AVR Project
PROGRAMMABLE LOGIC CONTROLLER

by Kevin Kirk

This month, we are looking at developing a project using the AVR. The difference between this, and other articles you are going to find in the magazine, is that we shall be working through the project stages step by step. In this way, you should be able to see how a project is developed, so you can go off and try one on your own. The AVR lends itself very well to this type of project development (i.e., learn as you go) because of the fact that it not only has in system programming capability but also it has a 1,000 reprogramming lives, which is more than ample for most applications.

The project itself is a mini PLC. A PLC (Programmable Logic Controller) is used in industrial environments to control machines. Very useful in a domestic environment, huh? In actual fact, they are, except that they normally cost a couple of hundred quid or so and so it would be a little tricky getting that one past the other half if you wanted one to control your trainset! This one should set you back twenty quid or so and it will be capable of providing the sort of performance that you can expect from the low end commercial PLCs. You can use the finished unit to control virtually anything that can be controlled digitally. Train layouts, central heating, burglar alarms, computer systems, school experiments (it is perfect for meeting the criteria of the national curriculum for Computer control) and robotics spring to mind. In fact, its uses are really only limited by your ingenuity.

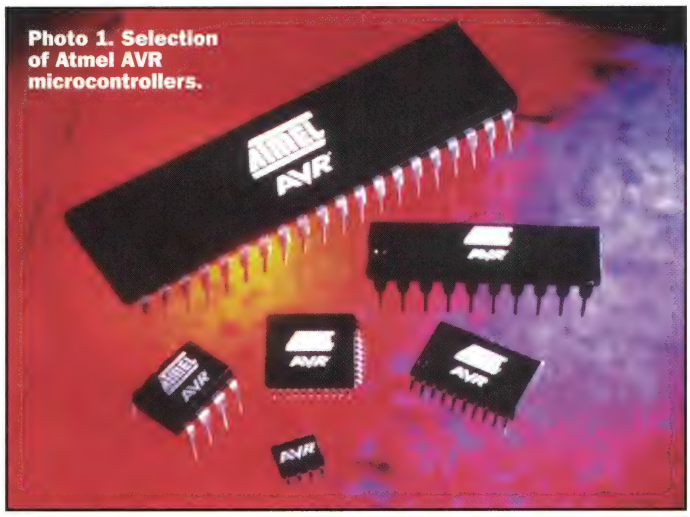


Photo 1. Selection of Atmel AVR microcontrollers.

The Hardware Design

Before we can rush off and design things, we need to sort out the specification of what we want the unit to do. So, this is always the first step. The unit itself is going to be a general purpose digital controller, with some analogue capability thrown in. So first of all, we need to look at the Input and Output requirements.

The initial I/O spec looks something like this:

Inputs: 6 Digital Inputs
2 Analogue Comparator Inputs
1 Interrupt input / uncommitted bit
Outputs: 6 Digital Outputs

This configuration has two distinct advantages. The first is that it gives you the maximum flexibility, the second is that it makes the software easier because we'll effectively be using one port for input and the other for output (the interrupt is the exception, but that's easy to get around).

The next stage is the actual hardware functionality. Now, we could opt for a very simple, non-isolated, system where we just feed in the digital inputs in raw and drive 'things' with the outputs, again, directly. The advantage here is that it is cheaper. However, it could give you problems, especially if you wanted to use this system to control mains devices and also in respect of common mode offsets. So in retrospect, we'll give the unit some isolation from the outside world. The easiest way to do this is via opto isolators. On the outputs, we'll use stand alone packages, one per output. The reason for this is that we can use either transistor-based devices (for controlling DC) or Zero Crossing TRIAC versions (for controlling mains). You just choose the one you want at build time, depending on your requirements. Note that they can be 'mixed and matched' with some DC and some AC circuits. The circuit options are shown in Figure 1a (for DC) and Figure 1b (for AC).

Watch the creepage distances if you are using mains though. This is the minimum distance you must allow between the mains carrying tracks and the rest of the circuit. BS EN 60950 gives the value as 4mm, but I recommend you double that. It goes without saying that you

Figure 1a.
DC output circuit.

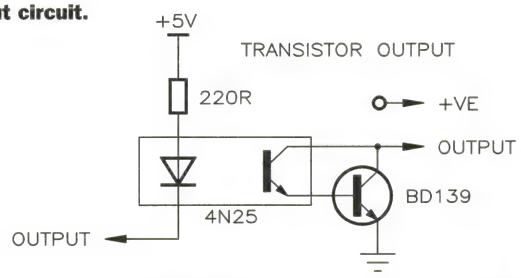
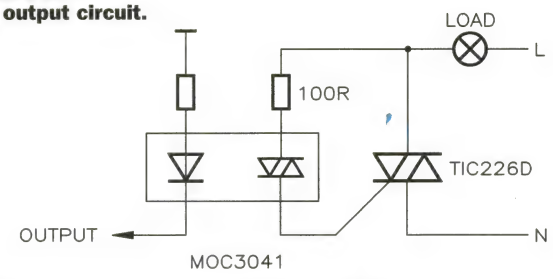


Figure 1b.
AC output circuit.



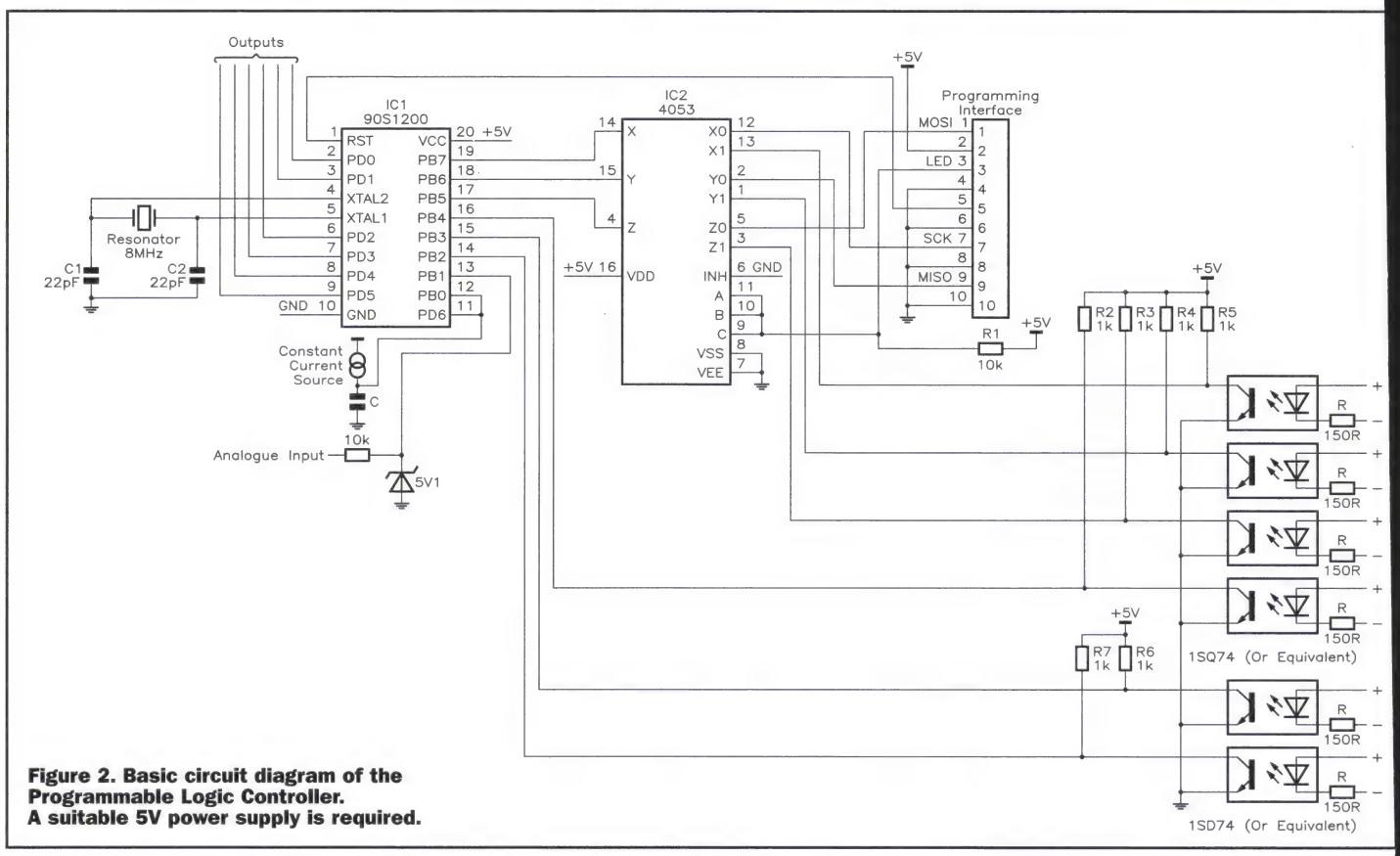


Figure 2. Basic circuit diagram of the Programmable Logic Controller. A suitable 5V power supply is required.

should keep your fingers off it while it is live! The inputs will also be isolated but here, we'll just use multiple opto isolator packages. Incidentally, because of the opto isolators, we now look for a '0' being a true value (i.e., ON) and a '1' as a false (i.e., OFF). This is the case for both inputs and outputs. So, we clear (set to a 0) to switch on and set (set to a 1) to switch off. Now, we'll choose ourselves a clock. Timing is not super critical in this application, so we can live with a ceramic resonator, rather than a crystal. To keep the maths easy, we'll choose an 8MHz version (Scientific, isn't it?).

The programming interface is next. Ideally, we should have the capability of swapping the programming port to general I/O for normal use; we need some sort of multiplexing. This is not as nasty as it sounds. We just need to divert 3 signals so we can plonk a 3-pole changeover switch in the circuit that is controlled from the microcontroller reset line (which the programmer itself uses to switch the microcontroller into programming mode). For this, we'll use a standard CMOS part called the 4053. Now we have the capability of programming in system so we can 'play' with the code to our heart's content. Finally, a power supply. A standard 5V regulator will fit the bill nicely, so we'll drop one of those in. The basic circuit diagram is shown in Figure 2. Note that the unused inputs are pulled high via a relatively high value resistor. The reason is that the AVR itself is CMOS and it is NOT a good idea to leave inputs floating.

We could add an analogue-to-digital convertor (ADC) if we wanted. This could be a very simple circuit based on the single slope principle. Essentially, all you do is to feed a capacitor with a constant current supply, which results in a straight line charge slope (rather than the rather nasty logarithmic slope that capacitors prefer). You then compare this constantly changing value with the input and when it exceeds (slightly) the input, then you know the value. To get the value, you simply set a timer running when the capacitor starts charging and then stop it when you reach the crossover point. This timer can be in software if you want, however, we have got a perfectly serviceable timer in the AVR itself, so we'll be using that. We'll cover this next month, but in the meantime, you can look at the components that the ADC uses, which are shown on the circuit diagram inside the dotted lines.

Program Design

Now, the tricky bit! There is one golden rule that you should bear in mind when working with microcontrollers, and that is that they are stupid. That's why we love 'em. They will only do what they are told to do and they will get all sulky if you don't give them something to do (thinking about it, they are the exact opposite of teenagers). So, we have to think the software through so that it covers all eventualities, because if they encounter something we hadn't planned for, then they'd just wander off and do their own thing (i.e., they'll crash!). Now, at this point, we could leap off into reams of flow diagrams, project planning, etc. We could, but we won't, because there is very little need in this case (I can hear the howls of protests from the micro teachers from here). What we will do is create a simple flowchart (without the symbols) that gives us a broad overview of what the system is trying to achieve, which will give us a framework around which we can plan the various software modules. To give it its scientific name, this is called 'top down design'. It looks something like this:

Set up
Scan inputs
Set outputs
Loop to Scan Inputs

It looks suspiciously simple, doesn't it? There is a very good reason for that, it is simple. All a PLC does is to look at the inputs then to change the outputs based on the combinational effects of the inputs. For example, if you have a machine with two protective safety guards on it, then you want to only switch the output on when the guards are in place AND the start button has been pressed AND NOT if the stop button has been pressed. This is a straight Boolean function which can be expressed as:

(Guard1 . Guard2 . Start) . !Stop

('.' is the symbol for AND and '!' is the symbol for NOT – incidentally '+' is the symbol for OR)

This can be rendered into the following software expression (assuming port D bit 0 is the output and port B bits 0 and 1 are the guards, bit 2 is the start and bit 3 is the stop):

This piece of code, which at first glance looks complicated, will actually cover the last three 'sections' of our functional flowchart. It needs a few bells and whistles to set it up and to provide some protection against noise spikes invading the circuit, but in essence, that's all it needs to perform the task. It may be worthwhile, if you are new to

```
loop1:  SBI   PORTD,0   ;switch output off – 1 is off
        SBIC  PINB,2    ;see if start button has been pressed
        RJMP  loop1     ;loop if it isn't
loop2:  SBIC  PINB,0    ;is guard 1 in place?
        RJMP  loop1     ;No! so loop until it is
        SBIC  PINB,1    ;is guard 2 in place?
        RJMP  loop1     ;No! so loop
        SBIS  PINB,3    ;Has stop been pressed?
        RJMP  loop1     ;Yes so switch off and wait for start again
        CBI   PORTD,1   ;If not then switch on
        RJMP  loop2     ;Then check for guards and stop switch again
```

is game, to read the code and ok up what each instruction is ping (the full instruction set as in last month's magazine) so ou can get a 'feel' for it. Note at the label (i.e., loop1) is on

```
LDI  R16,$0F          ;set up values for watchdog
     MOV  WDTCR,R16   ;Put them into watchdog control register
     LDI  R16,$00     ;Set port b to inputs (this is reset value!)
     MOV  DDRB,R16
     LDI  R16,$FB     ;Set port d to outputs except bit 2
                                              (interrupt)
     MOV  DDRD,R16
```

he left and is delimited with a olon, the operator (i.e., the struction) comes next and is dented (or tabbed) and finally, he operand (i.e., what it is orking on and where it should ick the result).

The I/O ports on the AVR are ue tri-state devices. That means at they have 3 possible states a , a 0 or off. The latter may be onfusing until you realise that it being driven neither high nor w but it is merely floating, i.e., will follow whatever value is resented to it, in other words, is is the input state. The AVR ses a special register called the Data Direction Register to

```
loop1:   SBI   PORTD,0    ;switch output off – 1 is off
and the:
         CBI   PORTD,1    ;If not then switch on
instructions:
         WDR
```

determine whether the port is used for input or output. A 0 in his register will set the corresponding port pin up as an nput and a 1 will set it up as an output. You may have noted that he code refers to the input orts as Pins whereas the outputs are Ports. The reason for hat is that the port is actually a atch and is a mirror of the output, whereas the pin reads he pin itself.

So, to start off, we need to set up the relevant ports as inputs or outputs. We'll use one of the general purpose registers to assemble the bytes required. As the top 16 (of 32) registers are capable of being used to load immediate values and for directly accessing the I/O, we will use R16. Incidentally, we'll be using the lower 16 registers for Scratchpad (temporary) storage as the project progresses. So, the first thing we do after a reset is set up the watchdog. This system is designed for control use and the last thing you want when you are controlling something is for it to go berserk. Thus, we use a watchdog to keep it in check. In this instance, we can live with the maximum prescaler value on offer which is /2,048, which with its 1MHz clock, gives us a timeout of around 2ms. We are not in any particular hurry, so this is fine. We will need to keep it sweet by resetting it and the less we have to do it, the better. Our start up code now looks like this:

There are a couple of points to note. The first is that the code, as it stands, makes no allowance for the interrupt vectors, which have been omitted for clarity, and which would normally require a jump from the reset vector (Vector is a Techie way of saying address) to the start of the code. The other is that if you add the previous piece of code to this end of this, then it will nearly work. What we need to do is to add our Watchdog reset instruction and we are in business. You need to add it into your main loop(s) so the following instruction must be added to the previous code:

The system will now work. In fact, because of the way the code has been written, you could actually leave the watchdog reset instruction (WDR) out of the code after the first instruction line. Can you see why? I'll leave you to work that one out.

The unit will, so far, perform the same task as a piece of relay-based ladder logic, but it can have extra functions added without having to resort to a soldering iron. It takes a lot less power too.

Next month, we will look at adding extra functionality such as an ADC and we'll also look at writing specific code to perform your own control tasks. Finally, it would be a worthwhile investment to get hold of the Atmel AVR data book, which fully describes all of the functions of the various registers, etc. The Maplin Order Code is **NR22Y** and the price is £11.75. Alternatively, there is also a fully fledged training system available for the device called the AVR Explorer (Order Code **NR41U**) which is on page 931 of the catalogue and is priced at a whisker under £99. This takes you from zero to full speed and comes complete with hardware, software and coursework. ELECTRONICS