Playbook

Writing Portable Embedded Software

WRITING PORTABLE EMBEDDED SOFTWARE

As embedded systems engineers and developers, we are often working close to the "metal", or the underlying hardware. This work is fun and rewarding, because we are working at the boundary between the ethereal world of ideas (software) and the physical world (hardware).

One of the most significant challenges in our field is an embedded software project's dependency on the underlying hardware and the RTOS. Developers are expected to turn out products at a faster and faster pace, and so anything viewed as "unnecessary up front" design is skipped. Embedded programs are designed in an ad hoc manner, making direct use of the processor vendor SDKs and RTOS APIs throughout the code base.

There is a real cost associated with these dependencies, especially in today's increasingly agile world and increasingly volatile electronics market. Inevitably, the house of cards comes crashing down. Electrical components can be in short supply, requiring us to qualify alternate components late in the product development cycle. Other components will reach end-of-life at an inopportune moment. Or perhaps a different processor is selected due to its significant power savings and integration of external peripherals, providing for a better user experience and simpler BOM.

When these situations occur, the illusion of agility comes crashing down. Qualifying an alternate component requires a month-long effort to update all of the references to that component scattered throughout the code base. Changing to a new processor requires an almost complete rewrite of the program because there was no clean separation between the application code and vendor SDK. These schedule delays are often an unexpected shock to the team, because they need to have the software ready for the manufacturing run scheduled in 45 days, but the software team is telling them the schedule will be delayed for 6 months!

This is not a new problem. In 1981, D.L. Parnas et al. observed that:

> ***Much of the complexity of embedded real-time software is associated with controlling special-purpose hardware devices****. Many designers seek to reduce this complexity by isolating device characteristics in software device interface modules, thereby allowing most of the software to be programmed without knowledge of device details. While these device interface modules generally do make the rest of the software simpler, their interfaces are usually the result of an ad hoc design process, and they fail to encapsulate the device details completely.* ***As a result, device changes lead to changes throughout the software, rather than just in the device interface module.***

We feel like we are being treated unfairly when management doesn't understand the software schedule delays that are caused by last-minute hardware changes. But these changes are a fact of life on embedded projects. For our embedded teams to be truly agile, we must be able to quickly adapt to changes in the underlying hardware without complaint.

Luckily, there is a single change in perspective that can take us from being "the developers who always complain about hardware changes and the schedule" to "the heroes on the software team." That change in perspective is **designing our embedded software for portability.**

WRITING PORTABLE EMBEDDED SOFTWARE

"Portability" is a quality that is commonly mentioned, but in our experience it is rarely one that is prioritized. In fact, we have never worked on a commercial project that gave any thought to portability. Like everyone else, we used vendor SDKs and RTOS APIs throughout our code base. We, too, complained about the late-stage hardware changes and the impact on the schedule. We, too, get tired of rewriting the same software over and over again on different platforms that use different vendor SDKs.

After one painful rewrite too many, we decided to focus on making our embedded software as portable as possible. What we found was that creating portable embedded software gave us superpowers beyond making it easy to change and reuse our software.

- We can run and debug code on our personal computers, giving us access to more powerful debugging tools like fuzzers and sanitizers
- We are able to start developing and testing significant portions of an embedded software program *before* the final hardware is available
- We can work remotely and continue to make progress *even without any hardware around*
- We can take advantage of automated unit tests and TDD, which are often skipped on embedded projects due to the difficulty in breaking hardware dependencies
- We can open source portions of our portable software, which builds relationships in the community and enables other developers to use and improve the drivers and libraries that we develop

By applying the strategies outlined in this playbook, you too can command these superpowers and adapt to hardware changes with no sweat.

The Three Key Principles Behind Portability

Before we dive into our top five strategies for developing portable embedded software, we want to outline the three key principles that serve as a foundation for building portable software:

1. Keep things loosely coupled
2. Hide the details that are likely to change
3. Define clear boundaries and responsibilities

1. Keep Things Loosely Coupled

Software industry leaders have been writing about the downsides of tight coupling since the 1960s, so we aren't going to rehash their work. If you're unfamiliar with the concept, here is some introductory reading:

- Wikipedia: Coupling
- Wikipedia: Loose Coupling
- Quality Code is Loosely Coupled

In practical terms, we can say that *tightly coupled* code makes direct references to the details of another module. This could be making direct use of functions provided by the VL53L1X Time of Flight sensor driver, invoking FreeRTOS functions, or using a processor vendor's SDK. Whenever the sensor, processor, or RTOS changes, every other file which references these functions must also be changed. With tight coupling, "simple" part changes can have far reaching impacts within the program.

On the other hand, *loosely coupled* code does not directly refer to another module. Instead, it interacts with other modules through mechanisms such as *abstract interfaces*, *registration functions*, and *callbacks*. By keeping references to external modules "generic", we can use alternate implementations without needing to modify all of the referenced modules.

If your embedded software is tightly coupled to the underlying hardware or OS, changing a single component of the system – such as the processor – can require you to rewrite significant portions of your software. To improve our software's portability and changeability, we want to think about how we can eliminate sources of tight coupling.

For more on coupling and the sources of coupling in embedded software, see "Musings on Tight Coupling Between Firmware and Hardware."

---

2. Hide the Details that are Likely to Change

In his classic 1972 paper, "On the Criteria to be Used in Decomposing Systems into Modules," D.L. Parnas observed that the default tendency for software developers is to compartmentalize a system according to the "program flow." We still observe this tendency today in many of the professional programs that we have worked on.

However, Parnas observes that this is not the ideal approach for building our systems:

> *"It is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."*

This design principle is called *information hiding*. The purpose of this principle is to minimize the impact of change by compartmentalizing the details that might change within different components. We do this by creating a stable interface that protects the rest of the program from the implementation details that are most likely to change. This interface should reveal as little as possible about its inner workings.

This design principle is often ignored in embedded systems software, leading to tight coupling throughout the program.

The most common examples are found in code that works with sensors. If we apply information hiding, a sensor interface would return samples in a specified format with conversion being handled behind the scenes (e.g. "returns 11.5 fixed point in g", or "returns an integral value in mV"). However, most embedded software that we see has sensors returning raw values, leaving the application code to handle data conversion. Changes to the sensor means that all modules working with the sensor need to change as well.

Another common violation of information hiding can be found when dealing with external storage. With information hiding, we might have a "file system" or "storage" module that provides operations like `open()`, `close()`, `read()`, `write()` while hiding the underlying details about where and how files are stored. However, most of the time we encounter application software that talks directly to an SD card, an internal EEPROM, an I2C EEPROM, SPI-NOR Flash, or Parallel Flash. This often requires the application code to manage details such as chip select pins, addresses, memory layout, page sizes, and the proper process for erasing and writing data. Under these conditions, changing to a new type of storage requires rework throughout a program.

One of our members shared a painful story about the impact of tight coupling when dealing with memory:

> *The worst I've seen was an external RAM chip on an EBI bus that didn't have enough address pins. All of the algorithms [throughout the program] had manual GPIO toggling of the most significant address pins to "switch RAM banks". I had to port that code to a system with sufficient internal RAM, and it took over a month. Without that coupling it would have taken no time at all, and they would just be [portable] algorithms.*

Remember, we want to create boundaries in our systems based on the principle of information hiding instead of following a flowchart.

For more on information hiding, as well as a comparison between the flowchart-based and information hiding-based approaches, read "On the Criteria to be Used in Decomposing Systems into Modules."

---

3. Define Clear Boundaries and Responsibilities

Another common problem that we see across embedded systems software is *code that tries to do too much*. Instead, we want to focus on creating smaller, coherent modules with clear responsibilities. This is the *single responsibility principle (SRP)*.

The SRP states that every module should have a *single responsibility* or *single concern* that is encapsulated by the module. By focusing on a single responsibility, we can make our module more robust against the impacts of change.

In *Timeless Laws of Software Development*, Jerry Fitzpatrick gives us another way to think about the SRP: we want to avoid "mixed metaphors." He outlines three real-world metaphors that we can use for evaluating our modules and abstractions:

1. Product - a representation of state, something that is created, a distinct entity in the system
2. Processor - an operation, a transformation, a monitor
3. Producer - something that generates products

Jerry points out that we often create modules and abstractions that mix these metaphors, which is how we get into trouble. If you encounter a module that uses mixed metaphors, you are certainly violating the SRP.

For example, we recently encountered a temperature sensor driver that read samples from the hardware component. This driver was also monitoring the returned temperature for unexpected temperature swings. When a temperature swing was detected, it notified other objects in the system that an unexpected swing had occurred. The temperature sensor was tightly coupled to these other modules because they were directly referenced in the source.

This temperature sensor design represents a mixed metaphor because it combines a *producer* (generating temperature samples) and a *processor* (monitoring for temperature swings). If we followed the SRP, we would instead want to separate the responsibilities in this temperature sensor driver into three separate modules:

1. The temperature sensor driver (producer), which communicates with the temperature sensor hardware to generate temperature samples (product).
2. The temperature monitor (processor), which consumes temperature samples (products) to check for swings and generates a notification event when a swing occurs.
3. A notification system (producer) which enables other modules to register for temperature swing notifications (products) and notifies listeners when the swing event is generated.

This design provides multiple benefits:

1. Each module is smaller, making each one easier to understand and work with
2. We've improved reusability
    - The temperature sensor is no longer tightly coupled to other modules in the system, and it can be used in another system without requiring the use of a temperature monitor or notification system.
    - The temperature monitor module can be used with other temperature sensors.
    - The notification system can be reused for other notification types.
3. We can independently modify these modules without needing to change the others (assuming that the interfaces remain unchanged)

As D.L. Parnas pointed out in "On the Criteria to be Used in Decomposing Systems into Modules," careful modularization also allows us to create a hierarchical structure of modules in our system based on the "uses" or "depends upon" relations between our modules. If our system can be organized in a hierarchical manner, we are able to cut off the upper levels of the software while still having a usable product…

The existence of the hierarchical structure assures us that we can "prune" off the upper levels of the tree and start a new tree on the old trunk.

… while also improving our ability to reuse our work:

A careful job of decomposition can result in considerable carryover of work from one project to another.

Five Practical Strategies for Creating Portable Embedded Software

Now that we understand the key principles underlying the concept of "portability", we can apply them with five essential portability strategies. By focusing on these strategies, you will drastically improve the portability of your software.

1. Keep platform- and component-specific details behind a barrier
2. Develop embedded software on your personal computer FIRST
3. Create minimal abstractions
4. Constrain tightly coupled code to isolated modules
5. Use your build system to enforce the rules

1. Keep Platform- and Component-Specific Details Behind a Barrier

Here's the most important strategy for keeping code portable: **any time you're using a platform/component/SDK-specific API in your code, put it behind some kind of abstraction**.

This advice applies to *any* SDK, OS, device driver, or other platform-specific interface. For example, instead of directly working with FreeRTOS functions, we can define a generic interface for OS functionality. Here are generic functions to create and lock a mutex:

```
uintptr_t mutex_create();
bool mutex_lock(uintptr_t handle, uint32_t timeout_ms);
```

Our program can operate with any implementation that provides the required abstract interfaces - pthread, FreeRTOS, ThreadX, or any other suitable OS. Whenever a new OS is required, we only need to define a new implementation for these generic OS functions. All of the source code in our program that uses the generic OS interfaces can remain unchanged.

Often, it can be the underlying data types that come back to haunt us. Use generic types in your abstract interfaces, whether an intermediate type you define (e.g. a general structure with relevant data), or a generic type such as `void*` or `uintptr_t`. We show this in the generic `mutex_lock` prototype above, where we use a `uintptr_t` for the handle instead of a platform-specific type.

Portable code must be kept "generic" by using these generic interfaces, standard language features, and portable libraries. Portable code should never directly reference any platform-specific modules or details. This includes vendor SDK driver interfaces, RTOS-specific APIs, direct register accesses, and platform-specific debugging functionality. These items prevent us from easily porting our code from one platform to another and belong behind a barrier of some kind.

Developing abstractions is an art in itself. We recommend reading "A Procedure for Designing Abstract Interfaces for Device Interface Modules", which describes an approach for designing and refining abstract interfaces. The paper also discusses strategies for addressing common design challenges faced by embedded systems software developers.

---

2. Develop Embedded Software on Your Personal Computer FIRST

The best way to create a portable embedded software design is to begin by writing and testing as much of the program as possible on your personal computer. Once a module is unit tested and functional, you can move it to the embedded target.

There's a simple reason we like to start on our personal computers: we know that we can't rely on all the facilities available to us on our computers once we move to an embedded environment. This simple change in perspective forces us to create abstractions that separate our code from the underlying system. The pthread library isn't available on our microcontroller target, so we create OS abstractions to hide those details. The Aardvark I2C/SPI adapter APIs aren't available on our microcontroller target, so we need to create generic I2C and SPI abstractions that support both Aardvark adapters and our microcontroller peripherals. A specific component on the microcontroller isn't available on our personal computers, so we need to create a mock, spy, or fake to test out the code without hardware.

Once we've built these abstractions, all we need to do to move our software to another platform is supply a new implementation for each abstraction. As a benefit, we've already tested our code, so we have increased confidence that it will work correctly when we migrate it to the target system. When it doesn't work, we've reduced our potential debugging problem space to the changes made during the migration.

3. Create Minimal Abstractions

We often see teams get stalled out with creating abstractions. Inevitably, it's because they are creating what we call *ambitious abstractions*. Whenever you have the feeling that you need to create the *perfect* abstraction that accounts for all possible implementation differences, you're probably entering into the realm of *ambitious abstraction.*

The risk of creating ambitious abstractions is three-fold:

1. We may give up on the effort altogether because it is too difficult compared to the status quo.
2. We may expose functionality that is not actually required by a user, which can introduce accidental coupling.
3. We run the risk of creating an abstraction which is subtly coupled to the underlying implementation because we failed to account for differences in other possible systems and components.

Plenty of platform-specific code cannot be easily abstracted. Initialization code in particular is often difficult to abstract, because initialization requirements and configuration options vary widely across OSes and components. It's much easier to just keep the initialization details constrained to an isolated module that is allowed to be tightly coupled to the OS. Then we can focus our efforts on creating generic functions like `mutex_lock()` and `mutex_unlock()` that can be used throughout our program.

You can identify other attempts at ambitious abstraction by stepping back and thinking about the responsibilities of your module. If your abstraction includes details that aren't needed by the users of your module, you should eliminate them.

For instance, many teams try to create a full-fledged SPI abstraction that supports all aspects of SPI communication, such as initializing the SPI controller, changing the SPI operating mode, setting bit ordering, and starting and stopping the driver.

```
// Abstract SPI Controller Interface
class SPIController
{
  public:
    virtual void configure(spi::baud_t baud) noexcept = 0;
    virtual spi::mode mode() const noexcept = 0;
    virtual spi::mode mode(spi::mode mode) noexcept = 0;
    virtual void start() noexcept = 0;
    virtual void stop() noexcept = 0;
    virtual comm::status transfer(void* data, size_t
length) noexcept = 0;
};
```

In this type of setup, our SPI device driver interacts with the SPI controller using the abstract interface:

```
// Construct an AX5043 object, taking in a
// reference to the generic SPI interface
AX5403(SPIMaster& spi);
```

The AX5043 is a radio IC. Let's step back and think about the core responsibilities of a radio device driver:

- Communication with the radio
- Initialization of the radio
- Adjusting radio settings
- Sending and receiving data over the radio

Note that the driver doesn't need to be responsible for initializing the SPI bus, changing SPI operating modes, setting bit ordering, or starting/stopping the SPI driver. Our radio driver only needs to know how to transfer data to the device. So why are we coupling our device driver to the full `SPIController` abstraction? We could instead use a single function pointer.

```
// SPI transfer abstraction.
// This function is used to send and receive data to
// a SPI device through a SPI controller.
//
// The first parameter is a buffer of data to tx,
// The second parameter is the length.
// The received data will be placed into the input buffer.
void (spi_transfer)(unsigned char*, uint8_t);

// Updated Constructor, the transfer function
// is saved as a private member variable
AX5403(spi_transfer func);
```

All a user would need to do to get this driver to work on a new system is to initialize the driver with an implementation that meets the requirements of the `spi_transfer` function.

```
void radio0_xfer(unsigned char* data, uint8_t length)
{
    wiringPiSPIDataRW(SPI_CHANNEL_RADIO_0, data, length);
}
```

Remember: our goal is to create *lightweight* abstractions. We merely want to separate portable code from the underlying system, making it easier to test without hardware and to migrate to a new platform. Use the minimum number of abstractions necessary. Use the minimum number of arguments you can get away with for those abstractions. *Keep it simple*.

For one of our favorite examples of minimal abstraction, check out the full case study on the portable AX5043 radio IC driver.

---

4. Constrain Tightly Coupled Code to Isolated Modules

While our goal is to minimize coupling between our embedded software and the underlying platform, we cannot completely eliminate it. Some part of our program needs to know how to initialize and configure the specific components in our system, even if the remainder of the program can safely work with the abstract interfaces. In other cases, platform-specific code may not be easily abstracted.

In order to manage this coupling, we will constrain the difficult-to-abstract code to a single, isolated location. This module may have its own generic interfaces that are used by portable software modules, such as an `initialize_hardware()` function that is called by `main()`.

This type of module is often called the "board support package" (BSP) or "hardware abstraction layer" (HAL), although these terms tend to be overloaded. For example, the STM32 HAL abstracts away differences in STM32 processors, but it is not the same kind of HAL described above – its use does not help us if we need to run our software on a non-ST processor.

Inside of this tightly coupled module, you can safely:

- Declare and initialize objects using the platform-specific types and APIs
- Configure the low-level processor peripherals for use with your system (e.g., DMA and interrupt configuration)
- Connect external hardware components with specific processor peripherals
- Initialize external hardware components
- Pass around function pointers, set handlers, and supply callbacks to the portable software modules

By keeping our tightly coupled code constrained within a minimal number of modules, we reduce the amount of code that needs to be updated whenever a change occurs. For instance, changing the time-of-flight sensor installed on our board will only require the creation of a new driver (which works with our system's generic interfaces) and updating the hardware initialization code. The rest of the software system, which is unaware of the specific hardware details, can remain unchanged.

For more information on constraining tightly coupled code, check out our article "Managing Complexity with the Mediator and Facade Patterns."

---

5. Use Your Build System to Enforce the Rules

The strategies outlined above will help you create portable software, but they don't prevent anyone from breaking the rules and directly using vendor SDKs and native OS function calls in portable modules.

This pollution is hard to prevent by default. Many projects are built as a single executable target that includes all of the source files and links against any vendor supplied libraries. In these setups, there is often a global include directory setting that automatically supplies the same include arguments to all files. This behavior is convenient and simple to implement, but it also prevents us from enforcing separation: any file can include any header within the global include search path.

To help us write portable software, our build should fail if we're trying to access a restricted API. This is possible, but we need to follow a specific strategy for building our embedded software application to make this work.

First, we need to strip platform-specific and component-specific include statements and definitions from "public" headers. This means that *none* of the headers for your portable software components and portable interfaces should include a restricted header.

Next, we need to isolate platform-specific code into standalone modules using the following steps:

1. Define library targets for every platform- and component-specific module. This enables you to hide the implementation details from external users and have fine-grain control over which modules can access "restricted" headers and which can only use the portable interfaces.
2. Update the excitable target's build rules so that the program links against the new library targets instead of directly compiling source files

Now that we've created these modules, we can remove global include settings from the build system. Instead, we will enforce a different set of include settings for each library and executable target. In practical terms, this means that only our FreeRTOS implementation for the generic OS interface uses the include directory flag for the FreeRTOS headers, while the rest of the program only uses the include directory flag for the generic OS headers.

Changing how we structure our builds yields dramatic improvements in designing portable software. We can eliminate the possibility of platform-specific code infecting our portable modules. We use this exact strategy for building processor-specific libraries and drivers, board support packages (BSPs), hardware abstraction layers (HALs), drivers, and OS-specific code.

For a detailed guide on accomplishing this strategy, check out our free article "Leveraging Our Build Systems to Support Portability."

Diving Deeper into Portability

If you want to learn more about writing portable software, as well as detailed dives into the strategies outlined above, check out the following articles on the Embedded Artistry website:

- Musings on Tight Coupling Between Firmware and Hardware discusses the pain that tight coupling causes, and looks at the sources of coupling in embedded software.
- Practical Decoupling Techniques Applied to a C-based Radio Driver is a case study in creating a portable and reusable radio driver by defining a single abstract interface.
- Prototyping for Portability: Lightweight Architectural Strategies outlines additional lightweight architectural strategies for building portable hardware.
- Leveraging Our Build Systems to Support Portability describes how we can leverage our build system and the structure of our source code repository to create portable software and enforce dependency rules. If you're not used to working with a build system, our Introduction to Build Systems course will help you build the necessary foundation.
- Managing Complexity with the Mediator and Facade Patterns discusses two classic design patterns and their applicability for managing complexity in embedded projects.
- A Classic Paper on Designing Hardware Abstractions for Embedded Systems introduces one of our favorite embedded systems papers: "A Procedure for Designing Abstract Interfaces for Device Interface Modules." This extremely practical paper provides us with a two-pronged approach for designing and refining our abstract interfaces, while also discussing strategies for addressing common design challenges faced by embedded systems software developers who are building abstract interfaces. Embedded Artistry members can also access our commentary.
- Virtual Devices in Embedded Systems Software expands upon the previous paper with example abstractions. In particular, this article shows how to manage both abstract interfaces and device-specific initialization when implementing a driver.
- A Case Study on Finding, Fixing, and Testing a Firmware Bug Without Hardware shows the power of having a portable software design. I walk through an actual debugging process for a FLEX paging system without powering on any hardware or test equipment.
- On the Criteria to be Used in Decomposing Systems into Modules, by D.L. Parnas, is a classic paper that introduces the principle of information hiding. Embedded Artistry members can also access our commentary.

We are always publishing new content. You can find our complete collection of articles on portability and practical software architecture by browsing our archives and selecting the Practical Architecture category.