker on its own, as long it has a relatively high impedance (e.g. 64 ohm). If you are using a lower impedance speaker (e.g. 8 ohm) you might be better off driving it with a transistor. The flowchart is shown in Figure 4.13; notice how the entire program is interrupt oriented and the main body of the program will simply be a loop.

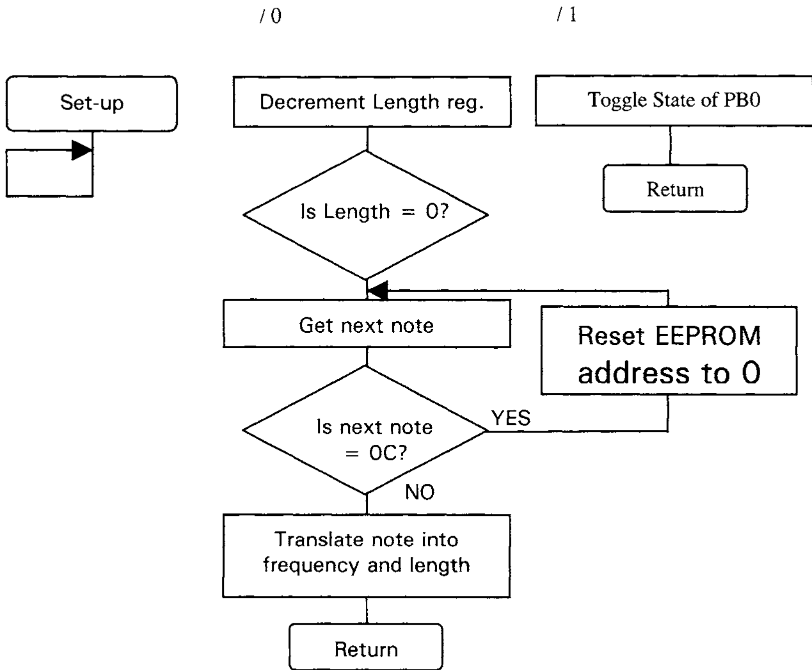

Figure 4.13

A 'note letter' value between 0x0 and 0xB will correspond to a note between 'C' and 'B'. The value 0xC in the 'note letter' part of the EEPROM byte will indicate the end of the melody and cause the chip to return to the start of the melody and repeat it over again. You may want to add extra functionality by including 0xD in the 'note letter' part of the byte, meaning end the melody and *do not* loop back (i.e. just wait until a manual reset), but this is not included in my version of the program. In the Init section, configure the inputs and outputs, the timing registers, and the stack pointer register (**SPL**). Enable the T/C0 Overflow and T/C1 Output Compare interrupts. The T/C1 will be used to create a signal of a certain frequency on the speaker pin, whilst T/C0 will be used to regulate the length of the note. Therefore, set up T/C0 to count at CK/1024, and T/C1 to count at CK. In the Init section you will also have to set up the first note; call a subroutine **Read_EEPROM** to do this, we will write the subroutine later.

At **Start:** you need only write one instruction which loops back to **Start**. Whenever the T/C1 Output Compare interrupt occurs the output will have to change state. This simply involves reading in PortB into **temp**, inverting it, and then outputting it back into PortB.

EXERCISE 4.21 Write the *four* lines which make up the T/C1 Output Compare interrupt section. Include a link to this section at address $004 in the program memory.

All that remains is the T/C0 Overflow interrupt section. **Length** will be a working register we use to keep track of the length of the note. At the start of the section, decrement **Length**. If it isn't zero just return; if it is, skip the return instruction and carry on. If sufficient time has passed, we need to change the note, but first there must be a short pause. This pause allows us to repeat the same note twice without making it sound like a single note played for twice as long. An easy way to insert a pause is simply to wait for the T/C0 Overflow interrupt flag to go high again. If it is, skip out of the loop, reset the flag and move on to the section that reads the next note. Call this section **Read_EEPROM**.

EXERCISE 4.22 Write the *eight* lines at the start of the T/C0 Overflow interrupt section. Include a link to this section at address $006.

The **Read_EEPROM** section copies the number in a working register called **address** into **EEAR**. Read the EEPROM into the **ZL** register, and mask bits 4–7, selecting the 'note letter' part of the byte. Then compare ZL with the number 12 (0xC); if it is equal, jump to a section called **Reset**. If it isn't equal test to see if it is less than 12 (**brlo**). If it isn't less (i.e. it is greater than 12) it is an invalid note letter, and so ZL should be reset to 0x0, for want of a better note. If it is less than 12, skip that instruction.

EXERCISE 4.23 Write the first *eight* lines of the **Read_EEPROM** section.

We will be using ZL to read values from a look-up table in the program memory (using the **lpm** instruction). As you may remember, **lpm** uses the *byte address* of the program memory, rather than the *word address*, so we need to multiply ZL by two (using the **lsl** instruction). The look-up table will start at *word address* 013. We can ensure this using the **.org** directive in AVR Studio. This says 'Let the next instruction be placed at address …'. Our look-up table starts as shown below (**.dw** is the directive which puts the word or words which follow in the program memory).

```
.org 13
.dw  0x7A12       ; frequency for C     (word address 013)
```

```
.dw  0x7338        ; frequency for C#    (word address 014)
etc.
```

We must therefore add 26 to ZL to correctly address the look-up table. Use **lpm** to read the lower byte, and move the result from R0 into a working register called **NoteL**. Then increment ZL and do the same, moving the result into **NoteH**.

EXERCISE 4.24 What *seven* lines perform this task?

We will need to perform some basic maths to derive the values for the look-up table. Taking the frequencies of the lowest octave to be played, shown in Table 4.5, and dividing by 4 000 000 (the oscillator frequency) by these values, we get a set of numbers indicating the numbers with which we wish to compare T/C1. To get higher octaves we will simply divide these values by two. My values are shown in the full version of the program in Appendix J; you may wish to check them, or else you can simply copy them.

To get the correct octave we again copy **EEDR** into **temp**, swap the nibbles, and then mask bits 2–7, leaving us with the 2 bits we are interested in – those that choose the octave. Label the next line **GetOctave**. First test if the result of the AND operation just performed is 0; if it is we can just move on to the next section – **GetLength**. If it isn't 0, we will divide the number spread over **NoteH** and **NoteL** by two, decrement **temp**, and then loop back to **GetOctave**.

EXERCISE 4.25 Write the *eight* lines that use bits 4 and 5 of the EEPROM byte to alter the frequency according to a specified octave.

NoteH and **NoteL** are now ready to be moved into **OCR1AH** and **OCR1AL**, but remember to write the *higher byte first*. We then read the length, using a similar method to **GetOctave**. Again read the **EEDR** into **temp**, mask bits 5–0, swap the nibbles, and rotate once to the right. This places the relevant bits in bits 1 and 2 of **temp**. This means the number in **temp** is 0, 2, 4 or 6. This is almost what we want, and by adding 2 to **temp** we get 2, 4, 6 or 8. This should be moved into **Length**.

EXERCISE 4.26 What *nine* lines make up the **GetLength** section and return from the subroutine, enabling interrupts.

The program is now finished. By programming different values into the EEPROM when you program the chip, it can be made to produce any tune. You may find a spreadsheet useful in converting notes, octaves and lengths into the hex number which represents them. You may also want to look into ways to input bytes to the EEPROM more easily. For example, you could use an array of push buttons in a keyboard arrangement, strobing them to lessen the number

of inputs needed, to input the melody. Another method might involve a seven segment display to display the note, with a series of buttons to scroll through the memory and change the note – this would require less skill as a pianist to enter the tune!