Electronics for Model Railways

Chapter 21

PICs and Arduinos

By Davy Dick

Electronics for Model Railways

By Davy Dick

In memory of Margaret

Contents

PICs and Arduinos

PICs and Arduinos are types of *'microcontroller'*.
Microcontrollers are everywhere - in your home, your office, factories, etc.
They can be found in your car, mobile phone, television, computer and printer, microwave oven, fire alarm, hearing aid, camera and probably even in your toaster.
They are also used in factory robots, office automation, traffic management, security systems, lifts, etc.

In the model railway and electronics world, you find them in DCC controllers and decoders, servo controllers, sound systems, layout control systems such as CBUS and EzyBus, 3D printers and test equipment.

How did we get here?

In the early days, we relied on switches and relays to control our layouts.
With the advent of transistors we were able to able to add new features such as controllers with acceleration/deceleration, flashing lights, etc.
When ICs came along, they dramatically reduced the number of components you had put together. This allowed us to have DCC controllers and DCC decoders and modules that were difficult to make, such as sound players.
Now programmable chips takes us a step further and open up even more possibilities.

What is a microcontroller?

While very useful, integrated circuits are dedicated to a single function, or set of functions. This led to the development of a range of microcontrollers which, like ICs, combine a lot of components into the chip, but allow the user to program its activities. This opens up a whole new approach, since the microcontroller can be used to carry out many different actions depending on what program it contains. There is no need for major re-soldering if you change your mind - you just change the program inside the microcontroller.

What are their advantages?

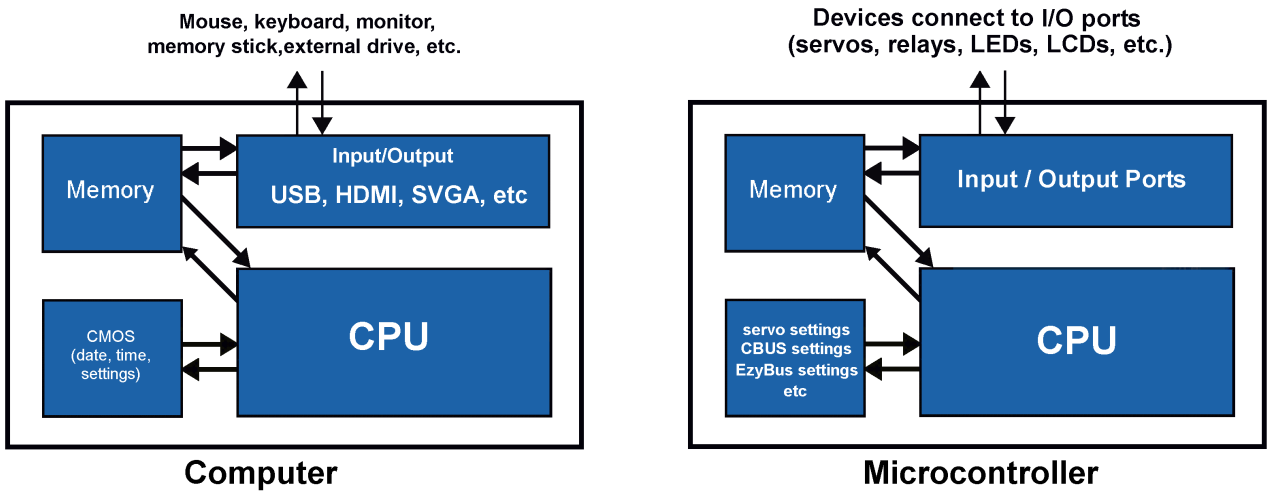
Both PICs and Arduinos share the same advantages:

- Low cost.
- Easily obtainable.
- Small size.
- Lots of Input/Output pins to connect to the outside world.
- Lots of processing options (e.g. ADC, PWM).
- Lots of interfacing options (e.g. CAN, USB, UART, I2C, Ethernet).
- Development tools are free.
- Easily programmed and reprogrammed.
- Many can be programmed in situ (i.e. the chip remains in the module it is designed for).
- Large body of support through documentation, source code, user groups, etc.

The exact faculties depends on the particular microcontroller. For example, a 40-pin version offers more I/O and interface options than an 8-pin version.

How do they work?

The most commonly used microcontrollers in model railways projects are the Microchip PIC and the Arduino.
Microcontrollers have similarities to your computer, as shown in this illustration.

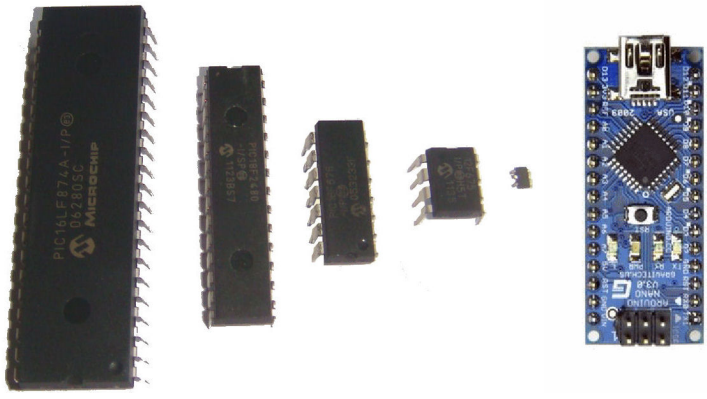


- They both have a CPU, the main brains of the system.
- They both have a memory area to store the program being used.
- They both have an area to hold data that the program is using.
- They both have a block of memory that can store data even after power is removed.
- They both have connections to the outside world.

Naturally, the computer system is more sophisticated. Then again,all the parts that you see for the microcontroller on the diagram are embedded inside the chip.
This arrangement makes microcontrollers very flexible.

What do they look like?

Microcontrollers look just like any other integrated circuit. They are available both in through-hole (the ones shown here with legs) versions or surface mount versions.
The Arduino range of modules are microcontrollers that have been mounted on printed circuits with additional components (more on these later).



Microcontrollers vs microprocessors

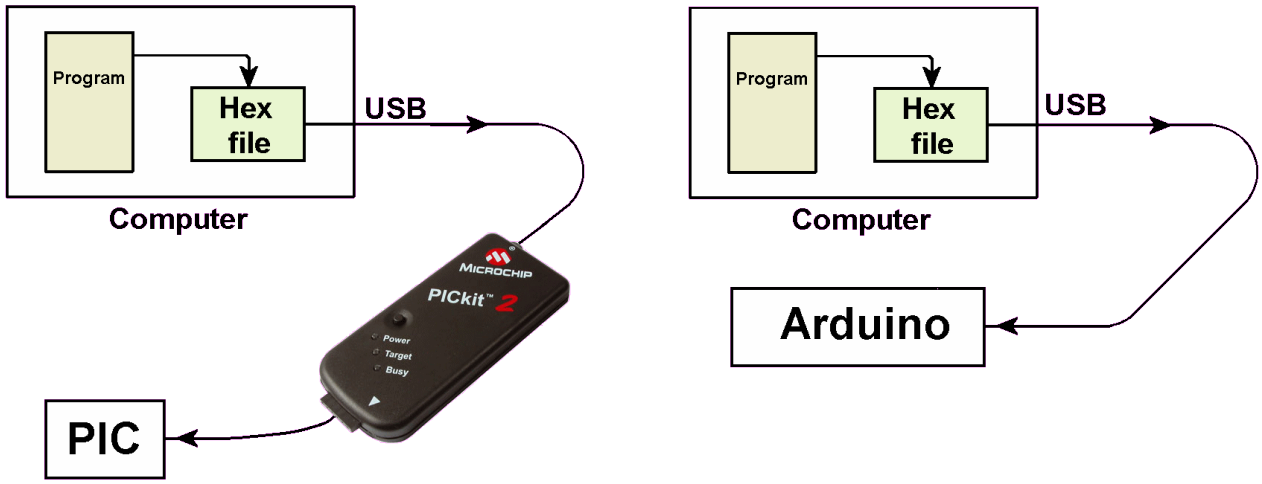
A short explanation of the differences.
A microprocessor is at the heart of all computers. It is an integrated circuit that contains the CPU (central processing unit). Although much faster and more powerful than a microcontroller, it relies on external memory and interface electronics. The memory and other additions are added to the computer's motherboard.
A microcontroller, as explained above, has both memory and interface facilities embedded in the chip.

How do you get a program into them?

To program a microcontroller you need two things:

- Software
- Hardware

You start with a program that you want to put on the PIC or Arduino. This may be a program you have written yourself, or may be one that you have downloaded.
The job of the software is transfer the program out of the computer and into the PIC or Arduino.



In the case of PICs, you mostly need a special programming module that connects between the computer and the PIC.
With Arduinos, you simply connect a USB cable between it and your computer, as the module already has interfacing electronics on board.

What is the hex file?

The hex file is the program that the microcontroller runs once it is loaded.
All computer programs are just a series of numbers.

- Some numbers represent instructions for the CPU to carry out (e.g. mathematical calculations, decision making, moving data around, etc.).
- Other numbers represent some of the data the program will use (other pieces of data will be generated as the program runs).

Since the numbers are in binary form, a program is really just a long sequence of zeros and ones. To make is bit easier for humans to appreciate, the sequence is readable as a set of hex numbers.
Here is an example of part of a hex file:

```
:020000040000FA
:1000000020282030AC000310A80DA90DAA0DAB0DBF
:100010000310A00DA10DA20DA30D031C1D28240883
:10002000A80725080318250FA90726080318260F77
:10003000AA0727080318270FAB07AC0B03280800F3
:100040000508B5001830AD00D230AE002230AF0048
:10005000B001E530B1005530B2009A30B300153030
```

Note

Although the program file is called a hex file, it is still sent out of your computer in serial format (a sequence of 1's and 0's).

How do we create the hex file?

If you download a hex file from the Internet, it already contains all the program as a binary sequence. It is ready to send to the microcontroller.
If you are writing your own program, there are a number of methods:

- Early programmers had to understand the architecture of the microcontroller they were working on and its particular instruction set. Since most CPU types have different constructions, the set of instructions varies between different microcontrollers. If you are a masochist, you could find a way to write a program in 1's and 0's or in hexadecimal numbers!
- The next stage up is to write in *'assembly language'*. You have to know how the CPU works and write a set of instructions in a slightly more readable format, as in this example:

```
;  40    delay_100ms(100)
                        movlw    100
                        movwf    v___n_5
                        clrf     v___n_5+1
                        call     l_delay_100ms
;  41    LED= low
                        datalo_clr v__porta_shadow ; x23
                        bcf      v__porta_shadow, 2 ; x23
```

  It is still very difficult to understand unless you know how the CPU functions.
  Once completed, the software *'assembler'* converts the instructions into the corresponding machine language.
  This method is still used where you need the most compact code possible, along with the maximum speed and accurate timing.
- Many programmers write in a *'high level language'* which need not know the internal workings of the microcontroller. It can be written in plain English and is more understandable, as in this extract:

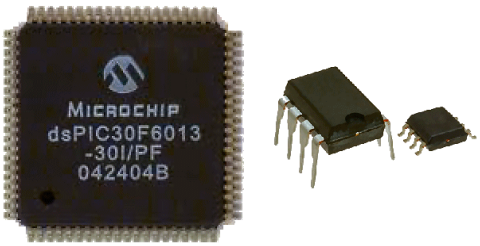
```
if switch==low then
  alarm=high
end if
```

  After the program is written, it is *'compiled'* into the final hex file for storage or programming.
  The process is similar for PICs and Arduinos and is covered later.

What are PICs?

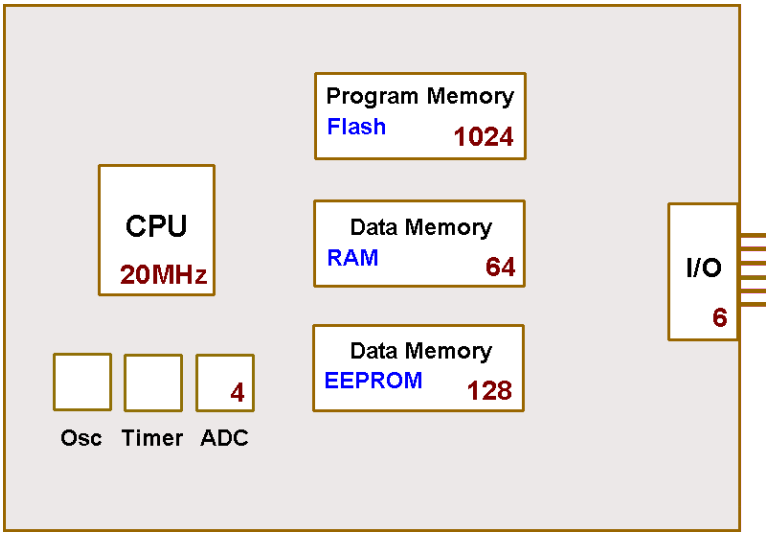
PICs are produced by Microchip Technology Inc and it has the largest share of the microcontroller market. In 2008, they had sold 6bn chips and this increased to 12bn by 2013. They have been around for about forty years and the range of PICs has developed and grown.
Apart from their use in many domestic, industrial, scientific and military applications, they also form the heart of many of the MERG range of kits. They can be found in the *'Servo4*' module, the *'Hector'* infrared detector, CBUS modules and many of the Pocket Money Kits. For example, the kits for the welder, lighthouse, automatic signals, EzyPoints, versatile timer, laser detector, steam emulator and AutoStop use exactly the same 12F675 PIC chip. Different programs have been written for each of those modules and then saved into the PIC chip.

PICs look very similar to ICs. Many have the same shapes, same number of pins, etc. However, they are easily distinguished by the markings on their bodies. For example, an L293D is an IC while a 12F675 is a PIC.



PICs are available in sizes from 6 pins to 144 pins, in either QFN (Quad Flat No Leads), DIL or surface mount pin formats.
Despite their size and pin differences, they have common features such as inbuilt oscillators and I/O (input output) pins to interface to external devices.

This is what you will find inside the 12F675 PIC chip.



Although it only has eight pins and is small in size, it contains the following elements:

CPU

This chip is relatively basic and only uses 35 different types of instructions. Higher-end PICs have more facilities and use up to 80 or more instructions. The 12F675 is an 8-bit CPU, which means that it handles eights bits of data in a single instruction. Higher-end PICs are 32-bit and are able to move data around four times faster.

Oscillator

The speed of the oscillator dictates how quickly the chip can carry out its instructions. The 12F675 PIC has a built-in oscillator that runs at 4MHz. This can be increased to 20MHz if you use an external oscillator. Higher-end PICs run at up to around 250MHz. While this is slow compared to a computer's microprocessor's speed that is measured in GHz, it is more than sufficient for most model railway applications.
The built-in oscillator is satisfactory for most purposes. If you need greater precision (e.g. for data communications) you can add an external ceramic resonator or a crystal.

Flash memory

This is where your main program is stored. The 12F675 only has 1k of program memory but that is sufficient for many of the small projects used in model railways. Being flash memory, its contents can be overwritten 100,000 times, allowing lots of scope for experimenting with programs and their parameters. The best thing about flash memory is that is contents (i.e. your program) is retained during power down.

RAM

This small block of memory is only 64 bytes. It is used to store temporary values as the program runs (e.g. results of calculations, keeping a count, storing temperature readings). The contents change during the running of the program but are lost when power is removed.

EEPROM

This a 128 byte chunk of Electrically Erasable Programmable Read Only Memory. Its contents can be altered during the running of the program and are retained during power removal. It is used in the sequencer and button servos kits and in the Servo4 module. In these examples, the user can alter values during the running of the program and expect them to be stored for use next time they power up the module. The ability to alter the contents of EEPROM up to 1 million times provides long-lasting use of the chip.

I/O

The PIC's digital Input/Output pins are interface between the program and external devices. The 12F675 has 6 pins that can be used as I/O (the other two pins are required for the power supply). Five of these pins can be configured to be input pins or output pins in any combination (all input, all output, or a mixture of both) while one pin can be only configured as an input pin.
As an alternative, three of the pins can be configured to work with analogue inputs instead of digital inputs. This facility is used in the Random Lights kit (to allow the user to adjust the switching delays), EzyPoints kit (to allow the user to set the servo endpoints and rotation speed) and the Automatic Coach Lighting kit (where it is used to sense reflection differences from the track).

ADC

When an input pin is configured as analogue, it uses an internal *'Analogue to Digital Conversion'* circuit inside the PIC. This allows the reading of analogue values from that pin. This can be used for reading the voltage levels from potentiometers, heat sensors, light sensors, etc. The ADC circuit inside the PIC converts these analogue values into digital equivalents that the PIC can process. The 12F675 PIC has four pins that can be configured as analogue inputs. Each of those is 10-bit resolution, which means analogue sample can be converted to any one of 1024 different values.

Timer

This provides the ability to generate signals in regular time intervals.
This is useful for data communications, counters, etc.

Higher-end PICs offer additional facilities such as:

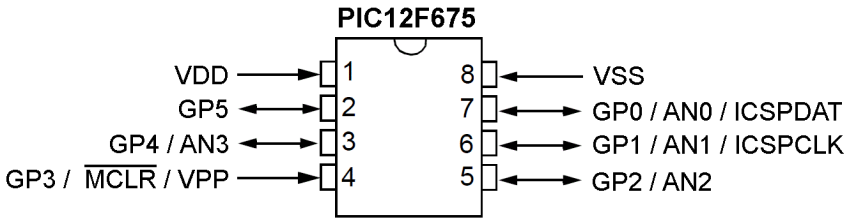
- Additional I/O pins, often able to be groups into sets of eight where they are known as 'ports' (e.g. Port A, Port B, etc.).
- Interfaces with commonly used communications systems, such as USB, CAN, UART, SPI, I2C, Ethernet, etc.
- PWM - Pulse Width Modulation. The ability to generate pwm outputs on one or more of a PIC's output pins. Useful for controlling motors, dimming LEDs, etc.
- DAC - Digital to Analogue Conversion. etc. The ability to turn stored digital values into analogue equivalents that are sent out of a PIC's pin. Useful for creating sound effects, playing sound samples (e.g. WAV files), etc.

A chart of the facilities for each PIC variety is available here:

http://ww1.microchip.com/downloads/en/DeviceDoc/30009630M.pdf

Typical PIC pin-out

The illustration shows the pins on the 12F675 PIC. This is the one that is used in many of the Pocket Money Projects kits.



It is an 8-pin microprocessor and has a notch at one end. The notch indicates that it is the end where pins 1 and 8 are located. In addition, some chips have a small indentation next to pin 1.

The label GP stands for General Purpose and AN indicates that it can be used for analog input (but not analog output). Bi-directional means that that pin can either be used for input or output.

As the table shows, pins can have more than one function, depending on how the chip is set up or how it is controlled externally. For example, pins 1, 4, 6, 7 and 8 are used while the chip is being programmed. After programming, pins 4, 6 and 7 are available for the other purposes shown in the table.

Pin	Name	Purpose
1	VDD	+5V
2	GP5	Bi-directional digital I/O
3	GP4 / AN3	* Bi-directional digital I/O * Analog input
4	GP3 / MCLR / VPP	* Digital Input (not bi-directional) * Master Clear (MCLR) external reset. When this pin is temporarily brought down to 0V, it forces the program to start all over again from the first instruction * During programming, the programmer raises this pin to 13.5V to inform the chip that it is in programming mode.
5	GP2 / AN2	* Bi-directional digital I/O * Analog input
6	GP1 / AN1 / ICSPCLK	* Bi-directional digital I/O * Analog input * During programming, this pin is used to ensure that the serial data transfer between the computer and the chip maintains synchronisation.
7	GP0 / AN0 / ICSPDAT	* Bi-directional digital I/O * Analog input * During programming, this pin is used to transfer the serial data from the computer into the chip.
8	VSS	0V

Programming a PIC

The precise steps depend on the programmer being used, but a typical process is:

- Write a program on a computer, using commercial or free software, then turn the program into the form that the PIC understands (called a compiled *'hex'* file).

OR

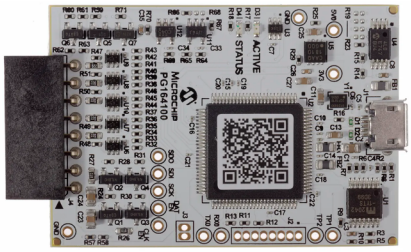
- Download the hex file you want from the Internet.

After that:

- Connect your PIC programmer module to the computer's USB port.
- Connect the PIC programmer to the circuit board being used.
- Insert the PIC into the circuit board.
- Open the PIC programming software.
- Load the hex file into the programming application.
- Click on the '*Write'* button to copy the hex file into the PIC.

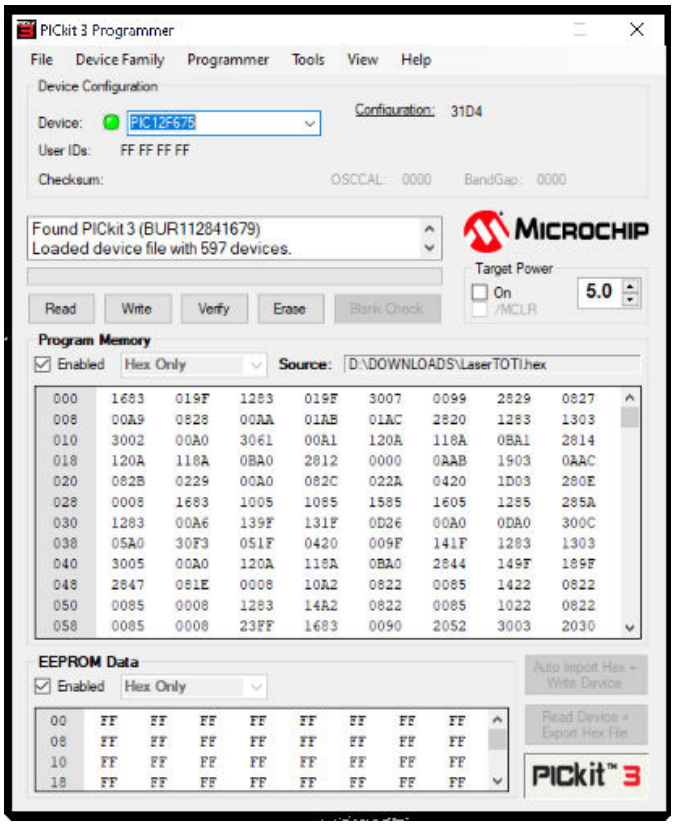
Microchip has produced a number of programmer modules (e.g. the PicKit3, the PicKit4 and the Snap). They all have a USB socket for connecting to the computer and a trailing lead to connect to the PIC chip.

Here are Microchip's PICki3, PICkit4 and Snap programmers.



Microchip, and other suppliers, provide their own software to accompany their programmers.

Here, for example, is the interface for the PICkit2 programmer.



The *'File'* menu lets you choose which hex file to work with.

Once loaded, confirmation appears in the message window ("*Hex file successfully imported"*) and the name of the hex file appears in the '*Source'* window.

The *'Blank Check'* button checks whether the PIC chip already has contents and informs you in the message window.

The *'Erase'* button clears all contents from a PIC chip.

The *'Write'* option copies the contents of the hex file into the PIC chip, then

compares the contents of the chip with the original hex file. This is sometime referred to as '*burning*' a chip.

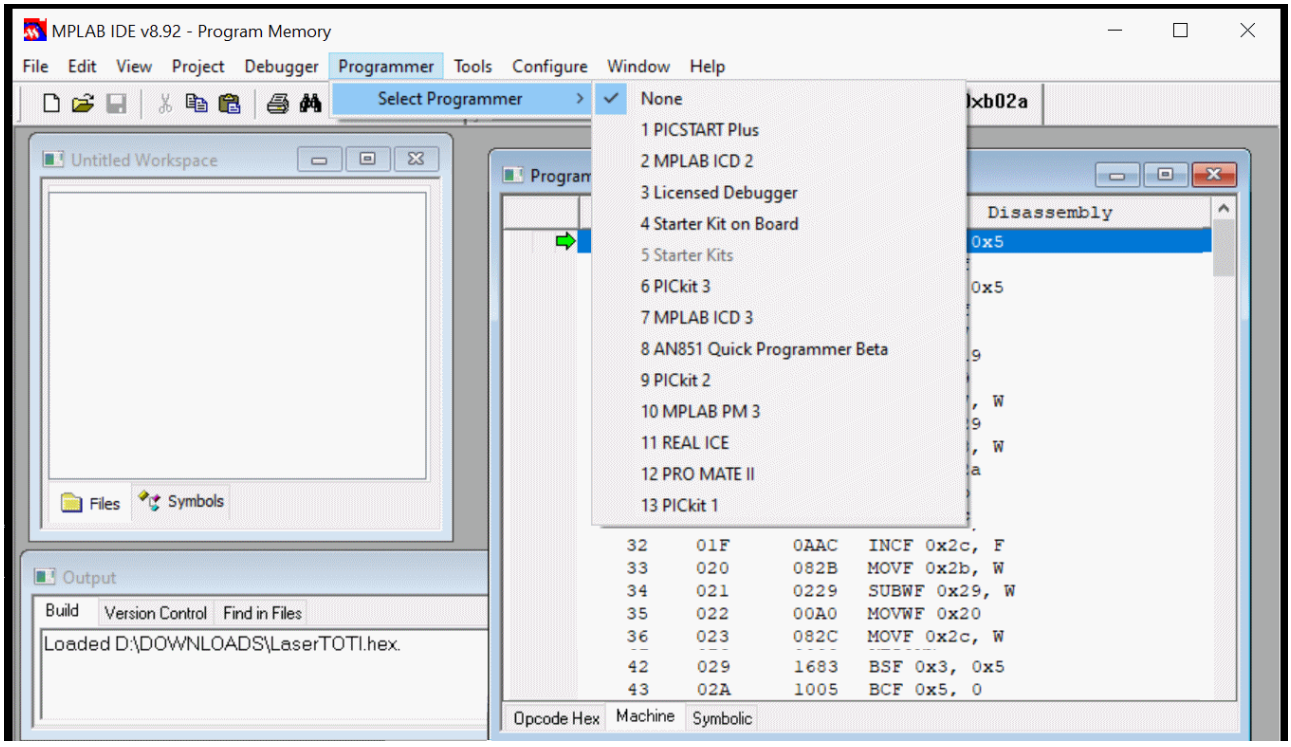
The large *'Program Memory'* window shows the contents of the hex file you have loaded.
The window below it shows the contents, if any, of the PIC chip's EEPROM contents.

Copying a chip

You can make a duplicate of an existing programmed chip, unless it has been read protected.
You insert the original chip into the programmer, then click on the *'Read'* button.
The contents of the chip are displayed in the Program Memory window and if the chip had any EEPROM contents (e.g. servo settings) they are displayed in the EEPROM window.
You then remove the original PIC from the programmer and insert the new PIC chip into the programmer.
Clicking the Write button then copies the contents on to the new PIC chip.

Using MPLABX IPE

Microchip's PICkit 2 software is designed to work with the PICkit2 programmer and the PICkit3 programmer uses its own PICkit3 software.
However, Microchip's MPLAB can work with a large range of microcontrollers.
As this illustration shows, you have to select which programmer you are going to use for programming your PIC.



The MPLABX IDE is used for editing your programs and the IPE is used for programming your chip.

If you have purchased a different programmer, it will have its own programming software.

How does the programmer work?

The programmer's job is to transfer the program's contents from the computer into the PIC chip, using the transfer software supplied with the programmer.

The computer's programming software communicates with the PIC chip using five connections.

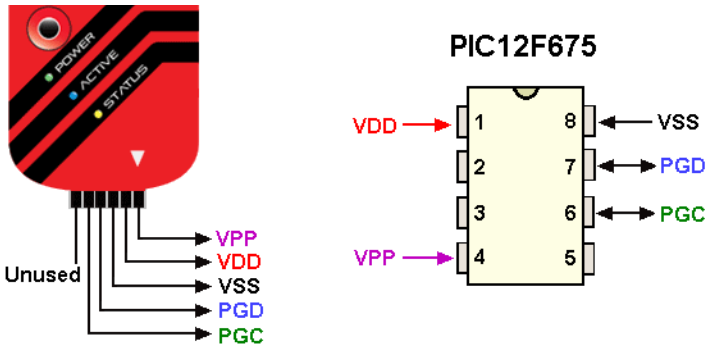
Vdd	+5V
Vss	0V
Vpp	This is the programming voltage. PICs enter programming mode when their Vpp pin is raised to around 13.5V.
PGD	This lines transfers the data from the computer to the PIC chip.
PGC	This line (the '*clock*') keeps the computer and the chip synchronised during data transfers.

Note

The PGD and PGC pins are also used if you want to read contents from a PIC chip into the user's IDE.

Making the connections

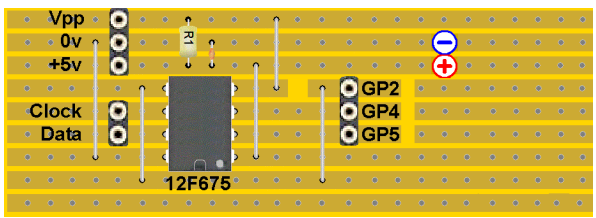
This illustration shows the outputs from a PICkit3 and where they connect to a 12F675 chip during programming.

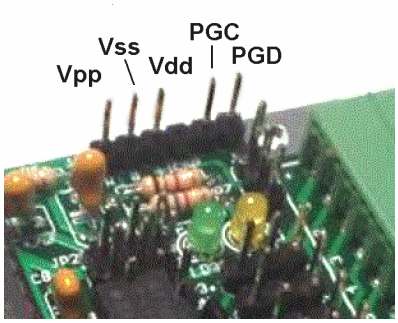


In practice, the connections are not direct. Instead, the cable from the programmer plugs into a module that holds the PIC chip.

This could be as simple as a piece of stripboard with an ICSP connection, or it could be a PIC development board.

Many modules also provide an ICSP connection, as shown below. This allows the module to be upgraded when faults are found or improvements are added. Some MERG kits use a pin order that is different from that on the Microchip programmer and a special connector cable has to be made up.

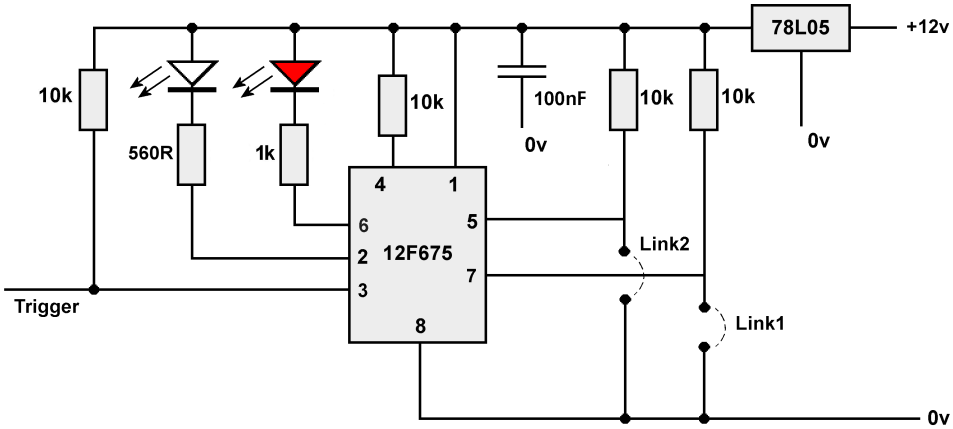




Typical PIC applications in MERG

Here are some typical examples of PIC microcontrollers for use with model railways.

Multi purpose flasher



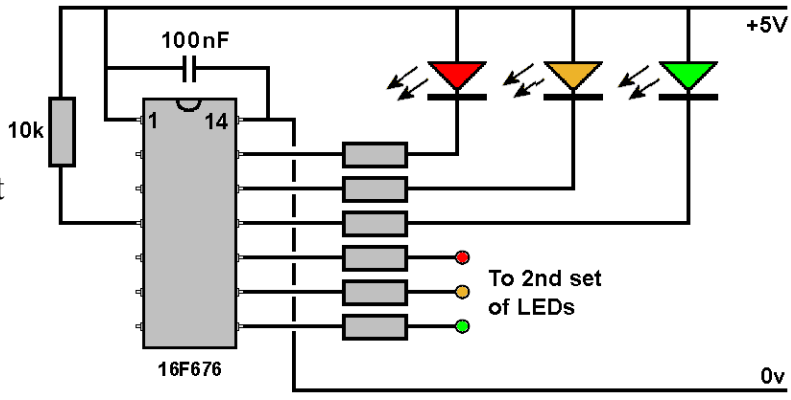
There are four different programs stored by the PIC, with the choice being determined by inserting links.

- A welder emulation, with bursts of a white or blue LED flashing, followed by a red LED slowly fading to provide the weld's afterglow (the use of PWM).
- Random flashes simulating arcing from a tram's trolley pole or an electric loco's pantograph pickup.
- Pulsing aircraft warning lights that are fitted on tall structures such as buildings, TV masts, power pylons, etc. (again the use of PWM).
- Simulating the flickering of a fluorescent tube on start-up.

The choices, sequences, random times and delays, plus the use PWM, would be difficult to design and would require many more components if it had to be produced with a collection of ICs and other components.

Traffic Lights

This circuit uses a larger PIC, as it needs more output pins than are available on an 8-pin PIC. Again, all the timing, sequencing and LED switching is being carried out by the single program stored inside the PIC.
It uses six pins as outputs, to provide a four-way traffic lights system.

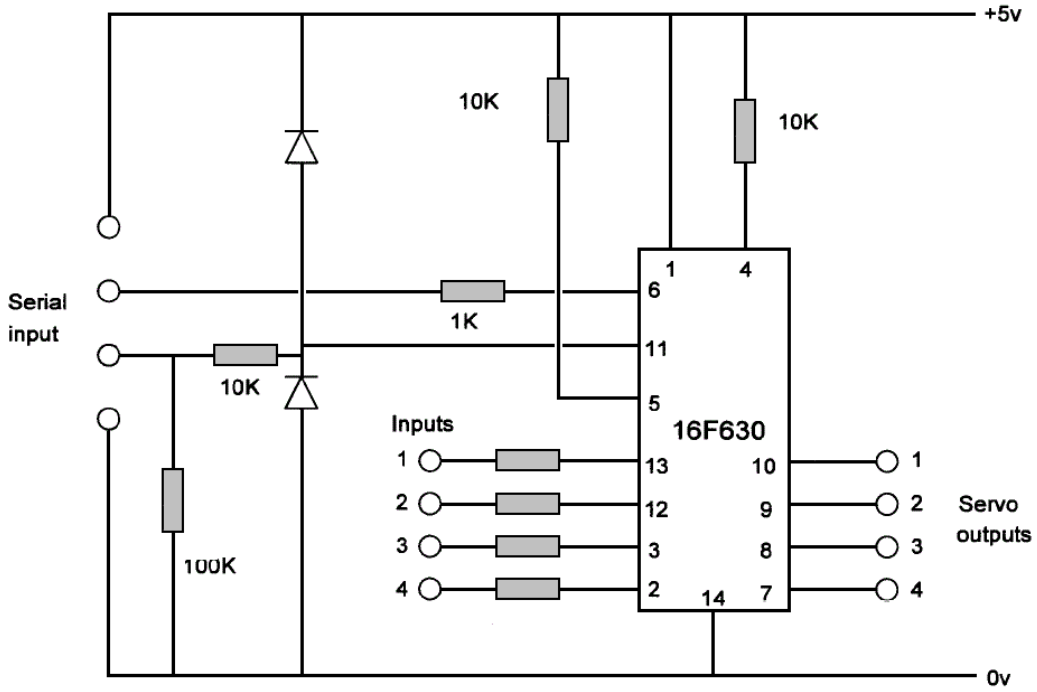


Note

The two modules above are covered in greater detail in the chapter on *'Pocket Money Projects'*.

Controlling 4 Servos

This is a simplified version of the MERG Servo4 board that is used to control four servos (for operating points and various animations).

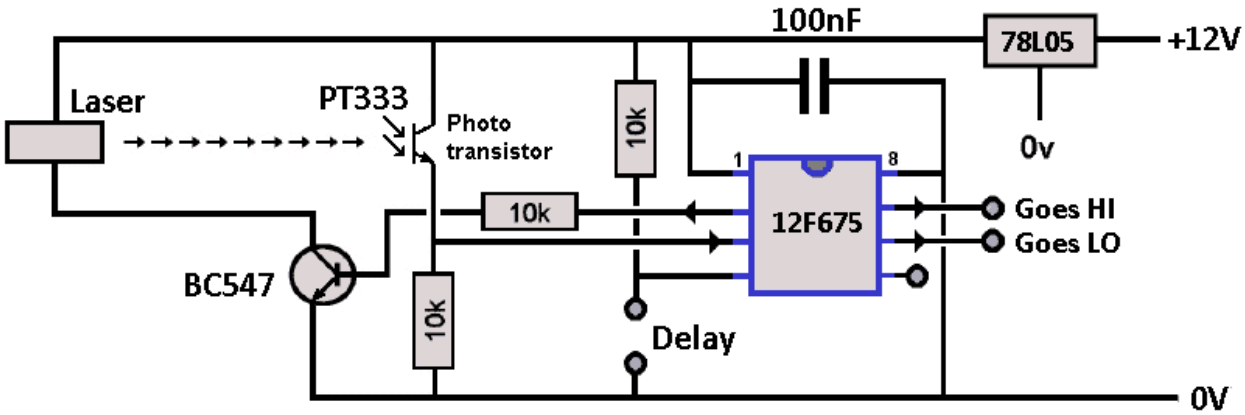


- Acts on inputs from switches, track occupancy detectors, a CANBUS module, etc.
- Each input, when switched, alters the corresponding servo's rotation .
- The PIC can be connected to a computer's serial port.
- The computer's software application can set the amounts of rotation (the '*endpoints*') for each servo.
- The final settings can be stored in the PIC's internal memory.

A number of PIC facilities are used here. It uses digital inputs, pulse width outputs and a serial interface. It is also able to store values after the power is removed.

Laser detector

The PIC takes analogue readings of the laser beam's intensity and switches at a preset change of level. It can also provide a short output change or a longer pulse, depending on whether the 'delay' link is fitted.



Writing your own programs

After a while, you may decide you would like to dabble in the world of programming chips.

Most enthusiasts start by modifying someone else's program until they become more familiar with programming language they have chosen.

For this, you use an IDE (Integrated Development Environment) which is a text editor with built-in syntax checking and compiling facilities.

Commercial IDEs are available, such as:

CCS PIC C Compiler
mikroPascal Pro for PIC
mikroBasic Pro for PIC
mikroC Pro for PIC
PICBASIC PRO

Perfectly good, and free ,versions are available, such as:

JALEdit (Just Another Language)

MPLABX

MicroPython

JAL and MPLAB have a large, active user base.

MPLAB X has two separate programs
IDE (the editor)
IPE (the programmer)

With JALedit, you create the hex file with the software and use any other programmer software (such as the PICkit or the MPLABX to carry out the programming side).

With MPLAB, you write your programs in the C language.
MPLABX can be downloaded from here:
www.microchip.com/mplab/mplab-x-ide

This installs both the IDE and the IPE

JALedit has fewer facilities but many find it easier to learn.
It can be downloaded from here:
www.justanotherlanguage.org/content/jallib/tutorials/tutorial_installation

Note

This chapter does not cover the writing of programs as that is a subject of an entire book - or more. Fortunately, there are many websites out there that are happy to guide you through your first steps.

Making life easier

MPLAB and JALedit are both high-level languages. As explained previously, this means that you do not need to know the intricate composition of the PIC chip you are using.

You just state what chip you are writing for and the compiler generates code that matches the chosen chip's architecture.

Even better, you can include code already written by someone else, to save you the trouble of working out how to do it.

These extra chunks of code are called *'libraries'* and can include routines that insert delays, read and write to EEPROM, write to LCD screens, etc.

This results in great savings in development time.

It is a two-way process. You may want to use a 4x4 matrix keypad in your program but the only library you find is for a 4x3 keypad. You amend it to suit your needs and then you can offer that new library for others to use.

Here are three lines of code from the JAL program that switch 10 LEDs on and off at random:

```
include 12F675
include random_10
include delay
```

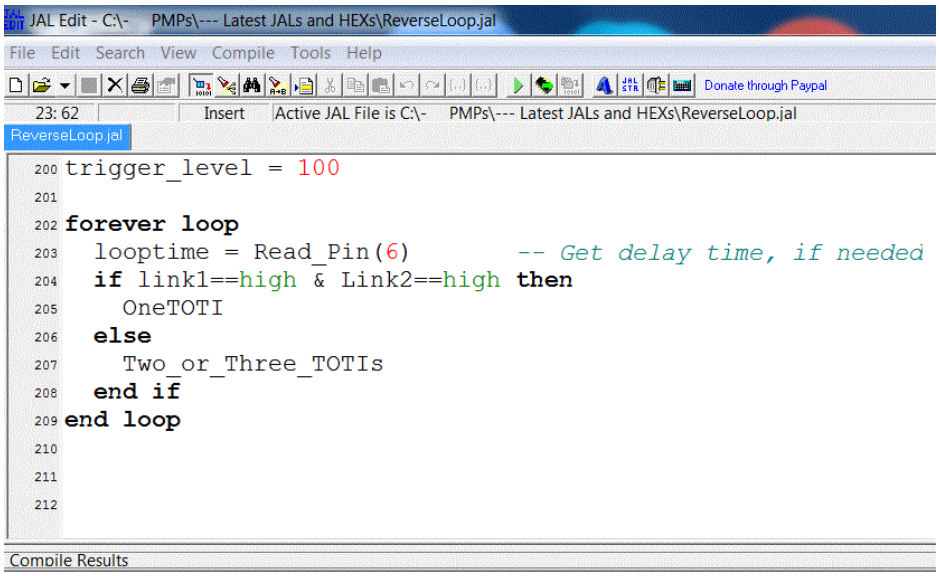
- The first line tells the compiler to include all the detailed routines that the 12F675 PIC uses.
- The second line adds in extra code that will generate random numbers between 1 and 10. This include started off as a library to generate six numbers for a dice program. It was easier to modify the dice library than start from scratch.
- The third line allows you to specify one, or many, different delays into your program. It is used in the random lights program to make the ten LEDs illuminate and extinguish in a random pattern.

These three extra libraries are stored on your hard drive and they are all added in to your code during the compilation process, so you end up with a single hex file.

JAL Edit

Here is what the JALedit screen looks like.

Notice that it colours different parts of the text to provide extra clarity.



Each line of code is numbered in the editor which is useful when you make a mistake, as the *'Compile Results'* window will tell you what line it found the fault in.

It is important to realise that while the compiler provides helpful advice on the faults it finds. It cannot know what is in your mind and what you are trying to achieve.

The compiler finds syntax errors but not logic errors.

Syntax refers to the specific set of words and their specific order of use inside computer instructions.

Examples of syntax errors

`includes 16f676` – includes is not in the set of defined instructions

`output1 = low high` – it can't be both at the same time in a single instruction

Examples of logic errors

`if door = closed then alarm = on` - you intended that the alarm will come on when the door is opened. You know that but the compiler does not; it is a valid statement to the compiler.

`totalprice = cost - vat` - you entered a minus instead of a plus. It changes the entire result - but is a valid statement to the compiler.

The syntax checker is very valuable but it cannot save you from all errors in your code.

Creating a JAL program

Here are the steps for creating a JAL application:

- Create a new folder for the program (helps to keep things organised).
- Open the JAL Editor.
- Close any files that may appear (JAL can be set to always load the last program you wrote).
- Click 'File' then 'New' to create a new JAL file.
- Type in your program code (the program instruction, any additional libraries, comments, etc.).
- When you click to 'Save' the program, it will be saved in the new folder with a .jal extension (e.g. if you called it flasher, it will be saved as flasher.jal).
- Click to 'Compile' the program.
- If there are no mistakes, a hex version and an asm version of the file will be created in the folder (e.g. flasher.asm and flasher.hex).
- If you have made any errors, the *'Compile Results'* window at the bottom of the screen tells you what mistake was made and what line it appeared on. Correct the mistake(s) and compile again. Repeat, if necessary, until the program compiles without errors.

Once you have a hex file, you can move on to using it to program your PIC.

For more information on PICs, just do a Google search for 'PIC tutorials' – there are more than you can read.

For information on the JAL programming language, a good starting point is this website:

http://justanotherlanguage.org/content/tutorial_book

What are Arduinos?

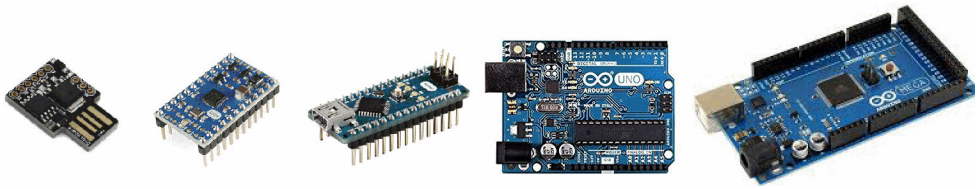
Arduinos are a range of module boards that use microcontroller chips produced by Atmel under the name of AVR (**A**lf and **V**egard's **R**ISC processor).
Atmel was subsequently purchased by Microchip, so the company now produces the two mostly commonly used types of microprocessors used by hobbyists.
The AVR range of microcontrollers maintained its name after the purchase.
The recorded sales figures are out of date but were a healthy 7 billion in 2015.
Although you can buy the Atmel microcontrollers as individual chips, they are more commonly available to hobbyists as Arduino modules. The microcontrollers are mounted on boards to provide extra features (voltage regulators, USB, serial and I^2C connections, etc.).

There is a constantly evolving family of Arduinos and the large list of types can be found here:

https://en.wikipedia.org/wiki/List_of_Arduino_boards_and_compatible_systems

You will find a long, long list of Arduinos types and compatibles.

At the time of writing, the most common Arduinos use in model railways are those shown below. They are, from left to right, and in increasing size:



ATtiny Pro Mini Nano Uno Mega

They uses the following microcontrollers:

ATtiny	ATtiny
Pro Mini	ATmega 328
Nano	ATmega 328, with older versions using the ATmega168
Uno	ATmega 328
Mega	ATmega2560

Like the Microchip PIC, Arduinos use three blocks of memory.

- Flash memory to store the program.
- RAM to store temporary variables while the program runs.
- EEPROM to store long-term information, after the Arduino is powered down.

Here are the characteristics of each type:

Name	Speed	ADC	I/O	PWM	EEPROM (Storage)	SRAM (Variables)	Flash (Program)
ATtiny	8MHz	4	6	15	512B	512B	8kB
Nano	16MHz	8	14	6	1kB	2kB	32kB
Mini	16MHz	8	14	6	1kB	2kB	32kB
Uno	16MHz	6	14	6	1kB	2kB	32kB
Mega	16Mhz	16	54	15	5kB	8kB	256kB

Where ADC is the number of analogue-to-digital converters, I/O is the number of digital Input/Output channels and PWM is the number of channels that can be used for Pulse Width Modulation outputs.

Typical Arduino pinouts

Here are the connections for the Arduino Nano.

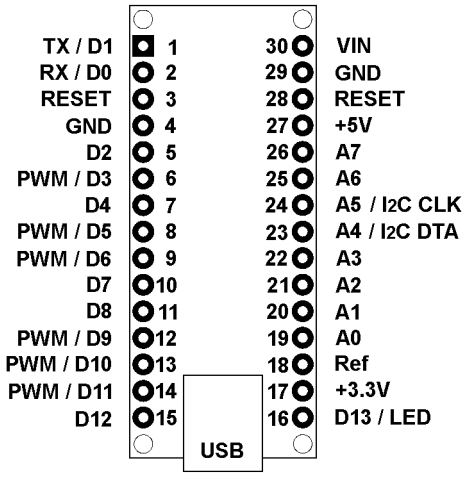
The Nano is the board that is used in the EzyBus Output Modules, the Sound Player, DC++ and many more model railway projects.

The chart shows the use of each pin.

Bi-directional means that that pin can be used either as an input or as an output.

The square connection marks pin 1.

There are four other holes round the board and these are only used for mounting purposes.



Pin	Name	Purpose
1	TX / D1	* Used when transmitting serial data * Bi-directional digital I/O
2	RX / D0	* Used when receiving serial data * Bi-directional digital I/O
3	RESET	External reset. When this pin is temporarily brought down to 0V, it forces the program to start all over again from the first instruction
4	GND	0V
5	D2	Bi-directional digital I/O
6	PWM / D3	* Pin is capable of outputting PWM (pulse width modulation) * Bi-directional digital I/O
7	D4	* Bi-directional digital I/O
8	PWM / D5	* Pin is capable of outputting PWM (pulse width modulation) * Bi-directional digital I/O
9	PWM / D6	* Pin is capable of outputting PWM (pulse width modulation) * Bi-directional digital I/O
10	D7	Bi-directional digital I/O
11	D8	Bi-directional digital I/O
12	PWM / D9	* Pin is capable of outputting PWM (pulse width modulation) * Bi-directional digital I/O
13	PWM / D10	* Pin is capable of outputting PWM (pulse width modulation) * Bi-directional digital I/O
14	PWM /D11	* Pin is capable of outputting PWM (pulse width modulation) * Bi-directional digital I/O
15	D12	Bi-directional digital I/O
16	D13 / LED	* Bi-directional digital I/O * Built-in LED (used for test purposes)

17	+3.3V	+3.3V output
18	Ref	ADC Reference Voltage. Connect a voltage (0-5V only) as a reference for analog conversions
19	A0	* Analog input * Can be configured as a digital I/O
0	A1	* Analog input * Can be configured as a digital I/O
21	A2	* Analog input * C* Analog input * Can be configured as a digital I/O
22	A3	* Analog input * Can be configured as a digital I/O
23	A4 / I2C DTA	* Analog input * Can be configured as a digital I/O * Transmits or receives data when used for I2C
24	A5 / I2C CLK	* Analog input * Can be configured as a digital I/O
25	A6	* Analog input * Can be configured as a digital I/O
26	A7	* Analog input * Can be configured as a digital I/O
27	+5v	* Can supply +5V to external sensors. etc. * Sometimes used as the main supply when fed with a regulated +5V power supply.
28	RESET	Duplicate of pin 3
29	GND	Duplicate of pin4
30	VIN	Power for the Nano. Can be a regulated or an unregulated supply of between 7v and 12V.

Apart from the LED that is connected to pin 13, there may be other LEDs on the board and these may be labelled as POW, RX and TX.

The POW LED illuminates when power is connected to the board.

The RX LED lights when the board is receiving data.

The TX LED lights when the board is transmitting data.

While being programmed, an Arduino can draw its power from the USB connection. Connecting external power is optional this stage.

The Arduino Uno uses the same ATmega 328 microprocessor as the Nano so its specification (speed, amount of memory, etc.) are identical.

The Mega is a larger board with a more powerful ATmega2560 microprocessor and provides superior performance (faster speed, more memory and a whopping 54 I/O pins).

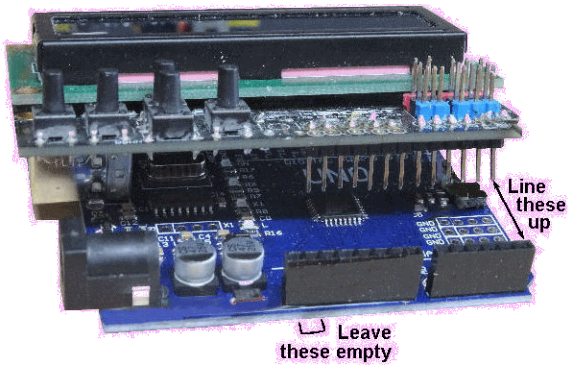
Adding shields

A huge advantage for some Arduinos is the ability to add extra functionality without having to solder in other modules.
An Arduino *'Shield'* is a board that plugs directly in to some of the Arduino modules, with no soldering. Unos and Nanos have strips of sockets down each side that the shields can plug into. A subtle difference in the pin spacing ensures that shields can't be connected the wrong way round.

Example shields for Uno and Megas are keypads, touch panels, joysticks, LCD screens, relays, MP3 players, motor drivers, prototype boards, wifi, Ethernet, GPS, etc.

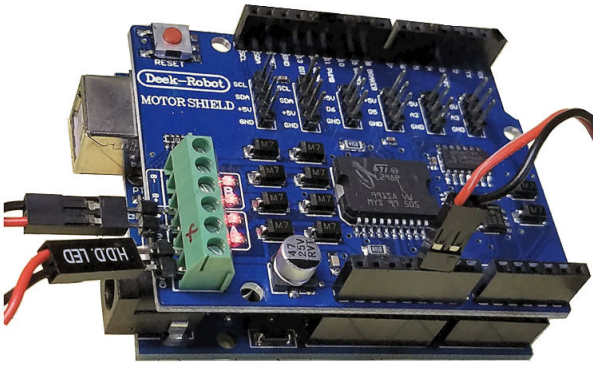
You will lots of examples described at:
learn.sparkfun.com/tutorials/arduino-shields-v2

This image shows an LCD screen, with buttons, being plugged into an Arduino Uno.



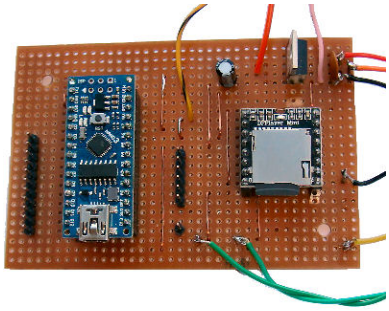
There is soldering involved and is all we need to create the Master Module for the EzyBus layout control system.

No hardware worries, just programming required.

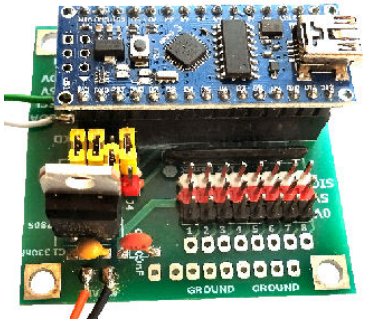


This image shows a motor shield plugged into a Uno, although a Mega could also be used here. This combination of Arduino and motor shield is the basis of the DCC++ system, providing a simple and cheap alternative to commercial DCC controllers. The software for this project is open source.

Not all Arduinos have the necessary socket strips to support a shield.
Also, shields are not available for every purpose.



You can mount an Arduino on a piece of stripboard. This method was used to embed an Arduino Nano in the Sound Player module.



You can also mount an Arduino in a custom designed PCB, as is the case with the EzyBus Output Modules.

Both of these non-shield methods are used on model railways to produce a variety of customised effects such as controlling servos, signalling systems, animations, light and sound effects, and so on.

Arduino libraries

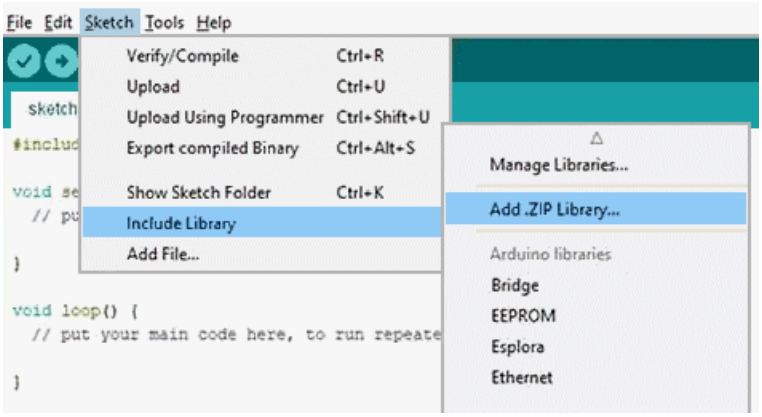
These are similar to the libraries available for PIC programmers, although they are written for Atmel microcontrollers.
These extra chunks of code that you can add to your program without any additional coding.
These *'libraries'* include routines that read and write to EEPROM, write to LCD screens, handle servos and steppers, communicate via SPI or Ethernet, etc.

All to save you time that would be spent doing it for yourself.

With PICs, you added instructions into the code you were writing, such as

```
include delay
include random_10
```

When you install the Arduino IDE, you automatically save a collection of the most popular libraries for possible use. To include a library in your software, you select it from menu options as shown below.



For example, if you chose EEPROM from the displayed list, it adds this line into your code for you.

```
#include <EEPROM.h>
```

If you buy a particular piece of electronics that is not already in the standard set of libraries, you can add the libraries its uses, by selecting the '*Add ZIP library'* option from the drop-down menu.
For example, the DFPlayer code has to be added to support that module.

```
#include <DFPlayer_Mini_Mp3.h>
```

Note

You do not unzip a library and try to integrate into the IDE.
Once you have the appropriate ZIP file, you simply choose to add it to the libraries and click on the ZIP file.

Adding an Arduino library automatically adds a number of example programs to show you how to use that library.
Many other includes are available to support other devices such as infrared detectors, I2C devices, the DFPlayer, keypads and a variety of sensors.

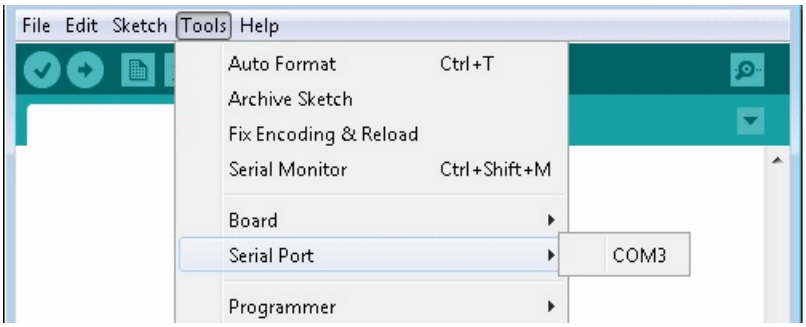
Sometimes you need a routine that is different or better than the ones in the standard list.
The EzyBus Output Module handles up to eight servos.
The VarSpeedServo library builds on the standard servo library by allowing up to 8 servos to move simultaneously or in a sequence. It also lets you set the speed of the moves. It adds this line to your code:

```
#include <VarSpeedServo.h>
```

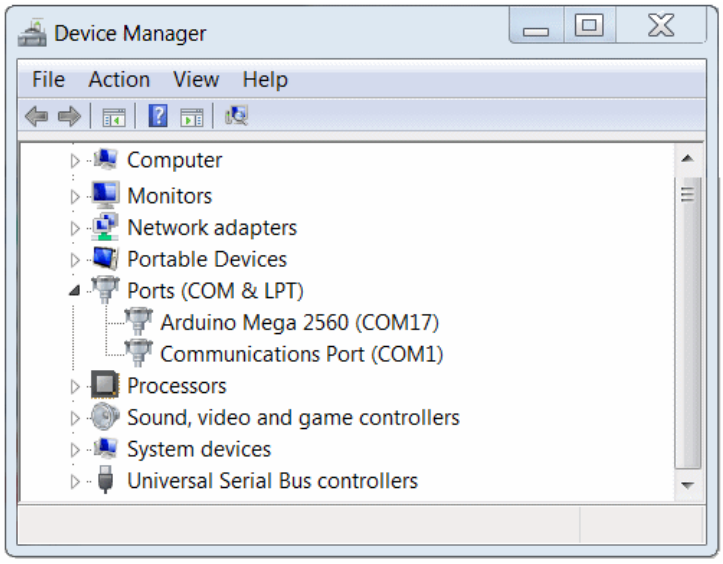
How do we program Arduinos?

There is no requirement for an external programmer between your computer and Arduino Nanos, Unos and Megas.
Their onboard electronics handle the USB interfacing.
A simple USB cable between the computer and the Arduino is all that is required. The ATtiny has a USB plug as part of its PCB and it plugs directly into your computer, with no need for a connecting cable.

The program's compiled code is sent out the computer's USB socket in serial format and so you have to choose which serial port you are using for the transfer.
Sometimes, only one port will appear on your list and you simply confirm it as your choice.



If more than one port is listed, you have to select the one that your Arduino is connected to.
You could simply try one at a time and see what happens.
A better method is to use the Windows Device Manager.
This utility lists all the hardware attached to the computer and you will find your Arduino in the the section called '*Ports'*. Take a note of the Arduino's COM port number, return to your IDE and select that port from the drop-down list.



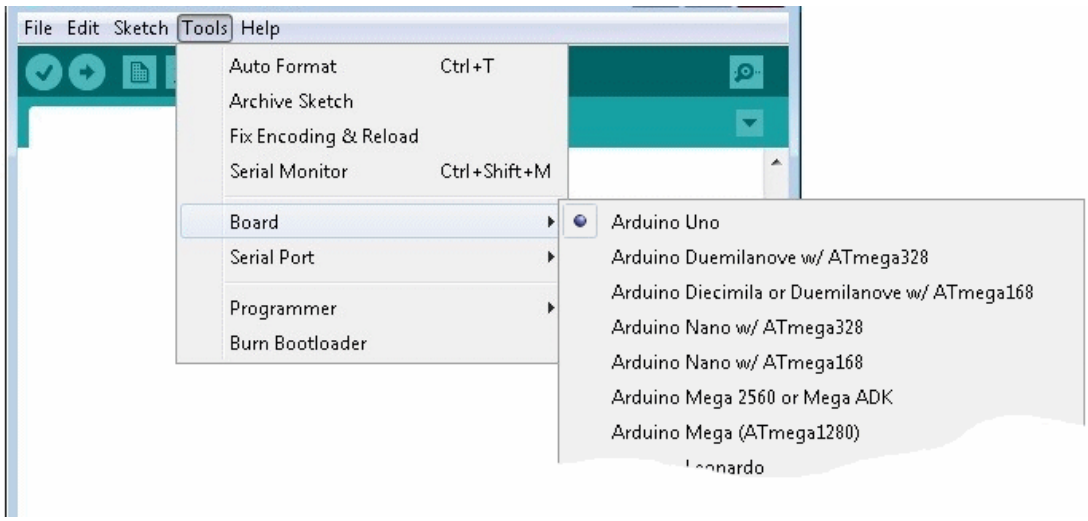
Some other Arduinos, like the Pro Mini do not has inbuilt USB support and need a programming adapter, similar to the one used for programming PICs.

Creating an Arduino program

With the PIC, you had to add a line of code to tell the compiler what type of chip the software was for.

```
include 12F675
```

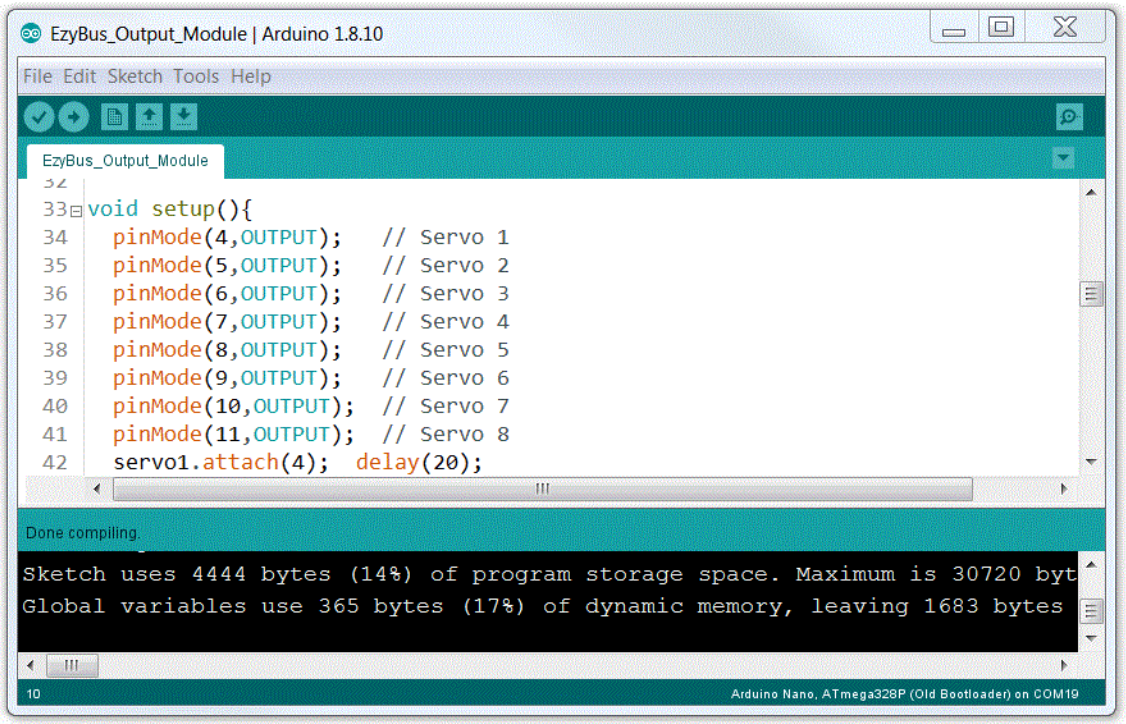
In the Arduino IDE, you choose the target board from a drop-down menu as below:



Once you have selected your target board, you can enter your programming instructions into the large code window.

When you click the arrow icon, or select '*Verify/Compile*' from the '*Sketch'* drop-down menu, any syntax errors are reported in the lower message window.

In the example, the code compiled correctly and the message box tells you how much of the microcontroller memory you have used.



Writing software

This chapter cannot hope to even start covering all the programming instructions for PICs or Arduinos. There are countless books that cover programming in great detail. This section just gives a taste of the methods you can use.

Programming for model railways is, in essence, about

Stuff in - switches, train detectors, etc.
Stuff out - lights, servos, animations, sounds, etc.

Sometimes a program is only about 'stuff out' (e.g. an animation that runs at fixed periods) but most programs make use of external inputs.

There are many instructions you can include in your program and the two most commonly used ones are:

- Decision making
- Looping

You will use these code instructions in most programs and here is what they can do.

Decision making

This is known as *'flow control'* and results in some line of code being run or not, depending on some condition.
For example:

```
if train at station
    then switch on station lights
```

```
if train moving
    then switch on coach lights
```

Looping

This is known as '*iteration'* and it results in a section of your code being run multiple times.
For example:

```
for 10 times
  switch LED on
  switch LED off
```

This will flash a LED ten times. You would insert a delay instruction after switching on and switching off, with their values determining the speed of flashing and the ratio between on time and off time.

Sometimes, you want to keep running the same section of code until something stops it.
For example:

```
while the button is pressed
   rotate the turntable one step
```

```
repeat
   rotate the turntable one step
until the button is pressed.
```

In the first example, the turntable will only rotate if the button is pressed. If you have not pressed the button, the program carries on to the next line of code.
In the second example, the turntable is rotated and then the button is checked to see whether you pressed it. That way, the turntable is rotated at least one step.

The PIC's and Arduino's code instructions also allow for many other activities such as

- Reading digital and analogue inputs
- Changing the status of output pins, between 0V and +5V
- Writing to eeprom
- Communicating with a variety of systems (e.g. serial, CAN, I2C, SPI)

Getting started

Many hobbyists begin programming by loading someone else's code and making small alterations to see if it then runs the way they expect.

The next step is write your own program from scratch. This can be fairly easy with a small program but is very rewarding nevertheless.

When you contemplate larger programs, you need an organised approach. There is a whole army of systems analysts out there, with formal approaches to determining what the customer wants and documenting the smallest detail. A long period can be spent just clearly laying out the tasks before a single line of code is written.

We don't need to be quite as strict as that but it is worthwhile stating your aims on paper before starting coding. If you can't describe it on paper, then you can't expect to be able to write a working program.

A useful method is called *'pseudo code'*. This is not code that can be compiled and loaded into a microprocessor. It is series of short statements, in plain language, of what the program should do step-by-step. This makes you consider the sequence of instructions and the logic involved. Documenting the program clarifies its structure, its functions and its data flow. As a result, it sorts out many errors before you start entering any program code.

You do not have to use pen and paper. Both JALedit and the Arduino IDE let you enter comments in your code. They are ignored at compile time but are normally used to explain what a piece of code is doing (very handy when someone else is trying to figure out what your program is doing).

Using pseudo-code, you can enter a list of comments into a blank edit screen. Once you are satisfied with the outline, you can flesh out the comments with actual code.

Example

This barge has to traverse back and forth along a canal, stopping at each end and also in the middle. Each time it stops, local activities should take place.

The pseudo-code for this is:

```
-- move barge to RHS
-- move barge leftwards to mid position
-- mid_activities
-- move barge leftwards to LHS
-- LHS_activities
-- move barge rightwards to mid position
-- mid_activities
-- move barge rightwards to RHS
-- RHS activities
```

At this stage, we have not yet worked out how to achieve these tasks with programming code.

Some hobbyists could decide to implement the program on a PIC while another may choose to program an Arduino. That is the beauty of pursue-code; it is independent of any programming language.

In practice, some lines of pseudo-code may be achieved with a single instruction.

In most cases, they will need several lines or even a whole chunk of code.

To save programs becoming one long program that it is difficult to decipher, programmers create *'sub-routines'*.

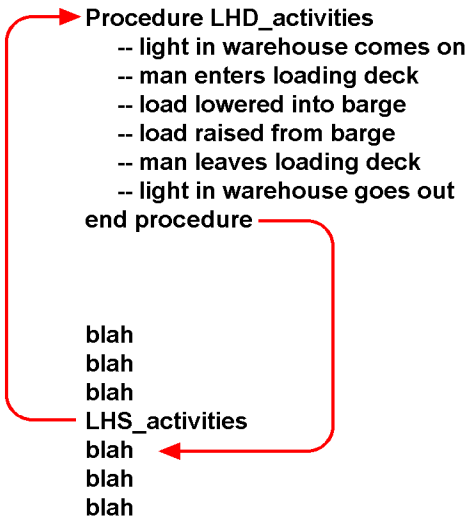
A sub-routine is written as an independent block of code within your program.

The sub-routine will have a name and it can be run by calling its name from a line of code somewhere in the program. Indeed it could be called in many different places in a program.

That saves programming space and means that you are not writing the same code over and over again in different parts of your program.

In the example pseudo-code, the mid_activities occur twice. We only write the code once but call it twice in different points in the program.

In JALedit, these subroutines are called *'procedures'*, while they are described as *'procedures'* when writing code for Arduinos.



This illustration shows what happens when the program is running.

At some point in the program, a line of code calls the LHS_activities procedure.

The program leaves the main program and runs the lines inside the procedure instead.

When all the lines in the procedure have been run, control passes back to the main program.

Notes

- When you first write the program, the procedure can be empty shells, waiting to be filled out at a later stage. Indeed, in the example, the procedure's lines are all comments and these will be augmented by actual code later. If one of these lines requires lots of independent code, the procedure may even call a further procedure.
- You can build up a program in stages and each stage can be checked as you go along.
- It makes fault-diagnosis much easier. If the left hand activities are not working properly, you know exactly where to look.
- Even a large application might have a main program with a small number of program lines but lots of calls to sub-programs. The Master Controller for the EzyBus system, for example, has almost 1,100 lines of code but the main program only needs 30 lines.
- Once you have written a sub-program you can cut and paste it into other programs for re-use.

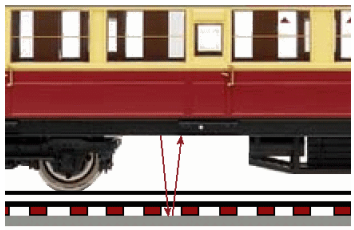
Data memory

Your program is tucked away in its own chunk of memory and will stay there unless you want to alter it.
There is a separate chunk of memory allocated for the temporary storage of data as the program runs. This temporary data will usually change its content as the program progresses.
One important element of all programs are *'variables'* that are stored in the data memory. Their values are used while the program runs and are lost when the microprocessor is powered down.

Examples

- If a program wants to flash a LED ten times, it needs to know how many times it has carried out that task as it runs. To achieve this, you allocate a small block of memory and give it a name (e.g. 'count'). That now becomes the store as the LED flashes. Each time the LED flashes, the value in the count variable is incremented. After the tenth time, the program spots that the variable called count has reached ten. At that point, the program moves on to its next instruction. As a programmer, you do not need to know the exact memory location. You just allocate a variable and let the compiler take over.

- The automatic coach lighting kit keeps checking the level of light reflecting off the track and reaching its detector. It compares that value with a nominated trip value. Every time it checks, it will result in the value being stored in a temporary location - e.g. in a variable called 'sample'. As the readings are being taken very frequently, the contents of the variable are constantly being updated.



- The EzyPoints kit has two variables that store the current position of the servo and also the new position the servo has to move to. To find out how far to travel, the contents of the two variables are subtracted and the result is placed in a third variable.

In JALedit, the formula would be expressed this way;

```
difference = newposition-current_position
```

In this example, a variable is being used to store the result of calculations within the program.

In JALedit, for the PIC, these variables would be declared like this:

```
VAR byte count        VAR byte sample      VAR byte difference
```

while with the Arduino IDE they would be:

```
byte count            byte sample          byte difference
```

In these examples, the variables have been allocated names that describe what they are storing. This helps us and others to understand their role in the program.
Consider the confusion, especially in a large program, if all the variables were called var1, var2, var3 and so on. Imagine returning to your program after some time away and trying to recall what the variables x, y, z, aa, qq, etc. were for.

Linking your software to your hardware

PICs and Arduinos have multiple pins that can be used for inputs or outputs.
You may want to connect switches or other input devices to some pins, while connecting LEDs or other output devices to other pins.

Naming your pins

The microprocessor needs to know which pins you want to use for these purposes.
In your code, you allocate a name for the each pin you are using.
Again, use meaningful names. Avoid using 'pin 16' or 'pin D5'
In JALedit, suitable assignments might be:

```
var bit RedLED is pin_GP1
var bit switch is pin_GP2
```

while for the Arduino it might be:

```
RedLED = 7        (i.e. this is pin D7 not pin 7)
switch = 8        (i.e. pin D8, not pin 8)
```

Once that is done, you can simply refer to 'switch' and 'RedLED' in your code and the compiler takes care of it all for you. You have used meaningful names in your program and the compiled version has converted them into their actual pins on the microprocessor.

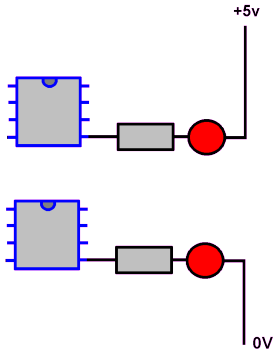
In or Out

At this stage, you have allocated names to pins but there is a further stage before you carry on. A digital pin can be used as an input or as an output and it is up to you to specify which use you are putting each pin to.
In JALedit, suitable assignments might be:

```
pin_GP1_direction = output
pin_GP2_direction = input
```

while for the Arduino it might be:

```
pinMode(RedLED, OUTPUT);
pinMode(switch, INPUT);
```



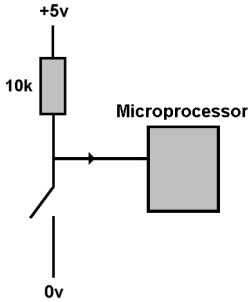
The program can any one of its outputs go high (+5V) or low (0V).
In these illustrations, a pin connects to a LED and its dropper which then either connects to +5V or 0V.
In the upper image, the LED will illuminate when the output pin goes low. This called *'sinking'* the current. When the output pin goes high, the LED will extinguish there is no voltage drop across it (it has +5V at each end).
In the lower image, the LED will illuminate when the output pin goes high This called *'sourcing'* the current.

Note

The output pins of microrocessors are not intended to provide high amounts of current. If you need to drive a high current device from an output pin, use a transistor or amplifying IC (see the chapter on Interfacing).

Digital inputs

PICs and 5V Arduinos expect the voltage on their input pins to one of two states; either fully on (+5V) or fully off (0V), although there is a degree of tolerance.
The illustration shows a 10K resistor holding an input pin high. When the switch is thrown, the input pin drops to 0V.



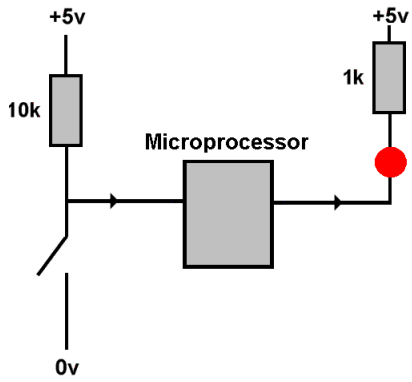
Warning

Be very careful that you never wire a switch to a pin you have configured as an output. If the output is sitting at +5V and you throw the switch, a high current will flow and the microprocessor will probably be damaged.

Example small program

If we combine the input and output illustrations above we have this very basic circuit.
It does not do a lot. It is only used to illustrate how simple some basic code can be.

- Throwing the switch brings on the LED.
- Releasing the switch puts the LED back off.



Possible code in JALedit would be:

```
if switch==high then
   RedLED=high        // extinguish the LED
end if
if switch==low then
   RedLED =low        // illuminate the LED
end if
```

while the Arduino version would be:

```
switch_state = digitalRead(switch);
if (switch_state == 0)
 {digitalWrite(RedLED, LOW);}   // illuminate the LED
if(switch_state==1)
 {digitalWrite(RedLED, HIGH);}  // extinguish the LED
```

In both cases, the first step is to check the state of the input pin. The state could be either +5V (also described a s 'high' or 'on') or 0V (also described as 'low' or 'off').
The program consists of two conditional tests.
If the switch state is low, then the output pin is brought low and the LED illuminates.
If the switch state is high, the output pin is brought high and the LED extinguishes.

The above code is part of a program (don't forget to add includes, pin assignments, etc.)
This simple program can be easily extended so that multiple activities are initiated from a single input.
Consider this piece of pseudo code for an automated good shed sequence:

When train arrives at the goods shed
    Sheds's outside light comes on
    Foreman comes out
    Shed's interior light comes on
    Sliding doors open
    Crane rotates round
    Truck comes round

The JAL code for this is bundled into a self-contained sub-program:

```
procedure shed_activities is
    exterior_light = on      delay_1S(2)
    man = on                 delay_1S(7)
    interior_light = on      delay_1S(1)
    door = on                delay_1S(7)
```

```
    crane = on                delay_1S(7)
    truck = on
end procedure
```

Some of outputs control lights and some control servos, and there are a few seconds delay between each activity.
This is only a snippet from a larger program but it demonstrates that even a large program is made up from lots of little sub-programs.

Note

This chapter only seeks to give you an insight into programming PICs and Arduinos, to improve or augment model railway activities. The next steps are up to you.

There is an abundance of support for those who want to take their first steps.

There are excellent books, online tutorials, and YouTube videos.

Also, check out the support groups:

www.microchip.com/forums/
https://forum.arduino.cc/

We have not even looked at the many existing projects that are out there:

- Stepper motors controlling turntables
- Sound players
- Signal sequencing
- Automated fiddle yards

The ideas and the implementations are constantly evolving.

Newer versions of both PICs and Arduinos continue to offer more memory, faster speeds, more facilities, etc.

It is an interesting and rewarding area for model railway enthusiasts.