PIC Microcontroller

C reference manual.

2011 Edition

by

John Main

Edits:

24/01/2007	: bit field updated.
05/07/2008	: Reference to bitwise examples and boolean examples removed These are contained in the course modules.
05/07/2009	: Added bitwise operator details.
27/01/2011	: Updated Links and introduction, tidy up page numbering.

1 Summary

This syntax reference manual accompanies the C programming course which you can find at:

www.best-microcontroller-projects.com

If you are interested in PIC Micros and want to start to using C for your projects then this course will teach you the basics using a practical project based approach. Two projects are used with theory reinforced using the projects to show how the code works.

The projects in the C course are built up slowly and allow introduction of new concepts in C programming which you can test out 'for real' on the hardware.

All the software modules have been tested out on 'real' hardware that matches the schematics given in the course. All you have to do is construct the hardware either soldering up a circuit or using solderless breadboard i.e. they are fully debugged.

One of the most difficult aspects of starting with microcontrollers is getting everything to work at the same time so having fully debugged software that has been used on real hardware is definitely an advantage.

The device used in the course is the 16F88 which has an internal reset and internal oscillator (since it is one of the more modern devices) and this makes setting up and using the device much easier.

Table of Contents

1 Summary....................2
2 C reference....................6
  2.1 Valid C Characters....................6
    2.1.1 Standard characters....................6
    2.1.2 Using characters....................6
    2.1.3 Escape characters....................6
  2.2 Keywords....................7
  2.3 Comments....................7
  2.4 Valid variable and function names....................8
  2.5 Variables....................8
    2.5.1 Variable assignment....................8
    2.5.2 Local variables....................9
      2.5.2.1 Local variable declaration....................10
    2.5.3 auto variables....................10
    2.5.4 volatile variables....................10
    2.5.5 register variables....................11
    2.5.6 static variables....................11
      2.5.6.1 local static variables....................11
      2.5.6.2 file static variables....................12
    2.5.7 global variables....................13
      2.5.7.1 external variables....................13
    2.5.8 Scope....................14
    2.5.9 Arrays....................15
      2.5.9.1 Array initialization....................15
      2.5.9.2 Array access....................15
    2.5.10 Multidimensional arrays....................16
      2.5.10.1 Multidimensional array initialization....................16
      2.5.10.2 Multidimensional array access....................16
    2.5.11 Strings....................16
  2.6 Constants....................17
  2.7 Types....................17
    2.7.1 Fundamental Types....................17
    2.7.2 Unsigned types....................19
    2.7.3 Floating point types....................19
    2.7.4 void type....................20
    2.7.5 Typedef....................21
    2.7.6 Enum....................23
    2.7.7 Struct....................24
      2.7.7.1 Declaring a useful typedef struct....................24
      2.7.7.2 Initializing a struct variable....................24
      2.7.7.3 Accessing a struct variable....................24
      2.7.7.4 Accessing a struct variable using a pointer....................25
      2.7.7.5 Example of nested structure....................25
      2.7.7.6 Arrays of structures....................25
      2.7.7.7 Usual struct declarations....................26

2.7.7.8 Using struct for bit field access 27
2.7.8 Unions..27
2.7.9 Casting..27
2.8 Operators and expresions..28
2.8.1 Arithmetic operators..29
2.8.2 Logical operators..29
2.8.3 Relational operators...30
2.8.4 Bitwise operators..31
2.8.4.1 Bitwise Inversion '~'...31
2.8.4.2 Bitwise And '&'...31
2.8.4.3 Bitwise Or '|'..32
2.8.4.4 Bitwise Xor '^'...32
2.8.4.5 Bitwise Shift left '<<'...32
2.8.4.6 Bitwise Shift right '>>'..33
2.8.5 Precedence...33
2.9 Increment and decrement...35
2.9.1 Increment..35
2.9.2 Decrement..35
2.9.3 Prefix and postfix...35
2.9.3.1 Postfix...35
2.9.3.2 Prefix..36
2.10 Loops : while, for, do...36
2.10.1 while loop..36
2.10.2 do while loop...37
2.10.3 For loop..37
2.10.3.1 Common for loop use..37
2.10.3.2 Advanced 'for' loop use...39
Advanced 'for' loop 1 : infinite loop..40
Advanced 'for' loop 2 : multiple expressions.......................................40
2.10.4 Break...40
2.10.5 Continue..41
2.11 Conditionals...41
2.11.1 if..41
2.11.2 If else...42
2.11.3 If else if..43
2.11.4 switch..43
2.11.4.1 default...44
2.11.5 ?:..44
2.11.6 Goto..45
2.12 Pointers...45
2.12.1 Pointer declaration 46
2.12.2 Using 'Address of' operator (&)..46
2.12.3 Pointer dereferencing..46
2.12.4 Pointer arithmetic...47
2.12.5 Void pointer...48
2.12.6 Pointer and Array equivalence..48
2.12.7 Pointer usage..48

2.12.7.1 Controlling variables....48
strcpy (string copy)....49
strlen (string length)....49
2.12.7.2 Pass by reference....49
2.12.7.3 Linked lists....50
2.12.8 Functions....50
2.12.8.1 Arguments....50
2.12.8.2 Return value....51
2.12.8.3 void types for functions....51
2.12.8.4 Function arguments details....52
2.12.8.5 Functions returning more than one argument....52
2.12.8.6 Arguments - pass by reference 52
2.12.9 File structure....53
2.12.9.1 C Source files....53
2.12.9.2 Header files....54
2.12.9.3 Prototypes....54
Function prototype....55
Function body....56
Using the function prototype....56
2.12.9.4 Constants and Macros....57
2.12.9.5 static functions....57
2.12.9.6 Function sets and libraries....58
Function sets....58
Libraries 58
2.12.9.7 Standard libraries....59
2.12.9.8 Hiding data within a file....61
2.12.9.9 Hiding functions within a file....61
2.12.10 Preprocessor....61
2.12.10.1 #include 62
2.12.10.2 #define – Constants....62
2.12.10.3 #define – Macros....63
2.12.10.4 #ifdef, #endif - Conditional compilation....63
2.12.10.5 #ifndef - Conditional compilation....64
2.12.10.6 #undef - Forget macro 64
2.12.10.7 #else - Conditional compilation....64
2.12.10.8 #elif – Conditional compilation....65
2.12.10.9 # - Argument to quoted string....65
2.12.10.10 ## - Token pasting....65
2.12.10.11 #pragma -Compiler directive....66
2.12.10.12 #line - Diagnostic....66
2.12.10.13 #error - Diagnostic....66
3 References....66

2 C reference

2.1 Valid C Characters

Here is a list of characters that C recognizes

2.1.1 Standard characters

- `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
- `a b c d e f g h i j k l m n o p q r s t u v w x y z`
- `! # % ^ & * ( ) - _ + = ~ [ ] \ | ; : ' " { } , . < > / ?`

$ and @ are commonly used but are not needed by the C standard.

2.1.2 Using characters

To assign a character to a variable you can use single quotes around the character:

```
char chr;
char = 'V';
```

2.1.3 Escape characters

In C the escape character is made up of a backslash (escape) followed by a normal character .

These are characters that can not be be printed. For Historical reasons the ASCII character set defines characters to control a line printer so that the paper could be advanced (form feed-FF), ring a bell (BEL), single line (LF), move the carriage to the start at left (CR).

BEL	\a	bell
BS	\b	backspace
FF	\f	form feed
NL	\n	new line
CR	\r	carriage return
HT	\t	horizontal tab
VT	\v	vertical tab
"	\"	double quote
'	\'	single quote
?	\?	query
\	\\\\	backslash

Some of them are a bit historical but \n and \r \t are definitely useful along with \”, \' and \\. These last three let you insert the character into a text string otherwise the compiler would think that the character ends the string.

e.g. You can write a string as :

“This is a string “

and if you wanted to put quotes into it you would do this

“This \”is a\” string “

The string would be stored as:

This “is a” string

Without the escape character the compiler would get confused as the double quote character defines the start and end of a string.

2.2 Keywords

Here is a list of keywords that are reserved by the compiler (you can not re-define these as your own functions or variables):

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

Microcontroller compilers include the keyword **asm** for entering assembler code directly into the C source – sometimes you will need the performance increase or speed saving by using direct assembler code.

2.3 Comments

There is one C commenting method:

/*	Defines the start of a comment.
*/	Defines the end of the comment.

Any text that is enclosed by these keywords will be ignored by the compiler and you can use them across multiple lines to comment out blocks of code.

There is also a C++ commenting method that is so useful it is often allowed by the C compiler – this is the double forward slash:

//	Single line comment

Anything after the keyword // up to the end of the line is ignored by the compiler.

2.4 Valid variable and function names

A name is made up of letters, underscores and numbers but can not start with a number and it can not be a keyword. So an example of a valid name is

my_fn1

an incorrect one would be

1_tyu3$

Note: Names are case sensitive so:

my_fn1

is a different name to

my_Fn1

2.5 Variables

A variable is a temporary storage area for your data. You can change the data throughout the use of the program – it is the RAM in the microcontroller. For the PIC Microcontoller the basic unit of storage is a byte but you can use groups of bytes for larger variables by using types (see section 2.7).

2.5.1 Variable assignment

You can assign numbers to variables easily

```
char my_char;
short my_short;
int my_int;
long my_long;
```

Different ways of assigning values to variables

Assing a number	**Assign a character**	**Assign a hexadecimal**	**Assign an octal**

my_char = 33;
my_short = 33;
my_int = 33;
	my_char = 'A';
	my_short = 'A';
	my_int = 'A';
		my_char = 0x43;
		my_short = 0x43;
		my_int = 0x43;
			my_char = 023;
			my_short = 023;
			my_int = 023;

my_long = 33;	my_long = 'A';	my_long = 0x43;	my_long = 023;

Note: numbers beginning with zero are defined as octal - I don't use this at all but all C compilers will let you write it so if you wrote:

```
my_num = 023;
```

Thinking that you're going to assign my_num with the value 23 you will actually be assigning an octal number (integer value 19)!

In the above examples all you have to remember is not to assign too large a value e.g.

```
my_int = 1231111;
```

.. will not assign that number to my_int – you will get something different:

The reason is that the compiler automatically casts the value into the variable - that is it makes it fit into the available space (here 16 bits). The hex value of the number is 0x12C907 so the last 16 bits are 0xC907 or 51463 or -14073 as a signed number.

Note: You have to be aware of the number range you are using to ensure that the variables can contain the numbers you need to use.

Casting is used in general so that you can use different types within the same expression for example:

```
int a;
a = my_char + my_short + my_int + my_long;
```

Here each variable is cast to the same type to allow the calculation to work (otherwise you would have to cast each variable manually) which would make the calculation look like this:

```
int a;
a = (int)my_char + (int)my_short + (int)my_int + (int)my_long;
```

.. and this just makes it look bad.

In general casting works to help you and makes the calculations work for the expression without you having to worry about casting details.

2.5.2 Local variables

All variables must be declared before you use them and in a function they must all be declared after the opening brace.

To declare a localvariable you write

```
<variable type> name [optional initialization] ;
```

e.g.

```
int a = 3;
char chr = 'V';
```

Note: The single quotes around the character V tells the compiler to assign the character V to a character variable.

2.5.2.1 Local variable declaration

```
void function(void) {
        ALL variable declarations here. // local variables
        program statements follow variable declaration.
}
```

These variables exist only within the function and when the function exits they are destroyed (except for static variables see section 2.5.6) they are known as local variables and they are also known as auto variables (section 2.5.3).

The local variable is only usable inside the the function and can not be seen by any other function. For example :

```
int functionA (int n) {
int alpha;
        alpha = 3;
        return alpha * n;
}
int funcitonB(int n) {
int alpha;
        alpha = 10;
        return alpha * n;
}
```

FunctionA and FunctionB both use the same variable name 'alpha' but because it is declared locally to each function each is totally separate from the other so Function a returns n * 3 and FunctionB returns n * 10 and there is no interaction between the two functions or variables. The local variables are said to have local scope. (See section 2.12.10.2 on scope).

2.5.3 auto variables

The **auto** keyword is totally redundant and you never need to write it – all local variables are auto variables.

Its purpose is to define a variable local to a statement block (only usable within the statement block and not visible outside it) but the compiler does this for you anyway i.e. any local variable has local duration for however long the function lasts (see section 2.5.2).

It is known a an auto variable as it is automatically created at the start of a function and destroyed when it is no longer needed.

2.5.4 volatile variables

The **volatile** keyword tells the compiler that a variable may be changed by a mechanism different to the normal program flow operation e.g. by an interrupt routine. It also tells the compiler not to optimize the code associated with the variable.

You can use this keyword in front of a variable declaration e.g.

```
volatile int a = 3;
```

Another useful example is for I/O port usage

If you defined an 8 bit port as follows

```
volatile char *PORTB = (char *)0x32;

*PORTB = 0x01;

*PORTB = 0x02;
```

If the port is not assigned as volatile then the compiler would think that the first statement PORTB = 0x01; is redundant and could optimize it away – not what you want when using hardware e.g. you might be sending data to a latch that needs a sequence of data strobes sent one after the other.

2.5.5 register variables

For the PIC Microcontoller the keyword register is redundant as any variable by definition is a register since all RAM space in the device is accessed by the machine code as a register.

In different architectures e.g. for an 8051 or a PC there is an external RAM memory mechanism but it also has internal registers. In these devices the keyword 'register' tells the compiler to try to optimize the code using the internal register so that the code will run faster (no external memory access needed).

If you did need it you would use it as follows:

```
register int a;
```

2.5.6 static variables

A static variable is one of two types:

1. **Local static variable** : Defined at the start of a function block – This declares a variable that is not destroyed at the end of the function – any data it holds is remembered for when the function is called again.

2. **File static variable** : Defined outside any function block a static variable is available only to functions within the same source file.

Option 2 is useful for keeping data within a set of functions (defined within the same file) e.g. a buffer for use in a communication system.

Note another use of the static keyword is in static functions (see section 2.12.9.5).

2.5.6.1 local static variables

Static variables are declared in the same way as an ordinary variable but are prefixed with the word **static** and when declared within a function the variable is never forgotten. Unlike a normal (auto or local) variable the static variable remembers its value between calls to that function.

The important difference between the local static variable and a global variable is the the local static variable has local scope (see section 2.5.8 on scope) i.e. it is only usable and visible within the scope of the function in which it is defined.

For example the function shown below is called three times if you called it a fourth time then the internal variable count would have the value 4.

```
void show_static(void)
static int count=0;
        count++;
}


void main(void) {
        show_static(); // 1st call
        show_static(); // 2nd call
        show_static(); // 3rd call

}
```

Note: The variable is initialized only once at the first call of the function and the variable is only available inside the function.

2.5.6.2 file static variables

Another place that the static keyword is used is for declaring variables with file scope this means that the variable is available to all functions within the same file but not available to functions declared in other files.

You can declare a static variable inside a file but outside of any function and its declaration is the same as a normal static variable section 2.5.6.1. It is best to place all the file static variables at the beginning of the file so it's easy to maintain the code:

```
// File static variables
static int count=0;


void show_static(void)
      count++;
}

void main(void) {
int a;
      show_static(); // 1st call
      show_static(); // 2nd call
      show_static(); // 3rd call


      a = count;
}
```

Note: Again the static variable is initialized but this time only when the program starts (at the initialization of the program) and again it's only initialized once.

In this case the variable is available to any function within the same source file so the variable a will have the value 3 at the end of the main function. This is a good way of sharing data between functions in the same file but stops you having to use a global variable.

Note this variable will only be visible/available in the file it is declared within.

2.5.7 global variables

A global variable is a variable defined outside of any function and is available to any other function. It keeps its contents throughout the entire life of the program and is never destroyed.

Don't use too many of these or think really carefully when you use them – too many and they lead to programs that are unmaintainable. In fact you should be able to code virtually anything without using a global variable.

Warning: Don't use globals in a professional environment if you do you are likely to make colleagues very frustrated. The reason is that global can be seen (scope and visibility) anywhere in the programming space. If you as a designer in a multi programmer team decide to use a global variable and unknown to you another designer does the same and uses a global of the same name then you overwrite each others data causing huge confusion, delays, random bugs - that are really difficult to find.

In fact any program can be made to work either using static variables (file static) to keep the variable local to your piece of code and use a function within that file to set and get the value of the variable.

Note: This is the main use of C++ classes – their great advantage is that the variables in a class can be private or public to other classes. It's exactly the same concept as using statics but instead of a variable being available only to a file it is available to a class and even better classes can be shared so the variable can be shared.

Take home lesson: Preferably: never use globals – if you have to, limit them to a few very well defined names and place them in a file that must be the only file with global definitions in e.g. globals.h. That way you know all the globals in the system.

2.5.7.1 external variables

You use the **extern** keyword to define a variable as external global variable.

If a global variable is defined in another file but the current file is compiled before that one the compiler will not know the type of the global variable and will complain.

The **extern** keyword lets you tell the compiler the type of the external global variable (that you want to use in the current source file) without actually declaring it (you only want to declare it once in one place).

Here is an example of using extern. The global variable glob_dpIdx is defined in gate.c and used in count.c

Source file that declares the global variable (gate.c)

```
    // globals for timer capture update
    unsigned short glob_dpIdx=20;
```

The above statement creates (declares) the variable glob_dpIdx.

Source file that needs to use the external global (count.c)

```
    // external globals for timer capture update
    extern unsigned short glob_dpIdx;
```

The above statement lets the compiler know that the variable glob_dpIdx is defined somewhere else but lets you use the variable in the current file (count.c). It does not declare the variable as you only want one declaration of the variable. Note how the compiler is not told exactly where the external variable is located – the location is resolved at link time.

Note: You could put the extern unsigned short glob_dpIdx; statement into a header file so that any other file using gate.h has access to the global variable in gate.c.

2.5.8 Scope

The visibility of a variable to another part of the program is called scope and the previous sections deal with how to declare different variable types and what their scopes are.

Here is a summary of variable scope:

Variable type	**Declaration**	**Visibility**	**Section**
auto (local)	Inside a statement block.	Visible only within the statement block.	2.5.2,2.5.3
static auto (local)	Inside a statement block.	Visible only within the statement block.	2.5.6.1
static (file scope)	Inside a file (outside of any statement block).	Visible anywhere within the file.	2.5.6.2
global	Declared outside all functions.	Visible anywhere in all files in the project.	2.5.7,2.5.7.1

When a variable has scope within a certain block of code then that variable is protected within that block e.g if a global variable is declared with the same name as a local variable then the function with the local variable is used i.e. the global is not affected from inside the function.

Similarly when the global is accessed outside the funtcion then the local variable inside the function is not affected.

The scope of the local variable is inside the function and no other part of the program can affect it.

This is the reason scope is powerful – it protects each variable from being affected by other variables with the same name and it works exactly as you expect it to.

2.5.9 Arrays

An array is a collection of objects of the same type arranged sequentially in memory in a contiguous block (of RAM). Since the objects are arranged in a block you can access each element using an index (just a number).

Declaration of arrays is similar to normal variables. You declare an array using square braces to define the number of objects:

```
int a[10]
```

declares an array of 10 integers.

Note: The first element of an array is at index zero and the last element of an array is one less than the declaration i.e. Here it is 9.

Note: In the mid range PIC microcontrollers the RAM is not contiguous so the maximum array size is limited by the physical size of the RAM. High end PIC microcontroller have contiguous RAM so they are easier to use when you need large arrays.

2.5.9.1 Array initialization

You can initialize an array as follows

```
int a[10]= {1,2,3,4,5,6,7,8,9,0};
```

or for auto initialization

```
int a[]= {1,2,3,4,5,6,7,8,9,0};
```

Here the compiler makes the array size the same as the number of objects.

2.5.9.2 Array access

To access an array just specify the object index

```
c = a[0];   // The first element – Note:indexing starts at zero
c = a[9];   // The last element - Note:indexing starts at zero
```

2.5.10 Multidimensional arrays

For multi-dimensional arrays you use 2 (or more) square braces

```
int a[4][3];
```

The last element changes fastest in memory i.e. The first index represents rows while the second index represents columns.

2.5.10.1 Multidimensional array initialization

```
int a[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
```

2.5.10.2 Multidimensional array access

Again just specify the array indices

```
c = a[0][0];  // first element
c = a[2][3];  // last element
```

2.5.11 Strings

Strings are supported in C using arrays of characters. But the are treated slightly differently to normal arrays:

A string array always ends in a zero (the value zero or '\0').

The zero commonly written as '\0' defines the end of the string.

You can initialize a string using a string constant.

You can initialize a character array with a string constant as follows

```
char str[] = “abcd”;
```

The compiler will allocate storage for the string – it allocates 5 bytes here because it always allows room for the last element of the string '\0'.

You can also write this using pointer notation:

```
char *str = “abcd”;
```

This works because arrays and pointers are interchangeable (see section 2.12.6). The difference is that you can move the pointer by adding an integer to it - see section 2.12.4 on pointer arithmetic (but don't forget to store a copy of the pointer before updating it otherwise you will loose the original data address!

The standard library <string.h> defines the following functions for manipulating strings.

String library	
memcmp	compare memory.
memcpy	copy memory.
memmove	move memory.
memset	set memory to a value.
strcat	concatenate strings (append).
strchr	locate character within a string.
strcmp	compare strings.
strcpy	copy strings.
strlen	return string length.
strncat	append strings no more than n characters.
strncpy	copy strings no more than n characters.
strspn	string span check chars from s2 that are in s1.

Note:Some of them may not be available – check compiler documentation.

2.6 Constants

Constants are declared in the same way as variables except that they are preceded by the keyword const. e.g.

```
const int a = 3;
```

All this does is to define a variable that will not change its value.

Note: Using the preprocessor #define statement may be a better way to define a constant value as it won't use up any memory space (see section 2.12.10.2). Although the compiler will probably manage this for you.

2.7 Types

The type of a variable defines the size of number that a variable can hold and it also tells the compiler how much memory (RAM) to allocate for each variable. The compiler also uses the type of a variable to check for illegal code e.g. if you assigned a floating point number to an integer type an error would be flagged.

2.7.1 Fundamental Types

The following keywords are fundamental types

char, short, int, long and float.

(There is another fundamental type 'void' see section 2.7.4).

Each of these types represents a different size variable that can hold different values (see Table 1: Fundamental types).

There are some other types that you will probably only find on higher bit size machines e.g. double (similar to float but uses twice the bytes to store the data – double precision floating point) and long double (similar to double but uses twice the byte to store data – super precision double). Since PIC microcontroller are not good at float anyway you are unlikely to see them in a PIC compiler.

Note: The C language allows the definitions of 'double' and 'long double' to be the same as 'float' – so you can use the names but they will not provide any difference to using a float.

Table 1: Fundamental types

Type	**Size**	**Range**
signed char	1 byte	-128 .. 127
short (int)	1 byte	-128..127 (default is signed)
int	2 bytes	-32768..32767 (default is signed)
long	4 bytes	-2147483648 .. 2147483647 (default is signed)
float	4 bytes	±1.17549435082 * 10^{-38} .. ±6.80564774407 * 10^{38} (Section 2.7.3)

Note: In C the size of these types is machine (and compiler dependent so if you port the code to a different target (or use a different compiler) the size of the types may not match.

This can cause problems e.g. depending on the compiler and target processor a short (integer) may be 8 or 16 bits You have to look at the compiler manual to find out the size of the type (or you can use the sizeof function to return it in a program).

Sidebar

A note on types when using different targets/compilers

*Note: This is where you have to be very careful about what these variable types mean **C does not define how the compiler implements types**.*

This is only important when you target a different processor/compiler. The same type in one target my be using a different RAM size in another target e.g. A 16 bit processor may define an integer type as using 16bits while a 64 bit processor may define an integer as using 64 bits. This does not sound like a problem when using a PIC microcontroller as it only uses 8 bits but there are newer processors in development.

Why is it a problem?

Lets look at an example. What if you had an integer type that you assumed had 32 bits as it's fundamental implementation. Say you were using it as a flag register to use each bit as a flag (saving memory space – you would have 32 flags available) perhaps you check one flag using &(1<<22)

If you move to a compiler that uses a 16 bit implementation then 1<<22 is undefined = crash.

2.7.2 Unsigned types

Adding the word **unsigned** in front of a fundamental type (not floating point type) frees up the top bit in the variable and the variable can then only hold positive numbers. The advantage is that the maximum positive value that the variable can hold is now larger (see Table 2: Using unsigned fundamental types).

Note: You can't apply unsigned to the float variable type (it does not make any sense to do so – floating point by definition can not be an integer).

Table 2: Using unsigned fundamental types

Type	**Size**	**Range**
unsigned char	1 byte	0..255
unsigned short (int)	1 byte	0..255
unsigned int	2 bytes	0..65535
unsigned long	4 bytes	0..4294967295

*Note: You can also write just '**char**' and this defines an unsigned character – unlike the other integer types the '**char**' type defaults to an unsigned value.*

2.7.3 Floating point types

With this variable type you can work on numbers that are far larger than an integer can hold. The range that float can hold is:

$$\pm 1.17549435082 * 10^{-38} .. \pm 6.80564774407 * 10^{38}$$

i.e. absolutely huge and this number is held in the same memory space as a long variable. It works by splitting the representation of a number into mantissa (number) and exponent (power) and allocating part of each variable space to each of these.

To achieve its operation a lot of code is needed to translate between the representation and the byte oriented operation of the microcontroller and this is why a floating point variable uses up a lot of resource – it needs support libraries to interpret the stored value and then do add, subtract, multiplication and division etc.

A floating point number does have limitations:

1. Inexact.
2. Slower.
3. Uses lots of resource.

Its nice to think of floating point values as exact because they look good but in reality they are not – all floating point numbers (for floating point using base 2 operation i.e. binary representation) are inexact because the representation must translate a 'real number' into 32 bits of data. 32 bits of data can not hold an infinite set of numbers so if the number does not translate exactly into the 32 representation then it must be approximated.

So the disadvantage of using floating point is that it takes up a lot of microcontroller resources and will be slow. The advantage is that it lets you easily handle fairly good precision variables (if you accept that there will be a certain level of error).

Note: Errors can be acceptable if you know that your input has errors anyway all you need to make sure of is that the errors must not be large compared to the input.

Note: Only use floating point types in a microcontroller environment if you absolutely have since floating point variables (and associated code) take up a huge amount of resources.

Table 3: Floating point types

float	4 bytes	$\pm 1.17549435082 * 10^{-38}$ .. $\pm 6.80564774407 * 10^{38}$
double	4 bytes	$\pm 1.17549435082 * 10^{-38}$ .. $\pm 6.80564774407 * 10^{38}$
long double	4 bytes	$\pm 1.17549435082 * 10^{-38}$ .. $\pm 6.80564774407 * 10^{38}$

For this compiler the floating point implementation sets all the different floating point types: float, double and long double to the same floating point representation i.e. 4 bytes long. In other compilers (non microcontroller compilers) double and long double will use more memory so that the precision of the result is higher.

2.7.4 void type

Although it sounds a bit strange the void type is similar to the number zero i.e. they both represent something that does not exist!

For declaring functions the void type declares either that the function has no arguments

```
int func(void) { ..... }
```

or that the function returns no value

```
void func(int a) { ..... }
```

or both

```
        void func(void) { ..... }
```

If you declare a pointer to a void type then it's used to imply that the pointer can point to <u>any</u> type (a generic pointer).

You probably won't need about void pointers but its useful to know if you come across one.

2.7.5 Typedef

You can define your own types, based on the fundamental types (section 2.7.1), which is simply an alias that lets you create type names that are easy to read.   For example to create a type labeled Byte you can write:

```
typedef  char Byte
```

This declares a new type that is exactly the same as the char type but named Byte. You can now use the following statement to declare a Byte variable.

```
Byte a_byte;
```

The real purpose of this feature is that it makes programs far easier to read as you are telling the reader (human) of the program more information about how the program operates. When you use Byte you know that it is to do with memory storage or port size i.e. Related to the microcontroller action whereas if you used a char it could be a character or part of a string.

Typedef also lets you reduce the size of a declaration. For example how about these for some ultra compact types

```
typedef unsigned short        U8;
typedef unsigned int          U16;
typedef unsigned long         U32;
```

These tell you two bits of information – that they are unsigned variables and how many bits they use. Here U8 is exactly the same as unsigned short etc.

These kind of declarations are commonly used in professional environments (although may be not as compact as this) a common declaration would be

```
typedef unsigned char  UINT8
typedef unsigned int    UINT16
```

Note: These declarations are used because you can not rely on the number of bits in the fundamental types being the same for different target processors. Using these types lets you declare them in one place and change them when you move the code to another processor.

2.7.6 Enum

Or rolling your own types...

enum stands for enumerated type and defines a set of constants that are grouped and that you can use as a typed variable.

The best way to understand them is to see an example

```
enum states { Starting, Action1, Action2, Action3, Finished };
```

This defines a new type labeled states and it has five constants defined within it. To use a variable with this type you would write

```
enum states machine1, machine2;
```

i.e. exactly the same way as a normal variable declaration.

Now within your code you can write :

```
machine1 = Action2;
machine1 = Action3;
```

This makes the operation of the code much clearer as you don't need to use integer values that you must remember e.g.

```
m1 = 1;
m1 = 2; ... etc.
```

You can also initialize the variables when you declare them:

```
enum states machine1=Action2, machine2=Starting;
```

You can also define any of the enumeration constants to have a specific value:

```
enum Sizes { byte=8, word=16, along=32 }
```

Note: A good use of the enum data types is for state machines.

2.7.7 Struct

Or rolling your own more complicated types...

The struct keyword lets you group related information together into a 'structure' which you declare as a type so you can make variables with it.

Within the structure you can have variables and even nested structures so you can make your program easier to read – you can also pass structures into functions (or pass by reference) and this just makes them very convenient.

Note: Remember to initialize struct variables.

2.7.7.1 Declaring a useful typedef struct

You can declare a structure as follows (note I find this the 'useful' way to declare it – see later for the other 'official options'. Which lead to this definition).

```
typedef struct  {
        int relay;
        int fan;
        int lock;
        float rpm;
} System;
```

This declares a type that is named System – you can now use this type for your own variables as follows:

```
System x,y;
```

Now there are two variables named x and y and each of these variables has the internal subset variables relay, fan, lock, rpm.

2.7.7.2 Initializing a struct variable

You can do this either when you declare it or by accessing each member variable individually (section 2.7.7.3). Example of initialization when the variable is declared:

```
System x= {5,55,555,5.5},y;
```

Initializes x but not y.

2.7.7.3 Accessing a struct variable

You use the dot operator '.' to access a struct variable to either get the contents or set the contents as follows:

```
x.fan = 100;       // Set variable fan in structure x to 100
y.lock = x.lock;   // Set lock in structure y to value of lock in structure x
```

(See later for examples of nested structure access).

2.7.7.4 Accessing a struct variable using a pointer

When you use a pointer to a struct there is a special pointer access operator '->' :

```
System *sys;      // Declare pointer to a System struct (section 2.12)
System z;         // Declare structure variable x.
sys = &z;         // point to the structure.
sys->lock = 10;   // Set the member value.
```

2.7.7.5 Example of nested structure

```
typedef struct  {
        System A;
        System B;
} Main_System;
```

Main_System S1,S2;

Now you have two variables S1 and S2 both with two sets of the System structure within them labelled A and B. You can access the 'System' variables as follows:

```
S2.A.fan  = 1121;
S2.A.lock = 1122;
S2.A.rpm  = 11222.2;
S2.B.fan  = 21;
S2.B.lock = 22;
S2.B.rpm  = 222.2;
S1 = S2;  // copy entire contents of structure S2 to S1!
```

2.7.7.6 Arrays of structures

You can also use arrays of structures that you initialize when you declare them:

```
typedef struct {
        char *str;
        int display_time;
} Message ;


Message messages[3] = {
        "Hello world", 1000,
        "Processing", 500,
        "Exiting", 300
    };
```

Note: you should be able to auto initialize structures as well where the compiler sets the size of the structure at compile time e.g.

```
Message messages[] = {
        "Hello world", 1000,
        "Processing", 500,
        "Exiting", 300
    };
```

*Note: See section 2.5.11 for information on strings (char *).*

2.7.7.7 Usual struct declarations

This is usually presented first but I find it just confuses the issue. If you use the declaration method in section 2.7.7.1 then you will find it easier to use and less confusing than this section.

A structure declaration:

```
     struct struct_name {
        int relay;
        int fan;
        int lock;
        float rpm;
    };
```

To declare a variable:

```
    struct struct_name x;
```

or you can declare the variable x with the structure:

```
     struct struct_name {
        int relay;
        int fan;
        int lock;
        float rpm;
    } x;
```

And you can declare a typedef

```
    typedef struct struct_name struct_name_type;
```

and declare variables using this type

```
    struct_name_type x;
```

But it is easier if you look here – section 2.7.7.1.

2.7.7.8 Using struct for bit field access

It is not a good idea to use struct types for accessing the bits within an integer as the ordering of the bits is not defined in the C standard (they may be allocated from left to right or right to left) you would have to consult the compiler documentation for this.

Also the C standard only specifies unsigned int and int as acceptable candidates for bit field manipulation – so you may be using more memory (if you want only a byte) – again consult the compiler manual.

The bit field object is defined in the structure using a colon followed by a number to define the number of bits used. For example :

```
typedef struct {
   unsigned int relay : 1;
   unsigned int fan : 1;
   unsigned int lock : 1;
   unsigned int state : 2;
} Flags_2;
```

Here the object 'state' uses two bits. which you can use as follows

```
Flags_2 the_flags2;
the_flags2.state = 3;
```

2.7.8 Unions

You declare unions in the same way as structures but replace the keyword struct with the keyword union.

A union is a group of variables in the same way as a structure but this time the variables occupy the same memory space. You can only use one variable (member of the union) at a time and when you use another the original is overwritten.

```
typedef union  {
        int relay;
        int fan;
        int lock;
        float rpm;
} System_u;
```

2.7.9 Casting

C is a fairly strongly typed language and the compiler is designed to do type checking all the time so that when you pass value between functions or use variables of different types in an expression the same type of variable is used.

This makes sense as you would not want to send a floating point value into a function that only accepted integer values so C would complain when you try and compile that code.

The process of changing a variable into a different type is known as casting and the compiler does this automatically for you most of the time (See section 2.5.1 on variable assignment).

Note: For closely related types the compiler will cast the variable for you so you can send an integer into a function that accepts longs because and integer is a subset of a long.

Sometimes however you will want to force the compiler to accept a certain type of value and you can do this by casting.

For example If you used a variable of type char to hold a port value but wanted to store it in an unsigned int you could cast it as follows:

```
char chr;
unsigned int a
a = (unsigned int) chr;
```

Casting can be useful when using pointers as you sometimes need to use a char pointer to look into a larger type e.g. A long and you have to cast the pointer address to make it work.

```
long a_long;
unsigned char *p;
p = (unsigned char *) &a_long;
```

This code would let you setep through the individual bytes within the long number. p is a pointer and it is a pointer to unsigned char (or bytes). To set the address of p to the address of the long you have to tell the compiler that you know what you are doing and its ok to allow the pointer (that normally should only point to the same type of variable to point to a different type).

So the address of a_long is cast into an pointer to unsigned char (the same type as the pointer p). This tells the compiler that you want to ignore the type checking that would normally flag an error in your code i.e. the compiler would flag the following as an error since the pointer type does not match the variable type:

```
long a_long;
unsigned char *p;
p = &a_long;
```

2.8 Operators and expresions

Or doing mathematics.

Expressions are formed using numbers, variables, operators, relational operators and bitwise operators. They are worked out using a precedence order which simply means some operators e.g. multiply '*' are worked out before others e.g. add '+'.

Expressions are used anywhere some maths need doing e.g. in calculating the new value of a variable or testing the condition within an 'if' statement e.g.

if (a>20) statement; // statement executes if a is greater than 20.

a = 3*3 + 20/2 + a*5; // division and multiplication 1st then addition.

Note: You can use parenthesis to alter the order of calculation e.g.

a = 3* (3+20/2 + a*5); // works out parenthesis expression before *3.

The rest of this chapter describes the details of expressions.

Operators include all the usual mathematics add '+', subtract '-', multiply '*', divide '/' but in addition include logical operators such as logical 'or' (the vertical bar symbol '|') as well as relational operators e.g. greater than '>'.

An important concept in using operators is precedence (which operator is applied first if there are no parenthesis to show the order you want).

2.8.1 Arithmetic operators

These are the operators that you already know (except ++ and --).

*	Multiply
/	Divide
+	Addition
-	Subtraction
%	Modulus (remainder from integer division).
++	Increment (see section 2.9)
--	Decrement (see section 2.9)

2.8.2 Logical operators

Logical operators are different from bitwise operators in that they treat operands as true if the result of a calculation is greater than 0 and false if the result is 0.

You can use them to perform logical operations on expressions that use integer types (not just binary numbers) and the always return 0 or 1.

The logical operators are:

!	Logical negation
&&	Logical AND
||	logical OR

So you can write an expression such as

```
x>4 && x<20
```

This would return 1 if the value of x fell between 5 and 19 but would return zero at all other times.

Note: The logical negation operator also returns 1 or 0 whatever is thrown at it e.g.

!4563	returns 0
!0	returns 1

2.8.3 Relational operators

Used to compare sizes of numbers or variables:

==	Test for equality
!=	Test for inequality
>	Greater than
<	Less than
>=	Greater than equal
<=	Less than equal

2.8.4 Bitwise operators

Bitwise operators perform calculations on each bit within the operand.

Even though the PIC microcontrller's basic word size is 8 bits these operators work on any integer type:

- unsigned char
- int
- long

The bitwise operators are:

~	Invert (complement) each bit is inverted.
&	AND
|	OR
^	XOR
>>	Shift right
<<	Shift left

Note: Be very careful not to confuse these operators with the logical operators (Section 2.8.2) - it is very easy to make a typo e.g. && instead of & or accidentally use ! Instead of ~.

2.8.4.1 Bitwise Inversion '~'

Action	**Binary value**
a = 0x79;	0111-1001
a = ~a;	1000-0110

2.8.4.2 Bitwise And '&'

Action	**Binary value**
a = 0x79;	0111-1001
a &= 0x60	0110-0000

2.8.4.3 Bitwise Or '|'

Action	**Binary value**
a = 0x79;	0111-1001
a |= 0x06	0111-1111

2.8.4.4 Bitwise Xor '^'

Action	**Binary value**
a = 0x79;	0111-1001
a |= 0xff	1000-0110

Note: This looks the same as inversion (it is) but its real power is to allow toggling of individual bits:

Action	**Binary value**
a = 0x79;	0111-1001
a |= 0x04	0111-1101

2.8.4.5 Bitwise Shift left '<<'

Action	**Binary value**
a = 0x79;	0111-1001
a <<= 1	1111-0010

Note: The '1' is the number of left shifts.

Action	Binary value
a = 0x79;	0111-1001
a <<= 3	1100-1000

2.8.4.6 Bitwise Shift right '>>'

Action	Binary value
a = 0x79;	0111-1001
a >>= 1	0011-1100

Again the '1' is the number of shifts right.

Action	Binary value
a = 0x79;	0111-1001
a >>= 3	0000-1111

2.8.5 Precedence

The following table shows a list of all operators and their precedence with the highest precedence operators listed from the top.

Precedence means the order in which operators are bound to the expressions or numbers they are associated with

Operand (highest precedence operands to the top)	Associativity
() [] -> .	Left to right
! ~ ++ -- + - * & (*type*) sizeof	Right to left

Operand (highest precedence operands to the top)	**Associativity**
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= /= *= &= %= ^= |= <<= >>=	Right to left
,	Left to right

An easy way to understand the precedence is to look at a simple mathematical example:

1 * 2 + 3 * 4

If there was no precedence then you could calculate the result working from left to right

1 * 2 = 2,

1 + 3 = 5,

4 *4 = 20.

But a different compiler may could work in a different way (right to left perhaps) so the standard includes the precedence order which defines how the expression is calculated. Using the table, multiplication has higher precedence than addition so the calculation is as follows:

1 * 2 = 1

3 * 4 = 12

1 + 12 = 13.

The higher precedence calculations are worked out first.

Note: You can change the order of precedence by using parenthesis in which parenthesised operands are calculated first – anything in parenthesis is calculated first starting at the innermost parenthesis.

e.g.

```
( ( (1 * 2) + 3) *4
```

would result in the same calculation as shown at the start .i.e.

```
1 * 2  = 2
2 + 3  = 5
4 * 5  = 20
```

Note: If you are not sure of the order of the calculation then use parenthesis to force the order you want.

2.9 Increment and decrement

2.9.1 Increment

Incrementing a variable is a fundamental operation – all it means is adding a value to a variable and storing the new value in the same variable.

Incrementing by one is such a common action that there is an operator just for this purpose :

```
++  this is the increment operator.
```

To use it:

```
i++;
```

This statement increases the variable i by one.

If you need to increment by more than one then use the standard expression

```
i += 2;
```

2.9.2 Decrement

Similarly to increment, decrement means subtracting a value from a variable and storing the new value in the same variable.

Decrementing by one is is also a common action and there is also an operator just for this purpose :

```
--  this is the decrement operator.
```

To use it:

```
i--;
```

This statement decreases the variable i by one.

If you need to decrement by more than one then use the standard expression

```
i -= 2;
```

2.9.3 Prefix and postfix

Postfix and prefix refer to using the increment or decrement operators before and after a variable. Both of these operations increment the variable but...

2.9.3.1 Postfix

The operator is placed after the variable. This is the most familiar use and the value of the variable is returned before the variable is incremented (or decremented). For example:

```
int x,y;

x = 3

y = x++;
```

The value is returned before the postfix operator is applied so at the end of the statements x will have a value of 4 and y will have a value of 3.

2.9.3.2 Prefix

The operator is placed before the variable. The value of the variable is returned after the variable is incremented (or decremented). For example:

```
int x,y;

x = 3

y = ++x;
```

This time the increment operator is placed before the variable and at the end of the statements x will have a value of 4 and y will have a value of 4.

Note: As you become more familiar in using C you will see where you can use these constructs.

2.10 Loops : while, for, do

There are three loop statements available:

- while
- do while
- for

2.10.1 while loop

This is the easiest to understand. The while loop is constructed as follows:

```
while ( expression ) {

        statements

}
```

All the statements within the braces are executed repeatedly while the expression (maths) remains true – that is while the expression is not zero (any number in C is regarded as true).

The expression can be any mathematical operation that returns a number:

while (1)	Always true so creates a loop that does not end - an infinite loop.
while (a>b)	Only execute statements in braces when a is greater than b. (a and b are variables changed within the braceed statements).
while (a!=0)	Only execute statements in braces when a is greater than zero (a is a variable)

2.10.2 do while loop

The do while loop is similar to the while loop but moves the while expression test to the end of the braces – this means that the contents of the statements within braces are always executed at least once – then the expression is tested.

This can be useful depending on the task you are trying to solve

```
do {

        statements

} while ( expression );
```

Note: you need a semicolon at the end of this loop structure.

2.10.3 For loop

The for loop is the most powerful of the loop structures and lets you execute a block of statements a number of times.

The most common use is presented first as it is the easiest to understand.

2.10.3.1 Common for loop use

You construct the for loop as follows:

```
for ( initialize; test; action ) {
        statements
}
```

To use a for loop in the most common way you need to use a loop variable that you declare before using the loop.

The three expressions within the parenthesis are used to set up, test and act on the loop variable. The first expression (initialize) is executed only once – that is why you can use it for initialization. The next expression (test) is executed before the statement block and if it returns non zero value the statements are executed otherwise the for loop finishes and the next instruction after the for loop is executed. The next expression (action) is the loop variable action and this is executed after the statements have executed – usually you increment the loop variable using this expression.

Here is one of the most common examples

```
int i;

for (i=0; i<8; i++) {

        statements

}
```

The for loop acts in the following way

Initialization

Here the loop variable is initialized to zero only the first time that the for loop is executed – so this defines the starting action of the loop.

Test

The next expression is the test and here the loop variable is tested to see if it is still smaller than 8 – If false then the for loop is finished and execution jumps out of the for loop.

Execute

If the test is true then the statement block is executed.

Note: If the test results in false (zero) at the start then the statement block is not executed at all.

Action (loop variable action)

At the end of the statement block the loop variable action is updated – in this case the variable i is incremented by one.

The for loop then repeats itself starting at Test.

Detailed examination of the example:

Table 4: Detailed examination of the for loop example

Loop variable	**Test**	**statement block**	**action**	**No. times statement block is executed**
0	true	executed	add 1 to i	**1**
1	true	executed	add 1 to i	**2**
2	true	executed	add 1 to i	**3**
3	true	executed	add 1 to i	**4**
4	true	executed	add 1 to i	**5**
5	true	executed	add 1 to i	**6**
6	true	executed	add 1 to i	**7**
7	true	executed	add 1 to i	**8**
8	false	NOT executed	none	

I have expanded the action of the for loop to show you that the for loop is executed exactly 8 times. The example lets you use a variable that within the statement block takes on values from 0 to 7 and this is extremely useful since arrays in C are indexed starting at zero.

It is a common task to use an index variable within the statement block to access an element of an array using

```
    for (i=0; i<8; i++) {

        a  = b[i];

    }
```

makes this simple and easy to do with the added advantage that you can see the total number of executions in the for loop code itself.

Of course if you wanted to index from 1 to 8 then you would write

```
    for (i=1; i<=8; i++)
```

2.10.3.2 Advanced 'for' loop use

The previous section shows a common use of the for loop but the for loop is in fact much more powerful. The real definition of the for loop is :

```
        for( expression1 ; expression2 ; expression3 ) {

                statements

        }
```

Advanced 'for' loop 1 : infinite loop

If you leave out the second expression it is replaced with the true statement (in the compiler) so

```
        for ( ; ; ) {

                statements

        }
```

executes the statement block forever – it is an infinite loop

Advanced 'for' loop 2 : multiple expressions

Expressions can use a comma to separate multiple expressions which are then executed from left to right. For example:

```
        int i = j = 0;

        char c = 'i';

        for ( i = 10, c = 'o', j=2 ; i<=20; i++, j+=2 ) {

                statements

        }
```

As you can see you can make up complex operations within the for loop itself. Here i, c and j are initialized once to 10, 'o' and 2 respectively. The loop variable i is tested against the value 20. Notice that you can put multiple statements in any expression in the for loop and the action part also does two things – incrementing i by one and adding 2 to j.

Note: You can make up very complex code using this technique but it is often easier to leave the code simpler and not use multiple expressions as it makes the code so difficult to read.

2.10.4 Break

In any of the previous loop constructs you can use the break statement which will exit the loop when it executes. You can use it to exit from a loop and in conjunction with an if-else statement you can exit when you decide. For example:

```
int i = 10;

while ( i-->0 ) {

    statements

    if (i==2) break;

}
```

This will exit from the while loop when i equals 2.

Note: The break statement is also used within the switch statement (see section 2.11.4).

2.10.5 Continue

In any of the previous loop constructs you can use the continue statement this will skip a loop iteration immediately. If you use an if-else statement you can skip an iteration when you decide. For example:

```
int i = 10;

while ( i>0 ) {

    if (i==5) continue;
    statements

}
```

This would go down from 10 to one but will skip the value 5. When the code reaches the continue statement execution will immediately go back the the 'test' expression (of the innermost loop construct) without executing 'statements'.

2.11 Conditionals

Conditional statements let you execute other statements based on the outcome of a test.

The test is an expression using numbers and variables and is calculated using the logical operators.

2.11.1 if

The basic structure of an if conditional is

if (expression) statement;

For example

```
if (x<4 || x>20) statement1;
```

executes statement1 if x is smaller than 4 or x is greater than 20.

Note: The statements executed after the condition can be a statement block:

```
if (expression) {
        statement  1;
        statement  2;
        ....
        statement n;
}
```

statements 1 to n are executed if the expression evaluates to true.

2.11.2 If else

The next part of the if structure is the else statement – the first statement executes if the test is true and the second statement executes if the statement is false.

The basic structure of an **if else** conditional is

if (expression) statement1; else statement2

For example

```
if (x==0) statementa; else statementb;
```

statementa executes if x is zero and statementb executes if statement2 is not zero.

Again statement you can use statement blocks

```
if (expression) {
        statementa 1;
        statementa 2;
        ....
        statementa n;
    } else {
        statementb 1;
        statementb 2;
        ....
        statementb n;
    }
```

statements a1 to n are executed if the expression evaluates to trueand statementsb1 t on are executed if the expression evaluates to false.

2.11.3 If else if

You can use this conditional construct to test different conditions one after another.

```
    If (expression1)
        statement1;
    else if (expresion2)
        statement2;
    else if (expression3)
        statement3;
```

This if else if construct prioritizes the tests, testing expression1 first then expression2 then expression3. As soon as a test is true the corresponding statement is executed.

Note: The switch statement is more convenient if you have lots of tests to do.

2.11.4 switch

You can use the switch construct to test a variable against multiple integer constants. The switch statement conveniently lets you match several integers and execute a single piece of code (see example).

The basic structure of a **switch** conditional is

```
    switch (expression) {
        case expression-constant : statement;
        case expression-constant : statement;
        case expression-constant : statement;
    }
```

In use you need to end each set of statements with a **break** keyword otherwise execution continues onto the next case statement. Here is a more practical example:

```
    switch (gate_time) {
        case 1      : delay_ms(1);    break;
        case 10     : delay_ms(10);   break;
        case 20     :
        case 30     :
        case 100    : delay_ms(100);  break;
        case 1000   : delay_ms(1000); break;
        case 33     : delay_1s(); break;
        default     : delay_ms(100);
    }
```

Note: You can use more statements separated by semicolons before the break keyword so you can build up complex actions. The case 20 and case 30 blocks if matched to gate_time fall through and execute the case 100 statement (because there is no break statement after each one).

Although only integers are matched you can test character constants as the character constant will be cast to integer by the compiler e.g.

```
    switch (chr) {
        case 'a' : action = 1; break;
        case 'b' : action = 2; break;
        case 'c' : action = exit
    }
```

2.11.4.1 default

The default statement is an optional statement that executes if none of the preceding case statements matched the input – it must be placed at the end of the switch statement before the last bracket.

2.11.5 ?:

When I first saw this construct I though what on earth is that. But if you look closely at it, it gives an amazingly compact representation of the if else statement and it makes you look clever if you use it!

You can rewrite the following statement using the ?: construct...

```
    If (a!=0)
        x = a;
    else
        x = b;
```

... and you get

```
x = (a!=0) ? a : b;
```

Now you simply cannot love the elegance of that both pieces of code give exactly the same result!

To understand it compare the two pieces of code:

First of all the condition in both cases comes 1st:			
If (a!=0)	The 'if' starts the statement	x = (a!=0) ?	The '?' triggers the statement (although it is really an expression which is why it follows the condition keyword '?').
Next comes the statement executed if the condition is true			
x = a;	The expression returns 'a' if the condition is true.	a	The expression returns 'a' if the condition is true.
Next comes the separator (separating true/false statements)			
else	Separates true/false statement sets .	:	Separates true/false statement sets .
Next comes the statement executed if the condition is false			
x = b;	The expression returns 'b' if the condition is false.	b	The expression returns 'b' if the condition is false.

The condition comes first followed by the query (?) then the result if the condition is true followed by a colon (:) followed by the result if the condition is false.

Here's a summary:

```
x = <condition> ? <expression if true> : <expression if false>;
```

Note: The ?: construct can be used as an expression as it returns the result directly.

When you get into it this provides a super compact conditional statement.

2.11.6 Goto

Just when you thought you had left the horrors of BASIC behind you find that C HAS A GOTO STATEMENT!!!!

If you use this construct you will be shot.

Personally I have never used it and its use is extremely discouraged and the reason is the old BASIC programs that used line numbers using goto statements all the time makes code unreadable – and in fact with a little thought you can **solve any task without using goto so just don't use them**.

The only time to use them is in nested control loops where you need to abort processing, other than that there is no need for them.

For completeness the goto construct is written as:

```
goto label;
```

where label is written

```
label: statement
```

This can be applied to any statement.

2.12 Pointers

A pointer is an object that points to a memory address .

Pointers have three main uses

1. Controlling variables.
2. Pass by reference.
3. Creating linked lists.

2.12.1 Pointer declaration

Confusingly a pointer is declared using the indirection operator (*). To declare a pointer you put the indirection operator before the variable name as follows:

```
int *p;
```

this declares a pointer to integer labelled p. This means that the pointer p will assume that it points to integer data (the compiler enforces this assumptions and complains if p does not point to an integer).

*Note: A pointer to void can point to any type void *p – but this is a special case (see section 2.7.4).*

When you assign a number to a pointer you are telling the pointer object what address to point to so...

```
p = 0x4323;
```

... makes p point to address 0x4323.

2.12.2 Using 'Address of' operator (&)

Setting a pointer address directly is sometimes useful but in general it is more useful to point at existing variables in the C program and you do this by using the operator (&). This operator returns the address of a variable. So...

```
int x=90;   // Declare integer initialized to value 90.
int *p;       // Declare pointer to integer.
p = &x;       // Make pointer p point to x i.e. P equals address of x.
```

The above statement sets the address held by the pointer to the address of x.

Note: You can think of the operator (&) as 'address-of' operator so reading the last code statement you can say "p equals the address of x" which is exactly what is going on.

2.12.3 Pointer dereferencing

Dereferencing is just a fancy name for "**get the value that the pointer is pointing to**" but I suppose it is shorter!

Annoyingly the dereference operator is the (*) symbol (yes it is the same as the indirection operator and the multiplication symbol – why; I don't know).

Using the same example and adding a dereference operation:

```
int x=90;     // Declare integer initialized to value 90.
int *p;       // Declare pointer to integer.
int z;        // Declare another variable.
p = &x;       // Make pointer p point to x i.e. p equals address of x.
*p = 42;      // Set the "pointed to value to" 42
```

The above statement sets the contents of the address pointed to by p to 42 and since the address was set to the address of x then after this statement x will have the value 42.

This also works the other way around so..

```
z = *p;       // Set z to the "pointed to value of" p
```

After this statement z will have the value 42.

Note: the type of the pointer must match the type of the variable. The exception is the void pointer (see later) also you can cast the variable to force the type you want.

Some interesting pointer use : incrementing what a pointer points to

```
*p = *p +1        // Increments what p points to by 1
*p += 1           // Increments what p points to by 1
```

2.12.4 Pointer arithmetic

Interestingly you can change the address that a pointer points to by using some normal arithmetic operators e.g.

```
int ar[10];   // Declare an array of 10 integers.
int *p;       // Declare pointer to integer.
p = &ar[0];// Point p to 1st element of array.
```

```
    p++;
```

Moves the pointer forwards to point to the 2nd element of ar.

```
    p = &ar[0];// Point p to 1st element of array.
    p+=3;
```

Moves the pointer forwards to point to the 4th element of ar.

Note how the pointer is moved by the size of the type it is declared with.

All this means is that pointers move through arrays correctly as long as the pointer and the array are declared with the same type.

Note: You can also compare pointers using relational operators (section 2.8.3)'>' '<' etc. as long as the pointers are used in the same array.

Allowed pointer arithmetic

- Adding or subtracting integers to pointers – OK.
- Subtracting pointers – OK.
- Setting a pointer equal to another pointer of the same type – OK.
- Comparing pointers if they are in the same array -OK.
- Compare pointers to zero – OK.
- Setting pointers to zero – OK.

Invalid pointer arithmetic (not allowed) – basically everything else is not OK.

- Adding pointers
- multiplying pointers.
- Dividing pointers.
- Shifting pointers.
- Masking pointers.

Note the following syntax:

*p++	Increments the pointer after getting the value from memory.
(*p)++	Increments the object that is pointed to.

2.12.5 Void pointer

The void pointer declares a pointer that can point to any type but it is only for very advanced use and you probably won't need it.

2.12.6 Pointer and Array equivalence

Pointers and arrays are closely related but using pointers will be faster – this is because the compiler has to work out each index for an array.

Example Array index calculation e.g. a[10]

```
address = index * sizeof(pointer type) + base address;
```

Example pointer stepping e.g. p++;

```
address = address + sizeof(pointer type).
```

You can use either pointers or indexed arrays to solve the same problem because the following is true:

```
a[n] is the same as *(p+n)
```

That is the element at array index n :	a[n]
is the same as pointer arithmetic (and value) :	*(p+n)

True if p points to the start of the array which you can set as follows:

```
p= &a[0] or p=a
```

Note you can set p to an address inside the array: p = &a[4];

2.12.7 Pointer usage

2.12.7.1 Controlling variables

Here are some interesting example programs. Using pointers makes them extremely compact but a little difficult to understand.

strcpy (string copy)

The function accepts two strings copy s2 to s1

```
void strcpy (char *s1, char *s2) {
    while(*s2)
        *s1++ = *s2++;
}
```

As you can see pointers let you write compact code. The actions is – while the contents of pointer s2 is not zero i.e. not at the end of the string perform the action. Expanding the code gives:

```
*s1 = *s2;
s1++;
s3++;
```

You can combine the above statements because of operator precedence into

```
*s1++ = *s2++;
```

The pointer dereference (*) binds to the pointer before the increment so first of all a character is copied from pointer s2 to pointer s1 then the pointers are moved forwards 1 character. The process repeats until reaching the end of s2 (the zero terminator of string 2).

strlen (string length)

The function accepts a string and returns an integer.

```
int strlen (char *s) {
int c=0;
        while (*s++) c++;
        return c;
}
```

Again the while loop steps through the string until it reaches the zero termination and at each step it increases c by 1. So the length of the string is found (note it does not include the terminator character in the count) i.e. it is what you expect.

2.12.7.2 Pass by reference

Pass by reference is a term used to describe passing arguments to a function but instead of passing the data to the function a reference to the data is sent instead.

The reference is an address (of the variable) and the pointer is the mechanism used to handle this address within the function.

This allows the function to work on the address of the variable and therefore change the data in the calling function – normally data passed to a function is changed to a copy of the data and the function only works with the copy not the original.

For more details on pass by reference see section 2.12.8.6.

2.12.7.3 Linked lists

Since pointer objects can point to other pointer objects you can make lists of information linked by pointers i.e. linked lists. This gives a flexible way of storing, ordering and searching through information.

You can also order the information using two pointers to other objects to create a tree-like structure.

Linked lists are probably more suited to larger resource devices using dynamic memory allocation (malloc and free).

2.12.8 Functions

Functions are the building blocks of C programs and you can have as many functions as you need. The one function that must exist in you program is the **main** function as this is where your program starts (the compiler looks for the main function so it must be present in any C program).

You could write an entire program in just this one function and it would compile but the power of using functions is that you can split a task into manageable blocks that you can code and debug separately. You can then use these tested blocks in new functions for reuse.

In this way you can build up a program so that in the end it is much more likely to work. You can also build these debugged functions into libraries that you can use in other C programs (this is in fact what the standard C libraries are).

*Note: The **main** function must exist in your program as it is defined as the starting point for the compiler to begin program operation.*

2.12.8.1 Arguments

As its name implies a function performs an action and returns a result and as with a mathematical function you pass information into a function by supplying data to the function e.g.

Mathematical	fn(120, 232)
C program	function_name(120, 232)

Note: Valid function names are detailed in section 2.4.

The data input to a function are known as arguments and for a C programming you have to tell the compiler the type of argument you are giving the function. Here's an example

```
num1( int a, int b)
```

This tells the compiler that wherever the function num1 appears in the C code the types of the arguments entered into the function must match the types in the function definition .

```
So...
int int_one, int_two;
char one_char;
num1( int_one, int_two);     // This is valid code
num1( int_one,  one_char);  // Invalid code
```

So the compiler is ensuring that your code does not have mistakes in it – known as type checking.

2.12.8.2 Return value

To get data out of a function you use the **return** keyword as follows

```
int num1( int a, int b) {
        if (a>b)
                return a;
        else
```

```
                return b;
        }
```

Notice how you declare the return type declared just before the function name.

2.12.8.3 void types for functions

The special type void means that there are no arguments and you can also use this to declare that a function does not return data e.g.

```
        void function1 (void) {
                statements
        }
```

funiction1 takes no arguments and returns no data. You might think that there is not a lot of point to this function but there are other ways of getting data into and out of a function or the function could perform some 'no-data' action e.g. output a fixed value to a port.

You can use a file scope static variable see section 2.5.8. This is a variable that is available to all functions within the same file. For example it could be a buffer for a serial port and this function could work on that buffer needing no input or output – perhaps it checks and corrects invalid characters.

Note: You often declare main as

```
        void main(void)
```

Since in microcontroller programming there is no way of getting data to main. In a PC program you can send command line arguments to main using void main(int argc, char *argv[]);

2.12.8.4 Function arguments details

Here are some important facts about functions that let you understand them better.

Function arguments inside a function are copies.

When you give information to a function by passing arguments into the function a copy of the information is used inside the function so you can work on the value of an argument or even use the argument variable without affecting the calling routine.

You can not pass an array into a function.

To pass array information you have to use 'pass by reference' see section 2.12.8.6.

2.12.8.5 Functions returning more than one argument

A C function can only return one result and sometimes you want to process something and return more than one result.

You can do this by using 'pass by reference' see section 2.12.8.6.

2.12.8.6 Arguments - pass by reference

Passing arguments by reference refers to giving a function the address of a variable rather than using a copy of the variable within the function (passing a variable normally – section 2.12.8.4). There are three reasons to do this:

1. You need to return more than one variable from the function.
2. You want the function to work on the data in the calling routine.
3. The data to be processed is large e.g. a large structure variable.

Reason 3 depends on how much data needs to be sent to the function – remember that the function uses a copy of the variable and if there is a lot of data it may use up too much resource – also it takes time to copy data. These notes are important for microcontroller use where resource and time are critical.

Reasons 1 and 2 are the more usual reason for using pass by reference

Pass by reference uses the (now familiar) pointer declaration (section 2.12) to declare the function arguments. For example here's a function declaration:

```
void swap(int *ptr_a, int *ptr_b) {
}
```

Note: You can use any argument names you want to but adding 'ptr' gives you a little hint that there is something special going i.e. using pointers rather than ordinary variables. This can become especially important in larger programs and organizations and there are books devoted to naming conventions!

In this example you don't need a to use the return value - you can if you want to and a common use is to indicate with a flag whether the function succeeded or failed. The function is working only using the pointer references – which can you can use simply as real pointers.

Filling in the detail:

```
void swap(int *ptr_a, int *ptr_b) {
int dmy;
        dmy = *ptr_a;           // Remember for a pointer value use (*)
        *ptr_a = *ptr_b;        // Transfer contents of b to a.
        *ptr_b = dmy;           // retrieve the original 'a' and store in 'b'
}
```

Here's an example using that function:

```
int num1,num2;
num1 = 20;
num2 = 99;
swap (&num1, &num2);       // Note use of address-of-operator (&)
```

The result at the end is that

num1 has the value 99

and

num2 has the value 20

i.e. the numbers are swapped – so the function has operated on the variables in the calling routine – exactly what you wanted.

2.12.9 File structure

So why put this section near the end? Well file structure is important only when your programs have to go beyond one file. This happens when you want to group functions into related operational blocks e.g. you may have a set of functions for accessing the serial port or processing input data etc.

This section shows you how to use multiple source files and also how to communicate information between them. Of course variables are the mechanism to transfer data between functions (run time) but this section shows you how to make functions that are defined in one file visible to your other source files using headers (compile time).

2.12.9.1 C Source files

You put all your program code into a C source file which is simply a text file with a file extension type of .c

For simple programs (less than ~500 lines) you can get away with putting all your C source code into one file but there are three reasons why splitting functions into separate files is a good thing:

1. Grouping related functions improves readability.
2. Files allow use of static variables to hide data from other functions.
3. Using static functions hides individual functions reducing complexity.

Grouping related functions also makes it easy to turn the source code into a library for easy use within other projects.

2.12.9.2 Header files

A header file is a text file that has an extension file type of .h and you use a header file from within a c source file by using the preprocessor directive #include “filename” (see section 2.12.10.1). This inserts the entire file into the current file at the #include statement at the compiler preprocessing stage (you won't actually see the included text).

You can use header files at the top of the C source file (or the head of the file) hence the term 'header file'.

The header file is a communication mechanism that lets you communicate information between your source files (.c) and it is usual to put the following information into your header files:

- Function prototypes (see section 2.12.9.3).

- Constants (see section 2.12.10.2).
- Macro definitions (see section 2.12.10.3).

Prototypes are explained in detail in section 2.12.9.3 which includes an example.

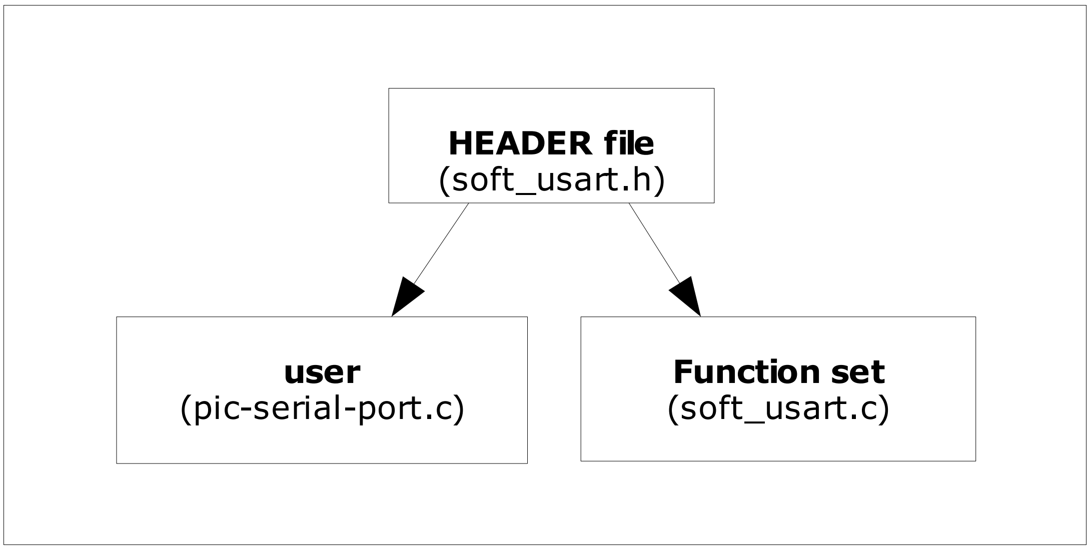
2.12.9.3 Prototypes

In fact you should have been using prototypes all the way through your program but I think they are only necessary when you are making up a complex program. A complex program is one in which you need to write functions in separate files (see section 2.12.9).

In the examples in this book the initial programs used very short examples that only require one function '**main**' and a prototype is not necessary because all the action is contained in one file (as long as all the functions are written down in the file before they are used elsewhere in the code – the compiler likes to know what a function is before you use it!).

Prototypes tell the compiler what types are expected in a specific function along with the return type so the compiler can check that you are using the same types when you use the function – known as type checking.

To explain prototypes lets look at a simple 'three file program':



The three files are:

12F675-pic-serial-port.c	- a user of the function set.
soft_usart.c	- the function definitions -the function set.
soft_usart.h	- the header file

Note: when you have debugged the functions fully you can make them into a library – which protects the files from alteration and lets you use them consistently throughout you projects.

What you typically do is write some useful functions (**Function set**) and provide a header (**Header file**) that gives access to only those functions that you want other source files to see (note see static functions section 2.12.9.5). Then you write you application code that uses the functions (**user**).

The header file ends in .h and you can include a header file into your current file using the #include statement e.g.

```
#include "soft_usart.h"
```

Note: These examples are taken from a real example that provides a soft serial port for RS232 (for a microcontroller without a built in USART).

Function prototype

A prototype is just a copy of the first part of the function without the body of the function.

The following code is written into the header file : soft_usart.h

A header file that defines the prototype
void _Soft_USART_Write(unsigned short chr);

The above code shows the prototype definition which is just the function return type, the function name and the arguments followed by a semicolon. Note you can write the arguments as follows:

```
void _Soft_USART_Write(unsigned short);
```

i.e. without the variable name (type checking does not need the variable name) but I find it easier to copy and paste the prototype from the C source file and add a semicolon – it's just less work.

Function body

The following code is written into the C file : soft_usart.c

A file that supplies the prototyped function

```
void _Soft_USART_Write(unsigned short chr) {
unsigned short mask=1,i;
unsigned int data;


        data = chr << 1;
        data &= ~0x0001; // START bit (=0).
        ... more code ...
}
```

Note: The above code is not the full function (it is left out as it is only illustrating the function body.

Using the function prototype

The following code is written into another C source file : pic-serial-port.c

A file that uses the prototyped function

```
#include "soft_usart.h"

void main(void) {
unsigned short chr=65;
        .. some code ...
        _Soft_USART_Write(chr);
        .. some more code ...
}
```

So this example shows you :

- How to share functions using prototypes.
- Where to put the prototype definition.
- Where to put the prototype body.

This is all you need to know about prototypes for their effective use.

Note: To make it all work you have to let the compiler know which files are in a project and you do this by either supplying the filenames on the command line (if the compiler is command line based) or using a built in project manager (when using a graphical IDE).

2.12.9.4 Constants and Macros

You can also put constants and macros into the header file so that they will be visible to which ever file they are included into.

For example changing header file : soft_usart.h

A header file that defines the prototype,constants and macros.
void _Soft_USART_Write(unsigned short chr);
#define BAUD_2400 2400
#define BAUD_9600 9600
#define get_byte_rate(bps) ((bps)/10)

These constants and macros would then be available in the file pic-serial-port.c (as that file includes soft_usart.h).

2.12.9.5 static functions

Normally all functions names are visible to all other files i.e. they have global scope – this means that you have to make sure that all function names are different but there is a trick.

The trick is to declare the function as a static function (just put the word **static** in front of the normal function declaration). This will then make the function name only visible in the file where it is declared.

So you can now declare functions that have the same name in different files and they will not interact.

Why could this be important? It saves you having to make up unique function names which you have to remember e.g. suppose you have LCD and a serial port code in separate files – you might want to use the word update to send data to each device.

Normally you would write the functions:

lcd_update

serial_update

But you could use the function name 'update' in each file as long as each is declared as a static function. Within each file the correct 'update' function will be linked in by the compiler.

OK this is a slightly artificial example but as your programs get bigger you may need longer function names and this technique reduces that ,keeping things simpler.

For instance what if you developed some equipment with 5 serial interfaces labelled serial_1 to serial_5 – you would have to have unique 'update' routines for each one e.g. serial_1_update, serial_2_update, serial_3_update, serial_4_update, serial_5_update – in each file you have to remember the correct function name whereas all you really want to do is update so here a static function name used within each file would help.

2.12.9.6 Function sets and libraries

Function sets

Section 2.12.9.3 on prototypes shows you how to organise your files to give access to functions using a header (soft_usart.h) and source code file (soft_usart.c). To add more functions you just add more to each file maintaining the header file as you write new functions. These functions are then available to any source code that includes the header file (within the same project).

Libraries

Libraries are simply pre-compiled function sets and depend on the compiler i.e. you can not use a library compiled using a different compiler.

Libraries make the total compilation process quicker as some work has already been done but the main advantage of a library is that you can use it within other projects

You can use a library just by including the header file (exactly the same header file as used for the function set) – the compiled library is put in a standard location (defined by the compiler manufacturer) along with the header file. Then any project can use the library.

Libraries are useful if you use a set of functions over and over in different projects otherwise you end up with the same source code existing in multiple projects and the temptation to add or tweak them is strong.

Once you duplicate code you will not know which is the master source code so using a library is also a way of controlling code – by keeping one master function set you can compile a library and use that in any other project. If you see a problem with the library operation you can go back to the master function set and alter that – in this way any other project that uses the library will benefit from the changes – without you having to individually update each instance i.e. the new library will be used when you compile the other project.

Obviously you have to be careful to ensure that the library works fully so it needs careful testing.

2.12.9.7 Standard libraries

Standard libraries are simply pre-made libraries shipped with your compiler they are standardized so that the same functions may be used with different compilers. This means that you can write code using the standard library and it should work on another compiler.

Note however that for microcontroller compilers the supported functions will be a subset of the available standard library functions and some libraries may not be supported at all. This is because you won't need the functions i.e. file handling will not be included as there is no file system in the microcontroller.

There are only a few standard libraries usually included with a compiler:

Libraries that may be included in a compiler.	
string.h	String manipulation.
stdlib.h	A collection of different functions including random numbers, max & min.
math.h	Floating point maths library including sin,cos,tan .
ctype.h	Functions to detect what class a character belongs to e.g. isalnum.

The most useful library is the string library with the following functions:

String library: string.h	
memcmp	compare memory.
memcpy	copy memory.
memmove	move memory.
memset	set memory to a value.
strcat	concatenate strings (append).
strchr	locate character within a string.
strcmp	compare strings.
strcpy	copy strings.
strlen	return string length.
strncat	append strings no more than n characters.
strncpy	copy strings no more than n characters.
strspn	string span check chars from s2 that are in s1.

Floating point maths library: math.h

acos asin atan atan2	ceil cos cosh exp	fabs floor frexp ldexp	log log10 modf pow	sin sinh sqrt tan tanh

Standard library : stdlib.h	
abs	Absolute value of integer (positive number only)
atof	ASCII to float (turn a string floating point number into a floating point value).
atoi	ASCII to integer (turn a string integer number into an integer).
atol	ASCII to long (turn a string long number into an long).
div	Division of integers – returns quotient and remainder in a structure.
ldiv	Division of longs – returns quotient and remainder in a structure.
labs	Absolute value of long (positive number only)
max	Maximum of two integers.
min	Minimum of two integers
rand	Random number between 0 and 32767
srand	Random number generator seed (randomize the random number generator
xtoi	Hex digit string to integer.

ctype.h			
isalnum isalpha iscntrl isdigit	isgraph islower isprint ispunct	isspace isupper isxdigit toupper	tolower isalnum

Note: There should be lots of non-standard libraries supplied with the compiler e.g. LCD, RS232, RS485, GLCD etc. These let you work at the hardware level more easily saving you creating common functions from scratch.

2.12.9.8 Hiding data within a file

Sometimes you want some data to be visible to all functions within a file but you don't want other files to see the data i.e. you want to make the data private to the file (c.f. C++).

For instance if you had a buffer system to accept input data in a circular buffer you would not want to let functions outside the file to control or overwrite that buffer so you would want that buffer to be visible only within the file.

You can make the variable (or array) private to the file by declaring it as a static variable (or array) - declared outside of any function block – see section 2.5.6.

Note: When you do this the only way of accessing the information is by using the functions provided within the function set. This is a good thing as the functions hide the detail of what is going on inside the function so you can change the operation of the function without affecting other code i.e. other code does not know about the internal variables.

2.12.9.9 Hiding functions within a file

Sometimes you will also want to make a function visible only within a file – you may want to break a piece of code into sections to make it easier to read but not want the separate functions to be used outside the function set (or file).

Again you can make the function private by declaring it as a static function see section 2.12.9.5.

Note In this case the header file does not contain the static function prototype.

2.12.10 Preprocessor

The preprocessor is the first part of the compiler processing action and is called pre-processor because it acts before all the other compiler processes. It is really a separate language incorporated in the C programming language.

All preprocessor commands begin with the hash character '#' and this is how the compiler understands that you want to do some pre-processing.

Typical things you can use pre-processing for are:

- Including other files (Section 2.12.10.1).
- Defining constants (Section 2.12.10.2).
- Defining macros (Section 2.12.10.3).
- Conditional compilation (Section 2.12.10.4).

This last one: conditional compilation is useful to cut out or to enable code based on another macro definition so you can have code for different targets all within the same file but enabled when required.

2.12.10.1 #include

There are three forms of the #include

```
#include "filename"

#include <filename>

#include macroname
```

Note: The last one is rarely used.

The include command followed by the text to the end of the line is replaced with the contents of the filename before the compiler starts doing anything else (preprocessing).

The difference between the first two is:

    #include "filename"

> The compiler searches for filename starting in the current project directory. If it is not found there it then searches are implementation dependent – typically it will search in the compiler's standard directory and after that in a user defined path.

    #include <filename>

> The compiler searches first in the compiler's standard directory. If it is not found there then searches are implementation dependent – typically it will search in a user defined path.

Note: By standard directory I mean the compilers directory of include files.

Typically angle brackets are used to pull in the built in libraries that you need to use in your program e.g.

```
#include <stdio.h>

#include <string.h>
```

Note: Normally a compiler will complain if these includes are not present and you use a function that is in the corresponding library – but the compiler used here automatically includes the standard library files so the includes are not needed. This is convenient as you can use any function you know about e.g. strlen without needing to find out which library it is in so you can put in the include information but if you move to another compiler you will need lines such as #include <string.h>

Typically quotes are used to pull in the header files of files in your project (see section 2.12.9.2).

2.12.10.2 #define – Constants

One of the most important uses of the preprocessor is in defining constants that do not use up any resources.

if you write...

```
#define MAX 20
```

...wherever MAX appears in the following C source code it is automatically replaced with the text “20” as a preprocessing compiler action. Using this method is better than defining constants since const are sometimes placed in RAM/program memory using it up (this depends on the compiler).

In fact #define is a macro command (see section 2.12.10.3)...

```
#define macro replacement_text
```

and wherever the defined macro appears the replacement text is put in its place.

Using defined constants in this way at the top of a file also lets you change the operation of the code without changing every instance manually i.e. all following code (to the end of the source file) where MAX is seen will be replaced by the replacement text.

2.12.10.3 #define – Macros

A macro uses the same format as the constant definition (section 2.12.10.2):

```
#define macro() replacement_text
```

but uses parenthesis to add arguments in its output replacement text:

```
#define setBit(var, bitnum)   (var)|=(1<<(bitnum))
```

This defines a macro that takes two arguments var and bitnum. You can use it as follows:

```
setBit(PORTB,3);
```

This will expand in line (after preprocessing) as :

```
(PORTB)|=(1<<(3));
```

and is a lot easier to read in the macro form

Note: Although the macro looks like a function it is not – there is no function call and the code is expanded in the source file before compilation.

Note: It is important the use parenthesis to enclose each macro argument as macro expansion may go wrong if you don't e.g. when the arguments are complex expressions.

Normally the macro ends at the end of the line it is defined on but to extend the macro to more lines add a backslash character at the end of each line e.g.

```
#define setBit(var, bitnum)  \
        (var)|=(1<<(bitnum))
```

Defines the same macro over two lines (for more lines add backslashes as needed).

2.12.10.4 #ifdef, #endif - Conditional compilation

You can control which parts of your program are included in the source using conditional compilation. Here is how you conditionally include source code

```
    #define SECTION1


    #ifdef SECTION1
        statements...
        statements...
    #endif
```

The preprocessor command #ifdef means if a macro is defined then the statements following the #ifdef are used.

Note you can also use #if defined macro – but not all compilers support it.

Notice how you don't need to supply a value for the macro in :

```
    #define SECTION1
```

the act of making a #define statement creates the macro SECTION1 that contains nothing but it does exist. This is why the #ifdef can detect the maco SECTION1.

2.12.10.5 #ifndef - Conditional compilation

Alternatively you can use #ifndef meaning 'if macro is not defined' to include source code:

```
    #ifndef SECTION1
        statements...
        statements...
    #endif
```

In this case the statements are included when the macro SECTION1 does not exist

Note: You can also use #if ! defined macro – but not all compilers support it.

2.12.10.6 #undef - Forget macro

To forget a previously defined macro use #undef e.g.

```
    #undef SECTION1.
```

There are

2.12.10.7 #else - Conditional compilation

You can use #else with the conditional preprocessing statements #ifdef and #ifndef to include code for the opposite part of the test e.g.

```
#ifdef SECTION1
        statements1...
        statements1...
#else
        statements2...
        statements2...
#endif
```

The above code includes statements1 if the macro SECTION1 exists and statements2 if it does not. Alternatively

```
#ifndef SECTION1
        statements1...
        statements1...
#else
        statements2...
        statements2...
#endif
```

The above code includes statements1 if the macro SECTION1 does not exist and statements2 if it does exist.

2.12.10.8 #elif – Conditional compilation

You can use elif to test multiple conditions e.g.

```
#ifdef SECTION1
        statements1...
#elif SECTION2
        statements2...
#else
        statements3...
#endif
```

The above code includes statements1 if macro SECTION1 exists, statements2 if macro SECTION2 exists and statements3 if neither exist.

2.12.10.9 # - Argument to quoted string

Within a macro definition using the # operator infront of an macro argument inserts the quoted argument text rather than the value of the argument itself.

e.g.

```
        #define varout(name) print(#name,name)


         dispvar(gate_time);
```

would give the source code: print(“gate_time”,gate_time);

2.12.10.10 ## - Token pasting

This is a token pasting operator so

```
        #define xypos(a,b) a ## _ ## b
```

gives x_y when you execute xypos(x,y)

2.12.10.11 #pragma -Compiler directive

Some compilers use this to control actions for the target microcontoller e.g. setting fuses in a PIC micro. The commands following #pragma are defined by the compiler manufacturer. There may be alternate methods used to set up the fuses e.g. a separate options screen instead of using this command.

2.12.10.12 #line - Diagnostic

Control compiler (diagnostic)

```
        #line const “filename”
```

Make compiler believe that const is the next line number and the current file name is 'filename'

This may not be implemented in your compiler.

2.12.10.13 #error - Diagnostic

Generate a diagnostic error output

```
        #error tokens
```

...would output an error message followed by the tokens.

This may not be implemented in your compiler.

3 References

C programming language, B W Kernighan, D M Ritchie

University of life.