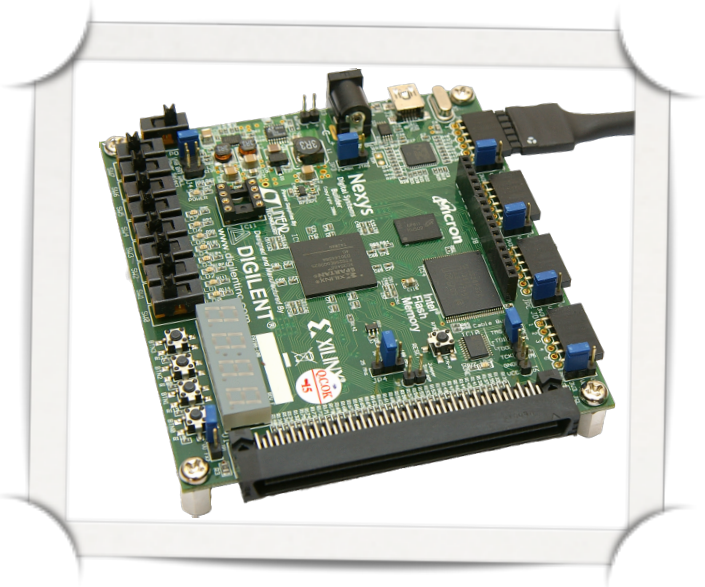
INTRODUCTION TO EMBEDDED C

by Peter J. Vidler

Introduction

The aim of this course is to teach software development skills, using the C programming language. C is an old language, but one that is still widely used, especially in embedded systems, where it is valued as a high-level language that provides simple access to hardware. Learning C also helps us to learn general software development, as many of the newer languages have borrowed from its concepts and syntax in their design.



Course Structure and Materials

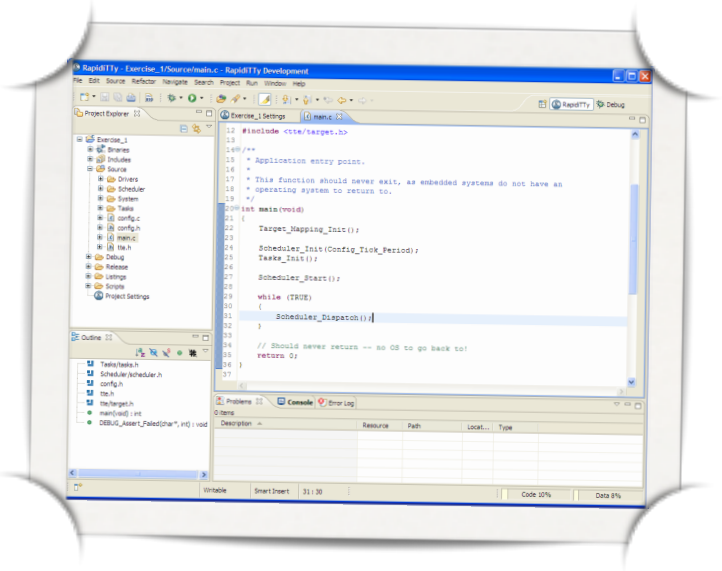
This document is intended to form the basis of a self-study course that provides a simple introduction to C as it is used in embedded systems. It introduces some of the key features of the C language, before moving on to some more advanced features such as pointers and memory allocation.

Throughout this document we will also look at exercises of varying length and difficulty, which you should attempt before continuing. To help with this we will be using the free RapidiTTy® Lite IDE and targeting the TTE®32 microprocessor core, primarily using a cycle-accurate simulator. If you have access to an FPGA development board, such as the Altera® DE2-70, then you will be able to try your code out in hardware.

In addition to this document, you may wish to use a textbook, such as “C in a Nutshell”. Note that these books — while useful and well written — will rarely cover C as it is used in embedded systems, and so will differ from this course in some areas.

Getting Started with RapidiTTy Lite

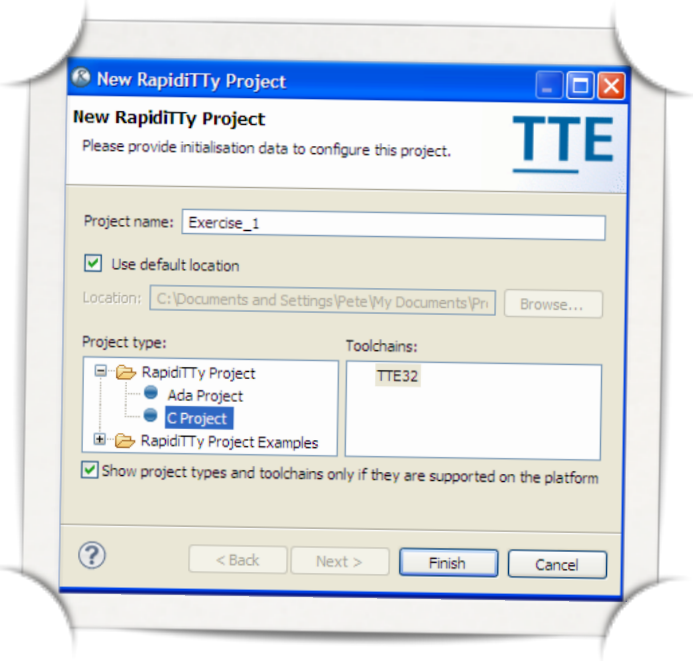
RapidiTTy Lite is a professional IDE capable of assisting in the development of high-reliability embedded systems. We can use it to develop in C, Ada and Assembly language; to statically analyse object code in order to determine all the code, data and stack memory requirements; and to gather measurements and statistics about runtime performance and timing behaviour of our system.



Throughout this course we will be making use of RapidiTTy Lite's TTE32 microcontroller simulator to work through exercises and get a feel for embedded systems development in C. If you do not have the latest version of RapidiTTy Lite installed, then you can download and install it by following the instructions provided on our web-site[1].

Basic concepts and project creation

In RapidiTTy — as with most IDEs — all development work is carried out on one or more *projects*. A number of projects are grouped together in a *workspace*; we typically only have one workspace open at a time. When starting RapidiTTy, you will be asked to choose a location for the workspace; if it doesn't exist yet, this will create an empty workspace in the chosen location. The welcome screen that greets us provides access to basic training videos for the IDE — it is recommended that you watch some of these before continuing.



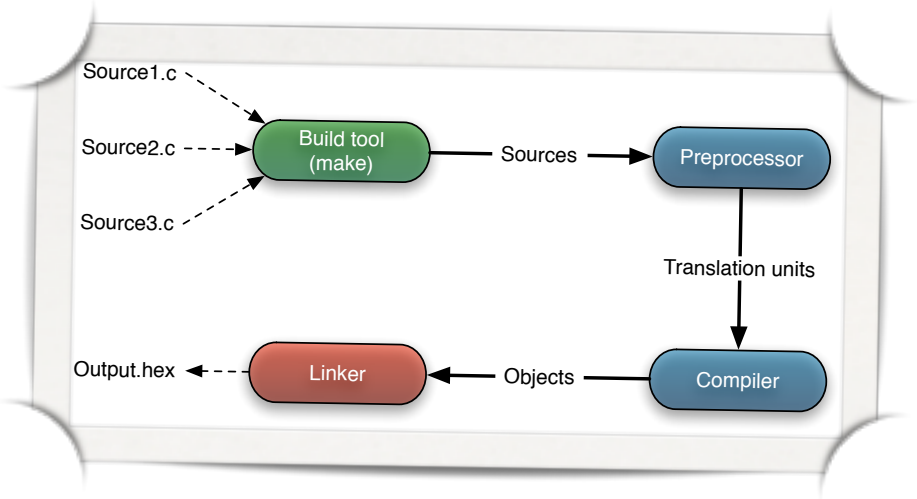
We can start a new project by selecting 'New' from the File menu and choosing 'RapidiTTy Project' from the resulting sub-menu. We can then select the type of project and choose a name. For the exercises in this course, we will always select the 'C Project' option. Pressing 'Finish' will create the project.

[1] See http://www.tte-systems.com/products/lite for further details.

Concepts of Compilation

A processor works by executing a series of simple, unambiguous instructions that are fed to it from memory. These instructions are stored and processed in a binary form that is easy for the processor to interpret, but impractical for a human to write. Instead, to make our lives easier, we develop software in higher level programming languages.

These languages involve writing *source-code*, which must then be translated into a binary form that the processor understands. This process is called *compilation* and is the main purpose of the *compiler*. In fact, the specific compilation process used for the C language involves a number of steps, such as pre-processing, compilation and linking. These steps allow us to use more than one source-code file at a time, which makes working on a large project easier.



In C, there are two main file types — source files and header files. These header files are used to contain the information necessary for more than one source file to work together, and can be *included* by source files using the '`#include`' pre-processor directive. When this directive is found in a source file, the pre-processor will simply replace the line on which it occurs with the entire contents of the given header file. The result of this combination is called a *translation unit*, which is what is passed to the compiler.

Every source file in the project must be pre-processed separately, which results in one translation unit per source file. Each translation unit is then compiled separately, resulting in one *object file* per translation unit. Object files contain executable code in a form that can be understood by the processor, but which is not yet complete. We must combine all the object files together using the *linker* in order to produce the final executable, which we can actually run on the processor.

Clearly this is not a process we want to repeat by hand every time we change a source file! Instead, there are a variety of *build tools* available, all of which can automate the process for us. RapidiTTy Lite includes this functionality, as you can see if you select the 'Console' view at the bottom of the screen. Here you will find each call to the compiler — which by default takes care of pre-processing internally — and the final call to the linker.

Parts of a Simple Program

Learning any new language is best done by writing some simple programs and increasing in complexity as we go. There is a lot of source-code required to get even the simplest of programs up and running — this is especially true in embedded systems where we may not have a full operating system to help! Before we can start looking at some real programs, it makes sense to take a quick tour of the C language itself.

Expressions

In C, an *expression* consists of the combination of *operators* and *operands*. They closely resemble arithmetic; for example, `'7*3-2'` is valid as both an arithmetic expression and as source-code. In that example, the `'*'` and `'-'` are operators; the `'7'`, `'3'` and `'2'` are operands. In fact, they are not simply operands but also *numeric literals*, because they are written as values directly in the source-code.

Infix operators

When an operator is written in this way, in-between its two operands, it is called an *infix* operator:

Operator	*Name*	*Type*
`+`	Sum	*Arithmetic*
`-`	Difference	*Arithmetic*
`*`	Product	*Arithmetic*
`/`	Quotient	*Arithmetic*
`%`	Modulo / Remainder	*Arithmetic*
`>`	Greater than	*Logical*
`<`	Less than	*Logical*
`>=`	Greater than or equal to	*Logical*
`<=`	Less than or equal to	*Logical*
`==`	Equal to	*Logical*
`!=`	Not equal to	*Logical*
`&&`	And (logical)	*Logical*
`||`	Or (logical)	*Logical*
`&`	And (bitwise)	*Bitwise*
`|`	Or (bitwise)	*Bitwise*
`^`	Exclusive or	*Bitwise*
`<<`	Shift left	*Bitwise*
`>>`	Shift right	*Bitwise*

From the table, there is clearly some overlap between logical and bitwise operators. It is important to remember that we should only use bitwise operators when we wish to produce a direct effect on the internal binary representation of a value — this is rarely used in desktop systems, but is often very useful in embedded systems. Logical operators, on the other hand, are most often used with control constructs to specify a condition that must be met; we will cover this situation in detail later.

Note that the above table is incomplete — later, we will also look at some *assignment* and *sequence* operators. There is also a *ternary* operator, which is a particularly strange beast and will be covered when we get to control structures.

Prefix and postfix operators

Although infix operators are the most numerous, there are also a number of *prefix* and *postfix* operators to consider. Prefix operators are placed before their operand, such as the '`-`' in the expression '`-27`'. Similarly, postfix operators are placed after the operand.

Operator	Name	Type	Pre or Post
`+`	Positive	*Arithmetic*	*Prefix*
`-`	Negative	*Arithmetic*	*Prefix*
`!`	Not (logical)	*Logical*	*Prefix*
`~`	Not (bitwise)	*Bitwise*	*Prefix*
`++`	Pre-increment	*Arithmetic*	*Prefix*
`--`	Pre-decrement	*Arithmetic*	*Prefix*
`++`	Post-increment	*Arithmetic*	*Postfix*
`--`	Post-decrement	*Arithmetic*	*Postfix*

Note that the above table is also incomplete — the *referencing* operators are missing; we will get to these much later on, when we cover pointers and addressing. Also note that — once again — there is an overlap between logical and bitwise operators. As before, logical operators should be used primarily in control structures or with boolean variables (which we will cover later). The bitwise operator inverts the internal binary representation of a variable, replacing every '`1`' with '`0`' and every '`0`' with '`1`'.

The post-increment and post-decrement operators differ from their prefix counterparts only in the time at which the operation takes place. The expression '`x++`' will result in the value of '`x`' *before* it is incremented. By contrast, '`++x`' results in the value of '`x`' *after* it is incremented. The increment happens in both cases, but the expression's value is different.

For example, if `x` is equal to `5` and we assign '`++x`' to `y`, then subsequently `x` and `y` will both equal `6`. If we then assigned '`x++`' to `y`, then `y` would equal `6` and `x` would equal `7`.

Statements

Expressions are useful, but by themselves they do very little for us. To employ expressions, we use them in *statements*. Statements are made up of one or more expressions that end with a semi-colon character; they are the basic building blocks of the C language. There are many different varieties, but for now we will focus on a simple application using only assignments and function calls.

Assignment statements

Probably the most immediately useful statement to learn is the *assignment* statement. This comes in a variety of forms, depending on which assignment operator is used — there are several of them, although most are simply short-hand for a longer expression. Here is an example of a simple assignment statement — we will use a trivial implementation of the Pythagorean Theorem:

```
// There is no 'squared' operator, so improvise:
c_squared = a * a + b * b;
```

Notice the first line — everything from '`//`' up to the end of the line is a *comment* and will be completely ignored by the compiler. We can use this to annotate our code with helpful notes to remind both us and our colleagues of the purpose of a particular section of code.

Often, it is considered bad practice to use comments to describe *what* the code is doing, as this should be obvious in any well-written program; instead, focus on using comments to describe the *why* and *how* of your source-code. The exception is comments for functions, as these are often intended to describe the use of the function to developers who may not be familiar with it (functions will be discussed in detail later).

As mentioned in the comment, there is no 'squared' operator in C. Instead, we must use the product operator to multiply each value by itself to achieve the same result. Note that the usual precedence rules of arithmetic apply here — multiplication will always happen before addition in the same expression, unless we specify otherwise with parentheses. The actual assignment happens last, and the result of the expression '`a * a + b * b`' will be stored in the variable '`c_squared`' for future use.

In the previous section, we discussed assigned '`++x`' and '`x++`' to `y`. Here is an example of what this might look like in source code — note that this style of commenting (describing exactly what the code does in painstaking detail) should be avoided in practice:

```
x = 5;
y = ++x; // After this line, y = 6 and x = 6
y = x++; // After this line, y = 6 and x = 7
```

Here is a brief summary of the assignment statements, with both short- and long-hand equivalents, by example:

Operator	*Example Usage*	*Equivalent Statement*
=	`a = b;`	
+=	`a += b;`	`a = a + b;`
-=	`a -= b;`	`a = a - b;`
*=	`a *= b;`	`a = a * b;`
/=	`a /= b;`	`a = a / b;`
%=	`a %= b;`	`a = a % b;`
&=	`a &= b;`	`a = a & b;`
|=	`a |= b;`	`a = a | b;`
^=	`a ^= b;`	`a = a ^ b;`
>>=	`a >>= b;`	`a = a >> b;`
<<=	`a <<= b;`	`a = a << b;`

Integer variables, types and declarations

In the previous assignment example, the expression '`a * a + b * b`' was assigned to the *variable* '`c_squared`'. This can be thought of as a chunk of memory that we can use to store a value. The exact size of this chunk, as well as the range and nature of values it can store, are specified by the variable's *type*. For now, we will limit ourselves to integers; that is, variables that can only ever contain whole numbers.

In C, the size of an integer may be specified in the source-code. This size can be either very specific (an exact number of binary bits to use), or a vague minimum value where the real size may vary from one compiler or platform to the next. The latter mechanism is older and has existed in C from the beginning, whereas precise sizing specifications were only added in the most recent language revision. At this point, enough compilers support the latest standard that we can often use the precise mechanism (although we will cover both here).

In addition, an integer may be either *signed* or *unsigned*. Being signed signifies that the integer may be either positive or negative; being unsigned signifies that the integer must always be positive. The advantage of specifying that a variable is unsigned is that the same chunk of memory will be able to represent a higher positive value than if it was interpreted as signed.

This difference is explained in greater detail towards the end of this course; for now it is enough to know that unsigned integers can have a higher value and can never be negative.

C is a statically typed language, which means that variables must be *declared* before we can use them. A declaration for the 'c_squared' variable must be found in the source-code before we assign a value to it, and might look something like this (note that this is also the variables *definition*):

```
uint32_t c_squared = 0; // The '= 0' is optional!
```

The 'uint32_t' is a new-style type specifying that 'c_squared' is an unsigned integer that takes up a chunk of memory 32 bits (4 bytes) in size. Here are some other integer types that we may use:

New Style Type	Minimum	Maximum	Old Style Approximation
int8_t	-128	127	signed char
uint8_t	0	255	unsigned char
int16_t	-32768	32767	signed short
uint16_t	0	65535	unsigned short
int32_t	-2147483648	2147483647	signed long
uint32_t	0	4294967295	unsigned long
int64_t	-9223372036854775808	9223372036854775807	signed long long
uint64_t	0	18446744073709551615	unsigned long long

Declaring and defining basic functions

Declarations, statements and expressions alone are insufficient for developing anything but the most trivial of programs. The only practical way to handle the complexity of a larger project is to break the program up into separate logical units, the smallest of which in C are called *functions*. Like variables, functions can be declared and defined; unlike variables, the difference is far more pronounced. The source-code that contains the actual implementation of the function is known as the function *definition*, and any definition will also count as a declaration for the source-code that comes after it.

A function must be declared before it can be used; so either we only use it in the same source-file that it is implemented (and below the definition), or we put a declaration at the top of any file that will use it. To make life easier these declarations are often placed in the header file, which is then reused in other source-files with the '#include' pre-processor directive. Here is an example of a function declaration:

```
// Squares the value and returns the result
uint32_t Square(const uint32_t number);
```

Notice the semi-colon at the end? This is required for declarations but is *never* used when writing a function definition. Here is the definition of the function declared above:

```
// Squares the value and returns the result
uint32_t Square(const uint32_t number)
{
    return number * number;
}
```

The function in the above example is called '`Square`', and it takes a single *parameter* or *argument* (written in parentheses after the function's name) called '`number`' which we can see from the type is a 32-bit unsigned integer (with a '`const`' qualifier to tell the compiler that it will be read-only); if more than one parameter is needed, they must be separated by a comma. Notice that this function will return a `uint32_t`, as seen by the specifier before the function's name.

There are two new types of statement in the above function. First is the *block* statement, which is the '`{`' and '`}`' characters and everything in-between. Every function definition has a block, which is used to group together and contain other statements. As blocks are statements that contain other statements, they may be *nested* within other blocks.

Second, within the block in the above example, is the *return* statement; these are used to exit a function and optionally return a value. Functions in C may only return a single value of a specific type, which is given in the declaration before the function's name. To create a function that does not return anything, we must give the type '`void`' here and either use a statement containing only '`return`' (and a semi-colon), or simply allow the execution to reach the end of the function (in which case a return statement is not required).

Calling functions

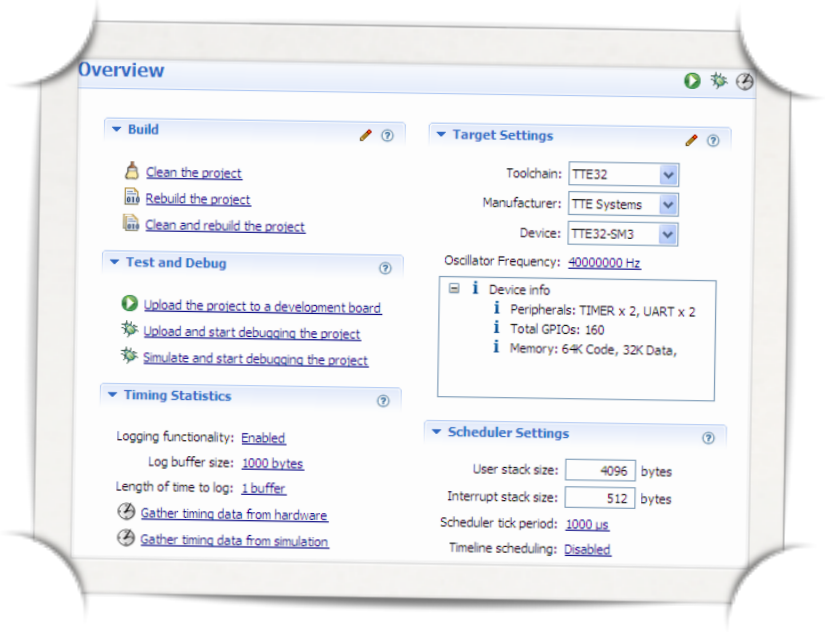
In C, every statement we write must be within a function definition's block. Variables and functions may be declared or defined outside of any function, provided they do not contain code to be executed — assignments with simple expressions that the compiler can evaluate to a number or value are fine, function calls or expressions using other variables are not. Note that every program begins by executing the `main` function, which we must define. In embedded systems with no operating system, this function can never be allowed to return.

A function may be called by itself as a statement, or in an expression as part of a larger statement. The former is often used for functions that do not a value; the latter can only be used when the function *does* return a value. When used as an expression, the return value becomes the value of that expression. For example, we use '`square`' like this:

```
// This calls a function, so must be put inside one!
uint32_t c_squared = Square(a) + Square(b);
```

Putting Theory into Practice

A RapidiTTy Lite project has a great deal of source-code generated for us at creation. Because there is no operating system, the project will have code to interact with the hardware (often known as the *drivers*), some assembly code to initialise the system at startup and a *scheduler* to execute specified units of code (often known as *tasks*) at very precise time intervals.



In a normal project, there will only be one task created: `Flashing_LED`, which can be found in the '`tasks.c`' file, under the '`Source/Tasks`' folder. If we scroll to the bottom of the file we can see the task itself, which is just a function:

```
void Flashing_LED(void)
{
    static gpio_value_t led_state = GPIO_LOW;

    GPIO_Write(LED_Pin, led_state);
    led_state = !led_state;
}
```

For now, this probably won't mean much to you. The '`void`' return type signifies that the function takes no parameters and does not return a value; notice the lack of a `return` statement, indicating that the function will only return when the last statement in its block has been executed.

As per the name, `Flashing_LED` is simply designed to toggle an LED on and off again. To this end, the scheduler will execute the task function every 500 milliseconds (so that the LED turns on once per second and off once per second). Sending a value to the LED's pin is accomplished by calling the `GPIO_Write` function, which is part of the General Purpose Input/Output driver found in the 'Drivers' folder.

For the remainder of this course, we will not be concerned with the intricacies of tasks and schedulers. Instead, we will simply modify the existing `Flashing_LED` task function to do whatever we are trying to accomplish. Later on, you may want to try writing your own tasks and seeing how they are executed by the scheduler.

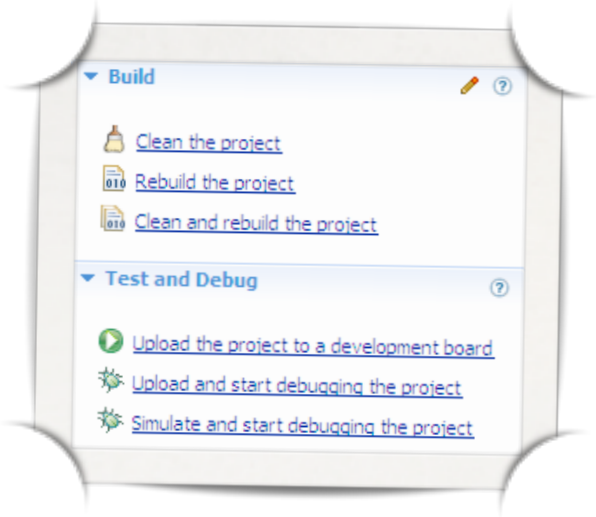
Printing some output

Most books and courses that teach C programming begin by outputting some text to the screen (for historical reasons, the phrase "Hello world!" is most common). This text is usually written using a function called '`printf`' from the C standard library. But we are writing software for embedded systems and so don't actually have a screen; instead, most compiler tool-chains provide a standard library that writes to the first serial port. Note that the `printf` function requires a lot of user stack space — we should increase this to `4096` on the project settings page, or the results may be unpredictable!

Let's try replacing the code in your `Flashing_LED` function, and see what happens (we must also add '`#include <stdio.h>`' to the top of the file to include the declaration of `printf`). Change the content of the `Flashing_LED` function's block to the following:

```
printf("Hello world!\r\n");
```

Compiling the program



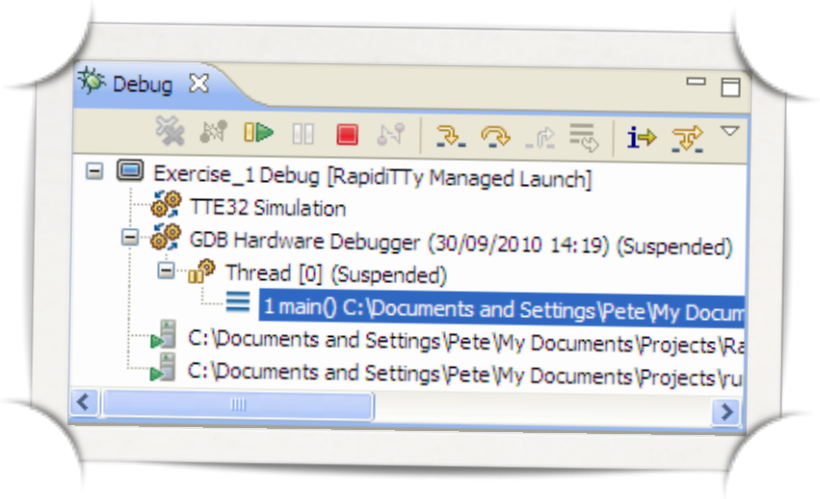
In order to see the result of our change, we must save the file and recompile the project. Open the 'Project Settings' file and you should see something like the screenshot to the right. At the top-left, we can select the 'Clean and rebuild the project' link in order to recompile the entire project.

We can think of this file as our window into the project. Here there are links that build, upload, debug and carry out timing analysis. We can see details of the targeted processor, oscillator frequency, stack sizes and scheduler settings. We also have the facility to look at the results of static analysis carried out on the source-code, which can be found on the 'Analytics' page at the bottom of the editor.

To actually run the program, select the 'Simulate and start debugging the project' link. This will start the simulator for the target processor (which is the TTE32 processor core) and then 'uploads' the compiled code to it. A dialog boxes will pop up during this process asking if you want to switch to the 'Debug' *perspective*, to which you should select 'yes'. A perspective is a group of views and editors that are specialised for a particular purpose, in this case debugging.

If you do not want to switch now, or you press 'no' button by accident, you can switch manually by pressing the 'Debug' button at the top-right of the window.

The program will now begin executing, only to pause on the first line of the 'main' function.

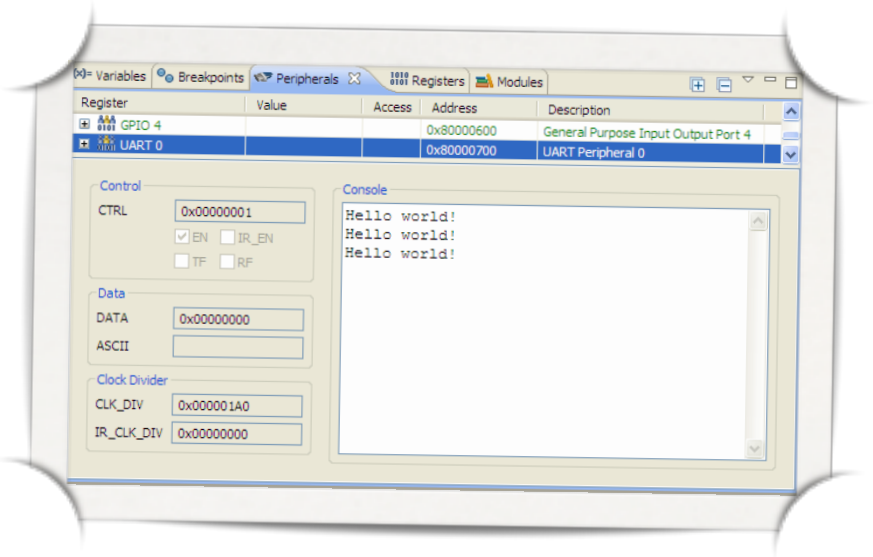


In the 'Debug' view — found at the top-left of the screen by default (and shown in the screenshot to the right) — we can use the buttons at the top to run, stop and step over or into individual functions.

Before doing anything here, we should select the peripheral view and arrange it as desired.

Viewing simulated peripherals

The peripherals view can be found in amongst the other views at the top-right of the debug perspective. You may find that you have to use the 'step over' button from the debug view once before the peripherals view is populated. Once we can see the list of peripherals, we can select 'resume' in the debug view.



As we can see in the screenshot above, the peripherals view contains all of the peripherals available on the target microcontroller, or at least those that are currently configured for us. After selecting the '`UART 0`' peripheral — and rearranging the view to make everything more visible — we will finally be able to see the "Hello world!" text that is outputted by our (now poorly named) `Flashing_LED` task function. Notice also the values under 'Address' at the top — numbers starting with '`0x`' are in hexadecimal notation, which can also be used in C source-code.

If the target is left running, you should eventually see more text appear in the console (as shown in the above screenshot). This is because the scheduler is executing the function repeatedly with a period of 500 milliseconds; of course, depending on the speed of your computer, the simulator will probably take longer than that. This is because it is providing simulation with an extremely high level of detail, right down to the cycle level.

Printing the values of variables

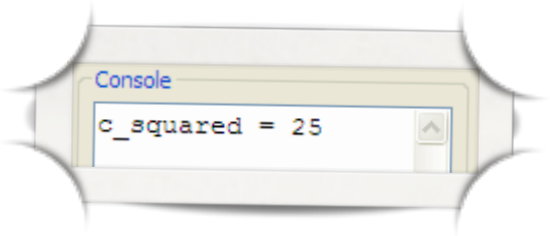
The `printf` function is very powerful, capable of far more than just outputting plain text — the '`f`' in the name stands for 'format', as it is also capable of formatting and printing the values of many types of variable. For example, we could do this:

```
uint32_t Square(const uint32_t number)
{
    return number * number;
}

void Flashing_LED(void)
{
    const uint32_t a = 3;
    const uint32_t b = 4;
    printf("c_squared = %lu\r\n", Square(a) + Square(b));
}
```

Notice that we are now passing two parameters to printf — the first is some text, and the second is the result of an expression. There is an important lesson here — expressions given as parameters are always evaluated *before* the function is actually called. The `printf` function in particular can take any number of parameters through the use of a variable-length argument list. These are not commonly used in embedded systems and will not be discussed further in this course.

The first parameter passed to printf is always the *format string*. A string represents text as a sequence of individual characters in an array (which we will cover later). In the example, the format string is passed as a string literal — just as numeric literals are numbers written directly in the source-code, so are string literals simply strings in the code. As shown, we use double quotes to tell the compiler about a string literal (although these quotes do not become part of the string itself).



Exercise: *Try creating a project and entering the source-code above in the correct place. Ensure that you can see the result shown in the screenshot above. Try using different values of the* `a` *and* `b` *variables and observe the resulting output.*

If you get no output at all, first check that the program has compiled correctly. If so, the problems view (at the bottom of the screen, by default) should be empty of warnings or errors. Next, ensure that the code is uploaded and debugging correctly and that you have correctly ended any previous debugging sessions (using the red square 'Terminate' button in the debug view). Finally, ensure that you did not miss the '`\r\n`*' at the end of the string literal. Without these escape codes the output buffer will not be sent to the UART.*

Entering escape codes

In addition to the usual alpha-numeric characters, any string may contain control codes that are not always visible, but have some effect when printed. Literals that start and end in quotes must also have a way to contain the quote character without ending the string. Both of these problems are solved using the backslash ('\') character, which is followed by another character that indicates the desired effect:

Control Code	Outputs
`\n`	*Newline*
`\r`	*Carriage return*
`\t`	*Horizontal tab*
`\v`	*Vertical tab*
`\b`	*Backspace*
`\e`	*Escape*
`\"`	*Double quote*
`\'`	*Single quote*
`\a`	*Bell (alert)*
`\f`	*Form-feed*
`\\`	*Backslash*
`\?`	*Question mark*

Note that some of these so-called *escape codes* are either obsolete (such as the form-feed used for printer-based terminals), or will not work in our embedded console — the '`\b`' code may produce a backspace, but in some consoles may not have a visible effect at all.

When we use `printf` to write to a screen or serial port, the string is not sent immediately. Instead, the output is stored in a buffer and only normally sent when newline or carriage return escape codes are encountered. For this reason we must have '`\r\n`' at the end of our string, or the output will never be seen.

Exercise: *Try some of these for yourself! Alter the string to produce different effects.*

Entering format specifiers

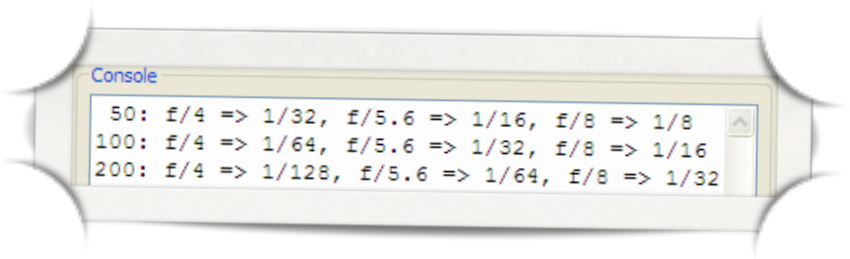
The above escape codes should work (with differing degrees of usefulness) with any string variable. In addition, the `printf` function's format string may contain *format specifiers*. For each variable that we wish to print, there must be one format specifier in the format string — no more and no less! If we provide fewer format specifiers in the string than we provide as parameters, the last values in the list will not be printed.

If we provide *more* format specifiers, then the results will be highly unpredictable. On desktop systems, this can result in the program crashing; on an embedded system, this can result in less predictable behaviour, but it often simply prints a random value.

Format specifiers always begin with a percent ('`%`') character, which is followed by a series of other characters that represent the type of the variable to display, as well as the various formatting options. If we wanted to actually print a percent character, we can double it up (as with the backslash in escape codes) by writing '`%%`' instead. The order of specifiers in the format string must match the order of the variables passed to `printf`. Here are the specifiers for the integer types:

Format Specifier	*Type*
%hhd / %hhu	int8_t / uint8_t
%hd / %hu	int16_t / uint16_t
%ld / %lu	int32_t / uint32_t
%lld / %llu	int64_t / uint64_t

Exercise: *Many cameras allow the user to control the aperture (the size of the hole inside the lens that controls the amount of light allowed through). When this setting is altered, the camera must change the shutter speed in order to maintain the correct exposure for a given light level. Typical shutter speed values (in seconds) for given light levels and apertures are as follows:*



Aperture	*Light Level 50*	*Light Level 100*	*Light Level 200*
f/4	1/32	1/64	1/128
f/5.6	1/16	1/32	1/64
f/8	1/8	1/16	1/32

Write a function to calculate and print the shutter speeds for all three possible aperture settings above, for any given light level (using only integer variable types). Call this function from the flashing LED task several times, each with a different value for the light level. Your function should match the following declaration:

```
void Print_Shutter_Speeds(const uint32_t light_level);
```

*This function should only output **one** line as shown in the screenshot above. You would need to call it three times to get the full output as shown. As a hint, the format string may be as follows: "`%lu: f/4 => 1/%lu, f/5.6 => 1/%lu, f/8 => 1/%lu\r\n`".*

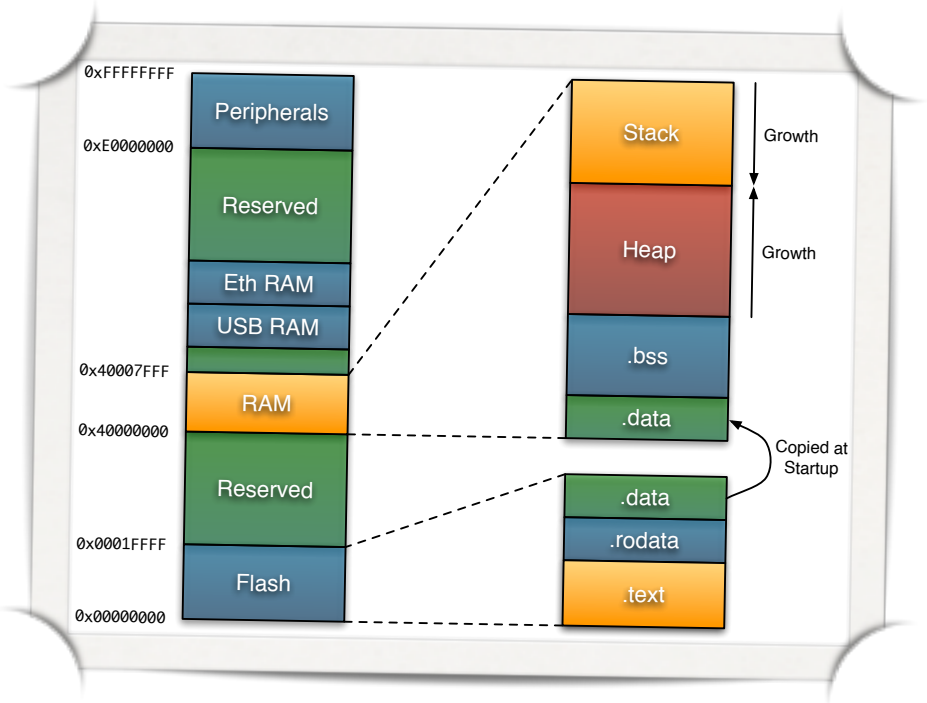
Memory in Depth

In embedded systems, the amount of memory available is often limited, especially when compared to what we can use in desktop systems! In many architectures separate areas of memory are used for code and data, as shown above.

Programming in C gives us a great deal of control over where our variables are stored in memory. For example, the variables that we define outside of any function are called *global* variables, and are assigned to specific locations in data memory by the linker. It would be wasteful to do this for the variables that are only used inside our functions (*local* variables), as they are only needed for relatively short periods of time; the space they use may be shared with the variables from other (currently unused) functions. This shared area of memory is known as the *stack*.

A typical microcontroller memory layout

The diagram to the right shows a typical layout for the address space of a microcontroller (on the left). The code area is often located in flash memory, while the data section must be placed in a writable area, such as RAM.



The right side of this diagram shows the code and data areas in detail, as well as the sections that they often contain.

The compiler places all of our executable code — along with all variables and data — into different sections, which are then placed by the linker. The compiler decides which section to use for a variable, based on both its type, *scope* (local or global) and qualifiers (like '`const`'). Most compilers provide some way of manually specifying which section a given variable or function to be placed in, but there is no such facility in the official C standard.

Executable code is usually placed in the '`.text`' section, along with most numeric literals (which, being small and written directly into the source, become part of the actual object code). Most of the constant data will be placed in the '`.rodata`' section, which is usually then placed into flash memory (because this region is typically larger than the RAM area).

Global variables that are not initialised (or are set to zero) at definition will be placed in the '`.bss`' section, which is located in RAM and cleared by the startup assembly code. If a global variable is initialised to a non-zero value at definition, then it will be placed in the '`.data`' section. As shown in the diagram, this section starts in flash (so that the initial values are not lost), but is then copied to RAM by the startup code.

It should be clear now why we can use variable definitions with assigned values outside of functions. The compiler simply places these values into '`.data`' and the startup script takes care of copying them to the correct location. This is also why we cannot have more complex assignments here — this all happens at compile-time and the compiler cannot be expected to figure out what value a function call (for example) might return.

Temporary local variables and the stack

In the previous diagram we can see that the stack begins at the top of the available RAM, and will then 'grow' downwards. The stack is effectively a Last-In, First-Out (LIFO) data structure; we *push* data onto the top of the stack and then *pop* it back off the top. The top of the stack — the lowest stack address that is currently being used — is stored in the *stack pointer* by the microcontroller.

When the compiler generates executable code, it will usually add some additional code at the beginning and end of each function — this will adjust the stack pointer and make room for the function's local variables. The `Flashing_LED` function calls `printf`, so the stack pointer will first be reduced to make room for any local variables that `Flashing_LED` may have, and then further reduced for `printf`'s own locals (and then even further for any functions that `printf` may call).

Because we have limited memory available — and because the stack grows downwards while all other memory usage grows upwards — it is quite possible for the stack to grow too far and overflow into other memory areas. The result of this often includes unpredictable behaviour and the corruption of variables and data.

Obviously, we want to avoid these problems — we can achieve this by limiting our use of large local variables and expensive functions (such as `printf`), as well as by being careful with (or avoiding altogether) expensive techniques such as recursion. The question is, how do we even know if we are coming close to overflowing the stack?

RapidiTTy Lite's simulator is capable of measuring the stack usage at run-time as part of its timing analysis functionality; simply select the 'Gather timing data from simulation' link from the project settings editor and we should see (on the 'Timing Details' page) something like the result in the screenshot to the right.

Here we can clearly see that we are using a maximum of `1800` bytes of the user stack memory. Stack usage may also be determined from a static analysis of the object code, which can be seen in the Analytics page of the Project Settings editor. Unfortunately, the `printf` function makes use of recursion (when a function calls itself), so we cannot tell from a static analysis exactly how much stack memory might be required.

Creating and Using Variable Types

Up until now the only variable types we have considered are integers; we ignored floating point, enumeration and structure types, as well as array types and the `static` qualifier.

Floating point

Not everything can be represented accurately in integer form, and it is for these situations that we have the floating point types — `float`, `double` and `long double`[2] — which are used to hold real numbers in a binary form. Full details are beyond the scope of this document, for now all we need is a brief summary:

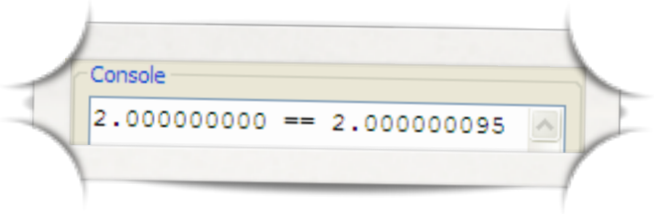
Type	*Size*	*Largest negative*	*Smallest non-zero negative*	*Smallest non-zero positive*	*Largest positive*
`float`	*32 bits*	-3.4×10^{38}	-3.4×10^{-38}	$+3.4\times10^{-38}$	$+3.4\times10^{38}$
`double`	*64 bits*	-1.7×10^{308}	-1.7×10^{-308}	$+1.7\times10^{-308}$	$+1.7\times10^{308}$
`long double`	*80 bits*	-3.4×10^{4932}	-3.4×10^{-4932}	$+3.4\times10^{-4932}$	$+3.4\times10^{4932}$

[2] Please note that `printf` will print the `double` (e.g. with '`%f`') and `long double` (e.g. '`%Lf`') types, but not `float`.

The most important thing to remember about any floating point type is that there is a limit on their accuracy. The *single-precision* `float` type, for example, cannot even accurately represent simple numbers such as '`0.1`' or '`0.2`'.

Consequently, we must be careful when comparing any two floating point values — the equality and inequality operators ('==' and '`!=`') should ***never*** be used! Instead, we must check that the difference between the two values is within an acceptable margin of error.

Exercise: *Try this code inside a task:*




```
// Neither are accurate.
const float first = 0.7;
const float second = 1.2;

// The printf function won't handle floats.
const double third = first + second + 0.1;

// '.9' prints 9 decimal places (6 is default).
printf("2.000000000 == %.9f \r\n", third);
```

Next, try changing the variable types until the two printed numbers are equal.

It might seem as if we should always use the maximum precision possible in calculations, to avoid such errors. Unfortunately, there is a significant cost involved in higher precision. First, the memory usage is significantly higher — each variable that you change from `float` to `double` will double in size (which should be obvious from the name).

More important is the fact that not all embedded processors have hardware support for floating point. In this case, when the compiler encounters the use of floating point types in your code it will insert function calls wherever they are found in an expression. These functions provide *emulation* for the operation, which can be very expensive in terms of execution time. Also, some embedded devices may provide support for single precision floating point in hardware, but rely on emulation for the `double` and `long double` types.

Fixed point

For the reasons mentioned above, using floating point types can be costly as they can result in large increases to both execution time and code size, sometimes to levels that we may find unacceptable. In this case, it is often preferable to avoid floating point altogether, as we did with the camera exercise earlier. This is not always easy — we might need more precision than we can normally get from an integer type.

One alternative is to use *fixed point*. This is not a supported type in C, but uses integer types to approximate real numbers by sacrificing some range. Essentially, this is the same as performing calculations in integer microsecond units, instead of floating point seconds.

For example, we might use a 32-bit unsigned integer (the `uint32_t` type) and use only the top 16 bits for integer results. We can use simple bit-shifting to adjust an integer up into this range, and then back again — the result will still be an integer, but the intermediate calculations will benefit from a degree of increased precision.

Exercise: *Try the following code inside a task:*

```
// Pure integer attempt at (3/2)*5.
const uint32_t a1 = 3;
const uint32_t a2 = a1 / 2;
const uint32_t a3 = a2 * 5;
printf("a3 = %lu \r\n", a3);

// Binary fixed point attempt at (3/2)*5.
const uint32_t b1 = 3 << 16;
const uint32_t b2 = b1 / 2;
const uint32_t b3 = (b2 * 5) >> 16;
printf("b3 = %lu \r\n", b3);
```

Note the relative accuracy of the results, compared to the perfect answer of '`b3 = 7.5`'.

An alternative to bit-shifting — which provides *binary* fixed point — is to use *decimal* fixed point. This involves scaling the integer by a power of ten, using multiplication and division to get back to the original integer. With decimal fixed point, we can also use the modulo operator to get the remainder of the division and provide the fractional part of the value separately. This approach is identical to performing calculations in a smaller unit (such as the microseconds and seconds example used previously).

Bit shifting is a very simple operation for embedded hardware, and is usually completed in a single cycle of the processor. Multiplication — even of integers — will take considerably longer on most microcontrollers. This makes the decimal form slower than binary, but it has the advantage that it can represent exact fractional powers of ten (such as `0.1` and `0.2`) where the binary form cannot.

Exercise: *Add to the code from the above exercise, to use a decimal fixed point approach with a scaling factor of ten. This time, your code should be able to print the exact answer of '`7.5`'. Hint: you will need a separate variable to store the fractional part when scaling the answer back down after the calculation.*

Exercise: *The standard library implementation that comes with RapidiTTy Lite was designed for embedded systems, and so contains additional non-standard functionality for this purpose. One of these is `iprintf`, which is the same as `printf` but which only supports integer variables. Try replacing `printf` with it in your code and observe the change in code size when you recompile. The code and data sizes are shown in the meters at the bottom-right of the screen.*

Creating and using enumerations

An *enumeration* (or enumerated type) is a special data type consisting entirely of a set of named constants, each of which is associated with an integer number. Defining a new enumerated type is done using the 'enum' keyword, an *optional* name and a block with only the list of names (and optionally their value). If a value is not given, the compiler will automatically assign a value one higher than the preceding constant (starting at zero). Here are some examples:

```
// Anonymous enum. No = 0, Yes = 1 after this.
enum { No, Yes };

// Named enum. False = 0, True = 5 after this.
enum Boolean { False, True = 5 };
```

In the example, notice the semi-colon at the end of the enum definition. This is required when defining a new type, because we have the option of defining a new variable of the type at the same time. The following example defines the enumeration 'Imperial' and a variable named 'size', in addition to the constants 'Inch', 'Foot' and 'Yard':

```
// Define the enum and a variable at the same time.
enum Imperial
{                           // Constants:
    Inch = 1,               // Inch = 1
    Foot = 12 * Inch,       // Foot = 12
    Yard = 3 * Foot         // Yard = 36
} size;
```

If we want to use an enumerated type to define a new variable outside of the definition, we can do so as follows. Note that we need the keyword 'enum' here as well, to inform the compiler that Boolean is an enumerated type.

```
// Using the enum to define a variable.
enum Boolean boolean_variable = False;
```

A variable of an enumerated type is simply an integer — any integer value may be assigned into an enumeration variable and any enumeration constant may be assigned to an integer variable (provided that the integer is large enough to hold it).

The benefit of an enumerated type over a normal integer is that it allows us to define a set of constants and specify that a given variable is *likely* to be equal to one of them. This specification can be very useful when, for example, we are writing functions for a library that will be used by other developers. If we have an enumeration variable as a function's parameter, it is obvious that we should pass one of the constants from the enumeration; when used appropriately, this can take us a long way toward making our code self-documenting.

Consider which of these you would prefer to come across in an undocumented library:

```
uint32_t To_Micrometers(uint32_t value, uint32_t unit);
uint32_t To_Micrometers(uint32_t value, enum Imperial unit);
```

In the second option it is clear which values we can pass, even without further information. This is extremely useful — particularly in a large project where many developers are trying to work together — so we should look for ways to get a similar effect for the other types.

Creating new type aliases

The type name '`uint32_t`' is not very descriptive — it tells us how the type will be used in hardware, but nothing about what variables of this type will represent. We can create new names for existing types using a '`typedef`' declaration.

```
// Create a new alias for uint32_t.
typedef uint32_t tyre_pressure;

// Both of these are effectively uint32_t.
tyre_pressure front_wheel_psi = 30;
tyre_pressure rear_wheel_psi = 28;
```

As we can see from the above example, a new type name ('`tyre_pressure`') can tell us far more about the purpose and meaning of the variables we define from it. What's important here is that using this new type is no different from using `uint32_t` — the generated code would be identical and the compiler places no new restrictions on it. A `uint32_t` variable can be assigned into a `tyre_pressure` variable (and vice versa) without conversion.

Another key use of type aliasing is to make life easier when working with enumerated types (as well as structures and unions, which we will get to later). Previously, when we declared new variables of an enumerated type we had to use the `enum` keyword. This gets old quickly, but is easy to avoid:

```
typedef enum Imperial imperial;

// Now we can create imperial variables:
imperial unit = Foot;

// Which will be identical to doing this:
enum Imperial unit = Foot;
```

This technique brings our own types in line with the compiler's built-in types, allowing us to use both in exactly the same way. We can also do this in the enumeration's definition, in which case there is no need to provide a name for it — we can just make the enumeration anonymous and use the alias instead. The following code demonstrates this approach, but note that the enumerated type's definition is *inside* the `typedef` declaration.

```
typedef enum
{
    Inch = 1,
    Foot = 12 * Inch,
    Yard = 3 * Foot
} imperial;
```

Notice that 'imperial' is ***not*** a variable but the alias of our newly created anonymous enumeration. This is because the 'imperial' symbol is part of the typedef declaration and not the enum definition.

Exercise: *Write a full implementation of the To_Micrometers function. Use it in a task to carry out a number of different conversions and print the results. (Note: 1 inch is 25400 µm).*

Exercise: *Try writing a 'metric' enumeration (starting at centimeters and increasing from there) and the corresponding 'To_Inches' function.*

Creating compound types with structures

A *structure* is a logical grouping of related variables that are combined to create a single compound type, which we can then use to declare or define new variables. Here's a simple example:

```
struct Coordinates
{
    uint32_t x;
    uint32_t y;
};

// Creating a variable from a struct:
struct Coordinates screen_coords;
```

By defining the screen_coords variable, we are actually creating two variables of type uint32_t instead. Using a structure allows us to group these related variables together and treat them as a single entity, such as when passing them to a function as shown below. This is a lot more convenient than using separate variables, especially for larger structures.

```
// Note that typedef works as with enums
typedef struct Coordinates coords;
void Print_Line(coords start, coords end);
```

Note the use of typedef to make life easier — just as with enumerations, if we didn't use it here we would be forced to repeat the 'struct' keyword each time.[3] The Print_Line function effectively takes four uint32_t parameters, wrapped up into structures.

[3] And just as with enumerations, we could have used the typedef with an anonymous structure to save some typing.

Inside `Print_Line` we can access the variables in each struct with the dot operator ('`.`').

```
void Print_Line(coords start, coords end)
{
    iprintf("(%lu, %lu) => (%lu, %lu) \r\n",
            start.x, start.y, end.x, end.y);
}
```

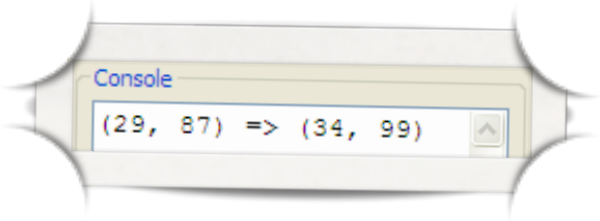
As of the C99 standard, there are two ways to create and initialise variables of our new structure type. We can simply define the variable as normal, using the dot and assignment operators to provide values manually, or we can use the new *designated initialiser* syntax:

```
// The old way, pre-C99:
coords line_start;
line_start.x = 29;
line_start.y = 87;

// Alternative method:
coords line_start = { 29, 87 };

// The new way, post-C99 (variables in any order):
coords line_end = { .y = 99, .x = 34 };

Print_Line(line_start, line_end);
```



Exercise: *Try creating a structure of your own and writing a function to print it out.*

Saving memory with unions

The syntax of a union looks very much like that of a structure, but with the `union` keyword instead of `struct`. While a structure combines multiple variables in memory (usually next to each other), a union will overlay all of its variables onto the *same* area of memory. This is typically used when we don't know beforehand exactly which type we will use for a variable, so we instead provide more than one choice (note the use of `typedef`, so we don't have to type '`union`' every time we create a variable):

```
typedef union
{
    uint32_t integer;
    float floating;
} int_or_float_union;
```

We now have a new type — `int_or_float_union` — which can be *either* an integer *or* a floating point value. Both variables will occupy the same 32-bit area of memory, so it's up to us to ensure that we only use the correct variable. This is usually done by keeping a separate variable (maybe even an enumeration) to specify exactly which of the available types is being used.

We can use a structure to keep all the relevant information together:

```
typedef enum
{
    INTEGER,
    FLOAT
} int_or_float_type;

typedef struct
{
    int_or_float_type type;
    int_or_float_union value;
} int_or_float;
```

Notice the use of capital letters for the enumeration constants? This is not required, but is commonly used to differentiate constants from other variables. It started when constants were most often created with pre-processor — as we will see later, we need to prevent clashes between pre-processor definitions and normal variables, which lead to using all capitals for such constants. This is still the convention used for constants by many, even when not using the pre-processor to create them.

The above structure could also have included the `enum` and `union` inside of it, although this may make the source-code a little more complicated. Now we can create and use a variable of this type and provide either an integer or a floating point value, as required. The following example shows how we might write a function to create an integer version of the union and return a new variable (of our structure's type) from it.

```
int_or_float Create_Int_Value(uint32_t integer)
{
    int_or_float value = { .type = INTEGER,
                           .value.integer = integer };
    return value;
}

// Put this in a task function:
int_or_float data = Create_Int_Value(1234567890);
printf("Value: %lu \r\n", data.value.integer);
```

Exercise: *Try printing the floating point value of `data`, to see why we must keep track of which type it is. Floating point values have a very different binary representation from integers...*

The more observant people reading this may have noticed that adding a second variable with the type of the union will require additional memory. In fact, this approach now uses just as much memory as we would use by having separate integer and `float` variables to begin with! Unions are not limited to single variables — it could have been two different (and large) structure types inside the union, reducing the significance of this overhead.

Casting from one type to another

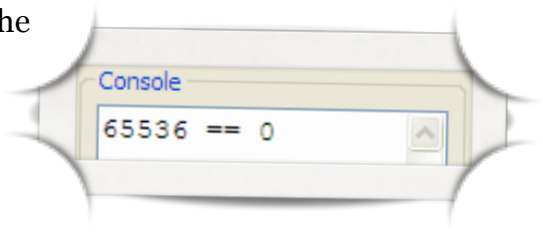
As we can see, nothing prevents us from reading a value from one of the union's variables after we have written to the other. This is *undefined behaviour* — the C standard does not guarantee what the result will be and, as such, it may behave differently when used with other compilers, platforms and even optimisation settings. That said, in some cases the behaviour is predictable enough that we can safely use a union to convert between types.[4]

We are allowed to simply assign one variable type to another, if the types are compatible[5] and the receiving type is capable of storing the assigned type completely. We can assign a floating point type to a more precise one, or a small integer type to a large integer type:

```
float a1 = 16.456f;     uint16_t b1 = 275;
double a2 = a1;         uint32_t b2 = b1;
```

If we convert from larger or more precise types to smaller or less precise types, then we have the potential to lose the excess information. The following code puts a number one higher than the maximum into a 16-bit integer, so the result will *probably* overflow back to zero.

```
uint32_t a = 65536;
uint16_t b = a;
printf("65536 == %hu \r\n", b);
```



Literals and expressions (even sub-expressions) have types as well — the '`f`' in '`16.456f`' above specifies that the literal is a `float`, not a `double` (which is the default for floating point). The result of the expression '`-28 + 12`' will probably have the type `int32_t`.

We can change the type of an expression by *casting* — just write the name of the required type in parentheses and put it before the expression whose type we want to change.

```
uint32_t a = (uint16_t) 65536;
printf("65536 == %lu \r\n", a);
```

Despite the fact that the only variable is a 32-bit integer, the value still overflows because of the cast to `uint16_t`!

Exercise: *Casting is often used to turn an integer into a floating point value to carry out an operation. Try casting `data.value.integer` from the previous section to a double before dividing it by `2.8` and passing it to `printf`, all without using any additional variables.*

---

[4] This is mostly the case when dealing with integer types and arrays, although the code will not be portable across the different microcontroller architectures.

[5] Types are compatible if they use the same internal representation — all integers are compatible with each other and all floating point types are compatible with each other, but integer types are *not* compatible with floating point types.

Arrays of types

An *array* is essentially a construct that allows us to group many variables together into one. They are conceptually similar to structures, except that the variables (or *elements*) of an array will all be of the same type and are accessed by numerical index, instead of by name.[6] The syntax for declaring an array is very different from that of a structure:

```
uint32_t two_integers[2];
```

We can assign values to individual elements of the array using the indexing operator ('`[ ]`'), or use initialisers to fill it directly in the definition. Once again, the latest standard (C99) provides some additional functionality in the form of designated initialisers.[7]

```
// Assigning individual values by index.
two_integers[0] = 27;
two_integers[1] = 52;

// Assigning values at definition.
uint32_t three_integers[] = { 1, 5, 2 };

// Assigning one value, the new C99 way.
uint32_t four_integers[4] = { [2] = 12 };
```

The first example shows the general usage of arrays — accessing elements by *indexing*. In this way we can use array elements in expressions, as if they were separate variables, by providing a zero-based index value. A significant benefit of arrays is that we can then use another variable for the index value.

Arrays can have more than one dimension. The syntax becomes a little more complicated, but we still have the same three choices that we had in the previous example (the first is omitted here):

```
// Two dimensional array, with initialisers.
uint32_t array_2d[][3] = { { 00, 01, 02 },
                           { 10, 11, 12 },
                           { 20, 21, 22 } };
iprintf("array_2d[1][2] == %lu \r\n", array_2d[1][2]);

// Three dimensional array, with designated initialisers.
uint32_t array_3d[7][3][2] = { [1][2][1] = 121,
                               [5][1][0] = 510 };
iprintf("array_3d[1][2][1] = %lu\r\n", array_3d[1][2][1]);
```

Exercise: *Try printing some of the other elements in `array_3d`, to see how they have been initialised.*

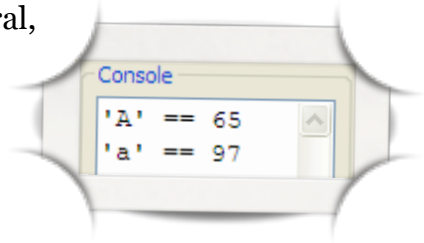
---

[6] The elements of an array will be consecutive in memory — this is also likely for a structure, but there is no guarantee.

[7] Note that we do not have to give a size for the first dimension of *fully* initialised arrays.

Characters and strings

We have been using string literals for quite some time now, but we have yet to cover strings as a variable type. Essentially, we have a type (`char`) dedicated to representing a single character. The `char` type is simply an integer — often but not always eight bits in size — so we can assign any integer to it. There is also a character literal, consisting of one character surrounded by single-quotes.




```
char ch = 'A';
iprintf("'%c' == %hhu \r\n", ch, ch);
iprintf("'a' == %hhu \r\n", 'a');
```

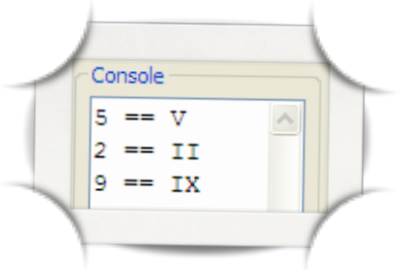
Characters are (by default) represented as integers with their equivalent ASCII codes. We can see above that the character '`A`' is equivalent to the number `65`, and that there is a separate code for the upper and lower cases. Also, note the '`%c`' format specifier, which tells the `printf` family of functions to interpret the integer as a character.

In C, a string is merely an array of characters, which ends with the number zero. There is also a '`%s`' format specifier, which allows us to output entire string variables in one go. Notice here that the initialiser is slightly different from that of normal arrays:

```
char str[] = "Mary had a little lamb";
// Same as = { 'M', 'a', 'r', ..., 'm', 'b', 0 };
iprintf("%s, its fleece was white as snow\r\n", str);
```

Note that we did not specify the size of the array and, once again, the compiler figured it out for us. Actually, the compiler also adds a terminating zero to the end of every string literal so that we don't have to. There are some situations where we are manipulating the array contents and must ensure that the zero is added manually — if it isn't there, `printf` and other string functions may continue printing gibberish until they reach one!

Exercise: *In Roman numerals, values one to ten are given as sequences of characters, as shown in the table below. Write a function (with the signature below) that will take as a parameter a number between one and ten and then print the appropriate roman numeral for it. Try it out by calling it with a several different values from a task.*




```
void Print_Numerals(const uint8_t number);
```

	1	2	3	4	5	6	7	8	9	10
Roman Numerals	I	II	III	IV	V	VI	VII	VIII	IX	X

Scoping rules for variables

Every variable has a *scope*, outside of which it cannot be accessed. A variable that has been defined outside of any function is said to be a *global* variable and is at the global scope, so it can be used anywhere — in any function, statement or expression — *after* its declaration. We should note that a variable can only be defined once, but may be declared many times.

The use of global variables is considered to be bad practice, as they may be depended on by code anywhere in the system, but can also be modified from anywhere. There are times when global variables (sometimes simply called *globals*) can be used — such as to facilitate communication between tasks or threads that cannot otherwise share data; this becomes dangerous only if the given tasks and threads have the ability to pre-empt each other, but that is beyond the scope of this document.

The alternative to the use of global variables is to use local variables instead (or function parameters, which amounts to the same thing). These have the advantage of only being accessible from within the function itself, and so do not suffer from the disadvantages of globals. However, they are stored on the stack, which means that we must be careful not to cause an overflow (as we discussed earlier).

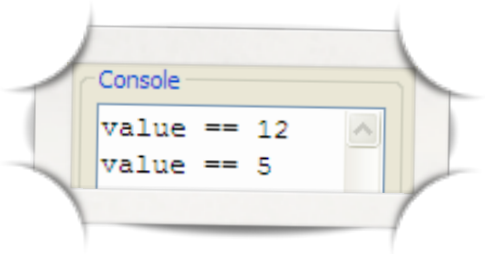
In general, local scopes exist within blocks (statements surrounded by '{' and '}'). As we are free to add our own blocks even within existing functions, we can have even narrower scopes to work with. If we declare a variable within a local scope that has the same name as variables from wider scopes, only the variable at the narrowest (most local) scope will be accessible. The other variables are said to be *shadowed* by it — some compilers will give warnings if this happens, as it can be a source of errors.

```
// At global scope.
uint32_t value = 5;

void Flashing_LED(void)
{
    // Now at local scope.
    uint32_t data = value + 2;

    // Create an even more local scope.
    {
        // This value shadows the global one!
        uint32_t value = data + 5;
        iprintf("value == %lu \r\n", value);
    }

    iprintf("value == %lu \r\n", value);
}
```



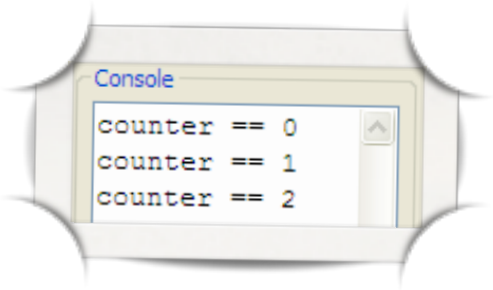
Adjusting a variables scope with static

Local variables are stored on the stack, which is cleaned up when a function returns — all variables that are local to a function lose their values when this happens. Suppose we want to use a variable in just one function, but have it retain its value between calls. We *could* make that variable global, but using globals is considered bad practice.

An alternative is provided by the `static` keyword. This has quite a large number of uses, but one of the main ones is to force the linker to provide permanent storage for a locally scoped variable.[8]

```
void Increment_And_Print(void)
{
    static uint32_t counter = 0;
    iprintf("counter == %lu \r\n", counter++);
}

void Flashing_LED(void)
{
    Increment_And_Print();
    Increment_And_Print();
    Increment_And_Print();
}
```



Notice in the above example that the initialiser will only set the static variable to zero the first time the function is called.

When we apply the `static` keyword to global variables and functions, it will limit their scope to that of the current translation unit. This can be useful, as truly global variables must have unique names — the names of static globals only have to be unique within their translation unit. Such 'static global' variables are often said to be at *file* scope.

Exercise: *Call the following function several times — guess what it will print each time!*

```
static uint32_t data = 2;

void Print_Some_Data(void)
{
    {
        static uint32_t data = 4;
        data *= 2;
        iprintf("data == %lu \r\n", data);
    }
    data *= 2;
    iprintf("data == %lu \r\n", data);
}
```

[8] It's a long running (but accurate) joke that every new C standard brings with it a new meaning for the `static` keyword.

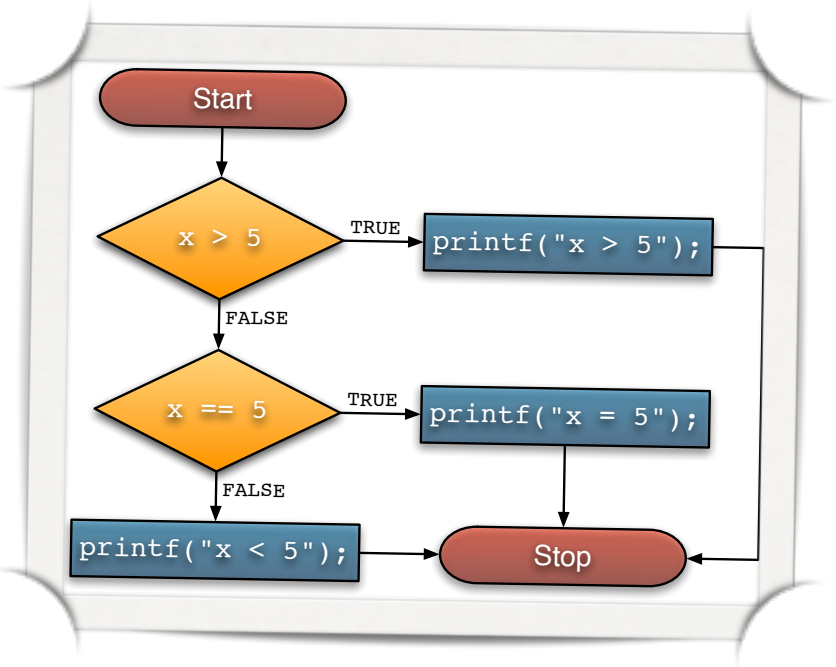
Controlling the Order of Execution

Running a C program can be thought of as executing a series of statements, one by one. We have already seen how we can use a function call to divert the order of execution into a separate set of instructions and back again (when the function returns); here we will look at a number of other statements with similar effects, starting with the '`if`' statement.

Conditional branching with the if statement

A function call is an *unconditional branch* in a program — a diversion to a different part of the program that happens every time the branching statement executes. Sometimes it is useful to perform a branch only when a *condition* can be met.

Look at the flowchart to the right. Here we have a variable ('`x`'), two conditions and three statements to be run. Notice that conditions are the same as expressions in C, with the exception that they always have *boolean* values.



A boolean value is one that can only be either true or false (often defined as '`TRUE`' and '`FALSE`'). An integer is false if it is equal to zero, and true if it has *any* other value.

Exercise: *Put the code below in a function and verify that it is same as the flowchart.*

```
if (x > 5)
    printf("x > 5");
else if (x == 5)
    printf("x = 5");
else
    printf("x < 5");
```

Try adding another call to `printf` in the first branch and note that the compiler gives an error. To use more than one statement in a branch, try placing them in a block with '{' and '}'.

Exercise: *Write a function taking one parameter and prints 'correct' if it equals `17`.*

Taking shortcuts with the ternary and comma operators

Suppose we want to assign different values to a variables based on the value of another. With our newfound knowledge of the if statement, this is a simple matter:

```
uint32_t a = 6;  // Try changing a to 15...
uint32_t b = 0;

if (a > 10)
{
    b = 5;
}
else
{
    b = 15;
}
```

Note that it is often considered good practice to include a block for each branch, even when it's not necessary; this is done to prevent the mistakes that can occur if we add an extra statement at the end of a branch. Unfortunately, it makes the code very long for such a trivial operation — one possible solution is to use some shortcuts to shrink things down:

```
uint32_t a = 6, b = a > 10 ? 5 : 15;
```

First, notice that we have put two variables into one definition, separated by the comma operator ('`,`'), which indicates that they both share the same type. The assignment to '`b`' is done with the ternary operator ('`?:`') — the part before the question mark is the condition; the part between the question mark and the colon is the result if the condition is true; and the final part is the result if the condition is false. Many developers believe both of these shortcuts to be bad practice, as they can be hard to read and so more difficult to maintain.

Unconditional branching with goto

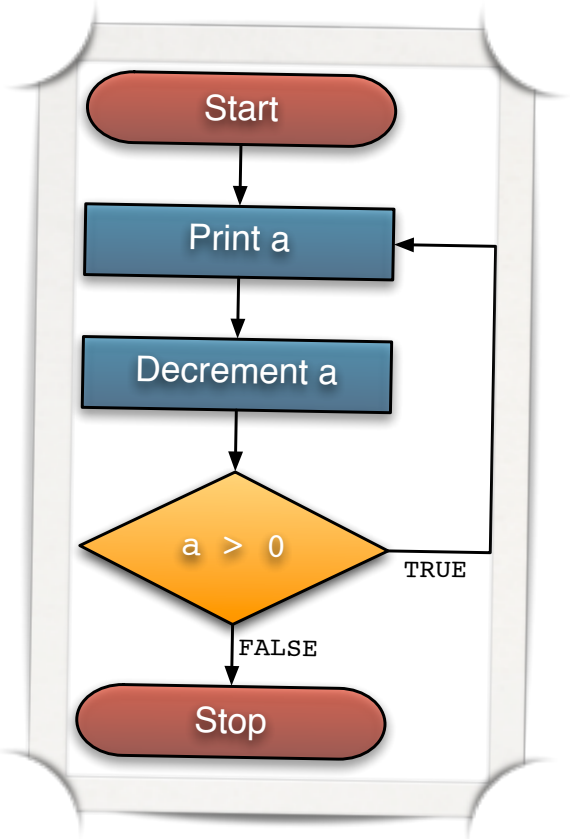
While function calls are the most widely used of the unconditional branches, the *goto* statement serves the same purpose. The goto statement has two parts. First, we must have a *label* as the target location — this can be prefixed to any statement within a function and consists of a name followed by a colon ('`:`'). To branch to a label, we can use the `goto` keyword, followed by the name of the label to branch to.

```
decrement_and_print:
printf("a is now %lu \r\n", a);
if (--a > 0)
    goto decrement_and_print;
```

By adding a condition of our own, we can control the `goto` and use it to repeat the `printf` several times. This repetition (called *looping*) is useful enough to have its own statements.

A comparison of looping constructs

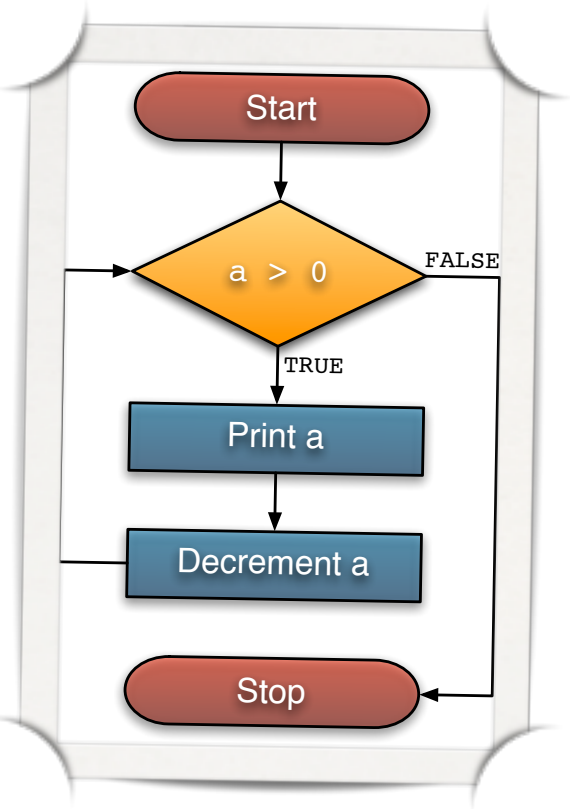
The goto statement is a very powerful, but very crude construct — developers avoid it simply as it does not take into consideration what is at the target label, which may be in the middle of an `if` statement's block (for example). Fortunately, C provides us with three other language constructs that are entirely dedicated to looping.



All loops have three basic elements — one branch used to provide the actual repetition, a condition used as an exit clause and a block of code that is repeated. The thing that separates the looping constructs is the time at which the condition is tested. Here is the goto-based code (shown in the flowchart to the right) rewritten as a `do` statement:

```
do
{
    printf("a is %lu \r\n", a);
} while (--a > 0);
```

This is better than the `goto` version — now we can clearly see which code will be repeated, and where the branch will go. The problem with both of these is that one line is printed no matter what the initial value of `a` is. In fact, an initial value of zero will cause problems!



The alternative is to test the condition before executing the block of code, which we can do with the `while` statement — which behaves as shown in the flowchart to the right.

```
while (a > 0)
{
    printf("a is %lu \r\n", a);
    --a;
}
```

Exercise: *These examples behave differently — try both with several values for 'a'.*

Taking another shortcut with the for statement

Looping through a range of values — to access elements of an array, as one example — is a common task. This typically requires a variable to use as the loop counter (and as an array index), as well as the loop itself. With a `while` loop, it might like as follows.

```
uint32_t values[20];

uint32_t i = 0;
while (i < 20)
{
    values[i] = i;
    iprintf("values[%lu] = %lu\r\n", i, i);
    i++;
}
```

The problem with the above code is that there is a lot of it, much of which must be repeated every time we want to do this — such as the loop counter definition and increment. To make life easier, C has the `for` statement; this has three parts, separated by semi-colons: first is variable declarations, then the loop condition and finally the increment operation.

```
for (uint32_t i = 0; i < 20; i++)
{
    values[i] = i;
    iprintf("values[%lu] = %lu\r\n", i, i);
}
```

The declaration part of the `for` statement can be used to define multiple variables of a given type using the comma operator, as described previously. One advantage of defining variables here is that they will only be valid inside the scope of the `for` loop's block, which means we don't have to worry about reusing the same name ('`i`') in subsequent statements.

In the condition and increment sections of the for statement, we can use any expression. The condition will be tested at the start of the loop, and the increment is executed at the end. As with `while` loops, the block may be omitted if we only need a single statement.

Exercise: *Create two arrays holding the following values, and write a function that will — using a loop — calculate and print exactly how much money a given sales person brought in (the sales person should be specified as a parameter). Try this with and without using floating point, and test your function by calling it several times from a task.*

	Sales of product 0	*1*	*2*
Sales Person 0	12	2	6
Sales Person 1	24	0	3
Sales Person 2	17	1	7

	Cost
Product 0	14.99
Product 1	29.99
Product 2	19.99

Selecting an alternative with switch and case

The final method of changing the course of execution that we will look at is the `switch` statement. We can use this to select one of a number of alternative sections of code to run based on a given integer value. For example, suppose we passed an enumeration variable to a function and wanted to have it print a string based on the given value — we could do this with `if` and `else` statements:

```
void Print_Length(const uint32_t length,
                  const imperial unit)
{
    if (unit == Inch)
    {
        printf("%lu inches\r\n", length);
    }
    else if (unit == Foot)
    {
        printf("%lu feet\r\n", length);
    }
    else // Ignoring yards for brevity!
    {
        printf("Unknown unit!\r\n", length);
    }
}
```

If we use a switch statement, the above code becomes a little simpler and much cleaner:

```
void Print_Length(const uint32_t length,
                  const imperial unit)
{
    switch (unit)
    {
    case Inch :
        printf("%lu inches\r\n", length);
        break;
    case Foot :
        printf("%lu feet\r\n", length);
        break;
    default :
        printf("Unknown unit!\r\n", length);
    }
}
```

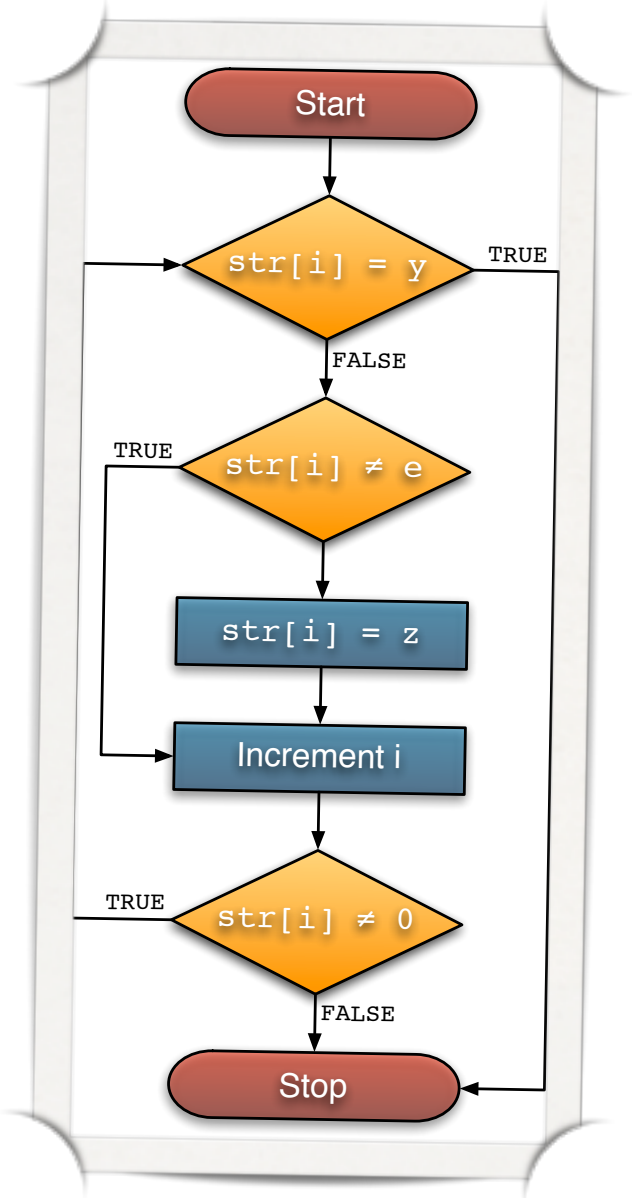
When the `switch` statement is reached it causes a branch to the appropriate `case` label. Note that the execution of a given `case` does not end at then next one; we have to break out of the `switch`. One advantage of this is that — when switching on an enumeration variable — the compiler can usually warn us if we miss a `case`.

Exercise: *Test the function by calling it from a task. Try it after removing a* `break`.

Breaking free and continuing on

In the previous section, we saw that we can use the `break` statement to exit from a `switch`. This can also be used to exit `do`, `for` or `while` loops. In addition, in these loops we can use the `continue` statement to end the current iteration, without exiting the loop.

These statements are often used in loops to provide a degree of additional control. For example, if we have a string of characters and want to replace all the 'e' characters with 'z' up until the first 'y' character, we might try this (also shown in the flowchart):




```
char str[] = "easy as pie";

uint32_t i = 0;
do
{
    if (str[i] == 'y')
        break;
    else if (str[i] != 'e')
        continue;
    str[i] = 'z';
} while (str[++i] != 0);

iprintf("%s\r\n", str);
```

The `continue` and `break` statements are almost always used in this way — triggered by a condition within an `if` statement. Many developers hold that these statements are no different from `goto` and so are bad practice.

Alternatively, the `for` loop can be used:

```
for (uint32_t i = 0; str[i] != 0 && str[i] != 'y'; i++)
{
    if (str[i] == 'e')
        str[i] = 'z';
}
```

Here we can see the use of two sub-expressions in the condition part of the statement. It behaves slightly differently from the previous example, in that it will still work if the given string is empty.

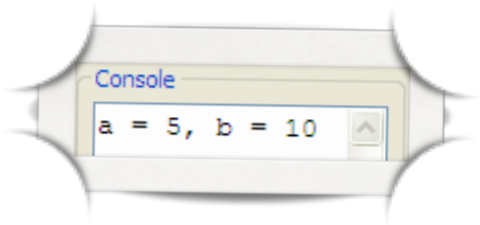
Exercise: *Try implementing the examples above, and also try it using a `while` loop.*

Causing Trouble with Pointers

Suppose we wanted to write a function to swap two integers — a naïve attempt might be to simply pass the two variables to the function and swap them internally:

```
void swap(uint32_t left, uint32_t right)
{
    const uint32_t temp = left;
    left = right;
    right = temp;
}

uint32_t a = 5, b = 10;
swap(a, b);
iprintf("a = %lu, b = %lu\r\n", a, b);
```



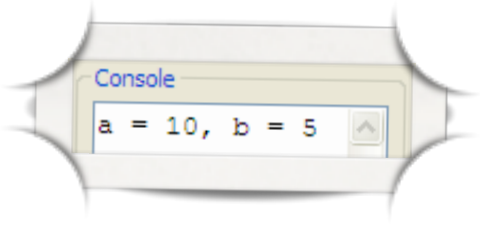
As we can see, this simply doesn't work — the reason is that parameters are passed *by value*. This means that `left` and `right` are actually separate variables, and when we call the function we are *copying* the values of the passed parameters into them. The same goes when returning values — the value or the variable that is returned is copied into a variable in the calling function.

To work around these restrictions, we have two new operators to consider — the address operator ('`&`') and the dereference operator ('`*`'), which are prefix operators. Applying the address operator to a variable will provide the address of the variable in memory, as you might expect. The dereference operator works in the opposite direction, given an address it will provide the content of the memory at that address.

In order to provide an address for the dereference operator, we must be able to store such addresses in variables. These 'address variables' have their own types, which are called *pointers* (they point to another location in memory). The dereference operator is used in a declaration to show that a variable is a pointer, but we must also provide the type of the variable at the memory address that it points to. We can rewrite the swap function:

```
// Two pointers, addresses passed by value
void swap(uint32_t *left, uint32_t *right)
{
    const uint32_t temp = *left;
    // Swap the contents
    *left = *right;
    *right = temp;
}

uint32_t a = 5, b = 10;
swap(&a, &b); // Passing the addresses
iprintf("a = %lu, b = %lu\r\n", a, b);
```



As we can see in the example, `left` and `right` are *still* separate variables that contain a copy of the value that is passed to them. That restriction no longer matters, as the passed values are not the actual `a` and `b` variables, but their addresses — when dereferenced with the '`*`' operator, we are directly modifying `a` and `b` themselves.

Multiple function outputs

Because passing pointers to functions allows us to alter the passed variable directly, it is known as *pass by reference*. This is inaccurate, as parameters are still copied by value but the values are addresses. It can be used to output multiple values from functions.

Normal functions may only return a single value — up until now, if we wanted to return more than one value we might have returned a structure. This can be expensive, as copying an entire structure can take a long time; whereas copying a pointer is usually only four bytes on a typical microcontroller. The same is true for passing structures as parameters — you will often see this done with pointers purely for performance reasons.

The classic example of returning multiple values from a function is `scanf`. This is the equivalent of `printf`, but focused on retrieving and parsing input values into variables. It is even larger and slower than `printf`, often too large to fit in a small microcontroller... fortunately, we can use the `iscanf` version instead.

```
uint32_t num1 = 0, num2 = 0;
iprintf("Enter two numbers separated by a comma: ");
iscanf("%lu,%lu", &num1, &num2); // Passing addresses!
iprintf("\r\nYou entered %lu and %lu.\r\n", num1, num2);
```

You may have to click on the console window before RapidiTTy allows you to enter a value. Note that `iscanf` will pause until you enter the values it asks for — it won't return until we have entered some data. As we can see from the code, `iscanf` is outputting more than one value without actually returning them; in fact, `iscanf` does return an integer that specifies the number of parameters that were matched in the input.

This is a common pattern in functions that must output multiple values — we pass output through parameters and use the return value as an *error code*. Whoever calls the function must then check that it was successful, which is most often signaled by returning zero.

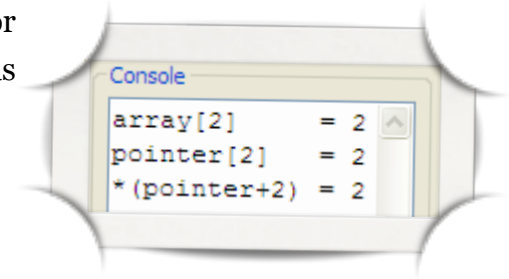
Exercise: *Try the above example and enter something invalid (i.e. not an integer). Now alter the program in order to fix the problem, reporting any invalid input with `iprintf`.*

Exercise: *Remember your `To_Micrometers` function? Try changing it to output its result as a parameter and return an error code to indicate that an invalid unit was given. Test it by calling it from a task function, with both correct and invalid inputs.*

Array parameters and pointer arithmetic

Arrays and pointers are far more similar than they appear. Usually, an array is a named variable that represents a number of elements in a block of memory, but when passed to a function all of that changes. Because arrays do not contain information about their size — and because we want to be able to pass different sized arrays to a function — the compiler cannot generate code to copy the entire original array into the parameter. As a result, it treats the array parameter as a pointer to the first element of the array.

This sounds like it would be confusing in the source-code, treating an array differently if used as a parameter, but that is not the case. In addition to the address and dereference operators, pointers make use of the indexing operator ('`[ ]`'), in exactly the same way as arrays. This works as follows:




```
uint32_t array[] = { 0, 1, 2, 3 };
uint32_t *pointer = array;

iprintf("array[2]     = %lu\r\n", array[2]);
iprintf("pointer[2]   = %lu\r\n", pointer[2]);
iprintf("*(pointer+2) = %lu\r\n", *(pointer+2));
```

There are a number of new concepts introduced in the above example. First, note that even when not used as a parameter, `array` is treated as if it were a pointer — we do not need the address operator when assigning it to `pointer`. Second, the indexing operator returns the *value* of the element and not the address (i.e. it does the dereferencing for us). Finally, we can do what is called *pointer arithmetic* — that is, directly altering the address contained in the pointer variable using the standard arithmetic operators.

Note that pointer arithmetic depends on the type of the variable that is being pointed to. In this case, adding two to the pointer adds eight to the address, because the compiler knows that it points to 32-bit values (which are 4 bytes each in size). If we changed the type of the pointer by casting (intentionally or otherwise), this can cause serious problems.

The problems caused by pointer arithmetic can be severe, which has led to most newer languages banning its use altogether. Generally speaking, it is probably best to avoid the confusion of pointer arithmetic wherever possible — use the indexing operator and please ensure that you do not try and access an element beyond the limits of the variable or array that is being pointed to.

This means that when we need to pass an array or pointer to allocated memory (which we discuss later) to a function, we should also be sure to pass the size as a separate parameter. The function can then correctly limit itself to the bounds of the memory being pointed to.

Revisiting strings of characters

If arrays and pointers are so similar — virtually identical when used as parameters — then this also applies to strings, which are merely arrays of characters. The standard library has an `strlen` function, for example, which returns the length of a zero-terminated string. This function is declared in the `string.h` file of the standard library (that can be included with '`#include <string.h>`'), but we can also write our own version:

```
// Could also be '(const char str[])'
size_t strlen(const char *str)
{
    const char *s;
    for (s = str; *s; ++s);
    return s - str;
}
```

Notice that, not only can a string be seen as a '`const char *`' type, but we can also use the array syntax without affecting the behaviour of this function. Also, note the `for` loop that uses pointer arithmetic to place the '`s`' pointer at the index where '`*s`' is zero (recall that zero is false and every other value equates to true). The loop has no statement attached — only the *null* statement ('`;`' on its own).

Finally, the return statement then uses more pointer arithmetic to calculate the distance between the start and end of the string — this value will be in bytes, but *only* because the type of the variable being pointed to is '`char`' (which is usually one byte in size).

Notice in this code that '`s`' is declared as constant, but is then modified in the `for` loop. We can do this because the `const` in this case applies only to the underlying type of the variable being pointed to and not the pointer itself. We will discuss this in depth later on.

There is one significant difference between using arrays and pointers to represent strings — string literals are usually stored in the code section, but pointers and arrays will be stored on the stack when created locally. For an array, the code must copy the entire string literal onto the stack; for a pointer, the compiler just copies the address of the literal. The array approach will take up much more stack space, but will allow us to modify the string. The pointer approach will simply point into the code section, which *may* not be modifiable.

```
void Some_Function(void)
{
    // The locals are stack-based, the literals are in .rodata
    char str1[] = "Hello";      // Writable, copied to stack
    const char *str3 = "Hello"; // Points to read-only memory
    // Writable variable, but may point to read-only memory
    char *str2  = "Hello";
}
```

Working with pointers to structures

As mentioned previously, it is common to use a pointer to a structure as a parameter — passing the structure itself can be expensive, depending on how many variables it contains. Accessing single elements of the structure from the point is a matter of first dereferencing, then using the normal dot operator. Alternatively, we can use the arrow operator ('->') as a shortcut.

```
void Print_Line(coords *start, coords *end)
{
    iprintf("(%lu, %lu) => (%lu, %lu)\r\n",
            (*start).x, (*start).y,
            end->x, end->y);
}
```

Remember `Print_Line` and the `coords` structure? This produces the same behaviour as before, but only passes two pointers (eight bytes) instead of four 32-bit integers (16 bytes). The downside of this is that we must now use the address operator when calling it:

```
coords line_start = { .x = 5, .y = 5 };
coords line_end   = { .x = 9, .y = 7 };

Print_Line(&line_start, &line_end);
```

Exercise: *Imagine you want to store employee information for your company. Each employee will have a first name (of up to `20` letters), a surname (of up to `20` letters), an employee number, a house number, a street name (of up to `50` letters) and a town name (of up to `30` letters). Create an appropriate structure to hold this information, and a function to print an element of the structure that is provided as a parameter.*

Exercise: *Create an array to contain the employee information listed in the table below. Write a function taking a string parameter and returning the index of the element whose surname matches the parameter. You may wish to write a string comparison function using a simple loop to test two strings for equality — comparing pointers (or arrays) without a loop will not produce the correct result!*

You should test your function by using `iscanf` to read a surname from the standard input into a character array, and then finding and printing the appropriate entry.

Surname	First name	Employee #	House #	Street	Town
Smith	John	1763235	32	Granby Street	Leicester
Dickens	Charles	187	1	Gad's Hill Place	Higham
MacLeod	Connor	1	7	Tor-an-Eas	Glenfinnan
Holmes	Sherlock	274	221	Baker Street	London

Allocating memory dynamically

So far all of the memory we have used has been either selected by the linker, or allocated dynamically (at run-time) on the stack by the compiler. In addition, we can use functions in the standard library to manually acquire and release memory (dynamically) from the heap. The heap is essentially the leftover space after the linker and stack space have been allocated.

To allocate memory from the heap, we can use the `malloc` function and we can release it with the `free` function. The advantage of manual memory management is that we do not need to guess in advance how much memory we will use. Instead, we can tell `malloc` to allocate exactly the right number of bytes (which it takes as a parameter) and it will return a pointer to the allocated memory. This pointer can (*must*) then be passed to `free` when we are done with the memory. Note that you will also need to '`#include <stdlib.h>`'.

```
void Some_Function(void)
{
    uint32_t static_version[5];
    uint32_t *dynamic_version = malloc(5 * sizeof(uint32_t));
    // ... use the memory here ...
    free(dynamic_version);
}
```

The above example shows two dynamic allocations. The first is `static_version`, which allocates 20 bytes on the stack (5 elements of 4 bytes each). Second is the heap allocated `dynamic_version`, which also allocates 20 bytes but which does so manually.

Note the use of `sizeof` — this is not a function, but is an operator that is converted into the correct size (in bytes) of a type or variable by the compiler. It is very important to use this with `malloc`, to specify the number of elements instead of just the number of bytes.

Basic linked list data structures

Suppose we are using an array to store data and we run out of room. With a static array, there is nothing we can do but handle it as an error condition. However, if the memory we are using was allocated with malloc, then it is possible to allocate a larger area and transfer our data across. The problem with this approach is that, having transferred everything, we must also update all pointers into the old area of memory to the new location.

An alternative is to use a more complex *data structure*. If we have a list of items that we need to store and we want to be able to add and remove elements at runtime, it is possible to do so using a *linked list*. This is essentially a structure that holds the data for a single element, and an additional pointer to a variable of the same structure (effectively a pointer to itself).

Here's an example of what this might look like in practice, using a list of coordinates:

```
typedef struct coordinates_tag
{
    uint32_t x;
    uint32_t y;
    struct coordinates_tag *next;
    struct coordinates_tag *prev;
} coords;
```

Note that, even though we are using `typedef`, we must still use the structure's tag (it's real name) inside it. This is because the alias is not available until the `typedef` statement is complete, after the end of the structure.

We can also see that coords has not only a '`next`' pointer, but also a previous ('`prev`') pointer as well. This makes it what is known as a doubly-linked list, in which we sacrifice some memory (a 4-byte pointer per element) in exchange for faster removal of elements from the middle of the list. Here is an example of how we might add a new element:

```
// A pointer to the 'head' of the list
coords *head = NULL;

coords *Add_Coords(const uint32_t x, const uint32_t y)
{
    coords *new_coords = malloc(1 * sizeof(coords));
    new_coords->x = x;
    new_coords->y = y;
    new_coords->next = head;
    new_coords->prev = NULL;
    head->prev = new_coords;
    head = new_coords;
    return new_coords;
}
```

In other words, the new coordinates are placed on the head of the list and their next pointer is set to point to the previous head. We can also loop through all the coordinates:

```
uint32_t i = 0;
for (coords *curr = head; curr != NULL; curr = curr->next)
{
    iprintf("Coords %lu: (%lu, %lu)\r\n", i++,
            curr->x, curr->y);
}
```

Exercise: *Try adding elements to the list and printing them out.*

Exercise: *Try writing a function to remove an element, then add code to your task that will call it and print the list out afterwards. Don't forget to have it call* `free`*!*

Pointers to functions

Just as we can have pointers to most other types, we can also have pointers to functions. In syntax, these are a little strange — the key to recognising them is that they look exactly like the function's declaration, but with the pointer symbol and an extra set of parentheses:

```
// Each line declares a variable named 'Pointer_To_...'
void (*Pointer_To_Void_Void)(void);
uint32_t (*Pointer_To_Int_Void)(void);
uint32_t (*Pointer_To_Int_Int)(uint32_t);
uint32_t (*Pointer_To_Int_Int_Int)(uint32_t, uint32_t);
```

Although they look just like odd function declarations — without the extra parentheses they *would* declare functions that return pointers — the four lines above actually declare variables that are all pointers to functions. It's much easier in practice to use a `typedef` for these things. In actual usage, they work as we might expect.

```
// Create a new type alias called func_ptr.
typedef void (*func_ptr)(uint32_t);

void Call_Function(func_ptr function)
{
    // Call the function, passing '5'
    (*func_ptr)(5);
}
```

When we actually assign a function to a pointer (or pass it as a function parameter), we do not use the address operator. Just as with arrays, the compiler knows that the function's name can only be assigned to a pointer type and so will automatically assign its address. Here is an example of passing a function as a parameter:

```
void Print_Value(uint32_t value)
{
    iprintf("Value: %lu\r\n", value);
}

void Flashing_LED(void)
{
    Call_Function(Print_Value);
}
```

Exercise: *Look at the `scheduler.h` file in the project — note the task data structure and the function pointer inside. Now open `config.c` and look at the definition of the task array; you should see the `Flashing_LED` task and its specification. The delay and period values specify when the task will start running and how often. The final values are beyond the scope of this document. Now add another task to the task array, with any `void (*)(void)` function!*

Reading and writing complex declarations

Any variable will have one basic type at its core. Beyond this, it can be decorated with a number of other specifiers and qualifiers. We have already seen how t use '`*`' to make a pointer, '`[]`' to make an array and '`()`' to make a function. What we haven't mentioned is that these can be combined to make variables with far more complex types.

```
// Pointer to a pointer to a uint32_t
uint32_t **ptr1;

// Array of 5 pointers to char
char *ptr2[5];

// Pointer to an array of 5 chars
char (*ptr3)[5];

// Array of 5 arrays of 2 pointers to char
char *ptr4[5][2];

// Array of 2 pointers to arrays of 5 chars
char (*ptr5[2])[5];
```

The trick is to start with the name of the variable, and work outwards. Array and function specifiers on the right take precedence, so we keep looking to the right until the reach an unmatched right parenthesis, then we go left until we reach the left parenthesis that matches it. We just repeat this until every part of the declaration is matched.

Let's try a worked example, step-by-step. The following is a complex expression:

```
uint32_t *(*ptr6[10])(int8_t *, float);
```

We start with the name, which in this case is '`ptr4`'. Moving to the right one place, we have a '`[10]`', so `ptr4` is an array of 10 elements. Next there's a '`)`' so we go left, where we get '`*`', making `ptr4` an array of 10 pointers. The parentheses are complete so we move further right and find '`()`' (with parameters inside) making `ptr4` an array of 10 pointers to functions. There is no more to the right, so we move back to the left and finish the rest.

In full, `ptr4` is an array of 10 pointers to functions that take two parameters — one pointer to an `int8_t` and one `float` — and returning a pointer to a `uint32_t`.

The C language was specifically designed so that declarations match expressions almost exactly. The type of an expression's result can always be found as the leftover parts of the declaration. For example, if we have '`ptr2[2]`' in an expression, we know that the result has the type '`char *`'. If we have '`*ptr4[1][2]`', then the result will have the type '`char`'. The expression '`*ptr1`' have the type '`uint32_t *`'. The expression '`**ptr1`' will result in the type '`uint32_t`', and so forth.

Using and Abusing the Pre-Processor

Right back at the beginning, we discussed the build process. Before the compiler even sees any of the source-code, the pre-processor gets to examine *and alter* it. The pre-processor does not actually parse C source-code at all, instead it simply recognises its directives and follows its own rules in producing translation units.

Including and guarding header files

We have already seen the '`#include`' directive several times. All pre-processor directives begin with the '`#`' character and this one in particular will replace the directive with the contents of the given file. There are two basic forms — either the file is specified with '<>' or with quote marks. The former will include one of the system's header files and the latter will include a file from the current directory (or one on the path).

When including header files, it is possible for the header file to include other headers itself. This can lead to circular inclusions, such as when the header '`file1.h`' includes '`file2.h`' and then '`file2.h`' includes '`file1.h`'. This will either be detected by the pre-processor, or simply go on forever.

To get around this, we have '`#define`' and '`#ifdef`' directives. The first will define a new symbol; whenever the pre-processor finds this symbol it will replace it with whatever came after it on the line in which it was defined, if anything. The second allows us to remove sections of source-code if a pre-processor condition is not met. For example:

```
// This goes in file1.h
#ifndef FILE1_H_INCLUDED
#define FILE1_H_INCLUDED

// The rest of file1.h goes here.

#endif

// This goes in file2.h
#ifndef FILE2_H_INCLUDED
#define FILE2_H_INCLUDED

// The rest of file2.h goes here.

#endif
```

This shows how we can use directives to provide *include guards*, to prevent the circular dependency problem. The entire file is placed between `#ifndef` and `#endif`, which will remove the code if the given symbol has already been defined. The first line inside these statements defines the symbol itself, ensuring that the content can only be seen once.

Macro constants and problems

The `#define` directive actually creates a *macro* — a symbol that will be replaced with its content wherever it is seen. This can be used to create zero-cost constants that will simply become literals:

```
#define PI 3.1415926536
```

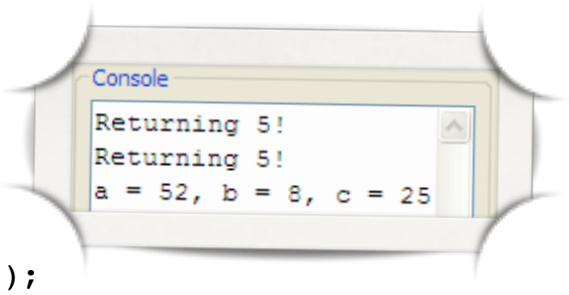
The trouble is that the replacement in carried out by the pre-processor, which has no knowledge of C source-code itself, so this constant will ignore all the usual scoping and naming rules. As well as constants, macros may take simple parameters to be used as if they were functions. These macro-style functions will be very fast, as they have no function call overhead (setting up the stack, etc), but also very difficult to produce safely.

```
#define ADD(x, y) x + y
#define SQUARE(x) x * x

uint32_t Get_Value(void)
{
    iprintf("Returning 5!\r\n");
    return 5;
}

void Flashing_LED(void)
{
    uint32_t a = ADD(2, 5) * 10;
    uint32_t b = SQUARE(2 + 2);
    uint32_t c = SQUARE(Get_Value());

    // This should print 'a = 70, b = 16, c = 25'?
    iprintf("a = %lu, b = %lu, c = %lu\r\n", a, b, c);
}
```



The results are not what we expected — it is clear why if we do the expansion manually:

```
uint32_t a = 2 + 5 * 10;
uint32_t b = 2 + 2 * 2 + 2;
uint32_t c = Get_Value() * Get_Value();
```

The first two problems are easy to fix, simply by adding parentheses around every use of the macro parameters and around the entire macro contents. The third problem — calling a function that has side effects — is much more challenging. Typical solutions include just ignoring the problem or creating a temporary variable inside the macro.

Creating variables in the macro brings more problems — we must also add a block around it or we won't be able to use the macro twice in the same scope. Developers often wrap the contents of macros in `do` statements with `FALSE` as the condition, as this also forces us to use a semi-colon after the end of the macro usage, as with normal C code.

Embedded Considerations

If you reached this point by reading all the previous sections and working through all the examples, then you may be glad to know that we have covered much of the C programming language. Further details can be found in any textbook, but for now we will be looking at some of the more low-level considerations that embedded developers must often deal with.

Representing unsigned integers in binary

Floating point representations are beyond the scope of this document, but should be aware of how integers are represented in binary. We already know that integers are limited in range by their size in memory — up until now we have most often used 32-bit unsigned values in all of our examples and exercises. In binary every 'digit' can only be zero or one, so if we look at a hypothetical 2-bit number we can see all of the possible values:

Binary	*Decimal*
00	0
01	1
10	2
11	3

If we were to add one to a 2-bit binary value of '11', the result would wrap back to '00' (as the real answer would be '100', but we do not have a third bit to work with). This is *modular arithmetic*, and it is used in the vast majority of microcontrollers and processors. As a result, we must always be aware of the potential for overflow — in C, this is technically considered *undefined behaviour* and so should be avoided wherever possible.

Exercise: *An important alternative to modular arithmetic is saturation arithmetic. In this case, instead of overflowing, the integer value will simply saturate at its maximum or minimum value. Adding any number to an integer already at its maximum would have no effect, as would trying to subtract from an integer already at zero.*

This behaviour is not supported natively in C, but it can still be very useful. Try writing functions to perform the saturating addition and subtraction of two integers. You should use the following declarations to assist in your implementation:

```
typedef uint32_t usat32_t;

static const usat32_t usat32_max = UINT32_MAX;
static const usat32_t usat32_min = 0;

usat32_t USat_Add(const usat32_t left, const usat32_t right);
usat32_t USat_Sub(const usat32_t left, const usat32_t right);
```

Boolean operations and accessing individual bits

Now that we have some idea of the binary representation of unsigned integers, we can look at how the various boolean and bit-manipulation operators work and their results:

Type	*Left*	*Operator*	*Right*	*Result*
Shift Left	001	<<	001	010
Shift Right	100	>>	001	010
And	011	&	101	001
Or	011	|	101	111
Exclusive Or	011	^	101	110
Not		~	101	010

Sometimes it is necessary to set (to one) or clear (to zero) individual bits of a variable. To achieve this, we typically use a combination of the above operators. Setting and clearing is done as follows.

```
uint8_t flags = 0;  // flags = 00000000 binary
flags |= (1 << 2);  // flags = 00000100 binary
flags |= (1 << 1);  // flags = 00000110 binary
flags &= ~(1 << 2); // flags = 00000010 binary
flags &= ~(1 << 1); // flags = 00000000 binary
```

This works because the expression '`1 << n`' results in an integer with only bit '`n`' set and nothing else — this is often referred to as a *bit-mask*. We can use the bitwise or operator to ensure the bit is set in the target variable. To clear bits, we use the opposite of the mask (obtained with the bitwise not operator) and then apply the bitwise and operator.

Integer variables are sometimes used in this way to provide *flags* — enumerations where each constant has a single bit associated with it. This allows us to apply several at once:

```
typedef enum
{
    OPTION_1 = (1 << 0), OPTION_2 = (1 << 1)
} flags;

void Some_Function(const flags options)
{
    if (options & OPTION_2)
        // Do something if OPTION_2 is set...
}

Some_Function(OPTION_1 | OPTION_2);
```

Representing negative integers in binary

The most obvious approach to representing negative numbers in binary is to allocate one bit as the 'sign' and keep the rest to represent the value. This approach has been used in platforms such as the IBM 7090, and is used in standard floating point representations. The problem is that it leaves us with two possible representations of zero.

To get around this, the developers of modern microcontrollers tend to employ the *two's complement* approach. In this scheme, the negative of a number is found by inverting it and then adding one to the result. Clearly this only produces a single value for zero — if we start with the eight bit number '`00000000`' and invert it, we get '`11111111`'. Add one and modular arithmetic overflows back to zero again, right back where we started.

Here are some examples of other numbers:

Initial value	*Inverted*	*Two's complement*
`00000001 / +1`	`11111110`	`11111111 / -1`
`11111111 / -1`	`00000000`	`00000001 / +1`
`01111111 / +127`	`10000000`	`10000001 / -127`
`10000000 / -128`	`01111111`	`10000000 / -128`

As we can see, the maximum positive number is `+127` and the minimum negative number is `-128`. Trying to negate `-128` simply leaves us back with the same value, as it clearly can't be represented with 8-bits using this method. However, as the sign-bit method only provides `-127` to `+127`, we still get a slightly better range.

Another advantage of the two's complement approach is that we can still use the same modular arithmetic methods and hardware that are used for unsigned numbers. This may not be obvious at first, but consider the sum '`1 + -1`' — this is equivalent to the result of the binary sum '`00000001 + 11111111`', which will just overflow back to zero again.

Exercise: *Implement a signed version of your saturation arithmetic functions from the previous exercise, with the following declarations as a guide.*

```
typedef int32_t sat32_t;

static const sat32_t sat32_max = INT32_MAX;
static const sat32_t sat32_min = INT32_MIN;

sat32_t Sat_Add(const sat32_t left, const sat32_t right);
sat32_t Sat_Sub(const sat32_t left, const sat32_t right);
```

Exercise: *Add functions for multiplication and division to both types of saturation arithmetic functions. Test them at both extremes as well as with non-saturating values.*

Optimisation and the volatile qualifier

All of your exercises so far have probably been compiled *without* optimisation, because this is the default. When we turn optimisation on — by using the 'release configuration' — we are telling to compiler to do whatever it can to make the program faster, preferably without changing its behaviour. To do this, the compiler has to make assumptions about our code that may not be accurate, which can lead to poorly optimised or incorrect code.

For example, the following code checks the value of a variable in a condition, immediately after it has been assigned a value. The compiler could optimise the entire `if` statement away, as the condition can never be true... unless the hardware can produce an *interrupt* whose handler function could result in the value of that variable changing. If this happens in between the assignment and the if statement, then the condition must still be tested, and the compiler may have generated incorrect code.

```
uint32_t a = 0;

void Some_Function(void)
{
    a = 5;
    // ...code which does not change a...
    if (a > 10)
    {
        // This could be optimised away!
    }
}
```

The example above is contrived, but the situation is all too real — just see the '`scheduler.c`' file for more examples. The `Scheduler_Tick` function is called from an interrupt handler that will execute every millisecond, whereas the `Scheduler_Dispatch` function is called repeatedly from `main`. As these functions access the same variables, they must all be given with the `volatile` qualifier, which tells the compiler not to make any assumptions about these variables. For example:

```
volatile uint32_t a = 0;
```

Another keyword that we can use to tweak the compiler's optimisation behaviour is the `inline` qualifier. This tells the compiler that the given function may be removed and every call to it can be replaced directly with its code — in the same way that we previously achieved with macros. This is only a hint, as the compiler is free not to follow it or even to inline functions that do not have the `inline` qualifier.

Generally speaking — when optimisation is turned on — a function that is both `static` and `inline` should be just as fast as a macro with the same behaviour. The difference is that macros have some associated pitfalls, but can still be stepped over when debugging.

Memory mapped peripheral registers

Microcontrollers talk to the outside world through *peripherals*. These perform functions such as reading and writing the serial data that we have used with the `printf` function in this course so far. These peripherals are often controlled by writing to (and reading from) hardware registers, which are usually mapped onto specific locations in memory.

If we look at the '`gpio.h`' file, which provides drivers for accessing the General Purpose Input and Output pins, we will see the following towards the end:

```
typedef struct
{
    uint32_t value;
    uint32_t direction;
    uint32_t padding[62];
} raw_gpio_register_t;

typedef volatile raw_gpio_register_t * const gpio_register_t;
static gpio_register_t gpio_register =
                                    (gpio_register_t) 0x80000200;
```

We have covered enough C to understand this complex source-code, but let's go through it anyway. The structure shouldn't contain any surprises, with the exception of the padding array — this is necessary because the `gpio_register` pointer is used with the indexing operator, which in turn uses pointer arithmetic in steps equal to the structure's size.

The GPIO registers are located at `0x80000200` for the first port[9] and `0x80000300` for the second. The padding, when added to the sizes of the other variables, brings the structure's size up to `0x100` (or `256` in decimal). Therefore, an expression like '`gpio_register[1]`' will dereference the pointer at the exact memory address for the second port, where we can then refer to the other variables directly (e.g. '`gpio_register[1].value = 1`').

Next we might note the use of `volatile`, which is here because the hardware can alter the value in the register at any time, and we cannot allow the compiler to optimise any of this code away. This works in exactly the same way that we discussed in the previous section.

Finally, note the use of `const` *after* the pointer symbol in the `typedef`. This indicates that it is the pointer itself that is constant and not the value that it points to. Of the three positions `const` can be placed around a pointer (e.g. '`const int8_t const * const`'), the first two signify that the memory being pointed to is constant and only the last refers to the pointer. This can be verified by the technique for reading declarations that we covered previously.

[9] Notice the hexadecimal number format. This consists of the prefix '`0x`' followed by the number in base 16, with the letters '`a`' to '`f`' representing the numbers `10` to `15` — case is ignored for the letters. Details of all the different number formats that we can use in C source-code are beyond the scope of this document, but can be found in any textbook.

Each GPIO port has 32 pins, with each bit in the `value` variable representing one pin. It should come as no surprise to see `GPIO_Set` implemented as follows. Note the use of a macro with the `do` statement and parentheses around each parameter use, as we discussed in the section about the pre-processor.

```
#define GPIO_Set(gpio)                                          \
    do {                                                        \
        gpio_register[(gpio).port].value |= (1 << (gpio).pin); \
    } while (0)
```

This macro example also shows the use of the backslash character at the end of each line. This is recognised by the pre-processor, which removes it and joins the next line into the current one. The entire macro is then on a single line, as required by the pre-processor.

Ignoring the extra code needed to deal with macros, we can see that the `gpio` parameter is intended to be a structure with `port` and `pin` variables (it can be *any* structure with those variables). This is used to select the correct port address and then set a given bit, *without* modifying the other pins — exactly as we discussed in the section on bit-manipulation.

Recursion and overflowing the stack

Suppose that the function being executed were to call itself — this is called *recursion*. For example, a recursive function to calculate the factorial of a number might look like this:

```
uint32_t Factorial(const uint32_t n)
{
    if (n < 2)
        return 1;
    return n * Factorial(n - 1);
}
```

If we try executing '`Factorial(2)`', it will execute the expression '`2 * Factorial(1)`' and return the result. Note that there is an `if` statement at the beginning that provides an exit clause; when the parameter is eventually below `2`, the function just returns `1`. This is a common pattern in recursive functions — we must have a way out when finished.

The problem with recursion is that, every time we call a function it allocates some space on the stack. Since we can't tell from the source-code how many times a recursive function will be calling itself, we have no idea of how much stack space it will use. Because stack space is limited in most embedded systems, we must be extremely careful with how much we are using — so recursion is generally avoided.

Exercise: *Any recursive function can be rewritten to work iteratively with one or more loops. Try rewriting the `Factorial` function to use a loop instead and test both styles to ensure that they produce the same results.*

Predictability and dynamic memory

When we use the standard library's `free` function to release allocated memory, it will place the block of memory onto a free list. Subsequent calls to `malloc` will first search the free list for a large enough block, before acquiring more memory from the heap.

For example, the time it takes to allocate a large block of memory will be much longer if we have already released many small blocks (there is more to search through). This lack of predictability and determinism make `malloc` and `free` very difficult to justify in low-level embedded systems, and they are often avoided as a result. For this reason, we should be aware that many coding standards forbid their use in safety critical systems.

Summary

We have reached the end of the document and the course. I hope you have enjoyed what you have learnt so far and did not experience too much difficulty with any of the exercises.

The concepts that we have discussed throughout this course are expanded on in the MSc in Reliable Embedded Systems, offered by the University of Leicester in partnership with TTE Systems Limited. The MSc is intended to be taken by working engineers who are in full-time employment. It runs on a part-time basis — there are six intensive (5-day) training courses spread over a two year period.

Further details can be found online at http://www.tte-systems.com/services/training/msc.

The RapidiTTy Lite IDE

Throughout this course we have used RapidiTTy Lite — a *free* IDE from TTE Systems Ltd that provides everything we need to develop software for embedded systems in C and Ada. RapidiTTy offers a number of advanced features, including static and dynamic analysis of stack usage requirements, full cycle-accurate timing analysis and schedulers capable of highly predictable operation.

Further details can be found online at http://www.tte-systems.com/products.

The TTE32 processor core

RapidiTTy Lite targets our own TTE32 processor cores, intended for use in safety- and mission-critical applications. Based on a 32-bit design with a five-stage pipeline, they have a Harvard architecture and a static interrupt overhead. They have been designed from the ground up to provide *guaranteed* memory-access and instruction-execution times.

Further details can be found online at http://www.tte-systems.com/tte32.