Embedded C Programming

Ben Nahill

January 27, 2013

Contents

Embedded C programming is, for the most part, the same as using C on any other platform; it uses the same compilers and therefore the same language specification but operates in a different ecosystem with very limited resources, no support from an OS (for the purpose of this document anyways...), and memory management which must be done entirely in software. Anyways, the similarities outweigh the differences and as such, this document focuses more than anything on the basics of good C programming.

Some prior experience with C or similar languages is assumed here. In my examples I try to provide a variety of methods to achieve common tasks and to highlight the trade-offs that are involved.

1 Toolchains

A toolchain is just as you would expect from the name: a chain of tools to turn your code into a binary to be executed on your device. Even if the details are abstracted away from you, this is the process that is going on when you build code and therefore it's important for understanding some of the intricacies of C to know a bit about the process. For most languages compiling to machine code (C included), the chain involves the following steps. Though sometimes steps are merged together in some implementations, these operations are still distinct.

Preprocessor

Quickly scan your code for syntax errors and apply your includes, macros, and conditional compilation steps.

Compiler

The compiler itself takes in preprocessed C code files and spits out assembly files. It knows about your processor architecture and, depending on your configuration, may optimize some operations.

The compiler still works on the file level. It doesn't perform any optimization of functionality that spans multiple files. Each file gets a symbol table defined here which indicates all of the symbols (variable and function names) which are to be connected to other files when we get to the linking stage. Exports are those that are contained in the current file, while imports are those symbols that we expect to find in other files. At this point, no addresses have been determined yet. Each segment of code or variables are assigned to a section and will only later get absolute addresses.

Assembler

This takes the assembly code and turns it into machine code. Since it is still on the individual file level, a symbol table is still necessary here. The outputs from this are usually called object files.

Linker

The linker looks through the object files together and places each bit of code into their appropriate sections based on a linker script. The linker script describes sections as memory ranges and details some of their properties. The linker script is hardware-dependent but is handled for you by most commercial tools. With all of the symbols piled into sections, they finally have absolute addresses and the tool can now connect the symbols to their appropriate values. Some linkers have options to perform some basic optimization during the linking process.

The output is a binary file which still contains a bit of extra information, usually for debugging purposes. The final step is to convert this to raw binary data to be loaded by the debugger/programming tool onto the device.

1.1 Toolchain Options

The ARM architecture is widespread usage and is therefore supported by quite a number of tools. The biggest currently are Keil MDK-ARM (what you are using), IARs ARM Embedded Workbench, and GNU GCC. The former two are commercial options including a GUI and debugging interface, while GCC is open-source and is really just a collection of tools that can work together to perform the steps listed above. No graphical component, build management, or hardware debug interface are included. That said, with proper setup (see Atollic TrueStudio), the GNU toolchain can be extremely powerful. Another open-source tool that is coming along for ARM and has shown great promise in other arenas is LLVM.

Disclaimer: I use GCC pretty much exclusively and highly recommend it to anyone serious about any low-level software development. For the purpose of this course, however, you should stick with Keil.

2 Syntax and The Preprocessor

The first pass at your processing your code is made by the preprocessor. As mentioned above, it parses based on the basic grammar of the language. Understanding how this all happens will help you to debug all sorts of mistakes. The preprocessor and most of the compilation process looks only at one file at a time. All interaction is put on hold until the last step and most error occur before this. It's much easier to debug when you realize that your errors in that case are limited to a single file.

Most files look like this at some level:

```
/*******************
 * Some information about the code and author
 *******************/

// A bunch of included header files>
#include <some_file.h>
#include <some_other_file.h>
...

// Some file-local macros
#define BUFFER_LEN 64
...

// Some file-local type definitions
typedef struct {
    int r, i;
} complex_t;

// Global statically-allocated variables
int number_available_to_everyone = 5;
...

// File-local statically allocated variables
static int my_private_number = 20;
...

// File-local function prototypes
static complex_t my_private_add_func(complex_t* a, complex_t* b);

// A bunch of functions
int main(){
    complex_t a = {1,20};
    complex_t b = {1,-1};
    my_private_add_func(&a, &b);
}
```

```
static complex_t my_private_add_func(complex_t* a, complex_t* b){
    complex_t ret;
    ret.r = a->r + b->r;
    ret.i = a->i + b->i;
    return ret;
}
```

Listing 1: Common file structure

2.1 Macros

Macros are simple find-and-replace directives that will perform substitutions in your code to make them easier to read. These substitutions are made before compilation so there's no performance hit. The text to be substituted in goes to the end of the line.

```
// #define <identifier> <value>
#define BUFFER_LEN 128

uint32_t some_array[BUFFER_LEN];
```

Listing 2: Macros

For many applications it's preferable to use `const` variables instead because they provide cleaner type-checking, usually at no performance cost since the compilers

You can also create functions with macros. Because they perform a blind substitution operation, there are no types associated with arguments. Since this isn't always a good thing, in many cases I recommend using inline functions instead where reasonable.

```
// Create shortcut for bit masks
#define BIT(x) (1 << x)

const uint32_t mask = BIT(1) | BIT(2);

// A multiline macro function (note "\")
#define GHETTO_INLINE_FUNC(x,y) if(x > 5) { \
                                    y += 1; \
                                }
```

Listing 3: Macro functions

2.2 Conditional Compilation

Sometimes it's nice to provide compile-time options to make it easy to switch behavior. Conditional compilation makes this possible without the performance hit of if-else statements and without the headache of moving around block comments.

```
// Check if the DEBUG identifier is defined
#ifdef DEBUG
    // Be more verbose
#endif

// #ifndef is the opposite

// I prefer this because it checks to make sure you've explicitly
// decided either way
// Make sure it's defined and condition on the value of DEBUG
#if DEBUG
    // Be more verbose
#endif
```

```
// General #if structure
#if <condition>
    // Something
#elif <other condition>
    // Something else
#else
    // Something else yet
#endif
```

Listing 4: Conditional compilation

2.3 Preprocessor Directives

2.3.1 Header Files

Now, saying that the compiler only looks at one file at a time is somewhat deceptive because of header files. Header files are no different from source files but usually have the *.h* file extension. When a header file is included using the `#include <some_file.h>` preprocessor directive, that file contents are substituted in place of that line. This makes them a surprisingly powerful tool if used carefully to be able to share code between files. The most common use is to share function prototypes, type definitions, variable declarations, and macros/constants.

Multiple-include Protection A common problem that can come up when including headers is that circular dependencies and repeated inclusions may occur. File a.h and b.h may both use types defined in file c.h. When a.c includes both a.h and b.h, c.h is included twice. Since this puts them all in sequence in the same file, anything in c.h will be repeated and may cause errors. A solution to this is to provide a protection block around the contents of c.h.

```
#ifndef __C_H_
#define __C_H_

#include <a.h>

// c.h contents

#endif
```

Listing 5: Header include protection (c.h)

This allows for only one inclusion per source file due to Conditional Compilation. `__C_H_` should be named something based on the filename so that it will be unique within your project.

2.4 Syntax

2.4.1 Operators

C provides a number of operators which are basically built-in functions using an alternate syntax that is more familiar or more appropriate for the case.

Arithmetic Basic arithmetic operations exist for numerical types:

```
a + b;
a - b;
a * b;
a / b;
a % b; // a modulo b (remainder of a / b)
a++; // Increment to a + 1, but return a
```

```
++a; // Increment to a + 1, return a + 1
a--;
--a;
```

Bit-wise Common bit-wise operations are also well-supported for most numerical types:

```
a | b;  // a or b
a & b;  // a and b
a ^ b;  // a xor b
a << b; // a left-shifted by b
a >> b; // a right-shifted by b
~a;     // not a
```

Logical Logical operators take boolean arguments and produce a boolean result:

```
a || b; // a or b
a && b; // a and b
!a;     // not a
```

Assignment Assignment operators change the value of the left-hand operand.

```
a = b;  // Assign the value of b to a

// Compound assignments
a += b; // a = a + b
a -= b; a *= b; a /= b; a &= b; a %= b; a >>= b; a <<= b; ....
```

Comparison Comparison operators compare two values and provide a boolean result for use in conditionals:

```
a > b;
a >= b;
a < b;
a <= b;
```

Pointers A few operators exist to help with pointers:

```
&a; // Address of a (as a pointer type)
*b; // The value at address a (since a is pointer)
```

See subsubsection 3.1.4 for a more information on this.

3 Variables and Functions

3.1 Datatypes

C provides a few built-in primitives but, for better or worse, allows free casting between them to the extent that such conversions are required to perform some tasks. Resource types must be declared explicitly when creating any object.

3.1.1 Integers

There are lots of integer types that are differentiated by their width and signedness for arithmetic operations. `char`, `short`, `int`, `long`, and `long long` are the most commonly known integer types which are usually signed in most toolchains. To make any one of them unsigned, simply apply the `unsigned` modifier before it (ie `unsigned char`). Their lengths vary depending on implementation so it is often ambiguous and I don't personally recommend using them for most embedded applications, though any negative effects are *usually* minimal and I think it's occasionally justified to use `int`s for simplicity.

The standard header stdint.h provides types of explicit length and signedness. It will include `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, ... through `uint64_t`. I recommend their use instead, as this header will contain the correct widths for your architecture and toolchain. It also includes useful macros like `INT8_MAX` and `INT8_MIN` to clearly address the limits of each type. In embedded C, its often important to know the worst case bounds of a given variable and size them accordingly. It's more important yet to be sure that you carefully analyze the possibility of overflowing a variable and handle it accordingly. Many frameworks and math libraries will also include their own types which may vary internally from platform to platform. stdint.h also provides a few other convenient and similarly specified types, such as `uint_fast8_t` for the fastest unsigned (or signed) integer of at least 8 (or 16, 32, or 64) bits available on the platform.

All numerical types in C support a shorthand incrementing syntax, allowing incrementing a variable in the same statement that uses its value. Depending on your intended use, you can *preincrement* a variable by writing it `++some_number`. This will provide the value `some_number + 1` in the current statement and commit the change back to some_number, such that `some_number == some_number + 1`. *Postincrementing* is written `some_number++` and has the same effect on `some_number` but provides the previous value to the current statement.

3.1.2 Boolean

Boolean types don't exist for your general use in C. Boolean types exist internally for logical operations such as conditionals. They are the result of comparison operations and can also be created implicitly as casts from integers or pointers. The condition expression for any conditional statement (including loop conditions) must cast to a boolean value or it won't compile. Logical operators exist to perform common logical operations.

As booleans aren't an available type for you in C, they are typically implemented with small integers and can be assigned easily from boolean operations with ternary conditionals.

3.1.3 Arrays

Any type may exist as an array which simply refers to a series of values of the same type stored in contiguous memory. That type can be anything. These values , regardless of the size of each cell (2 bytes for a `uint16_t`), will be packed together as tightly as possible. A big weakness of the type is that the data structure knows nothing about its own length. As such, it cant be passed directly as an argument since that function wont know the length, just the type. Examples below illustrate this.

Declaration examples:

```
// Allocates 10 integers but doesn't initialize them -- they could contain anything now
int an_array[10];
// Allocates space for 10 integers and only assigns the first 4
int an_array[10] = {1,2,3,4};
// Infers the length based on the data you provided
int an_array[] = {1,2,3,4,5};
Usage examples:
```

```
// Set the first value
an_array[0] = 5;

// Loop through an array
int i;
for(i = 0; i < sizeof(an_array)/sizeof(an_array[0]); i++){
    an_array[i] = 5*i;
}

// Use an array as a function argument
void func_using_array(int *that_array, uint32_t len){
    int *iter;
    // Another way to iterate using pointers (next section)
    for(i = that_array; i < &that_array[len]; i++){
        *iter = 5;
    }
}

// Call it
func_using_array(an_array, 10);
```

Listing 6: Arrays

3.1.4 Pointers

Pointers are a tricky feature for new C programmers coming from higher-level languages such as Java but are an extremely powerful tool for many complex operations. A pointer is actually an integer type of sorts which contains the address in memory of another object. Some confusion is about the syntax. The C implementation of pointers introduces two new modifiers for any variable: First "`&`" is synonymous with "the address of". Second "`*`" as a modifier is synonymous with "at the address of". "`*`" used as a type modifier (`int *`) doesn't quite mean the same thing. It means that we are referring to a pointer to a cell of that type.

```
int a_number = 1;
int * reference = &a_number; // pointer to int .. = the address of a_number
*reference = 5; // At the address of reference = 5
// a_number is 5 now
```

Listing 7: Pointer syntax

This creates an integer and assigns the value 1 to it, then creates a second value which refers to the address in memory of that value. In this simple example the value of a pointer isn't all that apparent, but consider the following more practical example illustrating how to efficiently pass arguments of arbitrarily large size:

```
// Declare a type that is way too big to pass
typedef struct { uint32_t header; uint32_t numbers[500]; } big_type_t;
big_type_t big_object;

// Use a pointer to that object to perform operations in place
void do_something_big(big_type_t *big){
    big->header = 50;
    // Probably do other stuff here
    // Then don't worry about returning anything
}
// Call it, passing a reference to that object
do_something_big(&big);
// Now that function has affected the data in place and we can use it again
```

Listing 8: Pointers as parameters

Pointer arithmetic is another tricky point. As mentioned before, pointers are integers internally, referring to an address. For arrays where we know the relative address of other items that may be of interest, we can do arithmetic on the pointer to access these efficiently. For the example of an integer array in the last section, this was done to move a pointer through an array. The type is 4 bytes long but in the example I only incremented by one for each cell. This is because the jump size is aligned to the data size.

The relationship between pointers and arrays is close. The explicit value of an array when it is declared is actually the pointer to its first item. If you pass the array to a function, you are passing it as a pointer to the first item. Just as arrays are treated as pointers, pointers can be treated like arrays. Given `int * a`, the following pairs are equivalent:

```
a[0] = 5; // The first value at a
*a = 0; // The data pointed to by a

a[1] = 4; // The second value in array a
*(a+1) = 4; // The data pointer to by the a incremented to the next cell
```

Listing 9: Pointers as arrays

A common problem with using pointers is that they allow use of memory that may not belong to the program. Operating systems and some hardware may catch onto this and kill the application (segmentation fault...). Therefore, you must be very careful with your pointers that you know that the address is always in the right place before you use it. This can be tricky to say the least, but if you can keep track of your pointers, you will unlock a very powerful tool.

A useful tool that often goes alongside pointer manipulation is the `sizeof` built-in function. This looks at the size of the type at compile time and provides it as an unsigned integer. For a fixed-size array declared in the same context `(sizeof(the_array)/sizeof(*the_array))` is an easy way to access the length of the array. It is the size of the array divided by the size of an element of the array. This was used in Listing 6.

3.1.5 Strings

String types don't really exist in C because they would generally require arbitrarily sized memory. Their construction is left to the programmer. We usually use the type `char *` (or equivalently `char []`) to denote string types. More generally, they are arrays of characters. As mentioned before, arrays don't explicitly have a length. In this array implementation, the length is denoted by having the last character be 0. Not the ASCII value '0', but actually the number 0 (these are 8-bit integers after all).

```
char a_string[] = "this␣is␣a␣string";

// Return the length of a string
uint32_t strlen(const char *str){
    int count = 0;
    // If this isn t actually a string, this loop could go forever!
    while(*(str++) != 0){
        count += 1;
}
return count;
}
```

Listing 10: String basics

3.1.6 Floating point types

The C programming language supports the IEEE754 standard for floating point numbers. Floating point numbers have the benefit of a non-uniform distribution across the real number space and allocates a lot more possible values closer to zero. Internally it is written like scientific notation so it has a much greater range as

a result. The two types available to support this are `float` and `double`. float is a 32-bit "single-precision" floating point number, while a double is 64-bit "double-precision" value. While most PC processors (and fancier ARM processors) support double-precision, the chips we use don't. If you use them, your arithmetic will be implemented in software at great computational cost. One common pitfall with floating point numbers is when comparing for equality. The number format stores only an approximation of any given value and due to numerous factors in computation, will likely have some degree of error. The == comparison operation doesn't accept even the most subtle of differences. Instead, it is common to compare based on a thresholded absolute value of the difference of two numbers.

```
#define ABS(x) ((x < 0) ? -x : x)

int float_compare(float a, float b, float threshold){
    if(ABS(a - b) > threshold)
        return 1;
    return 0;
}
```

Listing 11: Float comparison

3.1.7 Enumerated Types

C allows for types associated to human-readable names, called `enum`s or enumerated types.

```
enum {
    VALUE1 = 0,
    VALUE2,
    VALUE3
} enum_value;

enum_value = VALUE1;
```

Listing 12: Enumerated types

Internally, they are integers and can be treated as such if you are careful. If forced to boolean in a conditional statement, they are treated as an integer and evaluated on their equivalence to 0 (see If-else). The assignment of equivalent integer vales as in the example above is entirely optional. Further, it's guaranteed that the first value will be the number 0. There are still many applications where a mapping from string to integer is valuable and so it may be useful to provide integers for all of your labels. The length of the type is dependent on the number of options you provide.

This is a great alternative to using `#define` macros for enumerated values. Unfortunately, they share a namespace and must be used carefully. By convention, both use entirely uppercase letters. Further, you should prefix each with a relevant and unique contextual identifier. For example, when using an enumerated type for accelerometer data rate options:

```
typedef enum {
    ACC_ODR_0_5 = 0x0,
    ACC_ODR_10  = 0x1,
    ACC_ODR_50  = 0x2,
    ACC_ODR_100 = 0x3
} acc_odr_t;
```

Listing 13: Unique naming of enumerated types

3.1.8 Structured Types

C allows creation of types of arbitrary compound types in the form of the `struct`. A structure contains a list of fields to be placed in contiguous memory and each accessed by a string identifier. This allows for better organization than using arrays for some tasks, making it easier to see what's going on.

```
struct a_struct_type {
    int a,b;
    char name[50];
};
struct a_struct_type my_struct = {
    .a = 1,
    .b = 5,
    .name = "Some␣person"
};
```

Listing 14: Structures

The above example defines a `struct` type and shows how to initialize the data inside. This initialization method is only available as of C99 (make sure that option is set in Keil!) and is much more neater than it used to be.

The example created a new `struct` type called `struct a_struct_type`. It's a bit annoying to type that out every time, so I suggest defining a real type using `typedef`.

```
typedef struct {
    int a,b;
    char name[50];
} a_struct_type_t;
```

Listing 15: Typedef structure

It can now be used as the type `a_struct_type_t`. One problem with this method is when creating structures that reference their own type, as in a linked list. Since the type isn't defined to the compiler until after the `typedef` statement is complete, the type can't be used inside the structure. This must be handled by assigning a `struct` name as well.

```
typedef struct _ll_node {
    int              data[50];
    struct _ll_node* next;
} ll_node_t;
```

Listing 16: Recursive structures

Elements in a structure are usually accessed as `struct_name.element_name` but the syntax is different when using a pointer to a structure instead of the structure itself. In that case, they are accessed using an arrow (`->`) instead of a dot.

```
typedef struct {
    int a ,b;
} a_struct_t;
a_struct_t struct_instance;
a_struct_t* struct_pointer = &struct_instance.
struct_instance.a = 1;
struct_pointer->b = 5;
```

Listing 17: Structure member access

Bit fields For some (particularly embedded) applications it may be useful to specify the exact length of fields to be arbitrary length. This is useful when storage is tight, for example inside the header of wireless packets. This can be done manually with masking but are a supported feature of `structs` in C, which can make life a lot easier.

```
typedef struct {
    uintmax_t packet_type : 5;
    uintmax_t num_bytes   : 6;
    uintmax_t subtype     : 2;
    uintmax_t seq         : 19;
} a_packed_struct_header_t; // Total length is 32-bits
```

Listing 18: Structure bit fields

The total length can be anything but no guarantees are made by the compiler about the alignment of the structure. To align it and fix the length, I often declare them in a `union` with an integer array of the desired length. This also is often useful for applications that will involve individual byte-transfers to and from peripherals.

The bit field length has to be smaller than the declared type. I used `uintmax_t` here simply because it's faster to type than `unsigned int` and I do want to ensure that the type is unsigned.

3.1.9 Casting

Any type can be viewed as another type by using casting. This can be dangerous if not used carefully but is also very powerful. Casting is done by prefixing an expression with `(new_type_name)`. For integer and floating point types, a conversion is applied to represent roughly the same number in the new type. Other types simply consider it a new view on the same data.

```
float data[20];
uint8_t *iter;
for(iter = (uint8_t *) data; iter < &data[20]; iter++){
    some_peripheral_buffer = *iter;
}
```

Listing 19: Casting for byte-wise iteration

The above example moves a buffer of floating point data byte-by-byte to a peripheral which expects to receive bytes in the order that they may be transmitted.

3.1.10 Unions

Sometimes it's useful to have an area in memory with methods for access. This happens a lot in embedded applications where bit- and byte-level manipulation must be done to move data between peripherals. The `union` type allows for a single region of memory to be viewed as multiple types simultaneously without having to deal with casting, which can get messy in some cases.

```
union {
    float data[20];
    uint8_t bytes[80];
} float_buffer;
int i;
// Assign data as a float
float_buffer.data[0] = 0.5;
// ...fill other float buffers
for(i = 0; i < sizeof(float_buffer.data)/sizeof(*float_buffer.data); i++){
    // Copy each byte to a peripheral
    some_peripheral_buffer = float_buffer.bytes[i];
}
```

Listing 20: Unions for byte-wise iteration

In the example above, a `union` is used to provide byte-level access to an array of floats. Proper use of this provides a much better and safer solution than repeated explicit casting.
The union type can have any number of types in it and takes the size of its largest member.

3.1.11 Functions

Functions are simply modular ways to define actions to be performed. It may seem odd to include functions in the outline of different data types but it is destructive to think of them in any other way. That's what they are. A function is the starting address of some code to execute combined with a prototype explaining what the compiler should do with arguments and return values. They are just another type of data and can be passed around like anything else.
Now surely you've seen plenty of examples of functions right now, but I'll explain a few ways to deal with arguments and return values neatly:

- While the function parameters can be anything, it's preferred to keep them small. When calling a function, values are duplicated so that the called function has its own copy. If this is done on a large object, this is inefficient. The common way to deal with this is by passing a pointer to the object as seen in previous examples. Since the caller and callee are dealing with the same data, any changes made by the callee are seen by the caller.
- You only get one return value. This isn't always enough for all arrangements. Take the case where the function returns a buffer but the caller also needs to know the length. Since they can't be returned together, a common technique is to also pass a pointer to an int where that buffer size value can be stored.

3.2 Modifiers

When declaring variables and functions, you can specify a number of options to modify storage properties or compiler hints to improve performance or safety.

const

This modifier can be applied to any variable and indicates that the value is not to be changed. When declaring a const value, its value must be initialized. In addition to adding safety, it allows a few optimizations. In pointer types, this can be tricky to apply and refers to the type to its left unless there is nothing to the left.

```
// Create a constant integer with the value 5
const int an_int = 5;
// Same as above
int const another_int = 5;
const int *an_int_pointer = &an_int;
// Const pointer referring to constant int
const int * const another_pointer = &an_int;
// This is the same as above
int const * const yet_another = &an_int;
```

Listing 21: Constant declarations

static

This modifier carries two independent implications depending on the usage. When used for a function or global variable, this limits the visibility to the current file only. This is good for creating functions and

variables that are only used internally and arent to be called by anything else. This allows use of more generic names by avoiding symbol name collisions.
When used for a variable inside a function, static indicates how the variable is stored. Rather than being allocated on the stack (if inside a function), it will be statically allocated. This implies that the memory will persist across function calls, regardless of the context. This allows function-local persistent variables.

register

This hints to the compiler that it should try to keep this in a register if possible. While optimizing compilers are supposed to take care of this in a lot of cases, explicitly stating it where relevant can be of great value.

volatile

This tells the compiler that the value may change in the background without any indication. Any operation on the variable should be immediately written back and must be re-fetched on every read. This reduces the amount of optimization that may be done but is required for safety in many low-level applications where interrupts and peripherals can change values all the time.
Recklessly omitting this modifier when it should be used can very often lead to incorrect operation of your program.

restrict

This can only be applied to function arguments which are pointers and hints to the compiler that this function is the only code that may be modifying that memory. This allows for it to optimize memory operations. This is only available since C99 so make sure Keil is configured as such.

extern

Extern declares a variable or function without actually allocating it. This implies that it is in another file or sill somehow be found by the linker later on. Of course if it isnt found, the linker will throw an error about not being able to find a symbol. Be careful to make sure that you declare it to be the same type or function prototype, as the linker wont catch that!

inline

This asks the compiler to inline the function where possible. This essentially duplicates the function wherever it is called and integrated into the caller code to avoid calling overhead. This is a great alternative to using macros as functions. In most tools, functions will only be inlined if they are in the same file. A way around this is to write the function in a header so that every file has the function included by the preprocessor.
To force the compiler to inline (or throw an error if it cant) is to declare the function extern inline.

__attribute__( )

This isn't really a modifier on its own, but rather refers to a collection of compiler-specific modifiers that may specify all sorts of interesting stuff. Check the compiler manual for details.

4 Flow Control Mechanisms

4.1 Conditionals

4.1.1 If-else

As in most languages, the basic conditional structure is based on the `if` statement. The condition can be a pure boolean (ie the result of a comparison) or can simply be an integer, in which case the boolean value is determined based on equivalence to zero. All negative and positive values will evaluate to true. The `else` and `else if` sections are both independently optional. Not all cases have to be covered.

```
if(i > 5){
    // Do something
} else if(i > 2){
    // Do something else
} else {
    // Do something else yet
}
```

Listing 22: If-else structure

4.1.2 Ternary Conditionals

A shorthand way of assigning a value inside an if-else block is called a ternary conditional and looks like `((condition) ? result_if_true : result_if_false)`. These are computationally the same but may be more convenient in some cases. See Listing 11 for an example.

4.2 Loops

do-while

The do-while loop is a simple loop where a block of code is to be repeated depending on a condition that is evaluated at the end of each iteration. In this way, it is guaranteed that one iteration of the loop be executed.

```
do {
    // Some repeated operation
} while(<cond>)
```

Listing 23: Do-while loop

while

The loop can be altered to have the condition evaluated at the beginning, therefore allowing the possibility that the loop never execute.

```
while(<cond>){
    // Some repeated operation
}
```

Listing 24: While loop

for

The for-loop is a more concise way of writing many common looping patterns, as it allows initialization and an increment operation to be written into the loop construct.

```
<init>;
while(<cond>){
    <execute_this_each_time>;
    <do_at_end>;
}

// Written short-hand as a for-loop
for(<init>; <cond>; <do_at_end>){
    <execute_this_each_time>;
}
```

Listing 25: Generic for-loop

There is a bit more to the for-loop than that though. The parameters can take a bit of a different form than is normally allowed. Since semicolons have been taken here as the delimiter, we could normally only have one initialization statement or condition. In this case, we use commas and the following is valid:

```
for(<init>, <init2>; <cond>, <cond2>; <do_at_end>, <also_do_at_end>){
    <execute_this_each_time>;
}
```

Listing 26: More complex for-loop

It should also be noted that `<init>` and `<do_at_end>` can both be blank in the first example.

break

There are times when you might want to escape from a loop at a point other than the conditional evaluation in your selected loop type. For this the `break` statement is available. It may be executed anywhere within a loop and will result in execution continuing just after the end of the current loop. If loops are nested, only the outermost loop will be broken.

switch

The `switch` statement allows conditional execution based on specific values of a single integer (including `char` and `enum`) variable. It can often be implemented more efficiently than if-else statements and can also be preferable for clarity.

```
switch(<variable>){
case <some_value>:
    <to execute if <variable> == <some_value>>;
    break;
case <some_other_value>:
    <to execute if <variable> == <some_other_value>>;
    break;
default:
    <to execute if no others match>;
}
```

Listing 27: Basic switch

For most operations, a `break` statement is required at the end of each case to exit it. That's because if it's omitted the execution will continue through the next case. Be careful with this! There are lots of times when this is very useful but you can get some serious unexpected behavior if you omit by accident.

Most compilers will throw a warning or error if you don't provide complete coverage of all cases. With integers, this usually means that you provide a `default` but with `enum` you can get away with just coverage of the defined values. Since switches are often optimized, be careful in those cases to be sure that you provide a valid `enum`.

5 Libraries

6 Conventions

There are lots of variable, function, and file naming conventions out there that are intended to improve program readability but the most important thing is to be consistent.

6.1 Project Organization

There is a lot of confusion regarding what should go where in a project. There are many ways to organize a project and there are no clear black-and-white rules so use reasonable judgement to apply the following overview to your particular application. For the purpose of this class, you don't have to follow this exactly but expect any deviation to require substantial justification. That is, unless you have a really good reason, do not deviate from this.

6.1.1 Main

Your *main* file contains primarily the entry point to your application, `int main();`. Small support functions (eg. system-timing-related or functions that break your `main` into smaller chunks based on mode of operation) may also exist here but they should only be applicable to this high-level view of your application. No shared components should exist in your main file. Any declarations should not be public to other files. There is no reason to have a *main.h* file.

6.1.2 Modules

A good, readable, and reusable project will be broken into separate modules based on functionality. A module typically consists of a *.c* file and a *.h* file. It will contain any number of the following:

Public functions: Functions declared in header file and implemented in *.c* file. These functions are only those that must be called from another file (and thus need the header declaration). File names must be indicative of module membership (ie. `void adc_init()` for initializing an ADC module. Documentation should appear in the header file above the prototype.

Private functions: Functions declared (prototype) at the top of the *.c* file as `static`, implemented in the same file, and only referenced locally (ie. `static void update_value()`). The `static` modifier affects linkage so that no other file has access to it. Documentation should be in the file at the function declaration.

Public type definitions: Typedefs that *need* to be accessed by other files. They should be named in a way that indicates module membership (ie. adc_config_t for an ADC configuration structure). Documentation for these should appear in the header file alongside the type definition.

Private type definitions: Types that are declared for use only locally. Declare them in the top of your *.c* file and document them at the declaration.

Public variables: Global variables that need to be accessed from other modules. Declare these `extern` in the header file and then regularly in your *.c* file. That way other files are aware that it exists but will not recreate the variable locally. Document them next to the `extern` declaration in the header.

Private variables: Local variables that don't need to be accessed from other modules. Declare them `static` at the top of your *.c* file and document them there as well.

Public macros: Macros that need to be accessed from other modules. Declare them in a header, document them by the declarations, and make sure they are named appropriately for a global macro.

Private macros: Macros solely for local use. Declare and document them at the top of the *.c* file.

6.1.3 Interrupts

Interrupt handlers cannot be declared `static` (must be global) and require prototypes only for documentation purposes. Therefore just be careful to place them in the correct file and declare and document them as you would a private function (except not `static`).

6.2 Indentation

There are many styles of indentation. Pick one and be consistent. The two basic camps are tabs-only and spaces-only. I recommend a third camp using tabs for indentation (each indentation level gets one tab) and using spaces for character level alignment (mostly for multi-line statements). This guarantees that it will look correct on any machine regardless of the tab size set in the editor. I'm not saying this is the best way, but it's the best way. I stand by this for every language as well.

6.3 Line Length

It is common to not allow lines to extend past 80 characters in length. This ensures that there is no wrapping done in small windows. It's not always necessary to be so strict, but you should aim to keep your lines short and readable. If a function call, for example, is going out too long, try to break it down on argument boundaries in a way that preserves readability. Give all arguments equal treatment and align them. If it seems reasonable, perhaps group arguments together logically. You can even break up a single expression if it breaks logically (for example if you are adding a bunch of expressions together). There aren't really rules on the matter so just use whatever you think looks readable.

```
int some_really_long_function_name(int a,
                                   uint8_t *buff1,
                                   uint8_t *buff2,
                                   uint32_t *result);

// Call that function
a_variable = some_really_long_function_name(another_variable,
                                            first_buff, second_buff,
                                            result);
```

Listing 28: Limiting line length

6.4 Documentation

Learn Doxygen. Even if you don't use the generated output, it is a great human-readable format as well. As your projects grow and you perhaps do this in the working world, everybody around you (including bosses who want good high-level overviews that are well presented) will appreciate it. Further, you will do a better job of keeping track of your work in larger projects. Here's a basic tutorial to get started with. I highly recommend that everybody start doing this in this class. We TAs will appreciate it and all think that you will benefit greatly from it. If nothing else, it's a surprisingly powerful skill to put in at the bottom of a CV for software development.

7 Useful Design Elements

7.1 Queue

There are many asynchronous elements to embedded programming and queues help to handle that. Operating system constructs can really help here but in their absence we can still do quite a bit.

Linked List

This operation is great when sending data to peripherals. The code in the program provides the memory for each node in the list so dynamic allocation isn't necessary.

```
// Declare the type for each node. The type could include more than an int...
typedef struct _ll_node_s {
    int32_t data;
    // A flag for the caller to know that the data has been processed
    int32_t has_been_processed
    struct _ll_node_s *next;
} ll_node_t;

// This is my list. Rather, it's just a pointer to the head of the list.
static ll_node_t *my_list = NULL;

/*!
 @brief Add an item to the back of the queue
 @param node A pointer to the node
 */
void ll_enqueue(ll_node_t *node){
    ll_node_t *iter = my_list;
    if(iter){
        // Need to attach this to another node
        while(iter->next) iter++;
        iter->next = node;
    } else {
        // No existing node
        my_list = node;
    }
    node->next = NULL;
}

/*!
 @brief Dequeue an item from the front of the queue
 @retval Pointer to the item; NULL if empty
 */
ll_node_t *ll_dequeue(){
    ll_node_t *iter = my_list;
    if(iter == NULL){
        return NULL;
    }
    // iter is now the node to return
    // Dereference it from the list
    my_list = iter->next;

    // Return it
    return iter;
}
```

Listing 29: Linked list queue

Array (Circular Buffer)

The array implementation is better suited for having interrupts provide data for the application code to handle asynchronously. Since a bank of memory is allocated for it in advance, this will work until the queue runs out of memory.

```
#define BUFFER_ADDR_BITS 6
#define BUFFER_LEN       (1 << BUFFER_ADDR_BITS)

// Declare the type for each node. The type could include more than an int...
typedef struct {
    int32_t data;
    int32_t data2;
} node_t;

// This is my list. Rather, it's just a pointer to the head of the list.
typedef struct {
    node_t data[BUFFER_LEN];
    uint32_t rd_head, wr_head;
} circular_buffer_t;

void queue_init(circular_buffer_t* buffer){
    buffer->rd_head = 0;
    buffer->wr_head = 0;
}

/*!
 @brief Add an item to the back of the queue
 @param node A pointer to the node
 */
int enqueue(circular_buffer_t *buffer, node_t *node){
    node_t *tmp;
    __disable_irq();
    if((buffer->wr_head + 1) & (BUFFER_LEN - 1) == buffer->rd_head){
        __enable_irq();
        return 0;
    }

    tmp = &buffer->data[buffer->wr_head];

    // Copy data
    tmp->data = node->data;
    tmp->data2 = node->data2;

    buffer->wr_head = (buffer->wr_head + 1) & (BUFFER_LEN - 1);

    __enable_irq();
    return 1;
}

int dequeue(circular_buffer_t *buffer, node_t * restrict dst){
    __disable_irq();
    node_t *tmp;
    if(buffer>wr_head == buffer->rd_head){
        __enable_irq();
        return 0;
    }

    tmp = &buffer->data[buffer->rd_head];

    // Copy the data
    dst->data = tmp->data;
    dst->data2 = tmp->data2;

    buffer->rd_head = (buffer->rd_head + 1) & (BUFFER_LEN - 1);
```

```
    __enable_irq();
    return 1;
}
```

Listing 30: Circular buffer queue

7.2 Finite-State Machine

In a predominantly interrupt-driven application, it's common to see something like this:

```
int main(){
    // Do hardware initialization

    // Wait
    while(1);
    // Or in many cases, put the processor to sleep to be woken by interrupts
}
```

Listing 31: Interrupt driven system

For more complex applications, you may have some low-priority code that you want to execute in the background as well. For a lot of these cases, it makes sense to use a state machine inside of an infinite loop to handle this.

```
static enum {
    //! Brief startup routine
    ST_INIT,
    //! Waiting for action
    ST_IDLE,
    //! An error has occurred
    ST_ERROR,
    //! Currently running
    ST_RUNNING
} state;

static volatile uint32_t tick;

int main(){
    state = ST_INIT;
    tick = 0;
    // Do hardware initialization

    while(1){
        switch(state){
        case ST_INIT:
            // Wait for a button press
            while(!button_was_pressed);
            state = ST_IDLE;
            break;
        case ST_IDLE:
            // Nothing is happing; Just lay low
            led_set(0);
            if(bytes_received){
                state = ST_RUNNING;
                break;
            }
            break;
        case ST_ERROR:
            while(1){
                flash_led();
            }
            break;
```

```
        case ST_RUNNING:
            while(1){
                wait_for_tick();
                led_toggle();
                // Perform some actions
                if(something_bad_happened){
                    state = ST_ERROR;
                    break;
                }
                if(button_was_pressed){
                    state = ST_IDLE;
                    break;
                }
            }
            break;
        }
    }
}

void wait_for_tick(){
    while(!tick);
    tick--;
}

void SysTickHandler(void){
    tick = 1;
}
```

Listing 32: Finite-state machine