Chapter one is rather large but if you’ve never programmed PICs before then there’s a bit of a steep learning curve whilst you get to grips with some new technologies and ideas.

This chapter will cover the following subject areas:

- Brief introduction the BASIC programming language and why it’s great to use with PICs and other embedded CPUs.
- Brief (very) introduction to PICs.
- Creating your first BASIC program
- Assembling a simple PIC circuit on a breadboard

The BASIC programming language – a VERY brief history

BASIC – Beginners All Purpose Symbolic Instruction Code has its origins way back at the dawn of computing, and was created as a educational and teaching programming language. It found its way onto almost all modern home computers in the 80’s, and when Microsoft released their first version of the MS-DOS Visual Basic and then Windows Visual Basic, its continued place in history as one of the most used programming languages ever, was secured.

Now, many programmers don’t like BASIC and there are several reasons for this; some of them are even perfectly correct.

There are no standards for BASIC which is a shame. Manufacturers are free to create their implementation of BASIC in anyway they like, and this has caused a complete free-for-all.

The classic is the line-number debate is an example.

A very simple BASIC program used to be written like this:

10 PRINT “Hello World”
20 GOTO 10

This two line program would display the text “Hello World” on the VDU (or print it on an old fashioned tele-printer in the good old days). The program would then execute to GOTO instruction which said, jump back to line 10, and carry on executing from there. So in this case, the program would execute the PRINT instruction again. This program is of course rather stupid as it will just keep printing “Hello World” till either somebody either stops it, or the printer runs out of paper.
In case you’re wondering, line numbers typically went up in 10’s so that you could slip additional lines in-between if needed, like this:

```
10 PRINT "Hello World"
12 PRINT "I forgot to add this"
20 GOTO 10
```

Line numbers were a pain, but a necessary evil in the early days. Programs of hundreds or thousands of lines long had GOTO’s all over the place. You would see GOTO 1032 and have to go to that part of the program and look to see what happened there, than flip back to where you were originally looking and the longer the program, the worse the problem got. You would have computer program listing printouts, sometimes hundreds of pages long, and you would be flipping up and down trying to follow a program.

Then some bright spark decided that actually lines numbers were a bad idea, and we only needed them in certain circumstances.

```
10 PRINT "Hello World"
   PRINT "I forgot to add this"
 GOTO 10
```

The above makes a bit more sense. Only the first line has a line number on it because we need a “label” for the GOTO instruction on the 3rd line to reference. The next logical progression was when somebody thought that line numbers were meaningless now, and it would make more sense if we could have sensibly named labels, so we ended up with this:

```
STARTAGAIN: PRINT "Hello World"
   PRINT "I forgot to add this"
 GOTO STARTAGAIN
```

The colon at the end of the label is there to tell the computer that all the text to the left is a label. There are some general rules for labels; aren’t there always. Labels must start with a letter (A to Z), can contain digits if you want, cannot contain spaces and should be under a certain length – in our case 32 characters or less. There are also restrictions on the actual label name; it cannot be a name of an instruction or other reserved word that the language recognises.
To save space in the program listings, labels are now often placed on a line on their own, like this:

```
STARTAGAIN:
    PRINT "Hello World"
    PRINT "I forgot to add this"
    GOTO STARTAGAIN
```

If the label name is made something meaningful like “CALCULATETOTAL” or “CalculateTotal” then when you see “GOTO CalculateTotal” you know exactly what’s happening and probably don’t need to go find the CalculateTotal block of code. The only downside of labels is that you’ve no idea where the routine is. GOTO 10 or GOTO 10201 give you an immediate rough idea where to look; 10 will be up near the start of the program probably, and 10201 a good way down; CalculateTotal can be anywhere.

Because of this a convention started which makes a lot of sense to adhere too. Routines should be at the end of your program. But we will cover all this in more detail later on.

All modern BASIC’s now look pretty much like the example above.

**BASIC is slow... isn’t it?**

Programming languages are either compiled or interpreted.

Interpreted means that the computer has a piece of software pre-supplied that reads your program one line at a time, looks at it, checks to make sure that it makes sense, converts it into machine language that the CPU will recognise, and executes it. Then it moves onto the next line of your program. If your program has a loop; GOTO 10 for example, then it has to re-interpret line 10 every time it needs to execute it which in many ways is rather silly and a waste of time, especially if it’s already converted it to machine code previously.

Compiled means that the computer when instructed, runs through your program making sure that there are no obvious syntax errors, converts the program text into machine language, then executes your program.

Compiled programs run faster because your program has already been syntax checked and converted into something that CPU recognises and this only happens once, however, program development can take a bit longer as after every change you make, you have to re-compile your program. Fortunately, modern compilers and computers reduce this time to a few seconds; not like the many hours it used to take.
However, interpreted languages have their place. The programmer can stop the program running mid way through, look at the contents of variables, often change variable contents if required, sometimes even make changes to the program and let it then continue on from where it was stopped; interpreted programs are usually a breeze to debug and fix.

Traditional implementations of BASIC have always been interpreted as it made more sense when teaching students. It would be a slow learning process if after every program change or correction you had to set off a compile process that could take many minutes or even hours to complete.

Now we jump forward to today and from this point on, were only going to be concerned with BASIC for PICs and are going to ignore all the confusion that surrounds the use of BASIC on other platforms, and this is because programming PICs (and all embedded CPU’s) is completely different to programming computer systems.

The BASIC we will be using is actually compiled. You write the BASIC program in a special programming environment; the IDE (Integrated Development Environment) that runs on your PC. You press the compile button when your ready and the software on your PC checks your program for errors, and converts, via an indirect way, to machine code (HEX) that can be sent directly to the PIC for execution.

There are other PIC programming languages available like “C”, JAL and they tend to work the same way, but there are some exceptions like FORTH.

What this means is that what’s sent to the PIC is machine code that it can execute directly without having to interpret it.

So the only thing that makes one programming language execute faster or slower than another, is how well the software running on your PC can convert your program to machine code.

To quickly confuse the issue, there are some PIC based modules that do run interpreted BASIC. Do a web search for “PICAXE” or “Parallax BASIC STAMP”, but we won’t be covering these devices in these articles.
Portability.
Portability is how simple is it to move your code from one platform to another. In the PC and large computer world, this is a big issue but not so much in the embedded world; PICs and ATMEls for example.

Each PIC is different; some have CAN bus interfaces, some have USB support, some have lots of RAM, some have multiple PWM channels. Because they are different it’s not that common to need to move programs from one PIC to another. However, if this is required, it’s usually just a matter of changing a setting in your compiler to indicate a different PIC, making changes that reflect the different hardware resources available, and then hitting the compile button again.

One reason for moving from one PIC to another is either the original device has gone end-of-life and no longer available, or, program / hardware changes are required and the original device is no longer suitable.

Language Standards.
This is where some people’s blood starts to boil.

BASIC has standards, but they are more voluntary than enforced. The statement GOTO for example seems simple enough; the instruction tells the CPU to jump to a different part of the program and start executing from that point. Some BASICs would allow GOTO 100 or GOTO 100 + X (where X is a variable containing an integer). Some BASICs allow variable names of only a couple of characters in length, some allow huge variable names. Each manufacturer is free to “bend” BASIC in anyway they like, and this can cause problems for people moving from one version of BASIC to another, but all programming languages suffer from this to an extent.

“C” has ASNI (American National Standards Institute) standards. A panel of rather smart people sit down every now and again and release a new standard that says exactly how the “C” language will work in that revision. Programming languages MUST evolve over time as new technologies and techniques are developed and “C” is no exception. However, “C” doesn’t provide for control of hardware devices like USB or CAN bus etc. In fact, “C” has very few built in instructions.

So, the manufactures of “C” extended the language with libraries. These libraries contain chunks of “C” or machine code that make using these hardware resources by the programmer a lot simpler, but the ANSI committee don’t have any control over this. They don’t know if the “C” version you’re using is on an IBM mainframe computer, a mobile phone or a PIC, so each manufacture, in the belief that that way is the best and only way, implement libraries for their compilers in their own way and for their own needs. One manufacture may give you a USB
library that has a function called “USB_START”, another may call it “USB_OPEN”, or “USBOPEN”. They may also accept different parameters or the same parameters but in a different order.

**The Bottom Line.**

The bottom line is that I’ve tested some simple “C” programs written with a Microchip compiler against some BASIC programs written with the AMICUS18 BASIC compiler and on balance they averaged out the same execution time.

Also, no matter what programming language you write your code in, BASIC, “C” etc, it probably won’t be portable across different language compilers, often, even if the compilers are from the same manufacturer.

It’s also true to say that a bad programmer can write buggy, slow code in any programming language, and the language won’t do anything to prevent it.

**A brief Introduction to the PIC.**

Were going to take a brief tour of the PIC in this section and won’t go into too much detail here. Things will be explained in more detail as the need arises; it’s just a good idea that you have a brief understanding of PICs at a high level.

PIC’s are almost self-contained computers on a single chip as all they are missing are the physical input/output devices (VDUs, sound cards, keyboards etc) that we are used to seeing on more traditional computers.

As a minimum, PICs contain program memory, data memory and some pins that can be configured as digital inputs or outputs.

The model of PIC we will be using in this series also contains a lovely assortment of other internal goodies like EEPROM, communication devices, timers and other blocks that we will explore over time.

**CPU Core and memory**

At the heart of the PIC is a CPU core; this is the thing that is responsible for executing your programs. You write your program on a PC, transfer (sometimes called Flash) it into the Program Memory within the PIC IC, and the CPU then executes your program.
Memory is divided into four different types:

**Program Memory**

This is non-volatile memory that you program using a PIC programmer. Your program resides here as can any static data if required (constants for formulas or message texts etc). This memory is NOT lost when power is removed or the PIC is reset.

*Some PICs allow for self-modifying code. This means that the program loaded could update itself, and this is a common practice when using a piece of software called a BootLoader – The PIC we will be using supports this feature. Not all PICs do.*

Program memory of 32 Kbytes or more is now very common for PICs and you can get a lot of program in that type of space; if you’re careful.

You can, over time, wear-out program memory by constantly writing to it – constantly flashing a new program down to the device. It’s a bit like walking up and down the same piece of carpet; eventually it wears out. The datasheet for the device will tell you the maximum number of program cycles you can perform before you can expect to start seeing problems, but in reality, they can, and sometimes do fail prematurely. The PIC will program fine one moment, and then fail the next time you flash it, and will continue to fail at the same location. There’s nothing you can do about this except bin the PIC. Always have a spare handy!

**EEPROM – (Electrically Erasable Programmable Read Only Memory)**

This memory is non-volatile in that its contents aren’t lost when the power is switched off or the PIC is reset either. You can easily store and fetch data in this area from within your program so it’s a great place to store configuration settings or monitoring counters (how many times as a button been pushed since this project was built) etc. The PIC we will be using has 256 bytes of EEPROM available. Like program memory, you can wear this memory out if you write to it too often. Check the datasheet for the PIC in question for more details.

**RAM (Random Access Memory)**

This is where you store contents of variables, intermediate results of calculations, buffers and any other data that you need to read/write quickly but don’t mind losing if the PIC is reset or power is lost. Whilst it may not sound like much, the almost 4K (3894 bytes) is a lot of RAM for a PIC. You can read or write to this memory as often as you like and it should never wear out.
STACK and reserved

This PIC has a 31 level stack that is accessible from within your own programs if desired, but this isn’t common and changing things in here is quite likely to cause havoc. There will also be a small amount of RAM that CPU needs for itself to allow it to perform its tasks. This is why the full 4K (4096) bytes aren’t available – 202 bytes are missing for the PIC to use internally!

Clock

To control everything the PIC needs an accurate source of pulses – a clock source which is a analogous to a heart, and our PIC has several options available.

You can use the internal clock source which has the advantage of freeing up a couple of additional I/O pins for other uses, saves on a few external components and helps keep costs and physical size down. However, you get what you pay for and for some applications the internal clock just isn’t accurate enough and so it’s possible to use either a high precision external clock source, or a compromise using an external crystal and a couple of cheap capacitors. You can also opt for just resistor / capacitor combination if desired.

The internal clock has several pre-programmed speeds that can be selected for different applications. The maximum speed of the internal clock is 16MHz which is fairly fast. However, the PIC has a built in PLL (Phase Locked Loop) that will multiply the clock speed by a factor of four meaning the PIC actually runs at a clock speed of 64MHz. The PLL is also available for multiplying external clock sources as well.

PICs in general are extremely complex devices these days, and you would be well advised to read the datasheet available from the Microchip website. You may not understand everything at first as some of the features are extremely complex, but as you become more proficient it will all start to make sense. I’ll mention you reading the datasheet quite often !!

Power

The PIC also needs power to make it work. The 18F25K20 has a voltage range of 1.8v to 3.6v, and the K22 has a voltage range of 1.8v to 5.5v. We will be using the K22 exclusively because much of the hardware we will be interfacing with runs at 5v and so this PIC is the best choice. There are no other real differences between these two PIC devices.

The PIC has THREE power pins, four if you include the MCLR pin; but more on this pin later on.
Pin 8 and 18 MUST be connected to Vss (0v). The way the PICs internals are positioned on the silicone die, the PIC may not always work correctly if you don’t connect both the Vss pins. Pin 20 is the Vdd pin (+5v for our PIC) and this should be connected to the main power rail. 
*More on this later on as there are some other factors to consider.*

Once power is applied and assuming that the reset pin isn’t set, the PIC will start executing your program. Your program will configure the internal modules of the PIC as required, and start sending and receiving information over the I/O pins.

PIC pins are ALL multi-functional; this is a good… and a bad thing.

**PORTs and multipurpose I/O pins**

Remember we are ONLY concerned with the 18F25K22 in this series of articles, and whilst the theory is basically the same across all PICs, in some cases there are major differences. You **MUST** study the datasheet for the PIC you’re going to be using.

The PIC has physical I/O pins that allow communication with the outside world, and these pins are grouped onto PORTS, with each port supporting a maximum of 8 bits or pins.

Each PORT is accessed and configured via a series of five internal registers; that’s five 8-bit registers per PORT, and it’s important to learn how these registers work and how to manipulate and configure them. The AMICUS18 compiler will do a lot of the hard work for you, but you do need an appreciation of what’s going on and how to set things up the way you need them.

The datasheet for the 18F25K22 PIC states that the device has four PORTs, named PORTA, PORTB, PORTC and PORTE (notice no PORTD on this PIC) and since there are four, 8-bit ports this should give you 32 I/O pins… right?

Unfortunately, whilst each PORT is always 8-bits wide, sometimes not all those bits are physically implemented for each PORT. PORTA, B and C are full 8-bit ports whereas PORTE only has 1 physical I/O pin implemented. If you quickly add up the numbers, we have three, 8-bit ports (A, B & C), plus one, 1-bit port (E) which equates to twenty five physical pins. Add in the three power pins and we get twenty eight; and that’s the number of physical pins on the IC package and so, with the exception of the power rail pins, all the IC pins are physically mapped to one of the PICs PORTs.

Ok, so far so good. The PIC has PORTs, each PORT is a maximum of 8-bits wide, and each PORT bit maps to a physical I/O pin on the IC.
To keep things simple, there is a universal naming convention when dealing with PORTs and their bits: “PORTC.4” This refers to PORTC, and bit 4. Bit numbers ALWAYS start at 0, so PORTC.4 actually refers to the 5th bit of PORTC. This means you can physically have PORTC.0 to PORTC.7

So what do all these I/O pins do?

Within the PIC there are special blocks of logic; modules, that contain oscillators for PIC timing clock generation, timers, A/D (analogue to digital) converters, D/A (digital to analogue converters), comparators, communication subsystems (serial RS232 / RS485, I2C, SPI etc), PWM (Pulse Width Modulation) and of course, just digital logic level input detection, and output control.

Most of these modules have their own associated registers that allow you to enable or disable their functions and to configure their behaviour. There is a downside to all these lovely features of course. Firstly, they all require power to operate; this is usually only really an issue for battery powered projects but something that you may need to keep a watch-out for in some applications. However, the bigger issue is all these modules are desperate to connect to the outside world and there just aren’t enough pins to go around. This means that Microchip had to make some compromises and several modules often have to share physical pins, and so some module combinations aren’t supported. On more complex projects that can cause a real headache as careful decisions have to be made about which modules are really needed and what, if any, software workarounds can be used.

One last thing worth mentioning here; on this PIC, PORTE with its one solitary I/O pin can actually ONLY be configured as a digital input; you cannot configure this pin for output. Of course, if you’d read the datasheet you would have known this.

**Sink and Source currents**

At the start, we are only going to use the I/O pins for simple logic level work; reading and writing to digital devices by sending ones and zeros.

In digital I/O mode, each pin can sink or source up to 25 mA which is a lot of current, but be careful. The sink current on all pins combined cannot exceed around 200 mA, and the total source current cannot exceed 185 mA. If you exceed these maximums you will likely damage the PIC.
Assembling a circuit.

Enough theory and talk, it’s time to assemble a circuit that we can experiment with.

The circuit below is about the minimum you can get away with for a PIC. You have the PIC of course, some additional decoupling capacitors on the power rails, a reset circuit and a connection socket for an external PIC programmer.

There is also LED1 (with R2) which we are going to flash.

![Fig1. Minimum circuit for PIC.](image)

Let’s look at the above circuit in more detail.
The PIC contains a lot of logic that needs power to run, and this means that the power demands of the PIC can fluctuate quite rapidly depending on which modules within the PIC are currently in use. You should always place a decoupling capacitor, and a 100nf is ideal for this, as close to the power pins as possible. On circuit boards, you can solder one on the underside of the board if space is short. If the PIC is a fair distance away from the voltage regulator, a 10uf capacitor on the supply side of D1 can also be a sensible addition.

**ICSP and the diode**

This PIC, and most others these days, support ICSP; In-Circuit Software Programming. This means you can program the PIC whilst it’s in circuit without having to remove it and insert it into a PIC programmer. This saves a lot of time, wear and tear on the PIC and your fingers.

However, to support ICSP you need to add a programmer hook up point (SK1), and you should add a diode, D1. You will see that power flows from the +5v rail, through D1 and into the PIC’s Vdd pin. During ICSP, the programmer will actually apply power to the PICs Vdd pin when required. The problem is that when the programmer applies the +5v, this could flow back onto the main 5v rail and power up the rest of the circuit. Most PIC programmers are very limited on how much current they can supply and you could damage the programmer if it attempted to drive a large circuit; D1 prevents this. D1 does cause a voltage drop and you can use a SCHOTTKY diode to reduce this affect as its forward voltage drop is less than a normal silicone diode, if this voltage drop is a problem. I’ve used a standard 1N4001 silicone diode in these circuits as the voltage drop caused still allows to the PIC to run well above its minimum required voltage.

**The MCLR**

The MCLR pin has several possible functions depending on the PIC in question and on how the user wants to configure it.

The pin can be configured as a digital input pin – input only remember (but I would try and avoid using this pin for anything other than MCLR if possible).

Assuming this pin is not configured as a digital input pin, then holding this pin at Vdd allows the PIC to run normally. Bringing the pin to Vss forces the PIC to reset, and the PIC will restart when the pin is allowed back to Vdd. This means that for normal PIC operation, this pin should be held at Vdd and allowing this pin to float is a very bad idea as it can cause the PIC to randomly reset and exhibit other strange behaviour.

During ICSP, the programmer controls this pin and uses it to supply the programming voltage (around +12v) to the PIC. Because this pin should be tied to the Vdd, you should always include D2 and R1 in your circuits.
These two components allow the MCLR to be gently pulled up to Vdd during normal operation, but also allow the pin to be controlled by an external programmer when attached. D2 prevents the programming +12 from flowing back onto the circuits Vdd rail as this higher voltage would almost certainly damage the PIC and any other voltage sensitive components attached to the power rail.

Switch SW1 is a simple push button that pulls the MCLR pin momentarily to 0v forcing the PIC to reset.

LED1 is connected to the PIC on PORTC.4 (which is the 5th bit), and to 0v via a 470R resistor. This means that when the PORTC.4 is a HIGH logic level, the LED will switch on.
The picture in Fig 2 shows the circuit placed on a bread board. You will notice that there are two additional capacitors (top left and bottom left) on the power rails. These are 100nf are there to provide a bit of extra decoupling.

The small circuit board on the left is actually a 5v PSU that just plugs onto a standard breadboard (there are construction details on my website for this).

Also in the top left you can see a black six pin connector. This is the ICSP connector that allows for a PICKIT2 or PICKIT3 programmer to be directly attached. The simple way to make a suitable connector is to solder two, six way headers back to back as the pins of a single header are slightly too short on their own to hold the programmer securely on the board.

We will be using this circuit as the core for many of the future examples in this series so it may be an idea to leave this circuit built if you can.

In chapter 2, we will look at writing a first BASIC program for the PIC and then transferring this program to the PIC and running it.