

Chapter 1

Concepts for Developing Portable Firmware

“A good scientist is a person with original ideas. A good engineer is a person who makes a design that works with as few original ideas as possible. “

– Freeman Dyson

Why Code Reuse Matters

Over the past several decades, embedded systems have steadily increased in complexity. The internet’s birth has only accelerated the process as our society has been in a race to connect nearly every device imaginable. Systems that were once simple and stand-alone must now connect through the internet in a secure and fail-safe manner in order to stream critical information up into the cloud. Complexity and features are increasing at an exponential rate with each device generation forcing engineers to reevaluate how to successfully develop embedded software within the allotted timeframe and budget.

The increased demand for product features along with the need to connect systems to the internet has dramatically increased the amount of

Developing Reusable Firmware

software that needs to be developed to launch a product. While software complexity and features have been increasing, the time available to develop a product has for the most part remained constant with a negligible increase in development time (2 weeks in 5 years) as can be seen in Figure 1. In order to meet project timelines, developers are forced to either purchase commercial off-the-shelf (COTS) software that can decrease their development time or they need to reuse as much code as possible from previous projects.

Firmware for microcontrollers has conventionally been developed for a specific application using functional design methodologies (if any methodology has been used at all) that typically tie the low level hardware directly into the application code, making the software difficult if not impossible to reuse and port on the same hardware architectures let alone reuse on a different architecture. The primary driving factor behind developing throw-away firmware has been the resource constrained nature many embedded products exhibit. Microcontrollers with RAM greater than a few kilobytes and flash sizes greater than 16 kB were once expensive and could not be designed into a product without destroying any hope for making a profit. Embedded software developers did not have large memories or powerful processors to work with which prevented modern software design techniques from being used in application development.

Modern microcontrollers are beginning to change the game. A typical low end ARM Cortex-M microcontroller now costs just a few U.S. dollars and offers at a minimum 16 kB of RAM and 64 kB of flash. The dramatic cost decreases in memory, larger memory availability and more efficient CPU architectures is removing the resource constrained nature that firmware developers have been stuck with. The result is developers can now start utilizing design methods that decouple the application code from the hardware and allow a radical increase in code reuse.

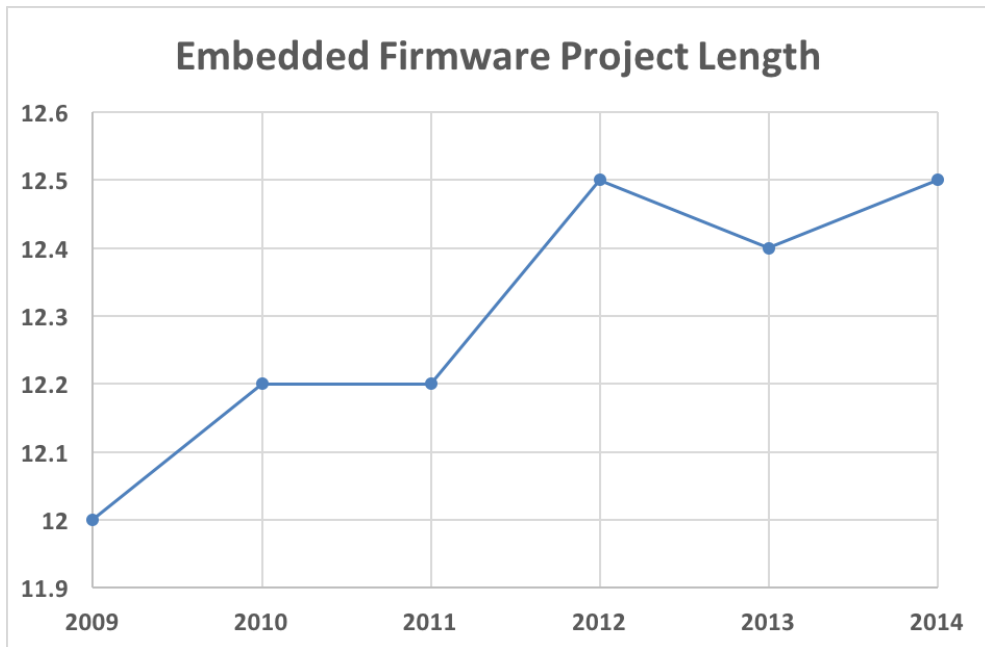


Figure 1 – Average Firmware Project Development Time (in months)¹

Portable Firmware

Firmware developed today is written in a rather archaic manner. Each product development cycle results in limited to no code reuse with reinvention being a major theme amongst development teams. A simple example is when development teams refuse to use an available real-time operating system (RTOS) and instead, develop their own in-house scheduler. Beyond wanting to build their own custom scheduler, there are two primary examples that demonstrate the issue with reinvention.

First, nearly every development team writes their own drivers due to the fact that microcontroller vendors provide only example code and not production ready drivers. Examples provide a great jump-start to understanding the microcontroller peripherals but still requires a significant time investment to get a production intent system. There could be a hundred companies using the exact same microcontroller and each and every one will waste as much as thirty percent or more development time getting

Software Terminology

Portable firmware is embedded software that is designed to run on more than one microcontroller or processor architecture with little or no modification.

their microcontroller drivers written and integrated with their middleware! I have seen this happen repeatedly amongst my client base and heard numerous corroborating stories from the hundreds of engineers I interact with on a yearly basis.

Second, there are so many features that need to be packed into a product and with a typical design cycle being twelve months¹, developers don't take the time to properly architect their systems for reuse. High level application code becomes tightly coupled to low level microcontroller code which makes separating, reusing or porting the application code costly, time consuming and buggy. The end result, developers just start from scratch every time.

In order to keep up with the rapid development pace in today's design cycles, developers need to be highly skilled in developing portable firmware. Portable firmware is embedded software that is designed to run on more than one microcontroller or processor architecture with little to no modification. Writing firmware that can be ported from one microcontroller architecture to the next has many direct advantages such as:

- Decreasing time-to-market by not having to reinvent the wheel (which can be time consuming)
- Decreasing project costs by leveraging existing components and libraries
- Improved product quality through using proven and continuously tested software

Portable firmware also has a number of indirect advantages that many teams overlook but can far outweigh the direct benefits such as:

- More time in the development cycle to focus on product innovation and differentiation
- Decreased team stress levels due to limiting how much total code needs to be developed (Happy, relaxed engineers are more innovative and efficient)
- Organized and well documented code that can make porting and maintenance easier and more cost effective

Using portable and reusable code can result in some very fast and amazing results as seen in the case study, “Firmware Development for a Smart Solar Panel”, but there are also a few disadvantages. The disadvantages are related to upfront time and effort such as:

- The software architecture needing to be well thought through
- Understanding potential architectural differences between microcontrollers
- Developing regression tests to ensure porting is successful
- Selecting real-time languages and understanding their interoperability or lack thereof
- Experienced and high-skilled engineers being available to develop a portable and scalable architecture

For development teams to successfully enjoy the benefits of portable code use, extra time and money needs to be spent up front. However, after the initial investment, development cycles have a jump start to potentially decrease development time by months versus the traditional embedded software design cycle. The long term benefits and cost savings usually overshadow the upfront design costs along with the potential to speed up the development schedule.

Developing firmware with the intent to reuse also means that developers may be stuck with a single programming language. How does one choose a language for software that may stick around for a decade or longer? Using a single programming language is not a major concern in embedded software development as one might initially think. The most popular embedded language, ANSI-C, was developed in 1972 and has prov-

Developing Reusable Firmware

en to be nearly impossible to usurp. Figure 2 shows the popularity of programming languages, for all uses and applications, dating back to 2002. Despite advances in computer science and the development of object oriented programming languages, C has remained very popular as a general language and is heavily entrenched in embedded software.

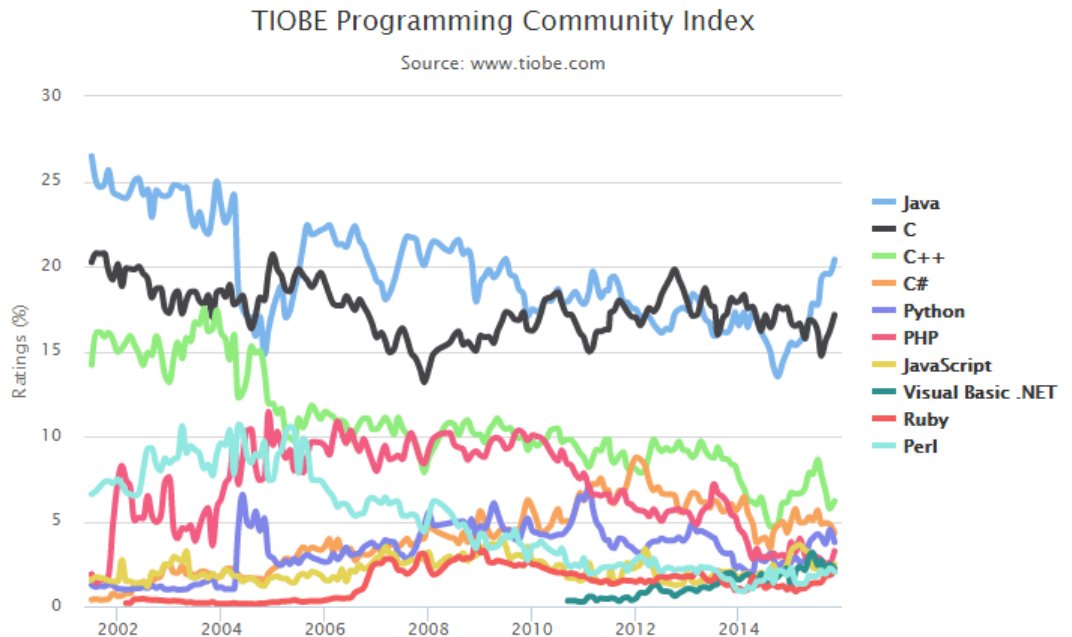


Figure 2 – TIOBE Computer Programming Index²

The C programming languages' popularity and steady use for decades doesn't appear to be changing anytime soon. When and if the Internet of Things (IoT) begins to gain momentum, C may even begin to grow in its use and popularity as millions of devices are developed and deployed using it. Developing portable and reusable software becomes a viable option when one considers the steady and near constant use that the C language has enjoyed in industry for developing embedded systems. When a development team considers the timelines, feature needs and limited budgets for the product development cycle, developing portable code should be considered a mandatory requirement.

Case Study – Firmware for a Smart Solar Panel

When it comes to product development, the single constant in the universe is that the development either needs to be done yesterday or by some not so distant future date. A few years ago on December 1st, I received a call from a prospective client I had been talking with for a better part of the year. The client, a start-up in the small satellite industry, had just received news that they had an opportunity to fly their new flagship spacecraft on an upcoming launch. The problem was that they had just six weeks to finish building, testing and then deliver their satellite!

One of the many hurdles they faced was that their smart solar panels, smart because they contained a plethora of sensors critical to stabilizing the spacecraft, didn't have a single line of firmware written. The solar panels firmware had to be completed by January 1st, leaving just four weeks over a holiday month to design, implement, test and deploy the firmware.

To give some quantification to the project scope, the following are some of the software components that needed to be included:

- GPIO, SPI, I2C, PWM, UART, Flash, ADC
- Timer and system tick
- H-bridge control
- Task scheduler
- Accelerometer
- Magnetometer
- Sun sensor
- Calibration algorithms
- Fault recovery
- Health and wellness monitoring
- Flight computer communication protocol

An experienced developer knows the above list would be impossible to successfully complete in four weeks from scratch. I2C alone could take two weeks to develop and the realistic delivery date for the project would be 3-4 months not weeks.

I accepted the project and leveraged the very same HAL and driver techniques presented in this book to complete the project. A day was spent pulling in existing drivers and making minor modifications for the microcontroller derivative. The second week was pulling together the application code and remaining drivers. Finally, week three was test, debug and delivery. Just in time for Christmas and to the clients' delight.

Developing Reusable Firmware

The decision to develop portable firmware should not be taken lightly. In order to develop truly portable and reusable firmware, there are a few characteristics that a developer should review and make sure that the firmware will exhibit. First, the software needs to be modular. Writing an application that exists in a single source file is not an option (yes I still see this done even in 2016). The software needs to be broken up into manageable pieces with minimal dependencies between modules and similar functions being grouped together.

Portable software should follow the ANSI-C programming language

10 Qualities of Portable Firmware

Portable Firmware

- 1) is modular
- 2) is loosely coupled
- 3) has high cohesion
- 4) is ANSI-C compliant
- 5) has a clean interface
- 6) has a Hardware Abstraction Layer (HAL)
- 7) is readable and maintainable
- 8) is simple
- 9) uses encapsulation and abstract data types
- 10) is well documented

standard. Developers should avoid using compiler intrinsic and C extensions because they are compiler specific and will not easily port between tool chains. In addition to avoiding these add-ons, developers should select a safe and fully specified subset for the C programming language. Industry accepted standards such as MISRA-C or Secure C might be good options to help ensure that the firmware will use safe constructs.

Developers will want to make sure that the reusable code is also well documented and contain examples. The firmware needs to have a clean interface that is simple and easy to understand. Most importantly,

developers will want to make sure that a simple, scalable hardware abstraction layer is included in the software architecture. The hardware abstraction layer will define how application code interacts with the lower, underlying hardware. Let's examine a few key characteristics that portable firmware should exhibit in greater detail before diving into hardware abstraction layers.

Modularity

On more than one occasion over the last several years I have worked with a client whose entire application, fifty thousand plus lines of code, was contained within a single `main.c` module. Attempts to maintain the software or reuse pieces of code quickly turned into a nightmare. These applications were still using software techniques from back in the 70's and 80's which was not working out so well for my client.

Modularity emphasizes that a programs functionality be separated into independent modules that may be interchangeable. Each module contains a header and source file with the ability to execute specialized system functions that are exposed through the modules interface. The primary benefit for employing modularity in an embedded system is that program is broken up into smaller pieces which are organized together based on purpose and function.

Ignoring the facts above and lumping large amounts of code into a single module, even if it is well organized or makes sense in the beginning, usually results in a decay into a chaos and a software architecture that resembles spaghetti. Breaking a program up into separate modules is so important when developing portable and reusable firmware because the independence each module exhibits allows it to be easily moved from one application to the next or in some cases even one platform to the next. There are a few advantages associated with breaking a program up into modular pieces such as:

Developing Reusable Firmware

- Being able to find functions or code of interest very quickly and easily
- Improved software understanding through the modules organization
- The ability to copy modules and use them in new applications
- The ability to remove modules from a program and replace them with new functionality
- Easing requirements traceability
- Developing automated regression testing for individual modules and features
- Overall decreased time to market and development costs

Each module added to a program does come with the disadvantage that the compiler will need to open, process, compile and close the module. The result in the “old days” would have been slower compilation times. Development machines today are so fast and efficient that increased compile time is no longer an excuse for writing bulking, clunky code.

Module Coupling and Cohesion

Breaking a program up into smaller, more manageable pieces is a good step forward towards developing portable firmware but it is only the first step. In order for a module to be truly portable, it must exhibit low coupling to other modules within the code base and a high level of cohesion. Coupling refers to how closely related different modules or classes are to each other and the degree to which they are interdependent. The higher the coupling, the less independent the module is.

Portable software should minimize the coupling between modules in order to make it easier to use in more than one development environment. Take for example the file dependency chart in Figure 3a. Attempting to bring the top level module into the code base will be a small nightmare or like peeling an onion. The top module will be brought in, only for the de-

veloper to realize that it is dependent upon another, which is dependent upon another and another and so on. In short order, the developer might as well have just brought in the entire application or simply started from scratch. Attempting to use modules that are tightly coupled is very frustrating and can cause the code size to balloon out of control if care is not taken.

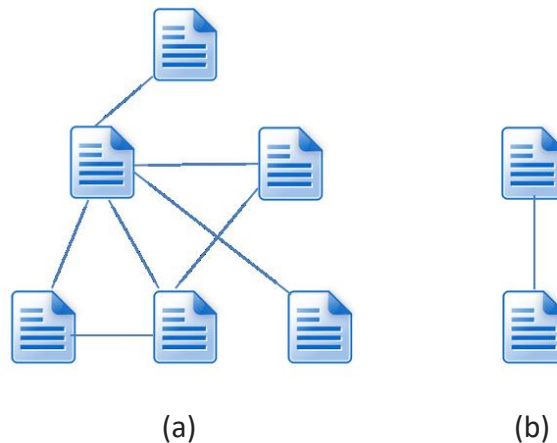


Figure 3 – Module Coupling

The software base in Figure 3b shows a completely different story. The modules in Figure 3b are loosely coupled. A developer attempting to bring in a top level module won't be fraught with continuous compiler errors of missing files and spend hours on end trying to track down all the dependencies. Instead, the developer quickly moves the loosely coupled module into the new code base and is on to the next task with little to no frustration. Low coupling is the result from a well thought out, and well-structured software design.

Software Terminology

Coupling refers to how closely related different modules or classes are to each other and the degree to which they are interdependent.

Cohesion refers to the degree in which module elements belong together.

Developing Reusable Firmware

Module coupling is only the stories first part. Having low module coupling doesn't guarantee that the software will exhibit easily portable traits. The desire is to have a module that has low coupling and high cohesion. Cohesion refers to the degree in which the module elements belong together. In a microcontroller environment, a low cohesion example would be lumping every microcontroller peripheral function into a single module. The module would be large and unwieldy. The microcontroller peripheral functions could instead be broken up into separate modules each with functions specific to one peripheral. The results would be benefits listed in the last section on modularity.

Portable and reusable software attempts to create modules that are loosely coupled and have high cohesion. Modules with these characteristics are usually easy to reuse and maintain. Consider what would happen in a tightly coupled system if a single module is changed. A single change would result in forcing changes in at least one other module if not more and could be time consuming to hunt down all the necessary changes. Failure to make the change or a simple oversight could result in a bug which in the worst case could cause projects delays and increased costs.

Following A Standard

Creating firmware that is portable and reusable can be challenging. For example, the C language has gone through a number of different standard revisions; C90, C99 and C11. In addition to the different C versions, there also exists non-standard language extensions, compiler additions and even language offshoots. In order to develop firmware that is reusable to the greatest extent possible, a development team needs to select a widely accepted standard version such as C90 or C99. The C99 version has some great additions that make it a good choice for developers. At the time of this writing there is limited support for C11 in firmware development and C11 is five years old! Adopting C99 is the best bet for following a standard.

The long term support for C and its general purpose use has resulted in language extensions and non-standard versions that need to be avoided. Using any construct that is not in the standard will result in specialized modifications to the code base that can obfuscate the code. Sometimes using extensions or an intrinsic is unavoidable due to optimization needs but we will discuss later how we can still write portable code in these circumstances.

In addition to using the C standard, developers should also restrict their use to well defined constructs that are easy to understand, maintain and fully specified. For example, standards such as MISRA-C and Secure-C exist to provide recommendations on a C subset that should be used to develop firmware. MISRA-C was developed for the automotive industry but the recommendations have proven to be so successful at producing quality software that other industries are adopting the recommendations.

Developers should not view a standard as a restriction but instead as a method for improving the firmware quality and portability that they develop. Identifying and following standard C dialects will take developers a long way in developing reusable firmware. Recognizing the need to follow the ANSI-C standard and having the discipline to follow it will take a development team towards creating embedded software that can be reused for years to come.

Portability Issues in C – Data Types

The most infamous and well known portability issues in the C programming language is related to defining the most commonly used data type, the integer. One needs only to ask a simple question to demonstrate a potential portability issue; What will be the value LoopCount contains when i rolls over to 0? The demonstration code that contains LoopCount can be found in of the loop iteration counter LoopCount shown in Figure 4 be when i rolls over to 0?

```
static int i = 0;
static LoopCount = 0;

for (i = 1; i != 0; i++)
{
    LoopCount = i;
}
```

Figure 4 – Integer Rollover Test

The answer could be 32,676 or 2,147,483,647. Both answers could be correct. The reason is that the storage size for an integer is not defined within the ANSI-C standard. The compiler vendors have the choice to define the storage size for the variable based on what they deem will be the most efficient and/or appropriate.

The storage size for an integer normally wouldn't seem like a big deal. For a code base an int will be an int so who cares? The problem surfaces when that same code is compiled using a different compiler. Will the other compiler also store the variable as the same size or different? What happens if it was stored as four bytes and now is only two? Perfectly working software is now buggy!

The portability issues arising from integers, the most commonly used data type, is solved in a relatively simplistic way. The library header file `stdint.h` defines fixed width integers. A fixed width integer is a data type that is based on the number of bits required to store the data. For example, a variable that needs to store unsigned data that is 32 bits wide doesn't need to gamble on int being 32 bits but instead, a developer can simply use the data type `uint32_t`. Fixed width integers exist for 8, 16, 32 and in some cases even 64 bits. Table 1 shows a list of the different fixed width integer definitions that can be found in `stdint.h`.

| Data Type | Minimum Value | Maximum Value |
|-----------|----------------|---------------|
| int8_t | -128 | 127 |
| uint8_t | 0 | 255 |
| int16_t | -32,768 | 32,767 |
| uint16_t | 0 | 65335 |
| int32_t | -2,147,483,648 | 2,147,483,647 |
| uint32_t | 0 | 4,294,967,295 |

Table 1 – Fixed Width Integers³

The library file `stdint.h` doesn't just contain the data types found in Table 1 but also a few interesting and less known gems. Take for example `uint_fastN_t` which defines a variable that is the fastest to process at least N bits wide. A developer can tell the compiler that the data has to be at least 16 bits but could be 32 bits if it can be processed faster using a larger data type. Another great example is `uintmax_t` which defines the largest fixed width integer possible on the system. A personal favorite is the `uintptr_t` which defines a type that is wide enough to store the value of a pointer.

Using `stdint.h` is an easy way to help ensure that embedded software integer types preserve their storage size no matter which compiler the code may be compiled on. A simple and safe way to ensure that integer data types are properly preserved.

Portability Issues in C – Structures and Unions

The C standards have some unfortunate ambiguities in the definition of certain language constructs; take for example structures and unions. A developer can declare a structure containing three members x, y and z as shown in Figure 5. As one might expect, when a variable is declared of type `Axis_t`, the data members will be created in the order x, y and z in memory; however, the C standard does not specify how the data members will be

Developing Reusable Firmware

byte aligned. The compiler has the option to align the data members in any way that it chooses. The result could be that x, y and z occupy contiguous memory or there could be padding bytes added between the data members that space the members by two, four, or some byte value that would be completely unexpected by a programmer.

```
typedef struct
{
    uint8_t x;
    uint8_t y;
    uint8_t z;
}Axis_t;
```

Figure 5 – Structure Definition

The unspecified structure and union behavior make it the developers job when porting the firmware to understand how the structure is being defined in memory and whether or not the structure is being used in such a way that adding padding bytes could affect the application behavior or performance. The structure could include padding bytes or even holes depending on the data type being defined and how the compiler vendor decided to handle the byte alignment.

Portability Issues in C – Bit Fields

The situation with structures get even worse when it comes to the definition of bit fields. Bit fields are declared within a structure and are meant to allow a developer to save memory space by tightly packing data members that don't occupy an entire data space. An example of using bit fields is to declare a flag within a structure that have a true or false value as can be seen in **Error! Reference source not found..**


```
typedef struct
{
    uint8_t x;
    uint8_t y;
    uint8_t z;
    uint8_t x_flag:1;
    uint8_t y_flag:1;
    uint8_t z_flag:1;
}Axis_t;
```

Figure 6 – Bit Field Definition

The problem with bit fields is that the implementation is completely undefined by the standard. The compiler implementers get to decide how the bit field will be stored in memory including byte alignment and whether or not the bit field is allowed to cross a memory boundary. Another problem with a bit field is that while they may appear to save memory, the resulting code to access the bit field may be large and slow which can affect the real-time performance for accessing them. The general recommendation when it comes to bit fields is that they are non-portable and compiler dependent and should be avoided for use in firmware that is meant to be reusable and portable.

Portability Issues in C – Preprocessor Directives

All preprocessor directives are not created equal. A developer will have different preprocessor directives available depending on whether GNU C, IAR Embedded Workbench, Keil uVision or any other compiler is used. ANSI-C has a limited number of preprocessor directives that are included in the standard and can be considered portable.

Compiler vendors have the ability to add preprocessor directives that are not part of the standard. For example, #warning is a commonly used

Developing Reusable Firmware

preprocessor directive that is not supported by C90 or C99! The `#error` preprocessor directive is part of the standard and `#warning` was added by compiler vendors in order to allow a developer to raise a compilation warning. Developers who rely heavily on `#warning` may port code to a compiler that doesn't recognize `#warning` as a valid preprocessor directive or may recognize it as having a different purpose!

A developer interested in writing portable code needs to be careful which preprocessor directives are used within the embedded software. The most obvious non-portable preprocessor directive is `#pragma` which can generally be considered to declare implementation-defined behaviors within an application. Using `#pragma` should be avoided as much as possible within an application that is expected to be ported to other tool chains.

Using `#pragma` or other specialized preprocessor directives and attributes cannot always be avoided without dramatically increasing code complexity and structure. One example where `#pragma` may be necessary is to specify an optimization that should be performed on an area of code. A developer in a similar situation can use compiler predefined macros and conditional compilation to ensure that the code is optimized and that if it is ever ported to another compiler an error is raised at compile time. Each compiler has its own set of predefined macros including a macro that can be used to identify the compiler that is in use. Figure 7 shows an example of a compiler defined macro that may be of interest to a developer.

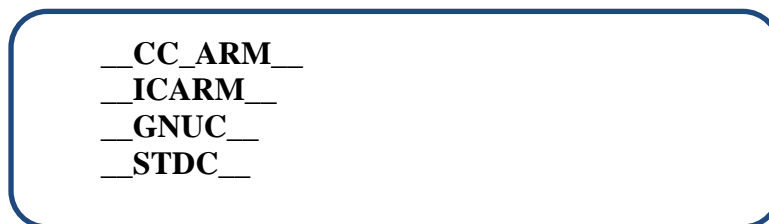
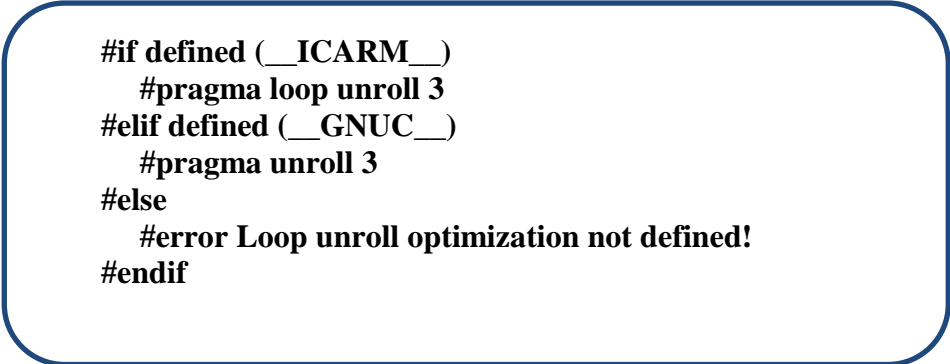


Figure 7 – Compiler Defined Macros

The predefined macros from Figure 7 that identify the compiler can be used as part of a preprocessor directive to conditionally compile code. Each compiler that may be used can then be added to the conditional statement with the non-portable preprocessor directive that is needed for the task at hand. Figure 8 shows how a developer might take advantage of the predefined compiler macros to conditionally compile a fictitious `#pragma` statement into a code base.



```
#if defined (__ICARM__)  
  #pragma loop unroll 3  
#elif defined (__GNUC__)  
  #pragma unroll 3  
#else  
  #error Loop unroll optimization not defined!  
#endif
```

Figure 8 – *Using Conditional Compilation for Non-Portable Constructs*

Developers interested in writing portable, ANSI-C code should consult the ANSI-C standard, such as C90, C99 or C11, and check the appendices for implementation defined behaviors. A developer may also want to consult their compiler manuals to determine the extensions and attributes that are available to developers.

Embedded Software Architecture

Firmware development in the early days used truly resource constrained microcontrollers. Every single bit had to be squeezed from both code and data memory spaces. Software reusability was a minor concern and programs were monolithically developed. The programs would be one giant fifty-thousand-line program all contained within a single module with little to no thought given to architectural design or reuse. The only goal was to make the software work. Thankfully, times have changed and while many microcontroller applications remain “resource constrained”, compiler

Developing Reusable Firmware

capabilities and decreasing memory costs now allow for a software architecture that encourages reuse.

Developing software that is complex, scalable, portable and reusable requires a software architecture. A software architecture is the fundamental organization a system embodies in its components, their relationship to each other, to the environment and the principles guiding its design and evolution⁴. In other words, a software architecture is the blueprint from which a developer implements software. A software architecture is literally analogous to the blueprint an architect would use to design a building or a bridge.

The software architecture provides a developer with each component and major software structure, supplies constraints on their performance and identifies their dependences and interactions (the inputs and outputs). For our purposes, we will only be looking at software architecture from the perspective of organizing firmware into separate software layers that have contractually specified interfaces to improve portability and code reuse. Each software has a specific function such as directly controlling the microcontroller hardware, running middleware or containing the systems application code. Properly architected software can provide developers with many advantages.

First, a layered architecture can provide a functional boundary between different components within the software. Take for example low level driver code that makes the microcontroller work. Including driver code directly within the application code tightly couples the microcontroller to the application code. Since application code normally contains algorithms that may be used across multiple products, mixing in low level microcontroller code will make it difficult and time consuming to reuse the code. Instead, a developer who architects layered software can separate the application and low level code, allowing both layers to be reused in other applications or on different hardware.

Second, a layered architecture hints at the locations where interfaces within the software need to be created. In order for a development

team to create firmware that can be reused, there needs to be an identifiable boundary where an interface can be created that remains consistent and unchanging as time passes. The interface contains declarations and function prototypes for controlling software in lower layers.

Third, a layered architecture allows information within the application to be hidden from other areas that may not need access to it. Consider the example with the low level driver. Does the application code really need to know the implementation details for how the driver works? Surely someone working at the application level would rather have a simple function to call and the desired result happens behind the scenes. This is the idea behind abstractions which is hiding the implementation behavior from the programmer and simply provides them with a black box. Developing a simple software architecture can help developers take advantage of these benefits.

Developers looking to create portable firmware that follows a layered software architecture model have many different possible models that can be chosen from and many custom hybrid models that they could undoubtedly develop. The simplest layered architecture can be seen in Figure 9 which consists a driver and application layer operating on the hardware. The driver layer contains all the code necessary to get the microcontroller and any other associated board hardware such as sensors, buttons and so forth running. The application code contains no driver code but has access to the low level hardware through a driver layer interface that hides the hardware details from the application developer but still allows them to perform useful functions.

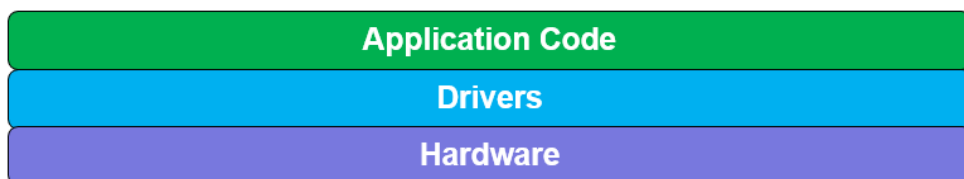


Figure 9 – *Two-layer Embedded Software Architecture*

Developing Reusable Firmware

The next model that a developer could choose to implement breaks the software up into three-layers similar to Figure 10. In a three-layer model, the driver and application layers still exist but a third “middle” layer has been added. The middleware layer may contain software such as a real-time operating system (RTOS), USB and/or Ethernet stacks along with file systems. The middleware layer contains software that isn’t directly the end application code but also does not drive the low level hardware. For this reason, components in this layer are often referred to as middleware.

Beyond the three-layer model, developers may find it worthwhile to start breaking the software into more refined layers of operation and may even provide pathways for higher level layers to circumvent layers and get direct access into lower software layers. The architectures can become quite complex and is well beyond the scope for this book. For now, a four-layer model will be as complex an example we will examine. For example, a developer may decide that the board support package, the integrated

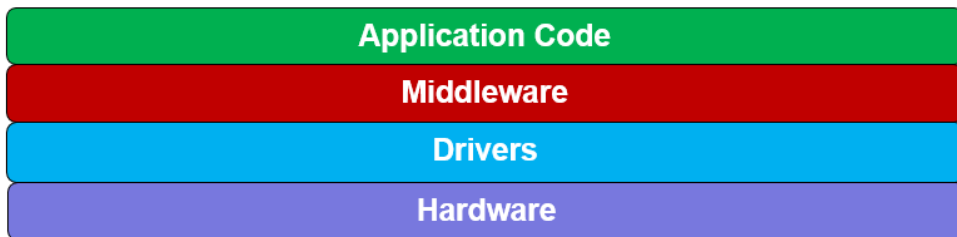


Figure 10 – Three-layer Embedded Software Architecture

circuits outside of the microcontroller, should be separated from the microcontroller driver layer. The board support drivers are usually dependent on the microcontroller drivers anyway and in order to improve portability probably should be separated. Doing this results in one possible four-layer model similar to the one shown in Figure 11.

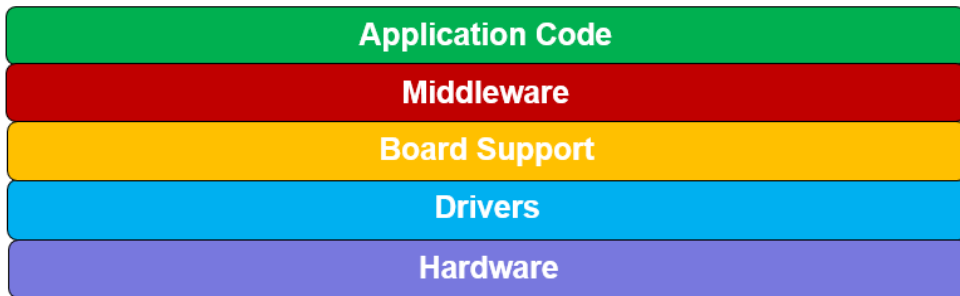


Figure 11 – Four-layer Embedded Software Architecture

Many formal models exist for developing layered software architectures including the well-known OSI model which contains over seven layers. A developer should examine their requirements, their portability and reuse needs and pick the simplest architecture that can meet their requirements. Don't be tempted to build a thirty layered software architecture if three layers will meet the requirements! The goal is to avoid complex spaghetti code that is intertwined and entangled and instead, develop layered lasagna code! (Just the thought makes my stomach growl)!

Hardware Abstraction Layers (HAL)

Each software layer software has at least one interface to an adjoining software layer. Depending on the software type that is contained within the next layer determines the name given to the interface. Each layer, if developed properly, can appear as a black box to the developer and only the interface specification provides insight into how to get the needed behavior and result. The interface has many benefits such as

- Providing a consistent method for accessing features
- Abstracting out the details for how the underlying code works
- Specifying wrapper interfaces for how to merge inconsistent code to the software layer

The most interesting firmware layer that developers now have the ability to utilize is the Hardware Abstraction Layer. A Hardware Abstraction

Software Terminology

Driver Layer refers to the software layer that contains low level, microcontroller specific software. The driver layer forms the basis from which higher level software interacts and controls the microcontroller.

Board Support Package refers to driver code that is dependent upon lower level microcontroller driver code. These drivers usually support external integrated circuits such as EEPROM or flash chips.

Middleware refers to the software layer that contains software dependent upon the lower lying hardware drivers but does not directly contain application code. Application code is usually dependent upon the software contained within this middle layer of software.

Application Layer refers to a software layer used for system and application specific purposes that is decoupled from the underlying hardware. The application code meets product specific features and requirements.

Configuration Layer refers to a software layer used to configure components within the layer.

Layer, HAL, is an interface that provides the application developer with a standard function set that can be used to access hardware functions without a detailed understanding for how the hardware works. Despite being commonly referred to as a HAL, it is not the infamous artificial intelligence from 2001 A Space Odyssey, although sometimes they can be just as devious.

HAL's are essentially API's designed to interact with hardware and a properly designed HAL provides developers with many benefits such as software that is

- Portable
- Reusable
- Lower cost (result of reuse)
- Abstracted (I don't need to know how the microcontroller does what it does)
- Fewer bugs due to repeated use
- Scalability (moving to other MCU's within a part family)

A poorly designed HAL can result in increased costs, buggy software and leave the developer wishing that they were dealing with the previously mentioned infamous HAL. An example software architecture that utilizing a HAL might look something similar to Figure 12. We will be discussing HAL design through-out the book.

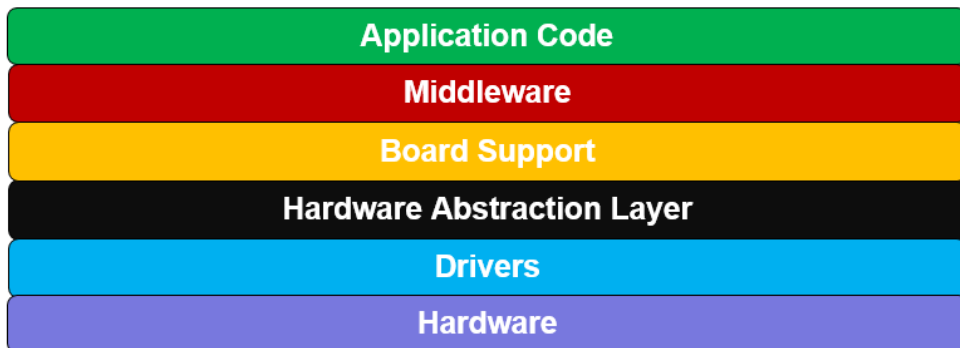


Figure 12- *Software Architecture with a HAL*

Software Terminology

Hardware Abstraction Layer (HAL) refers to a firmware layer that replaces hardware level accesses with higher level function calls.

Application Programming Interface refers functions, routines and libraries that are used to accelerate application software development.

Application Programming Interfaces (APIs)

Application Programming Interfaces, often referred to as API's, are a set of functions, routines and libraries that are used to accelerate the application software development. API's are usually developed at the highest software layers. There are many cases where developers will use the term API to include the HAL since the HAL is really a specialized API designed to interact with hardware. An example where the API might exist in a software stack can be seen in **Figure 13**.

A specific application may have multiple middleware components such as an RTOS, TCP/IP stack, file system and so forth. Each component may have their very own API associated with their software package. There could even be application level components that have their own API's in

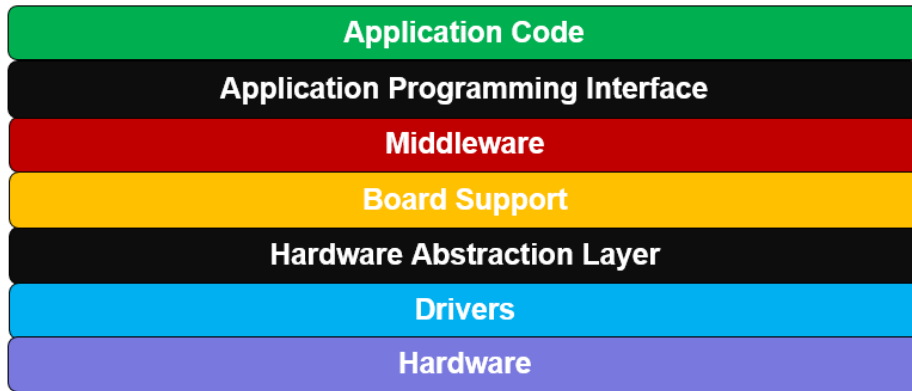


Figure 13 – Application Programming Interfaces

order to facilitate speedy development. The rule of thumb is that wherever you see two software layers' touch, there is an interface there which defines an API or HAL.

Project Organization

Organizing a project can help improve portability and maintainability. There are many ways that developers can organize their software but

the easiest way is to attempt to follow the software layer stack-up. Creating a file system and project folder structure that matches the layers makes it easier to simply replace a folder (a layer) with new software which would also include the components within that layer.

The project should also be organized in such a way within each layer that modules, tasks and other relevant code are easily locatable. Some developers like to create folders for modules or components and keep all configuration, header and source modules within the folders. Organizing the software in this way makes it very easy to add and remove software modules. Other developers prefer to break-up and keep header and source file separate. The method used is not important so much as being consistent and following a method is.

The following is an example organization that a developer may decide to implement to organize their project:

- Drivers
- Application
- Task Schedulers
- Protocol Stacks
- Configuration
- Supporting Files and docs

Getting Started Writing Portable Firmware

Developers who want to reuse software have a several challenges to overcome in order to be successful. These include:

- Endianness
- Processor architecture
- Bus width
- Ambiguous standards
- Development time and budget
- Modularity
- Code coupling

Developing Reusable Firmware

just to name a few. Getting started can be overwhelming and lead to more stress and confusion than simply writing very functional code that is discarded later. The key to successfully developing portable code is to determine how well your firmware currently meets the portable software characteristics. Once we understand where we are, we can decide where we want to go and set in motion the steps necessary to get there.

To determine where we are today with developing portable firmware, start by drawing a diagram similar to that shown in Figure 14. In the diagram, label each spoke with a portable firmware characteristic and select the top eight most important characteristics to you.

In each identified category, a developer can then evaluate how well their code exhibits these properties. For example, a developer who has been trying to transition into writing more portable code may evaluate themselves with a diagram result similar to Figure 15.

A quick look at Figure 15 can tell a developer a lot of information. First, we have strengths in documentation and modularity. That's a great step towards developing portable firmware and we are just getting started. The figure also shows us where our weaknesses are such as code coupling and cohesion.

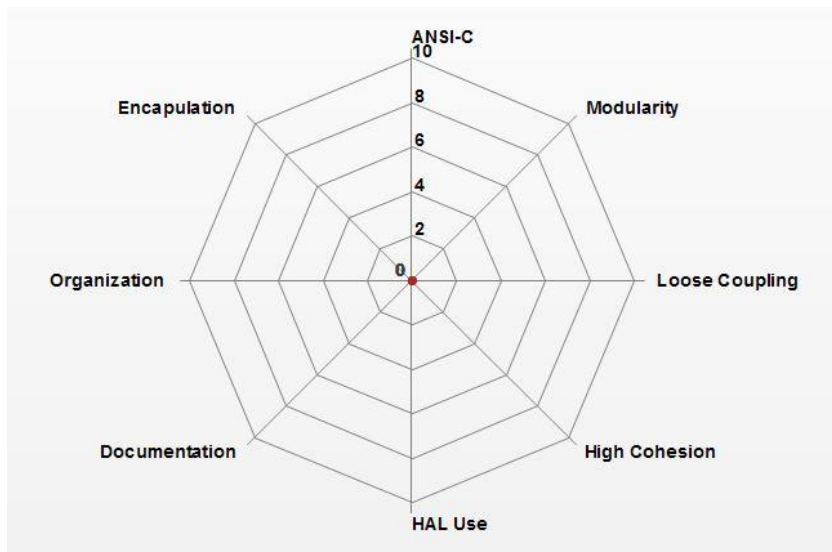


Figure 14 – Portable Code Evaluation

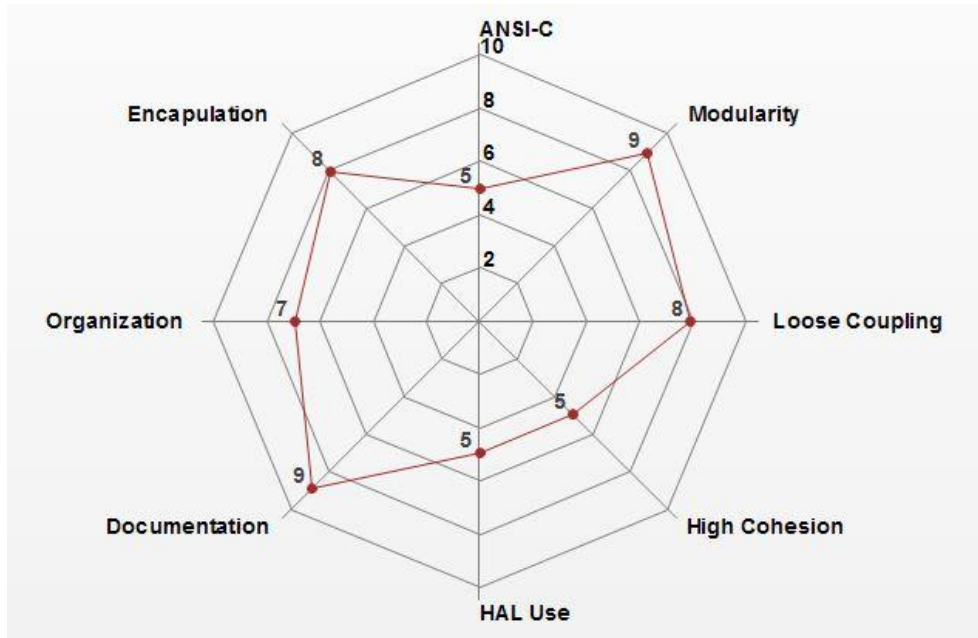


Figure 15 –Evaluated Firmware Characteristics

From this glance we can now determine where we should start to focus our attention. Which characteristic if improved by just a couple points will most drastically improve our code? Let's choose code coupling as example. If a developer is going to improve code coupling, they need to determine how they are going to go about making that improvement. They might decide that the best way to do this is to

- Schedule code reviews
- Find a tool that can provide a module dependency graph
- Use the dependency graph tool (just because we have a tool doesn't mean we have the discipline to use it)
- Develop a high level architecture that takes into account module coupling

A developer may decide that improving in one area is good enough to start or that all need to be done. The point is that we aren't going to start

Developing Reusable Firmware

writing perfect, reusable code overnight. The process is iterative and may take a few years before all the rough edges are smoothed but that is okay.

Below is a simple process that developers can use to improve their firmware portability:

- 1) Analyze their code characteristics
- 2) Identify strengths and weaknesses
- 3) Determine which characteristic to improve in the next 3 months
- 4) Identify what can be done to make the incremental improvement
- 5) Implement the improvement
- 6) After the specified period repeat

Going Further

Reading about portable and reusable code is one thing, actually doing it is a completely different story. Below are some suggestions on steps you can take to start developing firmware that is more portable:

- Select the language standard that will be used for your development effort(s) and spend 30 minutes each day reading through the language standard. Note areas that are not fully defined or could become pain points.
- Select two or three compilers, such as GCC, Keil and IAR. Download their user manuals and review their documentation on how they implemented the ambiguous areas in the selected standard.
- Purchase a copy of MISRA C/C++ and become familiar with the recommended best practices.
- Develop your own coding standard on the constructs that are allowed within an application and how compiler intrinsics and extensions should be handled.
- Review your typical software architecture. Does it have well defined layers? Does each layer have a well-defined interface? If not, now is the perfect time to spend a few minutes architecting your firmware stack-up. (Don't be concerned with defining the interface just yet. We'll be covering how to do this in the coming chapters).

- Review the last section on “How to get started Writing Portable Firmware”. On a sheet of paper, draw your own spider diagram and rank how well your code exhibits the portable firmware characteristics. Select one or two characteristics that you feel will have the biggest impact on your code and focus on improving those. Periodically review and reevaluate.

Chapter References

¹ Embedded Marketing Study, 2009 – 2015, UBM

² TIOBE Programming Community Index, November 2015, [www.Tiobe.com](http://www.tiobe.com)

³ ISO/IEC 9899:1999, C Language Specification

⁴ ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture

⁵ <http://whatis.techtarget.com/definition/layering>

⁶ <http://whatis.techtarget.com/definition/interface>