1
Introduction

An AVR is a type of microcontroller, and not just any microcontroller – AVRs are some of the fastest around. I like to think of a microcontroller as a useless lump of silicon with amazing potential. It will do nothing without but almost anything with the program that you write. Under your guidance, a potentially large conventional circuit can be squeezed into one program and thus into one chip. Microcontrollers bridge the gap between hardware and software – they run programs, just like your computer, yet they are small, discrete devices that can interact with components in a circuit. Over the years they have become an indispensable part of the toolbox of electrical engineers and enthusiasts as they are perfect for experimenting, small batch productions, and projects where a certain flexibility of operation is required.

Figure 1.1 shows the steps in developing an AVR program.

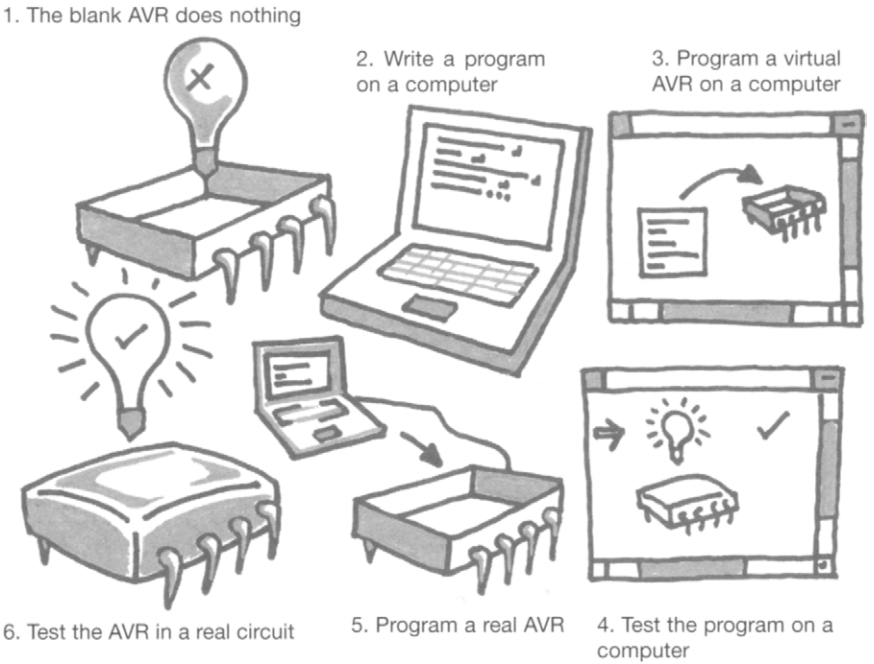


Figure 1.1

The AVR family covers a huge range of different devices, from Tiny 8-pin devices to the Mega 40-pin chips. One of the fantastic things about this is that you can write a program with one type of AVR in mind, and then change your mind and put the program in a different chip with only minimal changes. Furthermore, when you learn how to use one AVR, you are really learning how to use them all. Each has its own peculiarities – their own special features – but underneath they have a common heart.

Fundamentally, AVR programming is all to do with pushing around numbers. The trick to programming, therefore, lies in making the chip perform the designated task by the simple movement and processing of numbers. There is a specific set of tasks you are allowed to perform on the numbers – these are called *instructions*. The program uses simple, general instructions, and also more complicated ones which do more specific jobs. The chip will step through these instructions one by one, performing millions every second (this depends on the frequency of the oscillator it is connected to) and in this way perform its job. The numbers in the AVR can be:

1. **Received** from inputs (e.g. using an input 'port')
2. **Stored** in special compartments inside the chip
3. **Processed** (e.g. added, subtracted, ANDed, multiplied etc.)
4. **Sent out** through outputs (e.g. using an output 'port')

This is essentially all there is to programming ('great' you may be thinking). Fortunately there are certain other useful functions that the AVR provides us with such as on-board timers, serial interfaces, analogue comparators, and a host of 'flags' which indicate whether or not something particular has happened, which make life a lot easier.

We will begin by looking at some basic concepts behind microcontrollers, and quickly begin some example projects on the AT90S1200 (which we will call 1200 for short) and Tiny AVRs. Then intermediate operations will be introduced, with the assistance of more advanced chips (such as the AT90S2313). Finally, some of the more advanced features will be discussed, with a final project based around the 2313. Most of the projects can be easily adapted for any type of AVR, so there is no need for you to go out and buy all the models.

Short bit for PIC users

A large number of readers will be familiar with the popular PIC microcontroller. For this reason I'll mention briefly how AVRs can offer an improvement to PICs. For those of you who don't know what PICs are, don't worry too much if you don't understand all this, it will all make sense later on!

Basically, the AVRs are based on a more advanced underlying architecture, and can execute an instruction every clock cycle (as opposed to PICs which execute one every four clock cycles). So for the same oscillator frequency, the AVRs will run four times as fast. Furthermore they also offer 32 working regis-

ters (compared with the one that PICs have), and about three times as many instructions, so programs will almost always be shorter. It is worth noting, however, that although the datasheets boast 90–120 instructions, there is considerable repetition and redundancy, and so in my view there are more like 50 distinct instructions.

Furthermore, what are known as special function registers on PICs (and known as input/output registers on the AVR) can be directly accessed with PICs (e.g. you can write directly to the ports), and this cannot be done to the same extent with AVRs. However, these are minor quibbles, and AVR programs will be more efficient on the whole. All AVRs have flash program memory (so can be rewritten repeatedly), and finally, as the different PICs have been developed over a period of many years there are some annoying compatibility issues between some models which the AVRs have managed to avoid so far.

Number systems

It is worth introducing at this stage the different numbering systems which are involved in AVR programming: binary, decimal and hexadecimal. A binary number is a *base 2* number (i.e. there are only two types of digit (0 and 1)) as opposed to decimal – *base 10* – with 10 different digits (0 to 9). Likewise hexadecimal represents *base 16* so it has 16 different digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F). The table below shows how to count using the different systems:

binary (8 digit)	decimal (3 digit)	hexadecimal (2 digit)
00000000	**000**	**00**
00000001	**001**	**01**
00000010	**002**	**02**
00000011	**003**	**03**
00000100	**004**	**04**
00000101	**005**	**05**
00000110	**006**	**06**
00000111	**007**	**07**
00001000	**008**	**08**
00001001	**009**	**09**
00001010	**010**	**0A**
00001011	**011**	**0B**
00001100	**012**	**0C**
00001101	**013**	**0D**
00001110	**014**	**0E**
00001111	**015**	**0F**
00010000	**016**	**10**
00010001	**017**	**11**
etc.		

The binary digit (or *bit*) furthest to the right is known as the least significant bit or *lsb* and also as *bit 0* (the reason the numbering starts from 0 and not from 1 will soon become clear). Bit 0 shows the number of 'ones' in the number. One equals 2^0. The bit to its left (*bit 1*) represents the number of 'twos', the next one (*bit 2*) shows the number of 'fours' and so on. Notice how two = 2^1 and four = 2^2, so the bit number corresponds to the power of two which that bit represents, but note that the numbering goes from right to left (this is very often forgotten!). A sequence of 8 bits is known as a byte. The highest number bit in a binary word (e.g. bit 7 in the case of a byte) is known as the most significant bit (*msb*).

So to work out a decimal number in binary you could look for the largest power of 2 that is smaller than that number and work your way down.

Example 1.1 Work out the binary equivalent of the decimal number 83.

Largest power of two less than 83 = 64 = 2^6. Bit 6 = **1**

This leaves 83 – 64 = 19	32 is greater than 19 so bit 5 = **0**,
	16 is less than 19 so bit 4 = **1**,
This leaves 19 – 16 = 3	8 is greater than 3 so bit 3 = **0**,
	4 is greater than 3 so bit 2 = **0**,
	2 is less than 3 so bit 1 = **1**,
This leaves 3 – 2 = 1	1 equals 1 so bit 0 = **1**.

So **1010011** is the binary equivalent.

There is, however, an alternative (and more subtle) method which you may find easier. Take the decimal number you want to convert and divide it by two. If there is a remainder of one (i.e. it was an odd number), write down a one. Then divide the result and do the same writing the remainder to the *left* of the previous value, until you end up dividing one by two, leaving a one.

Example 1.2 Work out the binary equivalent of the decimal number 83.

Divide 83 by two.	Leaves 41, remainder **1**
Divide 41 by two.	Leaves 20, remainder **1**
Divide 20 by two.	Leaves 10, remainder **0**
Divide 10 by two.	Leaves 5, remainder **0**
Divide 5 by two.	Leaves 2, remainder **1**
Divide 2 by two.	Leaves 1, remainder **0**
Divide 1 by two.	Leaves 0, remainder **1**

So **1010011** is the binary equivalent.

EXERCISE 1.1 Find the binary equivalent of the decimal number 199.

EXERCISE 1.2 Find the binary equivalent of the decimal number 170.

Likewise, bit 0 of a hexadecimal is the number of ones ($16^0 = 1$) and bit 1 is the number of 16s ($16^1 = 16$) etc. To convert decimal to hexadecimal (it is often abbreviated to just 'hex') look at how many 16s there are in the number, and how many ones.

Example 1.3 Convert the decimal number 59 into hexadecimal. There are 3 16s in 59, leaving 59 – 48 = 11. So bit 1 is 3. 11 is B in hexadecimal, so bit 0 is B. The number is therefore **3B**.

EXERCISE 1.3 Find the hexadecimal equivalent of 199.

EXERCISE 1.4 Find the hexadecimal equivalent of 170.

One of the useful things about hexadecimal, which you may have picked up from Exercise 1.4, is that it translates easily with binary. If you break up a binary number into 4-bit groups (called *nibbles*, i.e. small bytes), these little groups can individually be translated into 1 hex digit.

Example 1.4 Convert 01101001 into hex. Split the number into nibbles: 0110 and 1001. It is easy to see 0110 translates as 4 + 2 = 6 and 1001 is 8 + 1 = 9. So the 8-bit number is **69** in hexadecimal. As you can see, this is much more straightforward than with decimal, which is why hexadecimal is more commonly used.

EXERCISE 1.5 Convert 11100111 into a hexadecimal number.

Adding in binary

Binary addition behaves in exactly the same way as decimal addition. Examine each pair of bits.

0 + 0 = 0	no carry
1 + 0 = 1	no carry
1 + 1 = 0	carry 1
1 + 0 + 0 = 1	no carry
1 + 1 + 0 = 0	carry 1
1 + 1 + 1 = 1	carry 1

Example 1.5 4 + 7 = 11

```
    1
  0100
+ 0111
  1011 = 11 in decimal
```

EXERCISE 1.6 Find the result of 01011010 + 00001111 using binary addition.

Negative numbers

We have seen how positive decimal numbers translate into binary, but how do we translate negative numbers? We have to sacrifice a bit towards giving the number a sign, so for a 4-bit signed number, the range of values might be –7 to +8. There are various representations for negative numbers, including *two's complement*. With this method, to make a positive number onto its negative equivalent, you invert all the bits and then add one:

Example 1.6
0111 = 7
Invert all bits: 1000
Add one: 1001
1001 = –7

Example 1.7
1000 = 8
Invert: 0111
Add one: 1000
1000 = –8 = +8 *FAIL!*

As you can see in Example 1.7, we cannot use –8 because it is indistinguishable from +8. This asymmetry is recognized as an unfortunate consequence of the two's complement method, but it has been accepted as the best given the shortcomings of other methods of signing binary numbers. Let's test these negative numbers by looking at –2 + 7:

Example 1.8 2 = 0010 therefore –2 = 1110

```
  1110  = –2
+ 0111  = 7
  0101  = 5    Which is what we would expect!
```

EXERCISE 1.7 Find the 8-bit two's complement representation of –40, and show that –40 + 50 gives the expected result.

A result of this notation is that we can simply test the most significant bit (msb) to see whether a number is positive or negative. A 1 in the msb indicates a negative number, and a 0 indicates positive. However, when dealing with the result of addition and subtraction with large positive or negative numbers, this can be misleading.

Example 1.9 69 + 120 = ...

```
        1
  01000101 = + 69
+ 01111000 = + 120
  10111101 = + 189 or –67
```

In other words, in the two's complement notation, we could interpret the result as having the msb 1 and therefore negative. There is therefore a test for 'two's complement overflow' which we can use to determine the *real* sign of the result. The 'two's complement overflow' occurs when:

- both the msb's of the numbers being added are 0 and the msb of the result is 1
- both the msb's of the numbers being added are 1 and the msb of the result is 0

The *real* sign is therefore given by a combination of the 'two's complement overflow' result, and the state of the msb of the result:

Two's complement overflow?	**MSB of result**	**Sign**
No	0	Positive
No	1	Negative
Yes	0	Negative
Yes	1	Positive

As you can see from Example 1.10, there is a two's complement overflow, and the msb of the result is 1, and so the sign of the answer is positive (+189) as we would expect. You will be relieved to hear that much of this is handled automatically by the AVR.

The *one's complement* is simply the result of inverting all the bits in a number.

An 8-bit RISC Flash microcontroller?

We call the AVR an *8-bit microcontroller*. This means it deals with numbers 8 bits long. The binary number 11111111 is the largest 8-bit number and equals 255 in decimal and FF in hex (work it out!). With AVR programming, different notations are used to specify different numbering systems (the decimal number 11111111 is very different from the binary number 11111111)! A binary number is shown like this: 0b00101000 (i.e. **0b**...). Decimal is the default system, and the hexadecimal numbers are written with a **0x**, or with a dollar sign, like this: 0x3A or $3A. Therefore:

0b00101011 is equivalent to 43 which is equivalent to 0x2B

When dealing with the inputs and outputs of an AVR, binary is always used, with each input or output pin corresponding to a particular bit. A **1** corresponds to what is known as *logic 1*, meaning the pin of the AVR is at the supply voltage (e.g. +5 V). A 0 shows that the pin is at *logic 0*, or 0 V. When used as inputs, the boundary between reading a logic 0 and a logic 1 is half of the supply voltage (e.g. +2.5 V).

You will also hear the AVR called a *RISC microcontroller*. This means it is a **R**educed **I**nstruction **S**et **C**omputer, i.e. has relatively few instructions. This makes life slightly harder for the programmer (you or me), but the chip itself is more simple and efficient.

The AVR is sometimes called a *Flash microcontroller*. This refers to the fact that the program you write for it is stored in *Flash memory* – memory which can be written to again and again. Therefore you can keep reprogramming the same AVR chip – for hobbyists this means one chip can go a long way.

Initial steps

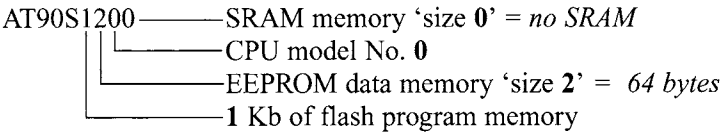
The process of developing a program consists of five basic steps:

1. **Select** a particular AVR chip, and construct a program **flowchart**
2. **Write** program (using Notepad, AVR Studio, or some other suitable development software)
3. **Assemble** program (changes what you've written into something an AVR will understand)
4. **Simulate** or **Emulate** the program to see whether or not it works
5. **Program** the AVR. This feeds what you've written into the actual AVR

Let's look at some of these in more detail.

Choosing your model

As there are so many different AVRs to choose from, it is important you think carefully about which one is right for your application. The name of the AVR can tell you some information about what it has, e.g.:



Memory sizes:

0	1	2	3	4	5	6	7	8	9	A	B
0	32 bytes	64 bytes	128 bytes	256 bytes	512 bytes	1K	2K	4K	8K	16K	32K

The meaning of these terms may not be familiar, but they will be covered shortly. The Tiny and Mega family have slightly different systems. You can get a decent overview of some of the AVRs and their properties by checking out Appendix A.

EXERCISE 1.8 Deduce the memory properties of the AT90S8515.

One of the most important features of the AVR, which unfortunately is not encoded in the model name, is the number of input and output pins. The 1200 has 15 input/output pins (i.e. they have 15 pins which can be used as inputs *or* outputs), and the 8515 has up to 32!

Example 1.10 The brief is to design a device to count the number of times a push button is pressed and display the value on a single seven segment display – when the value reaches nine it resets.

1. The seven segment display requires **seven** outputs
2. The push button requires **one** input

This project would therefore need a total of eight input/output pins. In this case a 1200 would be used as it is one of the simplest models and has enough pins.

A useful trick when dealing with a large number of inputs and outputs is called **strobing**. It is especially handy when using more than one seven segment display, or when having to test many buttons. An example demonstrates it best.

Example 1.11 The brief is to design a counter which will add a number between 1 and 9 to the current two-digit value. There are therefore nine push buttons and two seven segment displays.

It would first appear that quite a few inputs and outputs are necessary:

1. The two seven segment displays require seven outputs each, thus a total of **14**
2. The push buttons require one input each. Creating a total of **nine**

The overall total is therefore 23 input/output pins, which would require a large AVR such as the 8515 (which has 32 I/O pins); however, it would be unnecessary to use such a large one as this value can be cut significantly.

By strobing the buttons, they can all be read using only six pins, and the two

seven segment displays can be controlled by only nine. This creates a total of 15 input/output (or I/O) pins, which would just fit on the 1200. Figure 1.2 shows how it is done.

By making the pin labelled PB0 logic 1 (+5 V) and PB1, PB2 logic 0 (0 V), switches 1, 4 and 7 are enabled. They can then be tested individually by examining pins PB3 to PB5. Thus by making PB0 to PB2 logic 1 one by one, all the buttons can be examined individually. In order to work out how many I/O pins you will need for an array of X buttons, find the pair of factors of X which have the smallest sum (e.g. for 24, 6 and 4 are the factors with the smallest sum, hence 6 + 4 = 10 I/O pins will be needed). It is better to make the smaller of the two numbers (if indeed they are not the same) the number of outputs, and the larger the number of inputs. This way the program takes less time to scroll through all of the rows of buttons.

Strobing seven segment displays basically involves displaying a number on one display for a short while, and then turning that display off while you display another number on another display. PD0 to PD6 contain the seven segment code for both displays, and by making PB6 or PB7 logic 1, you can turn the individual displays on. So the displays are in fact flashing on and off at high speed, giving the impression that they are constantly on. The programming requirements of such an arrangement will be examined at a later stage.

EXERCISE 1.9 With the help of Appendix A, work out which model AVR you would use for a four-digit calculator with buttons for digits 0–9 and five operations: +, –, ×, ÷ and = .

Flowchart

After you have worked out how many I/O pins you will need, and thus selected a particular AVR, the next step is to create a program flowchart. This basically forms the backbone of a program, and it is much easier to write a program from a flowchart than from scratch.

A flowchart should show the fundamental steps that the AVR must perform and a clear program structure. Picture your program as a hedge maze. The flowchart is a rough map showing key regions of the maze. When planning your flowchart you must note that the maze cannot lead off a cliff (i.e. the program cannot simply end), or the AVR will run over the edge and crash. Instead the AVR is doomed to navigate the maze indefinitely (although you can send it to sleep!). A simple example of a flowchart is shown in Figure 1.3.

Example 1.12 The flowchart for a program to turn an LED on *if* a button is being pressed.

(The Set-up box represents some steps which must be taken as part of the start of every program, in order to set up various functions – this will be examined

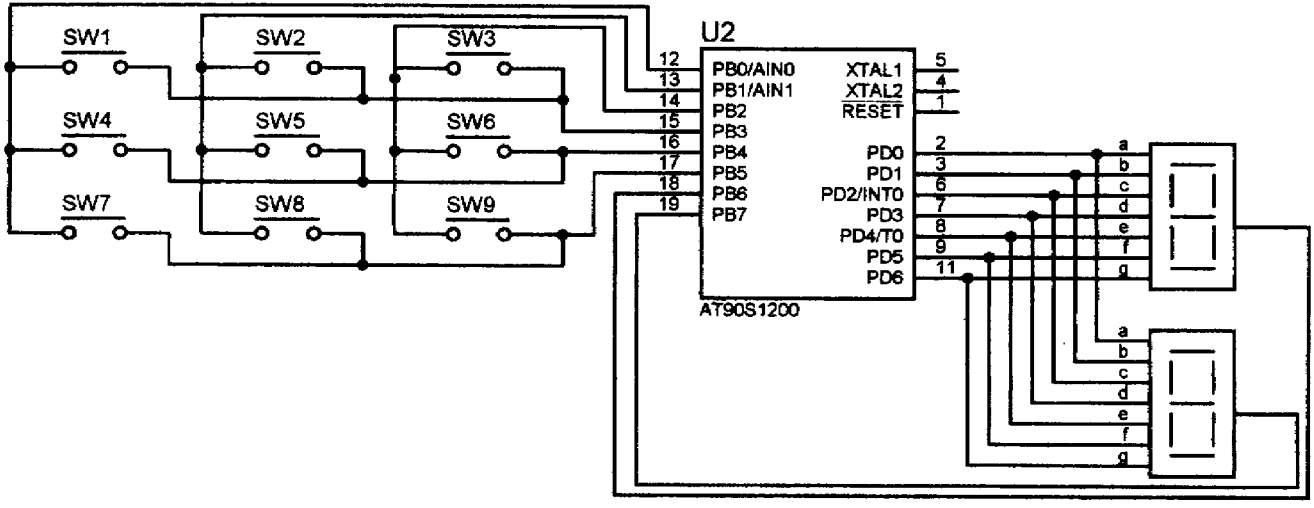

SW1
SW2
SW3
SW4
SW5
SW6
SW7
SW8
SW9
U2
12
13
14
15
16
17
18
19
PB0/AIN0
PB1/AIN1
PB2
PB3
PB4
PB5
PB6
PB7
XTAL1
XTAL2
RESET
PD0
PD1
PD2/INT0
PD3
PD4/T0
PD5
PD6
5
4
1
2
3
6
7
8
9
11
AT90S1200
a
b
c
d
e
f
g

Figure 1.2

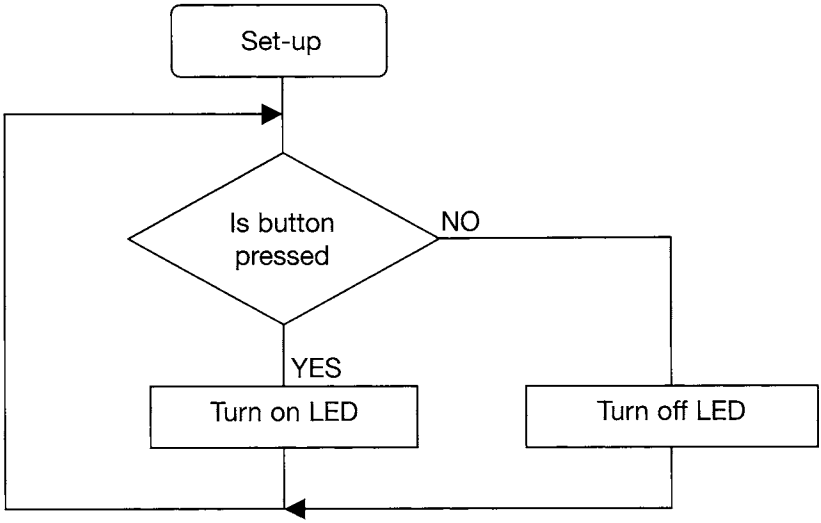


Figure 1.3

later.) Rectangles with rounded corners should be used for start and finish boxes, and diamond-shaped ones for decisions. Conditional jumps (the diamond shaped boxes) indicate '*if* something happens, *then* jump somewhere'.

The amount of code any particular box will represent varies considerably, and is really not important. The idea is to get the key stages, and come up with a diagram that someone with no knowledge of programming would understand. You will find it much easier to write a program from a flowchart, as you can tackle each box separately, and not have to worry so much about the overall structure.

EXERCISE 1.10 *Challenge!* Draw the flowchart for an alarm with three push buttons. Once the device is triggered by a pressure sensor, the three buttons must be pressed in the correct order, and within 10 seconds, or else the alarm will go off. If the buttons are pressed in time, the device returns to the state it was in before being triggered. If the wrong code is pressed the alarm is triggered. (The complexity of the answers will vary, but to give you an idea, my answer has 13 boxes.)

Writing

Once you have finished the flowchart, the next step is to load up a program template (such as the one suggested on page 19), and begin writing your program into it. This can be done on a basic text package such as Notepad (the one that comes with Microsoft Windows®), or a dedicated development environment such as AVR Studio.

Assembling

When you have finished writing your program, it needs to be *assembled* before it can be transferred onto a chip. This converts the program you've written into a series of numbers which can be fed into the Flash Program Memory of the AVR. This series of numbers is called the *hex code* or *hex file* – a hex file will have **.hex** after its name. The assembler will examine your program line by line and try to convert each line into the corresponding hex code. If, however, it fails to recognize something in one of the lines of your code, it will register an **error** for that line. An error is something which the assembler thinks is *definitely* wrong – i.e. it can't understand it. It may also register a **warning** – something which is probably wrong, i.e. definitely unusual but not necessarily wrong. All this should be made much more clear when we actually assemble our first program.

Registers

One of the most important aspects to programming with AVRs and microcontrollers in general are the *registers*. I like to think of the AVR as having a large filing cabinet with many drawers, each containing an 8-bit number (a byte). These drawers are registers – more specifically we call these the *I/O registers*. In addition to these I/O registers, we have 32 'working' registers – these are different because they are not part of the filing cabinet. Think of the working registers as the filing assistants, and yourself as the boss. If you want something put in the filing cabinet, you give it to the filing assistant, and then tell them to put it in the cabinet. In the same way, the program writer cannot move a number directly into an I/O register. Instead you must move the number into a working register, and then copy the working register to the I/O register. You can also ask your filing assistants to do arithmetic etc. on the numbers they hold – i.e. you can add numbers between working registers. Figure 1.4 shows the registers on the 1200.

As you can see, each register is assigned a number. The working registers are assigned numbers R0, R1, . . ., R31. Notice, however, that R30 and R31 are slightly different. They represent a double register called Z – an extra long register that can hold a 16-bit number (called a *word*). These are two filing assistants that can be tied together. They can be referred to independently – ZL and ZH – but can be fundamentally linked in that ZL (Z Lower) holds bits 0–7 of the 16-bit number, and ZH (Z Higher) holds bits 8–15.

Example 1.13

ZH	ZL	→	add one to ZL	→	ZH	ZL
00000000	11111111				00000001	00000000

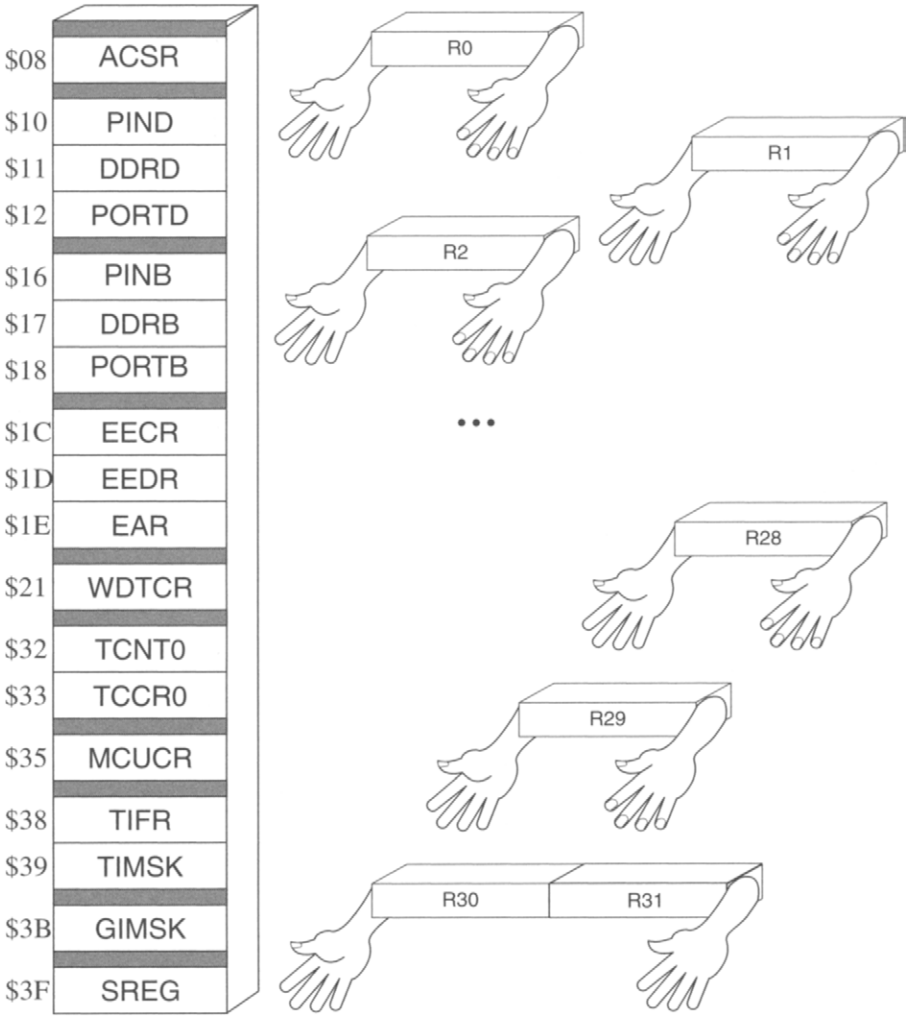


Figure 1.4

Example 1.14

ZH	ZL	→	add one to ZL	→	ZH	ZL
11111111	11111111				00000000	00000000

Note that this linking only occurs with certain instructions. Assume that an instruction *doesn't* have the linking property unless explicitly stated.

You will find it easier to give your working registers names (for the same reason you don't call your filing assistants by their staff numbers), and you will be able to do this. It is sensible to give them a name according to the meaning of the number they are holding. For example, if you were to use register R5 to store the number of minutes that have passed, you might want to call it something like **Minutes**. You will be shown how to give names to your registers shortly, when we look at the program template. We will also see later that the working registers numbers R16–R31 are slightly more powerful than the others.

The I/O registers are also assigned numbers (0–63 in decimal, or $0–$3F in hexadecimal). Each of these performs some specific function (e.g. count the passage of time, or control serial communications etc.) and we will go through the function of each one in due course. I will, however, highlight the functions of PORTB, PORTD, PINB and PIND. These I/O registers represent the ports – the AVR's main link with the outside world. If you're wondering what happened to Ports A and C, it's not really very important. All four (A, B, C and D) appear on larger types of AVR (e.g. 8515); smaller AVRs (e.g. 1200) have only two. These two correspond to the two on larger AVRs that are called B and D, hence their names.

Figure 1.5 shows the pin layout of the 1200. Notice the pins labelled PB0, PB1, . . ., PB7. These are the Port B pins. Pins PD0–PD6 are the Port D pins. They can be read as inputs, or controlled as outputs. If behaving as an input, reading the binary number in PINB or PIND tells us the states of the pin, with PB0 corresponding to bit 0 in PINB etc. If the pin is high, the corresponding bit is 1, and vice versa. Note that Port D doesn't have the full 8 bits.

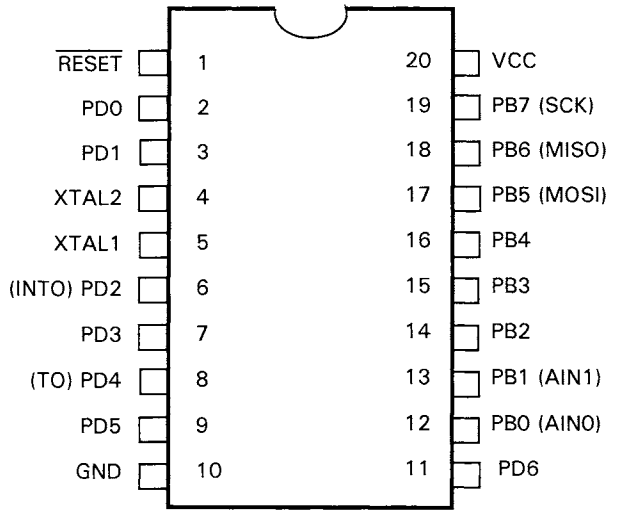


Figure 1.5

Example 1.15 All of PB0–PB7 are inputs. They are connected to push buttons which are in turn connected to the +5 V supply rail. When all the buttons are pressed, the number in PINB is 0b11111111 or 255 in decimal. When all buttons except PB7 are pressed, the number in PINB is 0b01111111 or 127 in decimal.

In a similar way, if the pin is an output its state is controlled by the corresponding bit in the PORTx register. The pins can sink or source 20 mA, and so are capable of driving LEDs directly.

Example 1.16 All of PB0–PB7 are outputs connected to LEDs. The other legs of the LEDs are connected to ground (via resistors). To turn on all of the LEDs, the number 0b11111111 is moved into PORTB. To turn off the middle two LEDs, the number 0b11100111 is moved into PORTB.

EXERCISE 1.11 Consider the example given above where all of PB0–PB7 are connected to LEDs. We wish to create a chase of the eight LEDs (as shown in Figure 1.6), and plan to move a series of numbers into PORTB one after the other to create this effect. What will these numbers be (in binary, decimal and hexadecimal)?

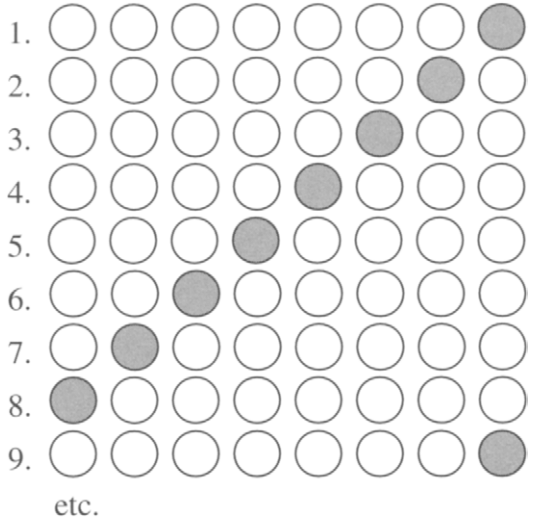


Figure 1.6

EXERCISE 1.12 PD0, PD1 and PD2 are connected to push buttons which are in turn connected to the +5 V supply rail. These push buttons are used in a controller for a quiz show. What numbers in PIND indicate that more than one button is being pressed at one time (in binary, decimal and hexadecimal)?

Instructions

We will now begin looking at some instructions. These are summarized in Appendix C at the back of the book. AVRs generally have about a hundred different instructions supported on them. This may sound quite daunting at first, but you will be relieved to hear that there is a fair amount of repetition. In fact there are only really about 40 that you *really* need to remember, and many are quite easy to remember with familiar sounding names like **add** or **jmp**. Fortunately, there a few general rules to help you decipher an unknown instruction. First, whenever you come across the letter **i** in an instruction, it will often stand for ***immediate***, i.e. the number which *immediately* follows the instruction or **I**/O register. A **b** will often stand for **b**it or **b**ranch (i.e. jump to a part of the program). Let's take a look at the format of an instruction line.

Example 1.17

```
(Label:)      sbi      portb, 0          ; turns on LED
```

The optional first part of the line is the label. This allows another part of the program to jump to this line. Note that a label cannot start with a number, and should not be given the same name as an instruction, or a file register (as this will confuse the AVR greatly!). The label is always immediately followed by a colon (this is easy to leave off and can be a common source of errors if you aren't careful). Note that the label doesn't actually have to be on the same line as the instruction it's labelling. For example, the following is just as valid:

```
Label:
              sbi      portb, 0          ; turns on LED
```

After the label comes the actual instruction: **sbi**, i.e. what you are *doing*, and then comes what you are doing it *to*: **portb, 0** (these are called the operands). Lastly, and just as important, is a semicolon followed by a comment on what the line is actually doing in your own words. It is worth noting that you can write whatever you want in an AVR program as long as it comes *after* a semicolon. Otherwise the assembler will try to translate what you've written (e.g. 'turns on LED') and obviously fail and register an ERROR. As the assembler scans the program line by line, it skips to the next line when it encounters a semicolon.

I must stress how important it is to *explain* every line you write, as shown above. There are a number of reasons for this. First, what you've written may make sense to you as you write it, but after a few coffee breaks, or a week later, or a month later, you'll be looking at the line and wondering what on earth you were intending to do. Second, you may well be showing your program to other people for advice. I am sent programs that, with alarming regularity, contain

very few or in some cases no comments at all. There is not much one can do in this situation, as it is almost impossible to deduce the intended operation of the program by looking at the bare code. Writing good comments is not necessarily easy – they should be very clear, but not too long. It is particularly worth avoiding falling into the habit of just copying out the meaning of the line.

Example 1.18

```
sbi     PortB, 0        ; sets bit 0 of register PortB
```

A comment like the one above means very little at all, as it doesn't tell you *why* you're setting bit 0 of register PortB, which after all is what the comment is really about. If you want to get an overview of all the instructions offered, have a good look at Appendix C and you can get a feel of how the different instructions are arranged. They will be introduced one by one through the example projects which follow.

Program template

Most programs will have a certain overall structure, and there are certain common elements needed for all programs to work. To make life easier, therefore, we can put together a program template, save it, and then load it every time we want to start writing a program. A template that I like to use is shown in Figure 1.7.

The box made up of asterisks at the top of the template is the program header (the asterisks are there purely for decorative purposes). Filling these in makes it easier to find out what the program is without having to scroll down and read the code and it helps you ensure that you are working on the most up-to-date version of your program. Note that the contents of the box have no bearing on the actual functioning of your program, as all the lines are preceded by semicolons. The 'clock frequency:' line refers to the frequency of the oscillator (e.g. crystal) that you have connected to the chip. The AVR needs a steady signal to tell it when to move on to the next instruction, and so executes an instruction for every oscillation (or *clock cycle*). Therefore, if you have connected a 4 MHz crystal to the chip, it should execute about 4 million instructions per second. Note that I say *about* 4 million, because some instructions (typically the ones which involve jumping around in the program) take *two* clock cycles. 'for AVR:' refers to which particular AVR the program is written for. You will also need to specify this further down.

Now we get to the lines which actually do something. **.device** is a *directive* (an instruction to the assembler) which tells the assembler which device you are using. For example, if you were writing this for the 1200 chip, the complete line would be:

```
;************************************
; written by:                       *
; date:                             *
; version:                          *
; file saved as:                    *
; for AVR:                          *
; clock frequency:                  *
;************************************

; Program Function:_____________________________________________
; ______________________________________________________________

.device     xxxxxxxx
.nolist
.include    "C:\Program Files\Atmel\AVR Studio\Appnotes\xxxxxx.inc"
.list

;==========
; Declarations:

.def        temp        =r16

;=============
; Start of Program

        rjmp   Init                 ; first line executed

;=========
Init:   ldi         temp, 0bxxxxxxxx    ; Sets up inputs and outputs on PortB
        out         DDRB, temp          ;
        ldi         temp, 0bxxxxxxxx    ; Sets up inputs and outputs on PortD
        out         DDRD, temp          ;

        ldi         temp, 0bxxxxxxxx    ; Sets pulls ups for inputs of PortB
        out         PortB, temp         ;    and the initial states for the outputs
        ldi         temp, 0bxxxxxxxx    ; Sets pulls ups for inputs of PortD
        out         PortD, temp         ;    and the initial states for the outputs

;====================
; Main body of program:
Start:
        <write your program here>
        rjmp   Start                ; loops back to Start
```

Figure 1.7

.device at90s1200

Another important directive is **.include**, which enables the assembler to load what is known as a *look-up file*. This is like a translator dictionary for the assembler. The assembler will understand most of the terms you write, but it may need to *look up* the translations of others. For example, all the names of the input/output registers and their addresses are stored in the look-up file, so instead of referring to $3F, you can refer to SREG. When you install the assembler on your computer, it should come with these files and put them in a directory. I have included the path that appears on my own computer but yours may well be different. Again, if the 1200 was being used, the complete line would be:

.include "C:\Program Files\Atmel\AVR Studio\Appnotes\1200def.inc"

Finally I'll say a little about **.nolist** and **.list**. As the assembler reads your code, it can produce what is known as a *list file*, which includes a copy of your program complete with the assembler's comments on it. By and large, you do not want this list file also to include the lengthy look-up file. You therefore write **.nolist** before the **.include** directive, which tells the assembler to stop copying things to the list file, and then you write **.list** after the **.include** line to tell the assembler to resume copying things to the list file. In summary, therefore, the .nolist and .list lines don't actually change the working of the program, but they will make your list file tidier. We will see more about list files when we begin our first program.

After the general headings, there is a space to specify some *declarations*. These are your own additions to the assembler's translator dictionary – your opportunities to give more useful names to the registers you will be using. For example, I always use a working register called **temp** for menial tasks, and I've assigned this name to R16. You can define the names of the working registers using the **.def** directive, as shown in the template. Another type of declaration that can be used to generally give a numerical value to a word is **.equ**. This can be used to give your own names to I/O registers. For example, I might have connected a seven segment display to all of Port B, and decided that I wish to be able to write DisplayPort when referring to PortB. PortB is I/O register number 0x18, so I might write DisplayPort in the program and the assembler will interpret it as PortB:

.equ DisplayPort = PortB or
.equ DisplayPort = 0x18

Another example of where this might be useful is where a particular number is used at different points in the program, and you might be experimenting and changing this number. You could use the **.equ** directive to give a name to this

number, and simply refer to the name in the rest of the program. When you then go to change the number, you need only change the value in the **.equ** line, and not in all the instances of the use of the number all over the program. For the moment, however, we will not be using the **.equ** directive.

After the declarations, we have the first line executed by the chip on power-up or reset. In this line I suggest jumping to a section called **Init** which sets up all the initial settings of the AVR. This uses the **rjmp** instruction:

```
rjmp    Init            ;
```

This stands for **r**elative **jump**. In other words it makes the chip jump to a section of the program which you have labelled **Init**. The reason why it is a *relative* jump is in the way the assembler interprets the instruction, and so is not really important to understand. Say, for example, that the **Init** section itself was 40 instructions further on from the **rjmp Init** line, the assembler would interpret the line as saying 'jump forward 40 instructions' – i.e. a jump *relative* to the original instruction. Basically it is far easier to think of it as simply jumping *to* **Init**.

The first part of the **Init** section sets which pins are going to act as inputs, and which as outputs. This is done using the Data Direction I/O registers: DDRB and DDRD. Each bit in these registers corresponds to a pin on the chip. For example, bit 4 of DDRB corresponds to pin PB4, and bit 2 of DDRD corresponds to pin PD2. Now, setting the relative DDRx bit *high* makes the pin an *output*, and making the bit *low* makes the pin an *input*.

If we configure a pin as an input, we then have the option of selecting whether the input has a built-in pull-up resistor or not. This may save us the trouble of having to include an external resistor. In order to enable the pull-ups make the relevant bit in PORTx high; however, if you do not want them make sure you disable them by making the relevant bit in PORTx low. For the outputs, we want to begin with the outputs in some sort of start state (e.g. all off), and so for the output pins, make the relevant bits in PORTx high or low depending on how you wish them to start. An example should clear things up.

Example 1.19 Using a 1200 chip, pins PB0, PB4 and PB7 are connected to push buttons. We would like pull-ups on PB4 and PB7 only. Pins PD0 to PD6 are connected to a seven segment display, and all other pins are not connected. All outputs should initially be off. What numbers should be written to DDRB, DDRD, PortB, and PortD to correctly specify the actions of the AVR's pins?

First, look at inputs and outputs. PB0, 4 and 7 are inputs, the rest are not connected (hence set as outputs). The number for **DDRB** is therefore **0b01101110**. For Port D, all pins are outputs or not connected, hence the number for **DDRD** is **0b1111111**.

To enable pull-ups for PB4 and PB7, make PortB, 4 and PortB, 7 high, all

other outputs are initially low, so the number for **PortB** is **0b10010000**. All the outputs are low for Port D, so the number for **PortD** is **0b00000000**.

We can't move these numbers directly into the I/O registers, but instead we have first to move them into a working register (such as **temp**), and then output the working register to the I/O register. There are a number of ways we can do this:

```
ldi     register, number    ;
```

This **l**oa**d**s the **i**mmediate number into a register, but it is very important to note that this instruction cannot be used on all working registers – *only on those between R16 and R31* (we can therefore still use it on **temp**, as that is R16). We can also use a couple of alternatives to this instruction if the number we wish to move into the register happens to be 0 or 255/0xFF/0b11111111:

```
clr     register            ;
```

This **cl**ears the contents of a **r**egister (moves 0 into it) – note an advantage of this over **ldi** is that it *can* operate on *all* working registers. Finally,

```
ser     register            ;
```

This **se**ts the contents of a **r**egister (moves 255/0xFF/0b1111111 into it), though like **ldi**, it *only works on registers between R16 and R31.*

We then need to move **temp** into the I/O register, using the following instruction:

```
out     ioreg, reg
```

This moves a number **out** from a register, into an I/O register. Make sure you note the order of the operands in the instruction – I/O register first, working register second, it is easy to get them the wrong way round! We can therefore see that the eight lines of the **Init** section move numbers into **DDRB**, **DDRD**, **PortB** and **PortD** via **temp**.

EXERCISE 1.13 Using a 1200 chip, pin PB0 is connected to a pressure sensor, and pins PB1, PB2 and PB3 control red, yellow and green LEDs respectively. PD0 to PD3 carry signals to an infrared transmitter, and PD4–PD6 carry signals from an infrared receiver. All other pins are not connected. All outputs should initially be off, and PB0 should have a pull-up enabled. Write the *eight* lines that will make up the **Init** section for this program.

After finishing the **Init** section, the program moves on to the main body of the program labelled **Start**. This is where the bulk of the program will lie. Note that

the program ends with the line **rjmp Start**. It needn't necessarily loop back to **Start**, but it does have to keep looping to *something*, so you may want to alter this last line accordingly. At the end of the program, you can write **.exit** to tell the assembler to stop assembling the file, but this isn't necessary as it will stop assembling anyway once it reaches the end of the file.