**Implementation of Simple C Programming Concepts in IDE: Bitwise Operations, Control Blocks and Functions**

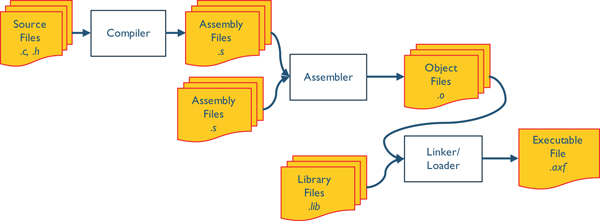
**1. Aim:**

To develop ‘C’ programs using STM32CubeIDE in the context of embedded systems.

**2. Introduction:**

Three major types of tools used for embedded software development are the program build toolchain, the programmer, and the debugger. The program build toolchain translates a program into a format that the MCU can understand, stored in an executable file. The programmer programs that information into the MCU’s program memory. This memory is nonvolatile (typically Flash ROM), so it will remain even after power is removed. The debugger enables the developer to control program execution and examine the program state (e.g., current instruction, values of processor registers, and data memory) as it runs on the processor. These tools are often grouped together in a single integrated development environment (IDE) to simplify development.

The program build toolchain is shown below.



The individual blocks of program build toolchain are explained below.

**2.1 Cross Compilers**:

Executable files created by compilers are usually platform dependent. An executable file compiled for one type of microprocessors, such as ARM Cortex-M3, cannot directly run on a platform with a different kind of microprocessors that support a different set of machine instructions, such as PIC or Atmel AVR microcontrollers. When we migrate a program written in a high-level language to a processor of a different instruction set, we usually have to modify and recompile the source programs for the new target platform. The binary machine program follows a standard called executable and linkable format (ELF). ELF, as its name suggests, provides two interfaces to binary files:

(i) a linkable interface that is used at static link time to combine multiple files when compiling and building a program.

(ii) an executable interface that is utilized at runtime to create a process image in memory when a program is loaded into memory and then executed.

**2.2 Assembly Code:**

To optimize our application for maximum execution speed or minimum memory size, writing pieces of our code in assembly language is one approach. The goal of the software is to combine two 8-bit variables into one 16-bit variable. If you cannot see the bug, look up the precedence of the two operators << and +.



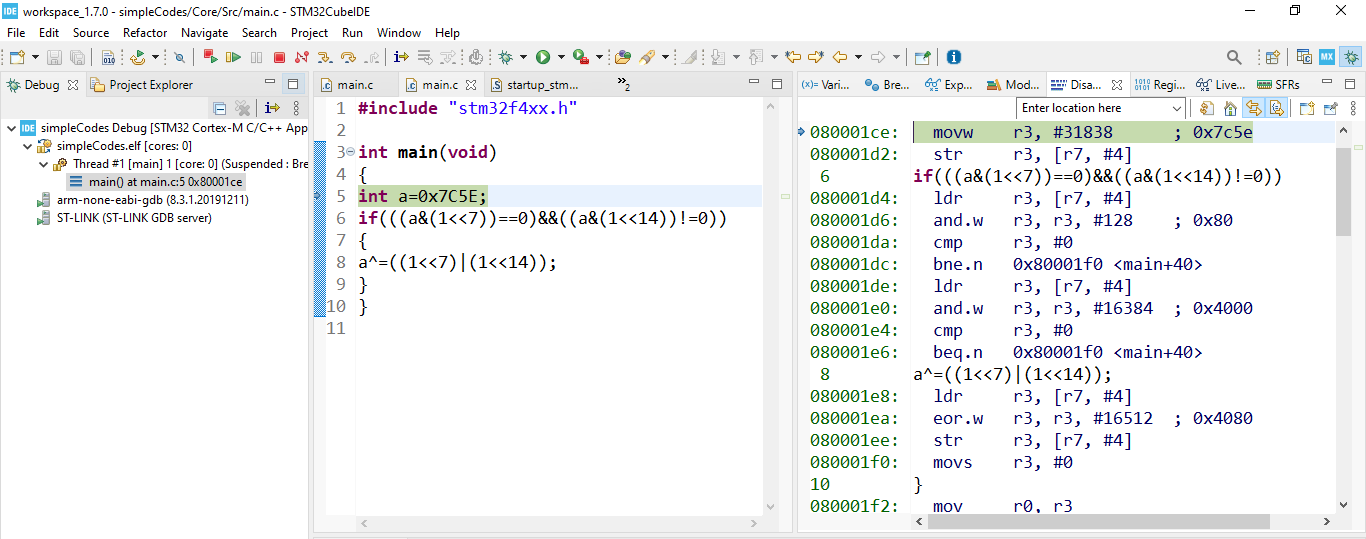
Good understanding of the assembly code generated by our compiler makes us better programmers.

**2.3 Debugger:**

Debugger enables the developer to control program execution and examine the program

state (e.g., current instruction, values of processor registers, and data memory) as it runs on

the processor. It allows user to observe and control program execution, variables, and processor registers.



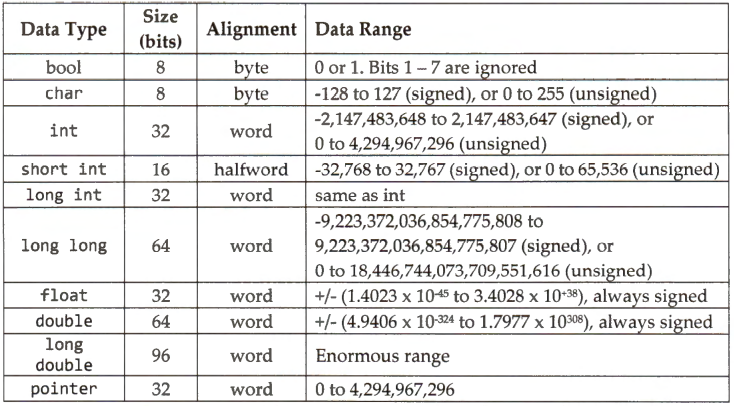
**3. Variables:**

Obvious considerations for creating a variable are the size and format of the data. Scope of the variable defines which software modules can access the data. A system is easier to design (because the modules are smaller and simpler), easier to change (because code can be reused), and easier to verify (because interactions between modules are well-defined) when we limit the scope of our variables. Since modules are not completely independent, we need a mechanism to transfer information from one to another.

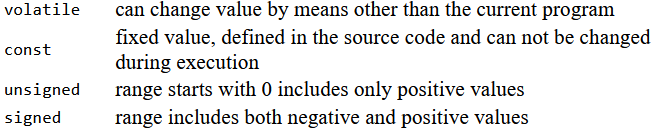
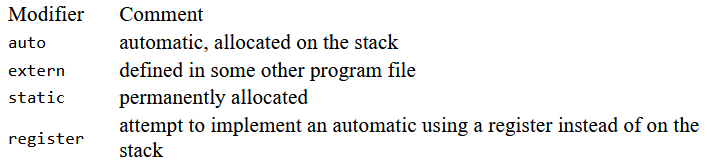
**3.1 Declarations:**

Describing a variable involves two actions. The first action is declaring its type and the second action is defining it in memory (reserving a place for it). Although both of these may be involved, we refer to the C construct that accomplishes them as a *declaration*. If the declaration is preceded by **extern,** it only declares the type of the variables, without reserving space for them. Here, the definition must exist in another source file. Failure to do so, will result in an unresolved reference error at link time.

**3.2 Typical ‘C’ data types:**



**3.3 Variable storage classes:**

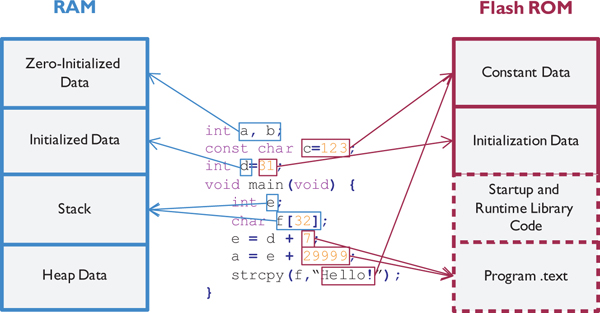


**3.4 Memory Allocation:**

A *variable* is a named object that resides in RAM memory and is capable of being examined and modified. It is used to hold information critical to the operation of the A *constant* is a named object that resides in memory (usually in ROM) and is only capable of being examined. *literal* is the direct specification of a number character or string. The difference between a literal and a constant is that constants are given names so that they can be accessed more than once.



**A program’s memory requirement:**



**3.5 Global vs. Static Variables:**

In an embedded system we normally wish to place all variables in RAM and constants in ROM.

The fact that these types of variables exist in permanently reserved memory means that static variables exist for the entire life of the program. When the power is first applied to an embedded computer, the values in its RAM are usually undefined. Therefore, initializing global variables requires special run-time software consideration. A **static global** is very similar to a regular global. In both cases, the variable is defined in RAM permanently. The assembly language access is identical. **The only difference is the scope. Static global** cannot be referenced by modules in other files.

**3.6 Static Variables:**

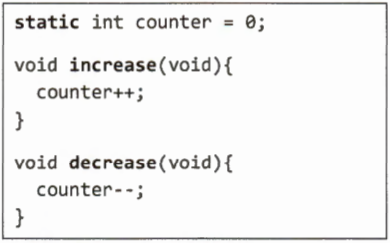
Different from local variables, a *static* C variable has a lifetime over the entire program runtime.

It is initialized only once at the compiling time no matter how many times this function is called.

A static variable can be either global or local. A static local variable can only be available within the scope of the function in which this variable is declared. A static global variable can only be accessed within the source file in which this variable is declared.

**3.7 Static Global Variables:**

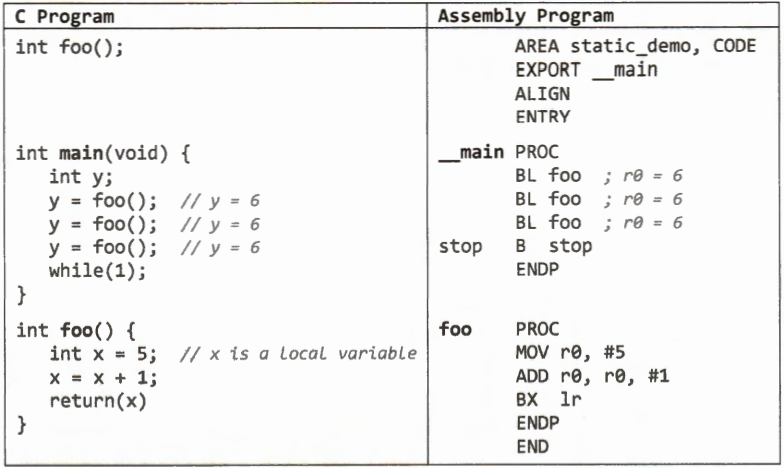
Always avoid global variables. If a variable is accessed by more than one subroutine within a source file, then it can be declared as global static variable.



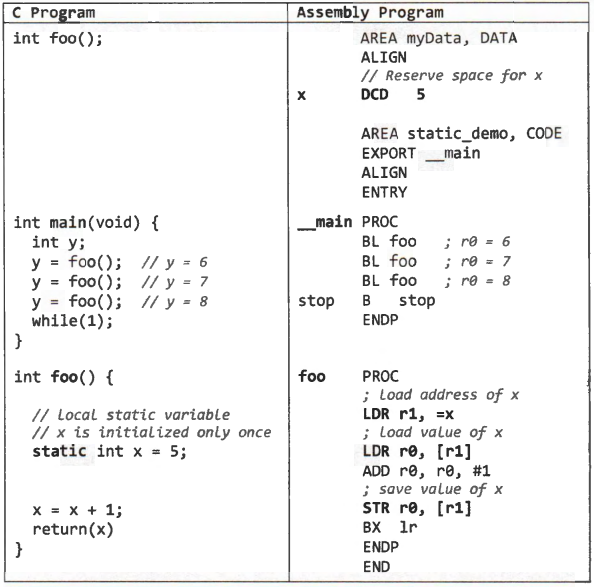
**3.8 Static Local Variables:**

All static variables are allocated in the data memory. A local variable is often stored in a register or the heap region of the data memory. A static variable is always loaded from memory first and then is stored back to the memory before exiting the subroutine. Therefore, if the subroutine is called again, the static variable keeps its previous value, instead of its initial value. The initialization is carried out at compile time instead of at runtime. The variable is defined in the data region with an initial value. No matter how many times subroutine runs, the variable is never initialized.

**Example of a local non-static variable:**



**Example of a local static variable:**



**3.9 External Storage Class:**

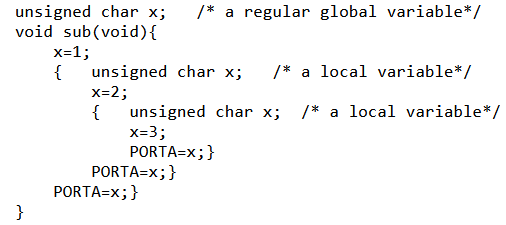
The compiler knows an external variable by the keyword **extern** that must precede its declaration. Only global declarations can be designated extern and only globals in other modules can be referenced as external. This global can be referenced by any function from any file in the software system. You normally use **extern** when you want to refer to something that is *not* in the current translation unit, such as a variable that's defined in a library you will be linking to it.

This tells the compiler, **“There’s a variable defined in another module called myGlobalVariable, of type integer. I want you to accept my attempts to access it, but don't allocate storage for it because another module has already done that.”**

**extern** int myGlobalVariable;

**3.10 Scope of a Variable:**

The scope of local variables is the block in which they are declared. Local declarations must be grouped together before the first executable statement in the block-at the head of the block. If we declare a local variable with the same name as a global object or another local in a superior block, the new variable temporarily supersedes the higher level declarations



**3.11 Volatile Variable:**

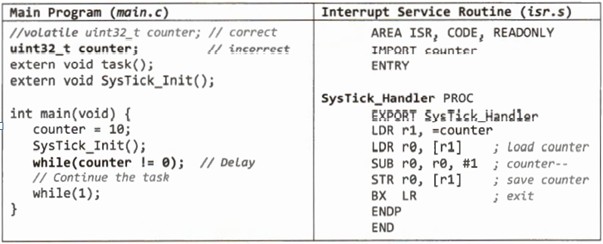
When the compiler optimizes a C program, a hard-to-find hidden error is that the program mistakenly reuses the value of a variable stored in a register, instead of reloading it from memory each time. To avoid such compilation error, the program should declare the variable as volatile, such as:

**volatile int** variable

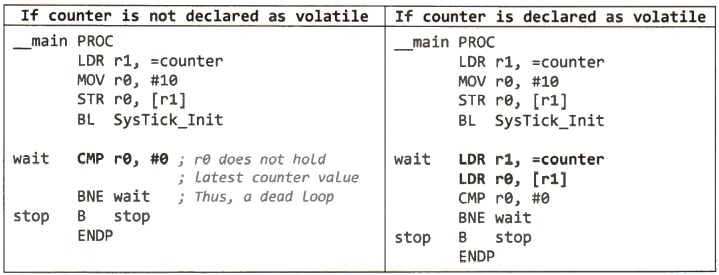
A volatile variable is a variable that may be changed by an external input or an interrupt handler. Therefore, the processor should not use a register to cache this variable to avoid using stale data. The keyword volatile forces the compiler to generate an executable, which always loads the variable value from the memory whenever this variable is read, and always stores the variable in memory whenever it is written.

This code illustrates the necessity of declaring a variable **counter as volatile,** shared by two

concurrently running tasks (**main function** and **SysTick\_Handler**).



Here compiler observes that, after counter is initialized to 10, the value of the counter Variable is not modified directly by **main()** or indirectly by any subroutine called from **main().**



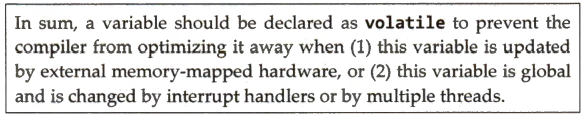
If the counter is not declared as volatile, the while loop is a dead loop. SysTick\_Handler periodically decrements the counter and stores its value in memory. However, the main memory repeatedly checks register r0, without reloading the latest value of the counter from memory. If the counter is declared as volatile, the dead loop problem is avoided.

**3.12 Memory Mapped Registers:**

A ‘C’ program should declare any variable that represents the data of a memory-mapped I/O register as volatile. Memory-mapped I/0 has been widely used to access peripheral devices.

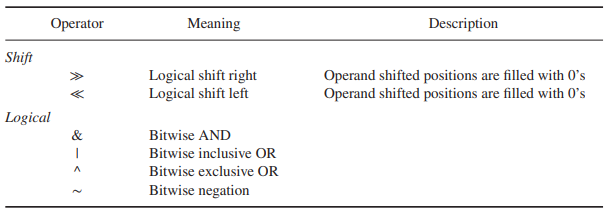
Data and control registers of external devices are mapped to specific memory addresses, and a program can use memory pointers to access these hardware registers.

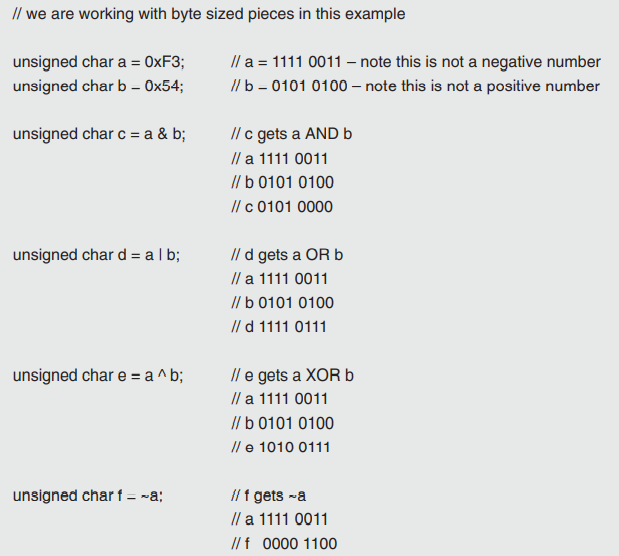




**4.1 Bitwise Operators:**

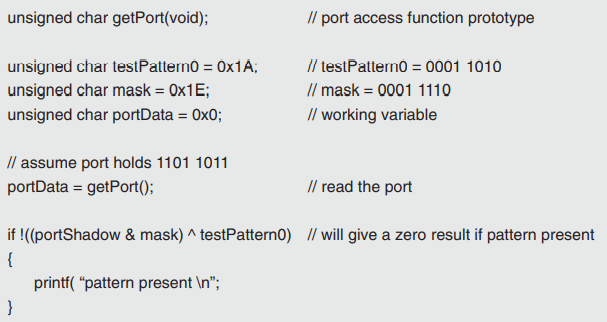
* Bitwise operators work naturally with embedded hardware by permitting the test and modification of individual signals coming into or going out of the embedded processor.
* These operators are intended for work at the hardware level – that is, with the registers and input or output ports on a target machine.
* When working at the bit level, often all of the bits in a word are important because they generally represent the state of some signal in the hardware of the machine.



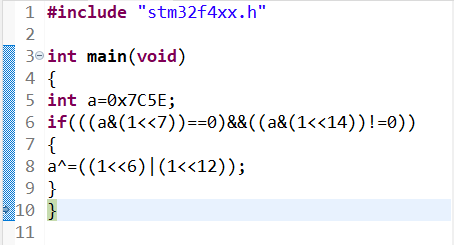


**Examples:**

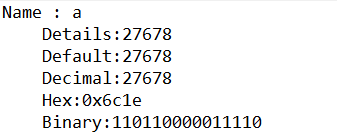
1. Using the Bitwise Operators to test for a Pattern within a Data Word



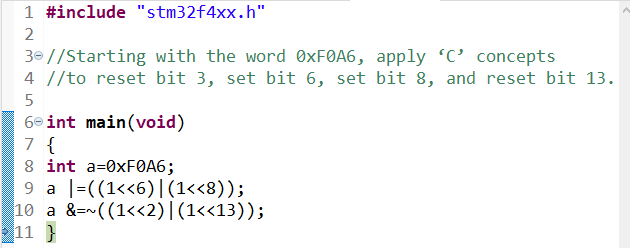
1. Starting with the word 0x7C5E, determine if bit 7 is reset and bit 14 is set. If so, complement bits 6 and 12.



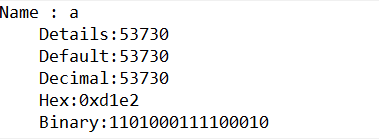
**Output:**



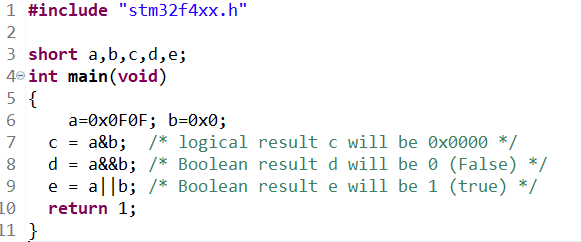
1. Starting with the word 0xF0A6, apply ‘C’ concepts to reset bit 3, set bit 6, set bit 8, and reset bit 13.



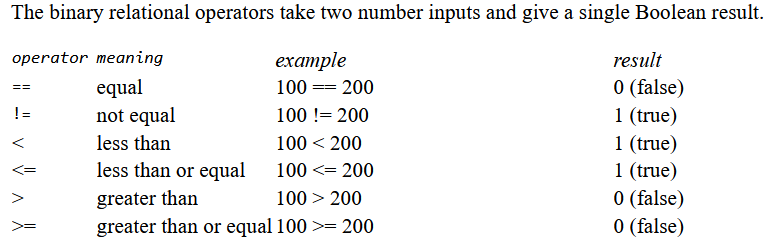
**Output:**

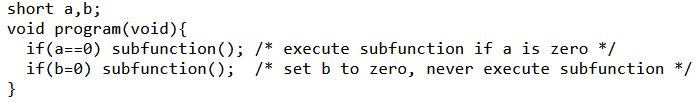


**4.2 Logical Vs Boolean Operators:**

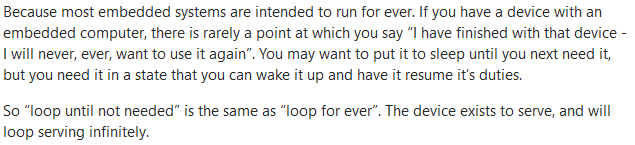


**4.3 Binary relational operators:**



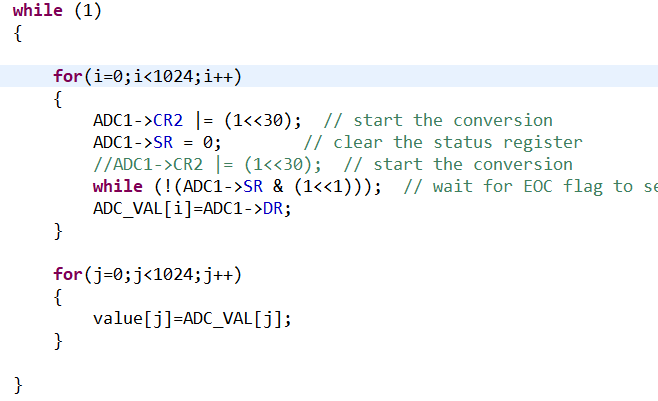


**Why do we use an infinite loop in embedded system software?**



The application in a **bare-metal** (without an OS) system must run forever, because the is nothing ‘after’ the application. The common way to do this is to use an infinite loop.

In bare metal programming without a RTOS a super loop coupled with interrupt service routines is the normal. Hence the use of a forever loop with varying priority of interrupts for time critical events.



**Return and busy waiting statement:**

The return statement is used within a function when a useful value is to be returned to the caller.

As its name implies, it does exactly nothing. Why to have a statement that serves no purpose?



**5. Modules:**

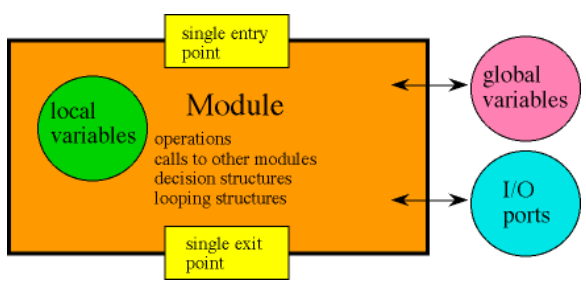
The key to effective software development is the appropriate division of a complex problem in modules. A module is a software task that takes inputs, operates in a well-defined way to create outputs. In C, functions are our way to create modules. A small module may be a single function. A medium-sized module may consist of a group of functions together with global data structures, collected in a single file. A large module may include multiple medium-sized modules.

Complex systems designed in a modular fashion are easier to debug because each module can be tested separately. Industry experts estimate that 50 to 90% of software development cost is spent.

in maintenance. All five aspects of software maintenance:

* Correcting mistakes
* Adding new features
* Optimizing execution speed or program size
* Porting to new computers or operating systems
* Reconfiguring the software to solve a similar related problem

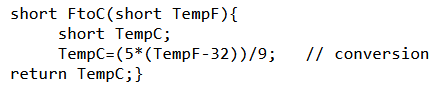
Above all are simplified by organizing the software system into modules, The approach is particularly useful when a task is large enough to require several programmers.



**5.1 Function in C programming:**

A mathematical function is a well-defined operation that translates a set of input values into a set of output values. In C, a function translates a set of input values into a single output value

Consider the function that converts temperature in degrees F into temperature in degrees C.



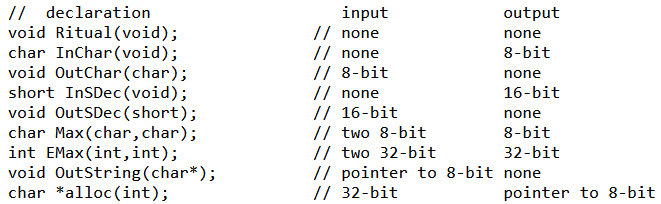
Similar to variables, C differentiates between a function declaration and a function definition

Similar to variables, C differentiates between a function declaration and a function definition

A declaration specifies the syntax (name and input/output parameters), whereas a function definition specifies the actual program to be executed when the function is called. A function must be declared (or defined) before it can be called.

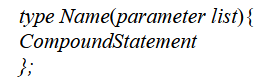
**5.2 Function declarations:**

A function declaration begins with the type (format) of the return parameter. If there is no return parameter, then the type can be either specified as **void** or left blank. Next comes the function name, followed by the parameter list. In a function declaration we do not have to specify names for the input parameters, just their types. If there are no input parameters, then the type can be either specified as **void** or left blank.



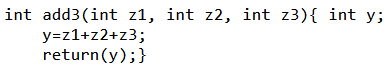
**5.3 Function definitions:**

Every function must be defined somewhere.



The **type** specifies the function return parameter. If there is no return parameter, we can use **void** or leave it blank. **Name** is the name of the function.

The **parameter list** is a list of zero or more names for the arguments that will be received by the function when it is called. Both the type and name of each input parameter is required. Compound statements may contain local declarations, simple statements, and other compound statements. It follows that functions may implement algorithms of any complexity and may be written in a structured style.



**5.4 Function calls:**

A function is called by writing its name followed by a parenthesized list of argument expressions.



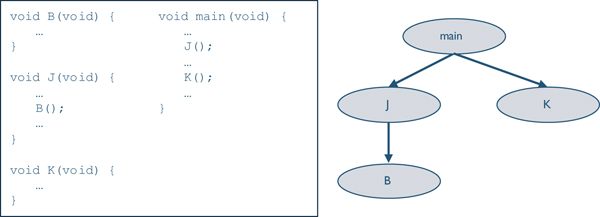
**Name** is the name of the function to be called. The**parameter list**specifies the particular input parameters used in this call. Each input parameter may be as simple as a variable name or a constant, or it may be arbitrarily complex, including perhaps other function calls. The resulting value is pushed onto the stack where it is passed to the called function. C programs evaluate arguments from left to right, pushing them onto the stack in that order. On return, the return parameter is located in **register R0**. The input parameters are removed from the stack at the end of the program.



**5.5 Argument Passing:**

With respect to the method by which arguments are passed, two types of subroutine calls are used in programming languages--*call by reference* and *call by value.* The *call by reference* method passes arguments in such a way that references (pointers) to the actual arguments are passed, instead of copies of the actual arguments themselves. Since C supports only one formal output parameter, we can implement additional output parameters using call by reference. The most important point to remember about passing arguments by value in C is that there is no connection between an actual argument and its source. Changes to the arguments made within a function, have no affect what so ever on the objects that might have supplied their values. C uses call by value that we can pass expressions, not just variables, as arguments.

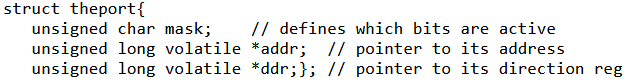
The function call graph is a diagram that shows possible function calls.



Every C program must have a function called **main.** The main function for embedded systems never completes, unlike a program you might run on your personal computer or smart phone.

**6. Structures:**

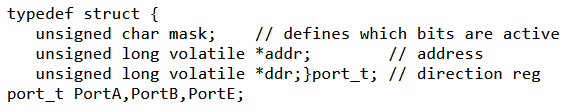
A structure is a collection of variables that share a single name. With structures we specify the types and names of each of the elements or members of the structure. To access data stored in a structure, we must give both the name of the collection and the name of the element. The structure must be declared before it can be used. The declaration specifies the tag name of the structure and the names and types of the individual members.



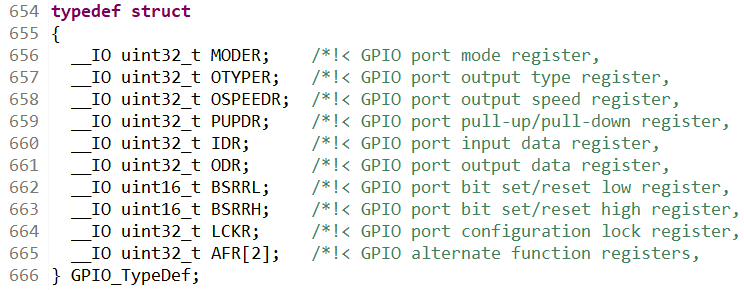
Declaration does not create any variables or allocate any space. To use a structure, we must define a global or local variable of this type. The above line defines the three variables and allocates 9 bytes for each of variable. Because the pointers will be 32-bit aligned the compiler will skip three bytes between **mask and addr**, so each object will occupy 12 bytes.



To improve code reuse we can use **typedef** to actually create a new data type (called **port** in the example below) that behaves syntactically like **char,** **int,** **short** etc.



**6.1 GPIO Port declaration of STM32F407:**

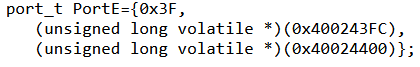


**6.2 Accessing the members of the structure:**

We need to specify both the structure name (name of the variable) and the member name when accessing information stored in a structure.

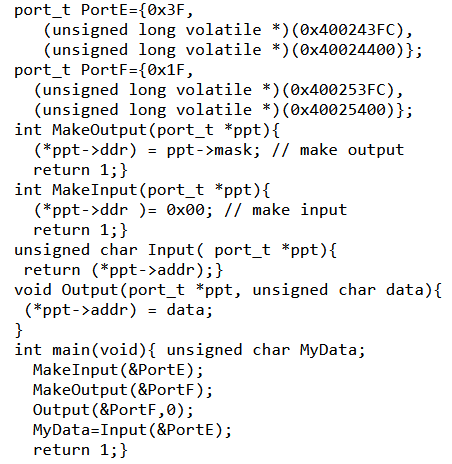


Just like any variable, we can specify the initial value of a structure at the time of its definition.



To place a structure in ROM, we define it as a global constant.

**Example: Passing structures to functions**



**7. Conclusion:**

In this experiment, embedded ‘C’ programming concepts like variables, data types, operators,

function, and structure are explored in greater detail with practice codes using STM32CubeIDE.