The Embedded Project Cookbook

A Step-by-Step Guide for Microcontroller Projects

—

John T. Taylor
Wayne T. Taylor

Apress®

The Embedded Project Cookbook

A Step-by-Step Guide for Microcontroller Projects

John T. Taylor
Wayne T. Taylor

Apress®

The Embedded Project Cookbook: A Step-by-Step Guide for Microcontroller Projects

John T. Taylor
Covington, GA, USA

Wayne T. Taylor
Golden, CO, USA

ISBN-13 (pbk): 979-8-8688-0326-0
ISBN-13 (electronic): 979-8-8688-0327-7
https://doi.org/10.1007/979-8-8688-0327-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Melissa Duffy
Development Editor: James Markham
Editorial Project Manager: Gryffin Winkler

Cover designed by eStudioCalamar

Cover image designed by Tom Christensen from Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

To Sally, Bailey, Kelly, and Todd.

—J.T.

Table of Contents

About the Authors

John Taylor has been an embedded developer for over 30 years. He has worked as a firmware engineer, technical lead, system engineer, software architect, and software development manager for companies such as Ingersoll Rand, Carrier, Allen-Bradley, Hitachi Telecom, Emerson, AMD, and several startup companies. He has developed firmware for products that include HVAC control systems, telecom SONET nodes, IoT devices, microcode for communication chips, and medical devices. He is the co-author of five US patents and holds a bachelor's degree in mathematics and computer science.

Wayne Taylor has been a technical writer for 27 years. He has worked with companies such as IBM, Novell, Compaq, HP, EMC, SanDisk, and Western Digital. He has documented compilers, LAN driver development, storage system deployment and maintenance, and dozens of low-level and system management APIs. He also has ten years of experience as a software development manager. He is the co-author of two US patents and holds master's degrees in English and human factors.

About the Technical Reviewer

Jeff Gable is an embedded software consultant for the medical device industry, where he helps medical device startups develop bullet-proof software to take their prototypes through FDA submission and into production. Combining his expertise in embedded software, FDA design controls, and practical Agile methodologies, Jeff helps existing software teams be more effective and efficient or handles the entire software development and documentation effort for a new device.

Jeff has spent his entire career doing safety-critical product development in small, cross-disciplinary teams. After stints in aerospace, automotive, and medical, he founded Gable Technology, Inc. in 2019 to focus on medical device startups. He also co-hosts the Agile Embedded podcast, where he discusses how device developers don't have to choose between time-to-market and quality.

In his spare time, Jeff enjoys rock climbing, woodworking, and spending time with his wife and two small children.

Preface

My personal motivation for writing this cookbook is so that I never have to start an embedded project from scratch again. I am tired of reinventing the wheel every time I move to a new project, or new team, or new company. I have started over many times, and every time I find myself doing all the same things over again. This, then, is a cookbook for all the "same things" I do—all the same things that I inevitably have to do. In a sense, these are my recipes for success.

On my next "new project," I plan to literally copy and paste from the code and documentation templates I have created for this book. And for those bits that are so different that a literal copy and paste won't work, I plan to use this cookbook as a "reference design" for generating the new content. For example, suppose for my next project I need a hash table (i.e., a dictionary) that does not use dynamic memory allocation. My options would be

1. Reuse or copy an existing module from this framework.
2. Adapt an existing module to meet my specific requirements.
3. Design and write the code from scratch.

For me, the perfect world choice is option one—copy, paste into a new file, and then "save as" with a new file name. Option two would be to use the material in this book as a reference design. Start with one of the code or documentation templates and adapt it to the needs of the new project. And option three would be the last resort. Been there; done that; don't want to do it ever again.

Even though nothing is ever a perfect world choice, I know from experience that I can reuse some of this code wholesale with hardly any changes. In fact, the entire impetus behind my early GitHub projects was to have a reusable repository of source code that was not owned by someone else that I could freely use as needed—both professionally and personally. And because you bought this book, I'm providing you with a BSD license to all the source code so you can use and reuse just as freely. And, in addition to the raw, reusable blocks of source code, I also have the building blocks for the framework, which is the automated test tools and simulators required for building and releasing embedded projects. In some ways, I think of this cookbook as the user manual for all my GitHub toys.

Beyond the obvious advantage of not having to rewrite code, there is also the advantage of having example documents and other materials that I can use when mentoring or training other engineers. In the past, when I've been trying to explain these concepts to new team members, it involved a lot of hand waving and hastily drawn boxes and arrows on the whiteboard. But now I have tangible examples of what I'm talking about at my fingertips. It's yet another thing I don't have to start from scratch. The next time I need to train or explain any of the best practices contained in this cookbook, I plan to say, "And if you want a better example of what I'm talking about, I know a really great book on this topic...."

—John Taylor, Covington, Georgia, March 2024

CHAPTER 1

Introduction

The purpose of this cookbook is to enable the reader to never have to develop a microcontroller software project from scratch. By a *project*, I mean everything that is involved in releasing a commercially viable product that meets industry standards for quality. A project, therefore, includes noncode artifacts such as software processes, software documentation, continuous integration, design reviews and code reviews, etc. Of course, source code is included in this as well. And it is production-quality source code; it incorporates essential middleware such as an OS abstraction layer (OSAL), containers that don't use dynamic memory, inter-thread communication modules, a command-line console, and support for a functional simulator.

The book is organized in the approximate chronological order of a software development life cycle. In fact, it begins with a discussion of the software development process and the software development life cycle. However, the individual chapters are largely independent and can stand alone. Or, said another way, you are encouraged to navigate the chapters in whatever order seems most interesting to you.

Note The focus of this cookbook is on software development—not the processes or deliverables of other disciplines. Other disciplines that participate in the process are typically only discussed in the context of their providing inputs for project artifacts or their consuming of project artifacts.

J. T. Taylor and W. T. Taylor, *The Embedded Project Cookbook*, https://doi.org/10.1007/979-8-8688-0327-7_1

Software Development Processes

Software development processes are different everywhere. No two organizations create software the same way, and in some organizations and companies, no two teams do it the same way. Additionally, processes that are intended to improve quality are not uniformly implemented: neither by companies in the same industry segment, nor, sometimes, by members of the same team. Consequently, there is no one-size-fits-all model or solution for professional software development. And yet, everybody ends up doing the same things.

For example, Figure 1-1 shows a straightforward model for developing a bit of software for an embedded system.

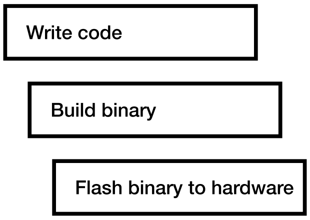


Figure 1-1. *A simple development model for embedded software*

At your discretion, you could add additional steps, or your organization might require additional processes. So the model might be expanded to something like what is shown in Figure 1-2.

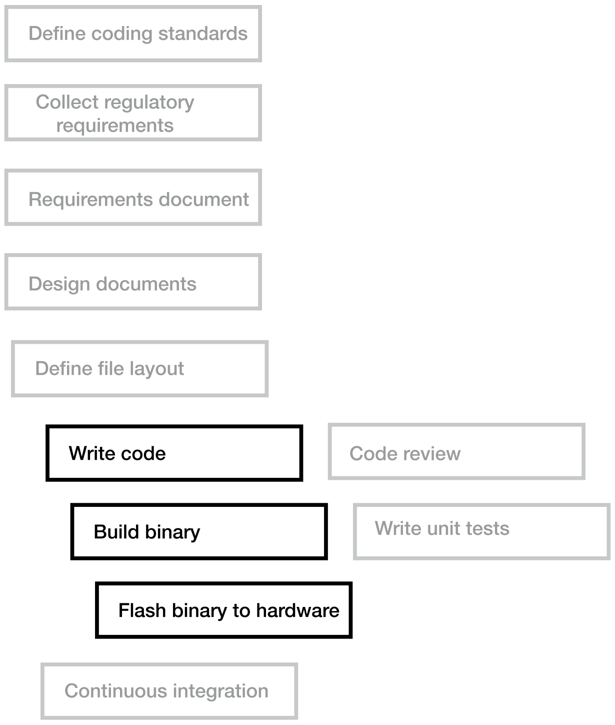


Figure 1-2. *Additional steps and processes for a simple development model*

The more additional processes and steps you add, the more sophisticated your development process becomes, and—if you add the right additional processes—the better the results. Figure 1-3 illustrates this continuum.

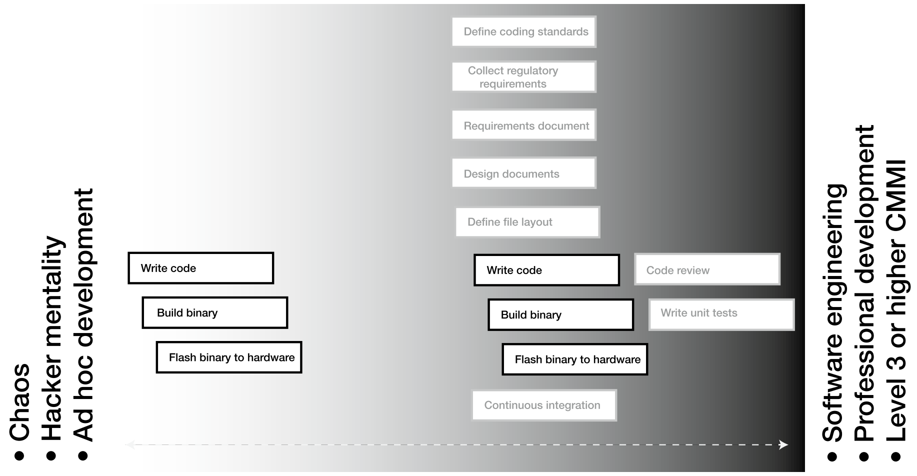


Figure 1-3. *A continuum of software development processes and practices*

There is no perfect set of processes. However, in my career, I have found myself using the same processes and steps over and over again. This book, then, is a collection of the steps and processes that I have found essential for developing embedded software in a commercial environment. I recommend them to you as an effective, efficient way to develop great code. Of course, you can skip any of these recommended steps or phases, but every time you do, there's a good chance that you're buying yourself pain, frustration, and extra work down the road. It is easy to say, "Oh, I can just clean up and refactor this module later so it meets our standards and conventions," but for me, clean-up refactoring is painful, and I have found it often gets skipped for the sake of schedule pressure. Personally, I try very hard not to skip steps because if I do, things don't get done any faster, and all I've done is start the project with technical debt.

In the end, it will come down to how willing you are to take on and adopt the engineering disciplines that these "software recipes" embody. Unfortunately, many people equate discipline with "doing stuff they don't want to do." And, yes, it's not fun writing architecture documentation or automated unit tests and the like, but it's the difference between being a hacker or a professional, spit-and-bailing wire or craftsmanship.

Software Development Life Cycle

Depending on your experience and background, you may have experienced four to eight stages in the software development life cycle (SDLC). This book focuses on the work, or stages, that runs from articulating the initial business needs of the product through the first production release of the software. My definition of the SDLC has the following three software development stages:

- Planning
- Construction
- Release

These three stages are waterfall in nature. That is, you typically don't want to start the construction stage until the planning stage has completed. That said, work within each stage is very much iterative, so if new requirements (planning) arise in the middle of coding (construction), the new requirements can be accommodated in the next iteration through the construction phase. To some, in this day of Agile development, it might seem like a step backward to employ even a limited waterfall approach, but I would make the following counter arguments:

- An embedded project—that is, one with limited resources and infrastructure—absolutely requires a certain amount of upfront planning and architecture before active coding begins.

- 80% or more of the work occurs in the construction stage, which is iterative and fits the Agile model.
- You will experience fewer hiccups in the construction stage if you're building on a solid foundation that was established during the planning stage.

Figure 1-4 outlines my software development life cycle and provides some representative activities that occur in each one. Note that only activities that are the responsibility of the software team are shown. That is, activities related to hardware development or formal software verification are not shown.

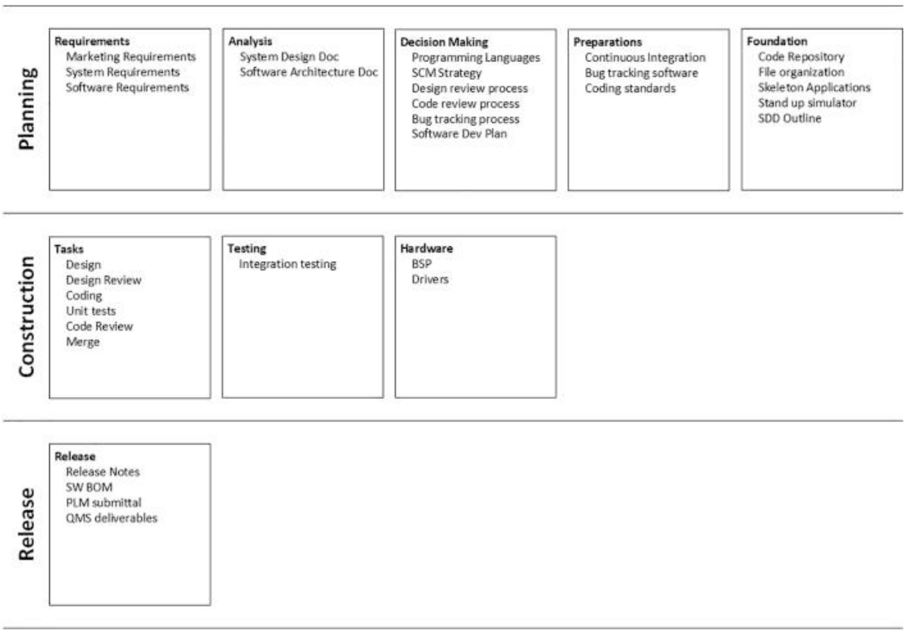


Figure 1-4. *Software development life cycle stages*

In this cookbook, I illustrate the work of these stages by defining and building a hypothetical Digital Heater Controller (DHC), which I like to call the GM6000. While the GM6000 is hypothetical, the processes, the framework, and the code I provide can be described as "professional grade" and "production quality." That is, everything in this book has been used and incorporated in real-life products. Nevertheless, there are some limitations to the GM6000 project:

- It is only intended to be a representation of a typical embedded project, not an actual product. Some of the requirements may seem unnecessary, but I've included them to illustrate certain concepts or to simplify the construction of the example code.
- Not all the requirements for the GM6000 were designed or coded because if the output of a particular requirement didn't illustrate something new or important, I was inclined to skip it.

Outputs and Artifacts

By applying the processes described in each of these stages, you can generate outputs or artifacts upon which you can build a releasable product. All these processes are codified in a framework that is built on a BSD-licensed, open source software that you have access to and which you can use to quick-start any microcontroller project.

What's different about the framework described in this book—that may not be found in other books about software development life-cycles—is this:

- It is specifically a cookbook for microcontroller applications, even though, having said that, the processes can be applied to software projects large and small.

- This cookbook prescribes the approach of "build and test software first; add hardware second." In real life, this allows you to develop significant amounts of production quality code even before the hardware is available, which dramatically reduces the start-to-release duration of a project.
- This cookbook prescribes continuous integration.
- This cookbook prescribes automated unit tests.

What You'll Need to Know

If you're directly involved in architecting, designing, implementing, or testing embedded software, you should have no problem following the concepts of this book. Additionally, if you have one of the following titles or functions, you might also derive some benefits from this book:

- Software architects and leads—The processes presented here identify the upfront planning and deliverables that can be used as a guide for creating production documentation. Personally, I look at documentation as a tool to be used in the development process, as opposed to busy work or an end-of-the-project scramble to record what was implemented.
- Software engineers—The processes presented here provide a context for processes that software engineers are often asked to follow. They also supply concrete examples of how to write architecture and design documents, write automated unit tests, and develop functional simulators.

- Software managers—The processes presented here provide specifics that can help justify project expenditures for tools like CI build servers or for training. It is material that can be used to champion the idea of doing it right the first time, instead of doing it twice.[1]

Coding in C and C++

The example code and framework code in this cookbook are written in C and C++, but mostly in C++. Nevertheless, if you have experience writing software in C, or a strongly typed programming language, you should be able to follow the examples. If you're skeptical about using C++ in the embedded space, consider that the Arduino UNO framework—written for an ATmega328P microcontroller with only 32KB of flash and 2KB of RAM—is implemented in C++. Nevertheless, there is nothing in the processes presented in this book that requires a specific implementation language.

All the example code and framework code in this book are available on GitHub, and the numerous appendixes in this book contain examples of all prescribed documents.

What Toys You Will Need

Here is a summary of what you will need to build and run the examples in this book and to create the final application code for GM6000:

- C/C++ compiler (e.g., Visual Studio, MinGW, etc.).
- Python 3.8 or higher.

[1] Paraphrased from John W. Berman: "There's never enough time to do it right, but there's always enough time to do it over."

- Segger's Ozone debugger software. This is available for Windows, Linux, and macOS (see `www.segger.com/products/development-tools/ozone-j-link-debugger/`).
- Target hardware.
    - STMicroelectronics' NUCLEO-F413ZH development board.
    - Or Adafruit's Grand Central M4 Express board (which requires a Segger J-Link for programming).

I use Microsoft Windows as the host environment, and I use Windows tools for development. However, the code base itself supports being developed in other host environments (e.g., Linux or macOS). Detailed setup instructions are provided in Appendix A, "Getting Started with the Source Code."

Regulated Industries

Most of my early career was spent working in domains with no or very minimal regulatory requirements. But when I finally did work on medical devices, I was pleased to discover that the best practices I had accumulated over the years were reflected in the quality processes required by the FDA or EMA. Consequently, the processes presented here are applicable to both nonregulated and regulated domains. Nevertheless, if you're working in a regulated industry, you should compare what is presented here against your specific circumstances and then make choices about what to adopt, exclude, or modify to fit your project's needs.

What Is Not Covered

There are several aspects to this software development approach that I don't spend much time defending or explaining. For example, I make the following assumptions:

- Software architecture is done before detailed design and implementation.
- Software architecture and detailed design are two separate deliverables.
- Detailed design is done before coding.
- Unit tests, as well as automated unit tests, are first class deliverables in the development process.
- Continuous integration is a requirement.
- Documentation is a useful tool, not a process chore.

Additionally, while they are worthy topics for discussion, this book only indirectly touches on the following:

- Multithreading
- Real-time scheduling
- Interrupt handling
- Optimizing for space and real-time performance
- Algorithm design
- User interface design
- How to work with hardware peripherals (ADC, SPI, I2C, UART, timers, input capture, etc.)

This is not to say that the framework does not support multithreading or interrupt handling or real-time scheduling. Rather, I didn't consider this book the right place for those discussion. To extend the cookbook metaphor a little more, I consider that a list of ingredients. And while ingredients are important, I'm more interested here in the recipes that detail how to prepare, combine, and bake it all together.

Conclusion

Finally, it is important to understand that this book is about how to productize software, not a book on how to evaluate hardware or create a proof of concept. In my experience, following the processes described in this book will provide you and your software team with the tools to achieve a high-quality, robust product without slowing down the project timeline. Again, for a broader discussion of why I consider these processes best practices, I refer you to *Patterns in the Machine*,[2] which makes the case for the efficiency, flexibility, and maintainability of many of these approaches to embedded software development.

[2] John Taylor and Wayne Taylor. *Patterns in the Machine: A Software Engineering Guide to Embedded Development*. Apress Publishers, 2021

CHAPTER 2

Requirements

Collecting requirements is the first step in the planning stage. This is where you and your team consolidate the user and business needs into problem statements and then define in rough terms how that problem will be solved. Requirements articulate product needs like

- Functions
- Capabilities
- Attributes
- Capacities

Most of these statements will come from other disciplines and stakeholders, and the requirements will vary greatly in quality and usefulness. Usually, good requirements statements should be somewhat general because the statement shouldn't specify how something should be done, just that it needs to be done. For example, this statement would be far too specific as a requirement:

> *The firmware shall implement a high pass filter using FFT to attenuate low frequencies.*

A better requirement would simply state what needs to be done:

> *The firmware shall remove high frequency interference from the device signal.*

In the requirements phase, then, the scope of the problem-solving is to “draw the bounding box” for the detailed solution. Here are some examples of how general requirements can be:

- The physical footprint shall be no larger than a bread box.
- The computing platform will be limited to a microcontroller.
- The total bill of materials and manufacturing costs shall not exceed $45.
- The device shall operate effectively in these physical environments: land, sea, and air.

These written requirements become the inputs for the second step in the planning phase. Most of the time, though, the analysis step needs to start before the requirements have all been collected and agreed upon. Consequently, don’t burden yourself with the expectation that all the requirements need to be defined before exiting the requirements step. Rather, identify an initial set of requirements with your team as early as possible to ensure there’s time to complete the analysis step. The minimum deliverable or output for the requirements step is a draft set of requirements that can be used as input for the analysis step.

Formal Requirements

Typically, requirements are captured in a table form or in a database. If the content of your requirements is presented in a natural language form or story form that is often referred to as a product specification. In my experience, a product specification is a better way to communicate to people an overall understanding of the requirements; however, a list of formal requirements is a more efficient way to track work items and features, especially when it comes to creating test plans and verifications. Either a product specification or a list of requirements constitutes formal requirements.

Generally, when I refer to formal requirements, I simply mean requirements that have been written down. The format is not important if the following minimum attributes are part of the requirements:

- Unique identifier—This identifier can be in any format as long as it is unique across all the entire scope of requirements for the project. After the requirement is written, the identifier should not be changed or reused. For this reason, you do not want to use a document's section numbering as the unique identifier because when the document is modified, the section numbering can change.
- Name—A text label assigned to a requirement that provides a short description or summary of the requirement.
- Description—This is the descriptive text of a requirement. Verbs such as *shall* and *should* are used to indicate whether the requirement is a "must-have" or a "nice-to-have." In addition, requirements must be testable; so phrases such as *shall be easy to use* are problematic. Rather, the description should incorporate measurable criteria such as *shall receive an average usability score of 8 or above in focus group testing*.
- Targeted release—This attribute allows the stakeholders to provide a road map for their product so that the current design cycle can accommodate future functionality.

Functional vs. Nonfunctional

There are two basic types of requirements: functional and nonfunctional. Functional requirements are those that define what a product or system is supposed to do; nonfunctional requirements (NFRs) define how the product or system should do it. NFRs are sometimes referred to as the "quality attributes" of a system. NFRs are also harder to write due to defining their pass/fail criterion. It is often difficult to articulate the tests that determine if the user interface is "intuitive" or that the user experience with the product is "first class." Some common words that are used in NFRs that should alert you to areas of concern are

- Availability
- Compatibility
- Reliability
- Maintainability
- Manufacturability
- Regulatory
- Scalability

As the name "nonfunctional" implies, the operation of the product is not impacted if one or more NFRs are not met. However, the stakeholder expectations or business needs may not be met if some of the NFRs are not achieved.

Sources for Requirements

Requirements will come to you from various sources and people involved in the product development process. For example, you may get requirements from people in the following roles:

- Product manager or marketing group—The product manager is typically the primary stakeholder and provides the top-level, most abstract requirements. I label these as Marketing Requirements Specifications (MRS).
- System engineer—The system engineering role is responsible for decomposing the MRS requirements to generate the system-level requirements that constitute a solution to the original MRS requirement. I label these as Product Requirements Specifications (PRS). Nominally the PRS requirements are traceable back to one or more MRS requirements. The PRS is essentially the project team's response to the MRS.
- Engineering (Software, Electrical, Mechanical)—The various engineering disciplines decompose the PRS requirements into yet more detailed requirements that are specific to their disciplines. This is typically the most detailed level of requirements. I label the requirements from software engineers as Software Requirements Specifications (SRS). Nominally the SRS requirements (as well as requirements from other engineering disciplines) are traceable back to either PRS or MRS requirements.
- Other roles (Quality, Regulatory, etc.)—Other functional groups within your company may provide you with requirements, and these requirements should be recorded as either MRS- or PRS-level requirements.

The initial set of requirements coming out of the planning stage will be a mix of MRS, PRS, and SRS requirements. This is to be expected as no development life cycle is truly waterfall.

Note Capturing and managing requirements is a system engineering role, not a software role. Consequently, a more detailed discussion of requirements, requirements management, and best practices is outside the scope of this book. Nevertheless, it is not unusual for a software engineer to fill the role of the system engineer.

Challenges in Collecting Requirements

In theory, this requirements step should be easy; it is primarily a collection effort where you go gather up all the requirements from your stakeholders. In practice, though, it turns out to be much more difficult because people always seem to want more information before they are willing to make a decision. You may find that you and your software team may find yourselves extracting requirements from stakeholders in the same way a dentist extracts teeth from recalcitrant patients. That is, there may be some browbeating involved in getting the stakeholders to commit to something. And sometimes it can get to the point where you just have to say, "Look, if you want me to build something in this decade, you're going to have to commit: Do you want it this way or do you want it that way?"

You and your software team will also need to make some strategic decisions about how to fill in the gaps from missing requirements. You may have to guess or "make something up" so the downstream work can progress. Filling these gaps is something of a leap of faith, and there is a good chance you'll get it wrong, which means some re-work will be required later in the project. Nevertheless, there are times when you just need to decide *something* so you can set up an implementation path. I have found, though, that if you make these decisions with a good understanding of what the product is supposed to be, the inevitable changes and adaptations that are required after the *real decision* is made will be less significant, less structural, and more cosmetic.

Exiting the Requirements Step

The requirements that are available when the first draft of the software architecture document is generated will only be a subset of the final requirements. In addition, some of the initial requirements will change (or be deleted) as the project moves forward. These changes in requirements dictate that the software architecture document be revisited to make sure it is still appropriate for the newest set of requirements. However, in my experience, if the initial core set of requirements was "mostly correct" in its intent, then the churn on the software architecture document is minimal. This is because the software architecture document is not detailed design; instead, it defines the boundaries of the detailed software solutions.

GM6000

Table 2-1 is the list of initial requirements for a hypothetical heater controller that I like to call the GM6000. This list is intended to illustrate the kinds of requirements that are available when you start to develop the software architecture in the analysis step. As you make progress on the software architecture, additional requirements will present themselves, and you will need to work with your extended team to get the new requirements included in the MRS or PRS requirements documents.

***Table 2-1.** Requirements captured at the start of the analysis step*

Req#	*Name*	*Requirement*	*Rel*[1]
MR-100	*Heating system*	*The Digital Heating Control (DHC) system shall provide indoor heating based on space temperature and a user-supplied heat setting.*	*1.0*
MR-101	*Heating enclosures*	*The DHC shall support at least three different heater enclosures. The heating capacity of each heater enclosure can be different from the other enclosures.*	*1.0*
MR-102	*Control board*	*DHC shall have a single control board that can be installed in many heater enclosures.*	*1.0*
MR-103	*Control algorithm*	*The heater control algorithm in the control board shall accept parameters and configurations that customize the algorithm for a specific heater enclosure.*	*1.0*
MR-104	*Provisioning*	*The DHC control board shall be provisioned to a specific heater enclosure during the manufacturing process. The provisioning shall include the heater control's algorithm parameters and configuration.*	*1.0*
MR-105	*Wireless input*	*The control board shall support connecting to a wireless module for communicating with a wireless temperature input.*	*2.0*
MR-106	*Wireless sensor*	*The DHC system shall support an external, wireless temperature sensor.*	*2.0*

(*continued*)

[1] Commercial release, where Rel 1.0 is the initial product release.

Table 2-1. (*continued*)

Req#	*Name*	*Requirement*	*Rel*[1]
MR-107	*User interface*	*The DHC unit shall support display, LEDs, and user inputs (e.g., physical buttons, keypad membrane, etc.). The arrangement of the display and user inputs can be different between heater enclosures.*	*1.0*
MR-108	*User actions*	*The DHC display, LEDs, and user inputs shall allow the user to do the following:* • *Turn the heater on and off* • *Set the maximum fan speed* • *Specify the temperature set point*	*1.0*
MR-109	*User information*	*The DHC display LEDs shall provide the user with the following information:* • *Current temperature* • *DHC on/off state* • *Active heating state* • *Fan on/off state* • *Alerts and failure conditions*	*1.0*
PR-100	*Sub-assemblies*	*The DHC heater closure shall contain the following sub-assemblies:* • *Control Board (CB)* • *Heating Element (HE)* • *Display and User Inputs (DUI)* • *Blower Assembly (BA)* • *Power Supply (PS)* • *Temperature Sensor (TS)*	*1.0*

(*continued*)

Table 2-1. (*continued*)

Req#	*Name*	*Requirement*	*Rel*[1]
PR-101	*Wireless module*	*The DHC heater closure shall contain the following sub-assemblies:* • *Wireless Module (WM)*	*2.0*
PR-103	*Heater safety*	*The Heating Element (HE) sub-assembly shall contain a hardware temperature protection circuit that forces the heating source off when it exceeds the designed safety limits.*	*1.0*
PR-105	*Heater element interface*	*The Heating Element (HE) sub-assembly shall have a proportional heating output interface to the Control Board (CB).*	*1.0*
PR-106	*Blower assembly interface*	*The Blower Assembly (BA) sub-assembly shall have a proportional speed control interface to the Control Board (CB).*	*1.0*
PR-107	*Temperature sensor*	*The Temperature Sensor (TS) sub-assembly shall use a thermistor for measuring space temperature.*	*1.0*

A complete set of final requirements for the GM6000 can be found in Appendix H, "GM6000 Requirements."

Summary

The goal of the requirements step is to identify the problem statement presented by the user and business needs. In addition, a high-level solution is identified and proposed for the problem statement. Both the problem statement and the high-level solution are captured in the form of formal requirements.

INPUTS

- User needs
- Business needs

OUTPUTS

- A subset of the documented final requirements. At a minimum, the subset must be sufficient to complete the analysis step.

CHAPTER 3

Analysis

In the analysis step of the planning stage, you will create three artifacts:

- System architecture (SA)—The system architecture document describes the discrete pieces or units of functionality that will be tied together to make the product. It consists of diagrams with boxes and lines whose semantics usually mean "contains" and "is connected to."
- Software architecture (SWA) documents—The software architecture, on the other hand, provides the designs for the system components that describe how each unit works. These designs usually contain diagrams that are more sophisticated in that they may be structural or behavioral and their lines and boxes often have more particular meanings or rules associated with them (like UML diagrams).
- Requirements trace matrix—The requirements trace matrix is generally a spreadsheet that allows you to map each requirement to the code that satisfies it and the tests that validate it.

In this chapter, I've pulled examples from Appendix I, "GM6000 System Architecture," and Appendix J, "GM6000 Software Architecture," to provide examples of how sections in these documents might look. Be aware that these examples use the "sparse language of work" because the intended

J. T. Taylor and W. T. Taylor, *The Embedded Project Cookbook*,
https://doi.org/10.1007/979-8-8688-0327-7_3

audience for these documents is other members of the development team who are expected to be familiar with these details. That is, I may refer to algorithms, standards, or constructs without much if any explanation. For example, when an example document specifies that the "enclosure for the project should be IP51 rated," it does not discuss or explain how the Ingress Protection (IP) rating system works, nor, when a document specifies that the "implementation must guarantee that there are no nested mutex calls when it locks its internal mutex," do I add that "critical sections are evil, and while they're handy-dandy and very easy to use, they will always get you in trouble."

System Engineering

I will note here that system engineering is not a software role. And while it is not unusual for a software engineer to fill the role of the system engineer, a discussion about the intricacies of developing the system architecture is outside the scope of this book. But, as it is an essential input to the software architecture document, Appendix I, "GM6000 System Architecture," provides an example of a system architecture document for the GM6000.

GM6000 System Architecture

Figure 3-1 shows an example of a system-level diagram for the GM6000. In Appendix I, "GM6000 System Architecture," each diagram is followed by a table of supporting text for each label block. However, in this excerpt from the appendix, as well as in all the following excerpts, only a few of the descriptions are shown to give you an idea of the level of detail expected.

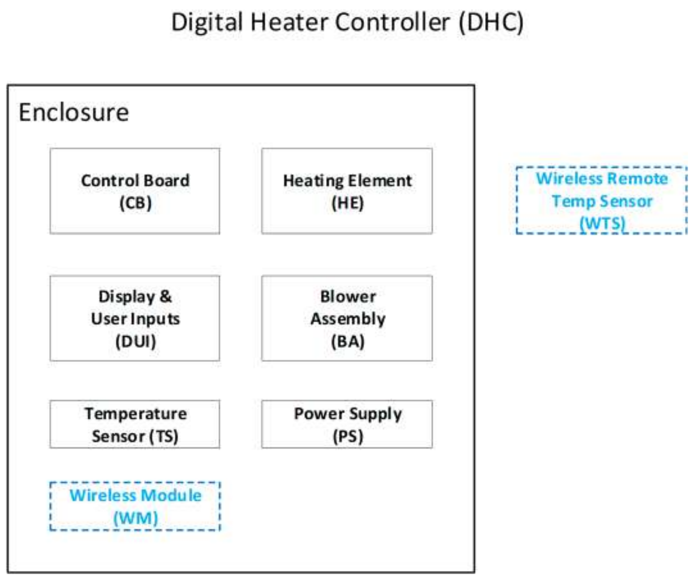


Figure 3-1. *System architecture block diagram*

Component	***Description***
Enclosure	*The box that contains the product. The enclosure should be IP51 rated.*
Control Board (CB)	*The board that contains the microcontroller that runs the heater controller. The CB contains circuits and other chips as needed to support the microcontroller unit (MCU) and software.*
Display and User Inputs (DUI)	*A separate board that contains the display, buttons, and LEDs used for interaction with the user. This DUI can be located anywhere within the enclosure and is connected to the CB via a wire harness.*
…	…

Software Architecture

There is no canonical definition of software architecture. This cookbook defines software architecture as

> *Identifying the solution at a high level and defining the rules of engagement for the subsequent design and implementation steps.*

From a top-down perspective, software architecture is the top of the software solution pyramid (even though software architecture is not a strictly top-down activity). When creating the software architecture, you will need to make design decisions in the following areas:

- Hardware interfaces.
- Performance constraints.
- Programming languages.
- Subsystems. This includes defining the rules for how these the subsystems will interface with each other and share data.
- Subsystem interfaces.
- Process model.
- Functional simulator.
- Cybersecurity.
- Memory allocation.
- ITC (Inter-thread communication) and IPC (Inter-process communication).

- File and directory organization.
- Localization and internationalization.
- Other areas of concern that are specific to your industry, company, or product. For example, a medical device must have an IEC 62304 software safety classification.

While you're doing all this, it is important to remember that your definitions should be as detail agnostic as possible. I know that sounds strange, but the software architecture is about

> *Defining the logical framework and boundaries for downstream detailed design.*

For example, Figure 3-2, in the "Hardware Interfaces" section that follows, is a hardware block diagram for this book's sample project, the GM6000 digital heater controller. The illustration was taken from the example software architecture document that is provided in Appendix J, "GM6000 Software Architecture." While at first glance it may seem fairly detailed—and it is detailed because it is delineating hardware interfaces and components—upon closer inspection, you will see that no specific hardware part numbers are called out. No specific type of serial bus is specified for communication between components (e.g., I2C, SPI, etc.). While "User Inputs" are called out, no specific type of button nor number of buttons is specified. And even though external storage is present, there is nothing that requires any particular type of storage nor amount.

In summary, then, the goal of this chapter is to describe how you can create the first draft of the software architecture. You will invariably have to create additional drafts as the two key inputs for this step—the extant set of requirements and the system architecture document—will evolve. Unfortunately, in the real world, these inputs are not waterfall handoffs; the details are derived over time during the planning stage.

As time goes on, it is important that when there is a "story" or theory-of-operations description of the system, the team clearly defines and enforces the canonical source for the requirements.

Moving from Inputs to Outputs

In Chapter 2, "Requirements," I provided one block diagram and approximately 20 requirements that were a mix of marketing and engineering requirements. But there are significant requirements and details missing. For example, there is nothing called out for

- A temperature control algorithm (e.g., error vs. PID control)
- The type of display (fixed segment vs. graphic, resolution, color depth, etc.)
- The UI workflow requirements
- The specifics of the interfaces to the Heating Element (HE) and Blower Assembly (BA) sub-assemblies
- Firmware updates
- Diagnostics
- Cybersecurity

Nevertheless, the inputs I provide are sufficient to create the first draft of the software architecture.

The following sections discuss the creation of the content of the software architecture document. While you are performing the analysis and making architectural decisions, you'll discover that there are still a lot of unknown or open questions. Some of these questions may be answered during the planning stage. Some are detailed design questions that will be answered in the construction stage. Either way, keep track of these open questions because they will eventually have to be answered.

Hardware Interfaces

Create a hardware block diagram in relation to the microcontroller. That is, outline the inputs and outputs to and from the microcontroller (see Figure 3-2). Whenever possible, omit details that are not critical to understanding the functionality of the inputs and outputs. For example, simply identify that there will be "external serial data storage." Do not call out a specific chip, storage technology (e.g., flash vs. EEPROM), or specific type of serial bus.

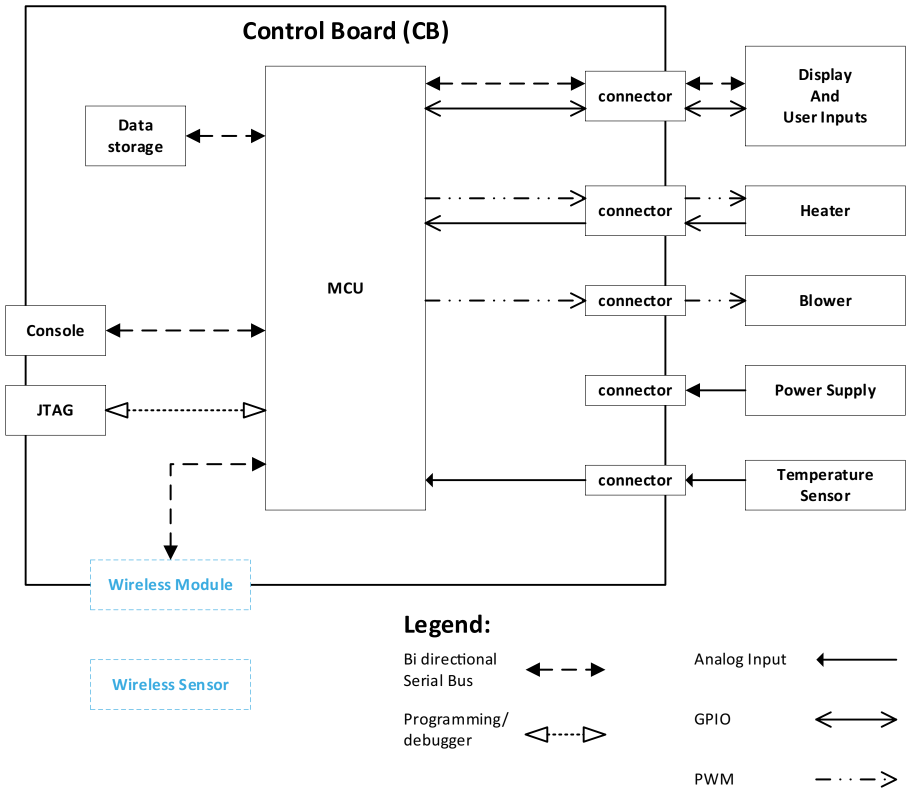


Figure 3-2. *Hardware block diagram*

Component	***Description***
...	...
Data Storage	*Serial persistent data storage for saving configuration, user settings, etc.*
...	...

In creating the hardware block diagram, I made the following decisions, which I fed back into the project's requirements prior to exiting the analysis stage:

1. The selected microcontroller must support at least four serial buses (HWR-200).
2. The display will have a serial bus interface (HWR-201).
3. The heater and fan control will be accomplished using a Pulse Width Modulation (PWM) signal (HWR-202, HWR-203).
4. The control board will have UART-based console for provisioning and debugging support (SWR-200, 201, 203).

Performance Constraints

For all of the identified hardware interfaces, you will want to make an assessment of real-time performance and bandwidth usage. Also you should make performance assessments for any applications, driver stacks, crypto routines, etc., that will require significant CPU usage. Since the specifics of these interfaces are still unknown, the assessment will be an approximation—that is, an order-of-magnitude estimate—rather than a precision value. Note that I use the term "real time" to describe contexts where stimuli must be detected and reacted to in less than one second. Events and actions that occur slower than 1 Hz can be achieved without special considerations.

The following are some excerpts from the software architecture document in Appendix J, "GM6000 Software Architecture," that illustrate the performance analysis.

Display

The microcontroller unit (MCU) communicates with the display controller via a serial bus (e.g., SPI or I2C). There is a time constraint in that the physical transfer time for an entire screen's worth of pixel data (including color data) must be fast enough to ensure a good user experience. There is also a RAM constraint with respect to the display in the MCU; it requires that there will be at least one off-screen frame buffer that can hold an entire screen's worth of pixel data. The size of the pixel data is a function of the display's resolution times the color depth. The assessments and recommendations are as follows:

- *The maximum size of the pixel data is limited to 64KB to meet the timing and RAM constraints.*
- *Assuming a 16 MHz serial bus, the wire transfer time for a full screen is 41 msec without accounting for protocol overhead, ISR jitter, thread context switches, etc.*

Temperature Sensor

The space temperature must be sampled and potentially filtered before being used as an input to the control algorithm. However, controlling space temperature is a relatively slow system (i.e., much slower than 1 Hz). Consequently, the assessments and recommendations are as follows:

- *No real-time constraints on space temperature sampling or filtering*

Threading

A Real-Time Operating System (RTOS) with many threads will be used. Switching between threads—that is, a context switch—requires a measurable amount of time. This becomes important when there are sub-millisecond timing requirements and when looking at overall CPU usage. The RTOS also adds timing overhead for maintaining its system tick timer, which is typically interrupt based. The assessments and recommendations are as follows:

- *The RTOS context switching time and tick counter overhead only needs to be considered when there are requirements for response or detection times that are less than 1 msec.*

Programming Languages

Selecting a programming language may seem like a trivial decision, but it still needs to be an explicit decision. The experience of the developers, regulatory considerations, performance, memory management, security, tool availability, licensing issues, etc., all need to be considered when selecting the programming language. The language choice should be documented in the software architecture document as well as in the Software Development Plan.

In most cases, I prefer to use C++. I know that not everyone agrees with me on this, but I make the case for the advantages of using C++ in *Patterns in the Machine*. I mention it here to say that you should not exclude C++ from consideration simply because you are working in a resource-constrained environment.

Subsystems

You will want to decompose the software for your project into components or subsystems. The number and granularity of the components are a choice that you will have to make. Some things to consider when defining subsystems are as follows:

- All of the source code that executes on the microcontroller should be contained within one of the identified subsystems. If you find yourself writing code that does not fit within the description of one of the subsystems, then you need to revise the software architecture document, or reevaluate if the new code is really needed for the project.
- Each subsystem must be traceable back to at least one formal requirement. That is, there must be at least one MRS, PRS, or SRS requirement that the subsystem satisfies or partially satisfies.
- Subsystems are not layers. In some instances, subsystems do map to layers, but they represent functionality.
- When selecting and naming a subsystem, start with a description of the scope of the subsystem. It is sometimes helpful to explicitly call out what is not included in the subsystem.
- Separate subsystems by functionality or behavior, not by anticipated code size. That is, it is okay to have a subsystem that is implemented with a minimal amount of code.

- Do not think of subsystems as directories or C++ namespace names. The code for a given subsystem can, and often will be, spread across multiple directories and namespaces.
- There is no minimum or maximum number of subsystems. That said, subsystems such as the Board Support Package (BSP), operating system (OS), operating system abstraction layer (OSAL), drivers, system services, and bootloader are ubiquitous for most microcontroller applications. It is not uncommon for many of the subsystems you identify for a given project to be the same as other projects, even when those projects are completely different.

The GM6000 control board software is broken down into the following subsystems. In Figure 3-3, the subsystems in the dashed boxes represent future or anticipated functionality.

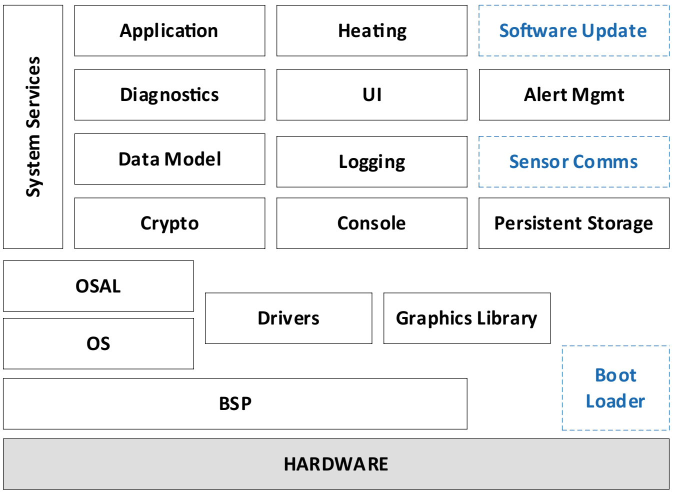


Figure 3-3. *Subsystems*

Here are some excerpts from the software architecture document in Appendix J, "GM6000 Software Architecture," that describe the scope of the subsystems.

Application

The application subsystem contains the top-level business logic for the entire application. This includes functionality such as

- *The top-level state of the entire device*
- *Creating, starting, and stopping all other subsystems*

BSP

The Board Support Package (BSP) subsystem is responsible for abstracting the details of the microcontroller unit (MCU) datasheet. For example, it is responsible for

- *The low-level code that directly configures the MCU's hardware registers*
- *Encapsulating the MCU vendor's supplied SDK, including any modifications or extensions needed*
- *Compiler-dependent constructs (e.g., setting up the MCU's vector table)*

Diagnostics

The diagnostics subsystem is responsible for monitoring the software's health, defining the diagnostics logic, and self-testing the system. This includes features such as power on self-tests and metrics capture.

Drivers

The driver subsystem is the collection of driver code that does not reside in the BSP subsystem. Drivers that directly interact with hardware are required to be separated into layers. There should be at least three layers:

- *A hardware-specific layer that is specific to a target platform*
- *A platform-independent layer. Ideally the majority of the business logic is contained in this layer.*
- *A Hardware Abstraction Layer (HAL) that separates the aforementioned two layers. More specifically, the hardware-specific layer implements the HAL, and the platform-independent layer calls the HAL.*

Graphics Library

The graphics library subsystem is responsible for providing graphic primitives, fonts, window management, widgets, etc. The expectation is that the graphic library will be third-party software. The minimum requirements for the graphics library are as follows:

- *It is platform independent. That is, it has no direct dependencies on a specific MCU or physical display.*
- *It supports a bare-metal runtime environment. That is, it can be used with or without an RTOS. This constraint is important because it allows the OSAL's event-based threads to be used directly for events, timers, and ITC messages within the thread where the UI executes. This eliminates the need for an adapter layer to translate the system services events into graphic-library-specific events.*

Heating

The heating subsystem is responsible for the closed loop space temperature control. This is the code for the heat control algorithm.

Persistent Storage

The persistent storage subsystem provides the framework, interfaces, data integrity checks, etc., for storing and retrieving data that is stored in local persistent storage. The persistent storage paradigm is a RAM-cached model. The RAM-cached model is as follows:

- *On startup, persistent record entries are read from nonvolatile storage, validated, and loaded into RAM. If the data is invalid, then the associated RAM values are set to factory default values, and the nonvolatile storage is updated with the new defaulted values.*
- *The application updates the entries stored in RAM via an API. When an update request is made, the RAM value is updated, and then a background task is initiated to update the values stored in nonvolatile storage.*

UI

The user interface (UI) subsystem is responsible for the business logic and interaction with end users of the unit. This includes the LCD display screens, screen navigation, consuming button inputs, LED outputs, etc. The UI subsystem has a hard dependency on the graphic library subsystem. This hard dependency is acceptable because the graphic library is platform independent.

Subsystem Interfaces

This section is where you define how the various subsystems, components, modules, and drivers will interact with each other. For example, it addresses the following questions:

- Are all the interfaces synchronous?
- What interfaces need to be asynchronous?
- How do drivers interact with application modules?
- Does using the data model architecture make sense?
- Is inter-thread communication (ITC) or inter-process communication (IPC) needed?

Depending on the scope and complexity of the project, there may be one or many interface designs. Do not try to force a single paradigm here. If things are similar but different, they are still different. This is also an area where it helps to have a good understanding of both the detailed design of the project as well as the specific implementation of some of the modules. Taking a deeper dive into the details that are available can be very beneficial at this stage.

The following is a snippet of the "Interfaces" section from the software architecture document in Appendix J, "GM6000 Software Architecture."

Interfaces

The preferred, and primary, interface for sharing data between subsystems will be done via the data model pattern. The secondary interface will be message-based inter-thread communications (ITC). Which mechanism is used to share data will be determined on a case-by-case basis with the preference being to use the data model. However, the decision can be "both" because both approaches can co-exist within a subsystem.

The data model pattern supports publish-subscribe semantics as well as a polling approach to sharing data. The net effect of this approach is that subsystems are decoupled. That is, there are no direct dependencies on other subsystems. In Figure 3-4, the arrows represent dependencies between the subsystems. Note that model point data is bidirectional between the subsystems.

A third option for a subsystem interface is a functional API. This should be used for subsystems (e.g., system services) that have no external dependencies with other subsystems and that typically only execute when their APIs are called by other subsystems.

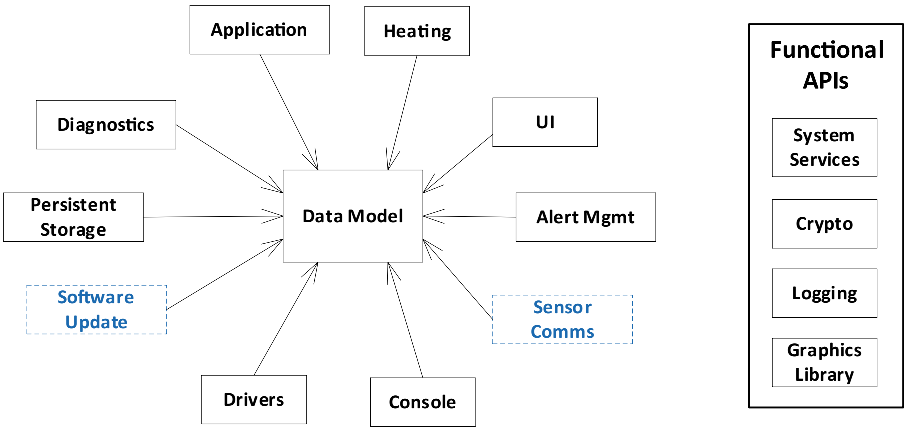


Figure 3-4. *Subsystem interdependencies using the data model*

Process Model

The following questions need to be answered in the architecture document:

- Will the application be a bare-metal or main-loop application?
- Will the application use threads?
- Will the application have multiple processes?
    - Is the system architecture a multiprocessor design? (e.g., is it a main board running Linux with an off-board microcontroller that performs all the hard real-time processing?)

If the system uses threads or processes, I recommend that you document why that decision was made. Articulate what problems or requirements having threads solves. If the system uses multiple processors, define the roles and responsibilities of each processor. When multiple threads and processes are being used, the software architecture needs to also address

- Thread and process priorities
- Data integrity when sharing between threads and processes

The following is the “Process Model” section from the software architecture document in Appendix J, “GM6000 Software Architecture.”

Process Model

The software will be implemented as a multithreaded application using real-time preemptive scheduling. The preemptive scheduling provides for the following:

- *Decoupling the UI from the rest of the application (with respect to timing and sequencing). This allows the UI to be highly responsive to the user actions without delaying or blocking time-critical business logic.*
- *Using deferred interrupt handlers. A deferred interrupt handler is where the time-consuming portion of the interrupt service routine is deferred to a high-priority thread in order to make the interrupt service routine as short as possible.*
- *Simpler sequential designs that utilize nonbusy, blocking wait semantics*

Thread Priorities

The application shall be designed such that the relative thread priorities between individual threads do not matter with respect to correctness. Correctness in this context means the application would still function, albeit sluggishly, and not crash if all the threads had the same priority. The exception to this rule is for threads that are used exclusively as deferred interrupt handlers.

Data Integrity

Data that is shared between threads must be implemented in a manner to ensure data integrity. That is, read, write, read-modify-write operations must be atomic with respect to other threads accessing the data. The following is a list of allowed mechanisms for sharing data across threads:

- *Data is shared using data model point instances. This is the preferred mechanism.*
- *Data is shared using the system services inter-thread-communication (ITC) message passing interfaces. ITC messaging is recommended for when the interface semantics have a many-to-one relationship between the clients and servers and the clients are sending data to the server.*
- *Synchronous ITC messaging—where the client is blocked while the server processes the message—is only allowed when the server actions are well bounded in time and are of short duration. For example, invoking an HTTPS request to an off-board website is considered an unbounded transaction because there are too many variables associated with determining when a request will complete or fail (e.g., TCP retry timing, routing, website availability, etc.).*
- *Asynchronous ITC messaging can always be used.*
- *Encapsulated API is allowed but discouraged. This is where the API implementation uses an internal mutex (which is not exposed to the API's clients) to provide atomic data access. In addition, the internal implementation must guarantee that there are no nested mutex calls when it locks its internal mutex. This approach should only be used as a last resort option and must be clearly documented in the detailed design.*

When sharing data between a thread and an interrupt service routine (ISR), the critical section mechanism shall disable or enable the ISR's specific interrupt. Relying on the MCU's instruction set for atomic read, write, read-modify-write operations to a memory location is strongly discouraged.

If the MCU is used, it must be clearly documented in the detailed design. The following guidelines shall be followed for sharing data between a thread and ISRs:

- *Temporarily suspend thread scheduling before disabling the interrupt, and then resume scheduling after enabling the interrupt. This ensures that there will not be a thread context switch while the thread has disabled the interrupt.*
- *Keep the time that the interrupt is disabled as short as possible. Your process should not be doing anything else except moving data and clearing hardware registers while it has an interrupt disabled.*
- *Only disable or enable a specific interrupt. Never globally disable or enable interrupts.*

Functional Simulator

If your project requires you to implement automated unit tests for your project, you will find that a lot of the work that goes into creating a functional simulator will be done while creating the automated unit tests. The reason is because the automated unit tests impose a decoupled design that allows components and modules to be tested as platform-independent code. Consequently, creating a simulator that reuses abstracted interfaces is not a lot of additional effort—if it is planned for from the beginning of the project. The two biggest pieces of work are as follows:

- Identifying the necessary abstraction layers (OSAL, HAL, etc.)
- Determining the strategy for breaking dependencies on the Board Support Package (BSP)

The planning and implementation of this separation strategy results in minimal additional effort to create a functional simulator.

Software architecture decisions that need to be made to implement a functional simulator are as follows:

- What platform will the simulator run on (e.g., Windows, Linux)?
- What portions will be simulated? The entire application? Just the algorithms? The entire system?
- How will the hardware elements be simulated? Mocked drivers? Full hardware emulation? Functional hardware simulation?
- Are there communication channels that need to be simulated?
- Is simulated time needed or desirable? Simulating time is very useful for projects that have algorithms that are time based or where running faster than real time is desirable (e.g., simulating one month of operation in ten minutes).

The following is the "Simulator" section from the software architecture document in Appendix J, "GM6000 Software Architecture."

Simulator

The software architecture and design accommodate the creation of a functional simulator. A functional simulator is the execution of production source code on a platform that is not the target platform. A functional simulator is expected to provide the majority of the functionality but not necessarily the real-time performance of the actual product. Or, more simply, functional simulation enables developers to develop, execute, and *test production code without the target hardware. Figure 3-5 illustrates what is common and different between the software built for the target platform and software built for the functional simulator. The architecture for the functional simulator is on the right.*

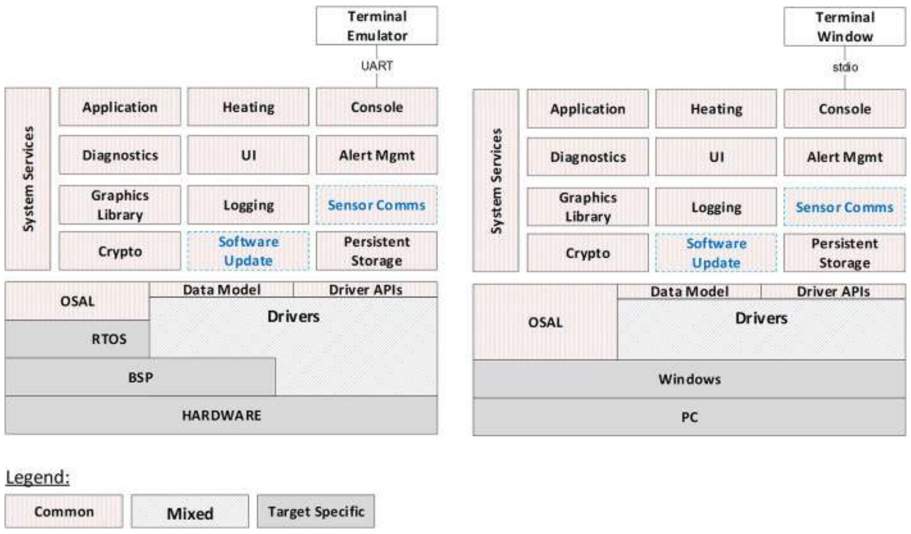


Figure 3-5. *Software architecture with and without a functional simulator*

The functional simulator has the following features, attributes, and limitations:

- *The functional simulator shall be a Windows console executable.*
- *The executable's* `stdio` *is the stream interface for application's console.*
- *The Windows file system will be used to "mock" the nonvolatile storage of the product.*
- *The bootloader is not simulated.*

During detailed design of the various drivers, make the decisions as to how each hardware driver will be simulated. Allowed options for simulating hardware are as follows:

- *Mocked–A mocked simulation is where you provide a minimal implementation of the device's HAL interface so that the application compiles, links, and does not cause aberrant behavior at runtime.*
- *Simulated–A simulated device is where only the core functionality of the device is implemented.*
- *Emulated–An emulated device is where you replicate a device's behaviors at its lowest, most basic level.*

Cybersecurity

Cybersecurity may or may not be a large topic for your project. Nevertheless, even if your initial reaction is "there are no cybersecurity concerns for this product," you should still take the time to document why there are no concerns. Also note that I include the protection of personally identifiable information (PII) and intellectual property (IP) as part of cybersecurity analysis.

Sometimes just writing down your reasoning as to why cybersecurity is not an issue will reveal gaps that need to be addressed. Depending on the number and types of *attack surfaces* your product has, it is not uncommon to break the cybersecurity analysis into its own document. And a separate document is okay; what is important is that you do some analysis and document your findings.

A discussion of the best practices and methodologies for performing cybersecurity analysis is beyond the scope of this book. For example, with the GM6000 project, the cybersecurity concerns are minimal. Here is a snippet of the "Cybersecurity" section from the software architecture document that can be found in Appendix J, "GM6000 Software Architecture."

Cybersecurity

The software in the GM6000 is considered to be a low-risk target in that it is easier to compromise the physical components of a GM6000 than the software. Assuming that the software is compromised, there are no safety issues because the HE has hardware safety circuits. The worst-case scenarios for compromised software are along the lines of denial-of-service (DoS) attacks, which might cause the DHC to not heat the space, yield uncomfortable temperature control, or run constantly to incur a high energy bill.

No PII is stored in persistent storage. There are no privacy issues associated with the purchase or use of the GM6000.

Another possible security risk is the theft of intellectual property. That is, can a malicious bad actor steal and reverse-engineer the software in the control board? This is considered low risk since there are no patented algorithms or trade secrets contained within the software and the software only has value within the company's hardware. The considered attack surfaces are as follows:

- *Console–The console's command-line interface (CLI) provides essentially super admin access to the software. To mitigate this, the console logic shall require a user to authenticate before being given access to commands.*

...

Memory Allocation

The architecture document should define requirements, rules, and constraints for dynamic memory allocation. Because of the nature of embedded projects, the extensive use of dynamic memory allocation is discouraged. When you have a device that could potentially run for years before it is power-cycled or reset, the probability of running out of heap memory due to fragmentation becomes a valid concern. Here is the "Memory Allocation" section from the software architecture document that can be found in Appendix J, "GM6000 Software Architecture."

To prevent memory leaks and fragmentation, no dynamic memory allocation is allowed. The application may allocate memory from the heap at startup, but not once the system is "up and running." This practice guarantees the system will not fail over time due to lack of memory.

For objects or structures that must be dynamically created or deleted after startup, the design is required to pre-allocate a memory pool on a per-type basis that will be used to construct the object or structure at runtime.

Inter-thread and Inter-process Communication

As discussed earlier in the "Process Model" section, the architecture may include multiple threads and processes. For this scenario, it is important to clearly define and restrict how inter-thread (ITC) and inter-process (IPC) communications are performed because ITC and IPC are, by definition, primary sources for race conditions and data corruption bugs.

The following is a snippet from the "Message Passing (ITC)" section in the software architecture document in Appendix J, "GM6000 Software Architecture."

Message Passing (ITC)

Data between threads can be shared using message passing. However, there shall be only one message passing framework. The framework has the following requirements and constraints:

- *The message passing model is a client-server model where clients send messages to servers. Messages can be sent asynchronously or synchronously.*

- *Data flow between clients and servers can be unidirectional or bidirectional as determined by the application. Because this is an inter-thread communication, data can be shared via pointers since clients and servers share the same address space.*
- *Data is shared between clients and servers using the concept of a payload. In addition, a convention of "ownership" is used to provide thread-safe access to the payload.*

File and Directory Organization

Source code file organization is very often done organically. That is, developers start writing code and organizing their files based on their immediate needs. File organization is a strategic best practice and should be well-thought-out before any source files are created. The file organization can either hinder or facilitate the construction of the build scripts and in-project reuse. This is especially critical for in-project reuse where you build multiple images and executables from your project's code base. Unit tests and the functional simulators are examples of in-project reuse. A well-thought-out, consistent file organization will also facilitate the creation and maintenance of continuous integration for your project.

The following is a snippet from the *File and Directory Organization* section in the software architecture document in Appendix J, "GM6000 Software Architecture."

Source code files shall be organized by dependencies, not by project. Or said another way, the code will be organized by C++ namespaces where namespaces map one to one with directory names. The exceptions to the namespace rule are for the BSP subsystem and third-party packages. In addition to namespaces, the following conventions shall be followed:

- *All in-house developed source code shall be under the top-level* `src/` *directory.*
- `#include` *statements in header files shall contain path information relative to the* `src/` *directory.*
- *There shall not be separate* `include/` *directories to contain header files. That is, do not separate header files and* `.c|.cpp` *files into different directories based solely on file type.*
- *Non-namespace directories can be created for organizational purposes. Non-namespace directory names shall be prefixed with a leading underscore.*

…

Localization and Internationalization

Localization and internationalization requirements may seem obvious. That is, you may be planning on creating a product for a single market or region. However, this requirement can change abruptly. Consequently, it is important to state the localization and internationalization requirements just to make sure everyone is on the same page. When it comes to localization, here are some examples of what to identify:

- Does your project need to display something other than numbers?
- Does your project require user input that is not menu driven or numerical? That is, does it require users to enter characters?
- What languages are you required to support? Specify these languages by country. The phrase "English only" is a very different thing than "US English only."
- If your project requires you to display characters, what encoding will you use? 7-bit ASCII has a low overhead, but 8-bit extended ASCII can give you the ability to display accent marks and other non-English characters. UTF-8 allows you to support the display of nearly every character in every language (assuming you can find a font), but it introduces the overhead that there is no longer a one-to-one mapping of the number of bytes to the number of characters in a string.

The following is the *Localization and Internationalization* section from the software architecture document in Appendix J, "GM6000 Software Architecture."

The product is targeted to be sold to North American consumers. The product is built for the US English-speaking market only (as determined by the Product Manager). Consequently, from a software perspective, the 7-bit ASCII character code is sufficient for all text presented to an end user.

Requirement Traceability

Requirement traceability refers to the ability to follow the path a requirement takes from design all the way through to a specific test case. There are three types of traceability: forward, backward, and bidirectional. Forward traceability is defined as starting with a requirement and working downward (e.g., from requirements down to test cases). Backward traceability is the opposite (e.g., from test cases up to requirements). Bidirectional is the ability to trace in both directions.

It is not uncommon for SDLC processes to include requirements tracing. In my experience, forward tracing requirements to verification tests is all part and parcel of creating the verification test plan. Forward tracing to design documentation and source code can be more challenging, but it doesn't have to be.

The following steps simplify the forward tracing of requirements to design artifacts (i.e., the software architecture and Software Detailed Design documents) and then to source code:

1. Label all "content section"[1] headings in the software architecture document with a unique identifier that does not change when the document is edited. For example, the GM6000 Software Architecture document uses the prefix `SWA-nn` to label each subsystem.

[1] Content section means any section that is not part of the boilerplate or the housekeeping sections. The introduction, glossary, and change log sections are examples of non-content sections.

2. All *subsystems* identified in the software architecture must be backward traceable to at least one formal MRS, PRS, or SRS requirement.

   a. If you have a subsystem that doesn't trace backward (e.g., you have a missing requirement), you need to revisit the requirements or your subsystem definition. (It's possible the subsystem is not needed.)

3. All SRS requirements must forward trace to at least one labelled section in the software architecture document.

   a. If an SRS requirement doesn't trace, then you need to revisit the requirements and the architecture to determine either what is missing or what is not needed.

There are similar rules for the detailed design document, which also support forward tracing from the software architecture to detailed design sections and then to source code. See Chapter 6, "Foundation," for details.

To see the requirements tracing for the GM6000, see the "Software Requirements Traced to Software Architecture" document in Appendix P, "GM6000 Software Requirements Trace Matrix." You will notice that the trace matrix reveals two orphan subsystems (SWA-12 Bootloader and SWA-26 Software Update). This particular scenario exemplifies a situation where the software team fully expects to create a feature—the ability to update software in the field—but there is no formal requirement because the software team hasn't reminded the marketing team that this would be a good idea … yet.

Even when your SDLC processes do not require requirements forward traceability to design artifacts, I still strongly recommend that you follow the steps mentioned previously because it is an effective mechanism for closing the loop on whether the software team implemented all the product requirements. It also helps prevent working on features that are not required for release.

In a perfect world, all traceable elements would have parents and children. That is, all PRS requirements would have at least one parent MRS requirement and all PRS requirements would have at least one SRS requirement. However, there will always be exceptions. For example, your Quality Management System (QMS) and your SDLC processes essentially impose implicit requirements. Or your SDLC may require design elements that facilitate the construction of automated unit tests. These implicit requirements are often not captured as formal requirements. This means that some common sense needs to be used when performing requirements tracing. Chapter 9, "Requirements Revisited," provides an additional discussion on this topic.

Summary

The initial draft of the software architecture document is one of the major deliverables for the analysis step in the planning stage. As discussed in the "Requirement Traceability" section, there is an auxiliary deliverable to the software architecture document, which is the trace matrix for the software architecture.

INPUTS

- An initial set or draft of the requirements
- An initial draft of the system architecture

OUTPUTS

- The first draft of the system architecture document
- The first draft of the software architecture document
- Requirements trace matrix for the software architecture

CHAPTER 4

Software Development Plan

Nothing in this step requires invention. The work here is simply to capture the stuff that the development team normally does on a daily basis. Creating the Software Development Plan (SDP) is simply making the team's informal processes formal.

The value add of the SDP is that it eliminates many misunderstanding and miscommunication issues. It proactively addresses the "I didn't know I needed to …" problems that invariably occur when you have more than one person writing software for a project. A written and current SDP is especially helpful when a new team member is added to the project; it is a great tool for transmitting key tribal knowledge to a new team member.

A large percentage of the SDP decisions are independent of the actual project. That is, they are part of your company's existing software development life cycle (SDLC) processes or Quality Management System (QMS). This means that work on the SDP can begin early and even be created in parallel with the requirements documents. That said, I recommend that you start the software architecture document before finalizing the first draft of the SDP. The software architecture will provide more context and scope for the software being developed than just the high-level requirements.

Another characteristic of Software Development Plans is that they mostly contain project-agnostic details. They capture processes and best practice details that apply to many projects. After you've created your team's first SDP, you essentially have a template for future SDPs, and 80% of the SDP work on the next project is already done.

It is also very helpful if your organization formalizes their software development processes and best practices. If they do, then the SDP can simply reference those documents instead of re-articulating them.

Project-Independent Processes and Standards

I recommend that you create the following development processes and standards for every project:

- Language programming standards and style guides, which include naming and file organization conventions
- Process descriptions of how you're going to run architecture and design reviews
- Process descriptions of how, and how often, you're going to run source code reviews
- Roles and responsibilities
- Process description of the bug tracking process, which includes how bugs are resolved and verified
- Software Configuration Management (SCM) repository branch model

- A description of how continuous integration (CI) will work
- Unit test requirements, which include how code coverage metrics will be determined
- Software best practices

Project-Specific Processes and Standards

Here are the project-specific topics that need to be addressed in your software planning:

- The SCM repository strategy—The simplest approach is a mono-repository model. However, if your project uses legacy code bases and third-party packages, you may be forced to implement a multi-repository model.
    - The management process for third-party source code and binaries.
- The integration testing and system testing roles and processes that will be performed by the software team—In my experience, this varies a lot depending on the project, team size, and schedules.
- The list of build tools—Typically, these decisions are constrained by your CI server, legacy code bases, and third-party packages.
- The list of release artifacts and the supporting documentation that will be required with each release

Additional Guidelines

You should also reference some additional guidelines that may not be under the control of your software team. For example, some important guidelines may be owned by the quality team, and you can simply include references to those guidelines. However, if those documents don't exist, you will need to create them. I recommended that you develop guidelines for the following items:

- Requirements traceability
- Regulatory concerns
- Development workflow, for example, Agile, Scrum, Waterfall, etc.
- Required architecture, design, and supporting documentation
- Other company or domain-specific quality processes

If these kinds of external guidelines don't exist, that is okay. But it does mean you will need to provide more definition in your SDP for these topics.

Care and Feeding of Your SDP

While fairly static, your SDP will need some care and feeding along the way. When you make changes, it is critical that all the relevant stakeholders—developers, program managers, managers, etc.—are made aware of the changes.

SDP for the GM6000

The following is the outline for the GM6000's SDP. If you use these as templates, feel free to change the organization and formatting. What's important is that you consider the content of each section, even for topics that ultimately do not apply to your project. The complete SDP is provided in Appendix K, "GM6000 Software Development Plan."

1. Document name and version number
2. Overview
3. Glossary
4. Document references
5. Roles and responsibilities
6. Software items
7. Documentation outputs
8. Requirements
9. Software development life cycle processes
10. Cybersecurity
11. Tools
12. SCM
13. Testing
14. Deliverables
15. Change log

Each of these sections should contain links and applicable references to your company's QMS documentation.

Housekeeping

Sections 1–4 and section 15, *Document name and version number, Overview*, *Glossary*, *Document references,* and *Change log*, are housekeeping sections for software documentation. They are self-explanatory, and it should be easy to fill in these sections. For example, the *Overview* section is just a sentence or two that states the scope of the document. Here is an example from Appendix K, "GM6000 Software Development Plan":

> *This document captures the software development decisions, activities, and logistics for developing all of the software that executes on the GM6000 Digital Heater Controller's Control Board that is needed to formally test, validate, manufacture, and release a GM6000.*

Roles and Responsibilities

This section identifies the various roles on the team. It describes the responsibility of each role in the software development process and the project as a whole, and it sets common expectations for the entire team. Also be aware that people often perform multiple roles.

Here is an example of a defined role on the team:

- ***Software lead****–Technical lead for all software contained within the GM6000 control board. Responsible for*
    - *Creating software architecture*
    - *Creating software detailed design*
    - *Defining SRS requirements*

- *Resolving software-specific technical issues and decisions*
- *Ensuring the software-specific processes (especially reviews) are followed*
- *Signing off on the final releases*

Also note that when a person is assigned a role, it does not mean that they are the sole author of the document or the deliverable. The responsibility of the role is to ensure the completion of the document and not necessarily to write it themselves.

Software Items

This section identifies what the top-level software deliverables are. For each software item, the following items are called out:

- The verification that is required
- The programming languages that will be used
- The coding standards that will be followed

Here is an example how software items can be identified:

1. *Software that executes on the GM6000 control board when it is shipped to a customer*
    a. *This software item requires formal testing and verification before being released.*
    b. *The software shall be programmed in C/C++ and conform to the SW-1002 Software C/C++ Embedded Coding Standard.*

2. *Manufacturing test software (which executes on the GM6000 control board) that will be used when manufacturing the GM6000*

    a. *This software item will be informally verified by engineering before being released to manufacturing.*

    b. *The software shall be programmed in C/C++ and conform to the SW-1002 Software C/C++ Embedded Coding Standard.*

Documentation Outputs

This section is used to call out non-source code artifacts that the software development team will deliver. It describes who has the responsibility to see that the artifact is completed and delivered and who, if it is a different person, the subject matter expert (SME) is.

Not all the artifacts or documents need to be Word-style documents. For example, Doxygen HTML pages, wiki pages, etc., are acceptable. Depending on your project, you may need to create all these documents, or possibly just a subset. And in some cases, you may need additional documents that aren't listed here. Possible artifacts are

- Software architecture
- Software detailed design
- Doxygen output
- Code review artifacts
- Design review artifacts
- Integration test plans, test procedures, test reports
- Developer environment setup

- Build server setup
- CI setup
- Release notes
- Software Bill of Materials (BOM)
- Release cover page for the Product Lifecycle Management (PLM) system.

Here are some examples from the SDP in Appendix K, "GM6000 Software Development Plan."

1. *The supporting documentation shall be created in accordance with the processes defined in the QMS-010 Software Development Life Cycle Process document.*
2. *A Software Architecture document shall be created and assigned a formal document number. The Software Lead is responsible for this document.*
3. *A Software Detailed Design document shall be created and assigned a formal document number. The Software Lead is responsible for this document.*
4. *The following documentation artifacts are captured in Confluence as wiki pages. The Software Lead is responsible for these items:*
    a. *Instructions on how to set up a developer's local build environment*
    b. *Instructions on how to manage the tools on the build servers*
    c. *Instructions on how to set up the CI platform*

Requirements

This section specifies what requirements documentation (with respect to software) is needed, who is responsible for the requirements, and what is the canonical source for the requirements. The section also includes what traceability processes need to be put in place. Basically, all of the processes discussed in Chapter 3, "Analysis," are captured in the SDP. Here is an example:

1. *The supporting documentation shall be created in accordance with the processes defined in the QMS-004 Requirements Management document.*
2. *The MRS is a formal document (with an assigned number) that captures all the top-level user and business needs. The Product Manager is responsible for this document.*
3. *The PRS is a formal document (with an assigned number) that captures the system-level requirements that are derived from the MRS. The System Engineer is responsible for this document.*
4. *The SRS is a formal document (with an assigned number) that captures the software-level requirements that are derived from the MRS and PRS. The Software Lead is responsible for this document.*

Software Development Life Cycle Processes

This section identifies the different software development phases and workflows for each phase. Ideally this section only consists of references to existing documents. Some of the topics that should be covered are as follows:

- The name and number of development phases
- The workflow for checking and reviewing code
- The workflow for reviewing design documentation
- The bug tracking processes
- The point in the process where formal testing can begin on the code that is being developed
- How the determination will be made that the software is "good enough" to release to manufacturing

Here are examples form the SDP in Appendix K, "GM6000 Software Development Plan."

1. *The software shall be developed in accordance with the processes defined in the QMS-010 Software Development Life Cycle Process document.*
2. *There are four phases: Planning, Construction, Verification, and Release.*
3. *The Planning phase shall consist of requirements gathering, software architecture, planning, and preparing the tools and infrastructure needed for the Construction phase.*
    a. *This process will generally follow an iterative, Agile Kanban process with tasks captured in JIRA.*

   b. *All code checked into GitHub during this phase requires a ticket. The ticket workflow shall be the same as the workflow described under the Construction phase.*

   c. *With respect to software development, the Planning phase is considered waterfall in that the Construction phase shall not begin until the Planning phase has completed.*

   d. *The Planning phase is exited after the following deliverables have been completed:*

      - *A reviewed first draft of the SWA-1327 GM6000 Software Architecture document*

      - *The foundational skeleton application can be successfully built by the CI server (including automated unit tests)*

4. *The Construction phase shall consist of detailed design, implementation, testing, and bug fixing.*

Cybersecurity

This section identifies the workflows and deliverables—if any—needed to address cybersecurity. Here is an example from the SDP.

1. *The cybersecurity needs of the project shall follow the processes defined in QMS-018 Cyber Security Work Instructions.*

2. *The cybersecurity analysis and control measures shall be documented in the software architecture document. The Software Lead is responsible for the cybersecurity content.*

Tools

This section identifies tools used to develop the software. Because it is sometimes a requirement that you can go back and re-create (i.e., recompile) any released version of the software, it is also important to describe how tools will be archived.

Here is an example from the SDP:

1. *The software that executes on the Control Board hardware shall be compiled with the GCC cross compiler for the specific microcontroller.*
    a. *The version of the compiler shall not be changed during the construction and release phases unless there is a documented compiler bug that impacts the software.*
    b. *The compiler toolchain shall be archived along with the source code in GitHub.*
    c. *The compiler toolchain shall be tagged and labelled when a formal build is performed.*

Software Configuration Management (SCM)

This section specifies the logistics of how you are going to version control your source code, including any third-party packages that will be used. Topics that should be addressed are as follows:

- SCM tools and repository server—For example, GIT, GitHub, etc.
- Repository strategy—For example, a single repository, multiple repositories, etc.
- Management of third-party packages—You might address issues like these:

- Will third-party packages be pulled from the Internet every time a build is done?
- Are there local copies of the packages?
- Are the packages stored in the company's GIT repository?
- Is the local copy forked from the original source?

- Branch strategy—For example, trunk, Git Flow, and merge rules (e.g., for pull requests)
- Versioning strategy—For example, a description of identifiers and SCM labeling

Here are examples from the SDP in Appendix K, "GM6000 Software Development Plan."

1. *GitHub private repositories shall be used to version control all the source code for the project.*
   a. *A single repository shall be used. The repository URL is `https://github.com/xxxxx/gm6000`.*
   b. *The repository will also contain all third-party packages and the cross-compiler toolchain used to build binaries for the target hardware.*
2. *The branching strategy shall be a modified trunk-based development model.*
   a. *The `main` branch shall be used for all candidate releases.*
   b. *A child branch (off `main`), called `develop`, shall be used as the stable branch for day-to-day development and pull requests.*
   c. *Each ticket will be used to create a short-lived branch off of the `develop` branch. The ticket number shall be part of the branch name.*

Testing

This section specifies details how the testing—which is the responsibility of the software team—will be done. Topics that should be covered are as follows:

- Unit test requirements and code coverage metrics as applicable
- Integration testing requirements
- How third-party packages will be tested
- Use of the non-target-based testing (e.g., using a functional simulator)

Here is an example from the SDP:

1. *The software team is responsible for unit testing and integration testing.*
2. *The source code is organized by namespaces (*per *SW-1002). Each namespace is required to have at least one unit test for the code it contains.*
    a. *If the namespace has a direct target platform dependency, the unit test shall be a manual test that executes on the target platform.*
    b. *If the namespace contains implementation for the UI, the unit test shall be a manual test that executes either on the target platform or the simulator.*

    Note *The automated unit test requirement is relaxed with respect to the UI because the test infrastructure does not include tools for automated verification of the UI's visual presentation.*

c. *All other namespaces shall have an automated unit that is a stand-alone application that returns pass/fail. Automated unit tests are executed as part of the CI process for all builds. All automated unit tests are required to meet the following code coverage metrics. Note that because of the tools being used (`gcc`, `gcovr`), the branch coverage metrics are not always correct–the branch coverage threshold is intentionally lower to compensate.*

- *Line coverage >= 80%*
- *Branch coverage >= 60%*

Deliverables

This section summarizes the various deliverables called out in the SDP document. This is essentially the software deliverables checklist for the entire project. Here is an example from the SDP in Appendix K, "GM6000 Software Development Plan":

Deliverable	***Phase(s)***	***Notes***
SDP-1328 GM6000 Software Development Plan	*Planning Construction*	*A reviewed first draft is required to exit the planning phase.*
SRS-1324 GM6000 Software Requirement Specification	*Planning Construction*	*A reviewed first draft is required to exit the planning phase.*
SWA-1327 GM6000 Software Architecture	*Planning Construction*	*A reviewed first draft is required to exit the planning phase.*
SWA-1329 GM6000 Software Detailed Design	*Construction*	
…	…	…

Summary

Don't skip creating an SDP. Lacking a formal SDP simply means that a de facto SDP will organically evolve and be inconsistently applied over the course of the project where it will be a constant source of drama.

INPUTS

- The project's user and business needs
- The company's quality processes
- The company's software development methodology, for example, Agile, waterfall, TDD, etc.
- The software development team's best practices and processes

OUTPUTS

- The first draft of the software development plan document

CHAPTER 5

Preparation

The preparation step is not about defining and developing your product as much as it is about putting together the infrastructure that supports the day-to-day work. The tools that you will be using should have been called out in the Software Development Plan (SDP), but if you have tools that aren't referenced there, this is the time to add them to the SDP along with the rationale for prescribing their use.

The SDP for the GM6000 calls for the following tools:

- Git server—For version control and file management
- JIRA—For ticket tracking and bug tracking
- Confluence—For the wiki server that is used to host some of the supporting documentation
- Jenkins—For continuous integration (CI)

In nearly all companies that I've worked for, tools such as JIRA and Confluence were already in place and were directly supported by the IT team, so it was not something the software team had to worry about. Nevertheless, I still needed to make sure that the team members had accounts and permissions to access the tools and that at least two members of the cross-functional core team had "full access" rights and permissions to the tools and systems, including the ability to create and manage users.

For actual code development, the companies used the GitHub tools unless they had commercial tools like Perforce or ClearCase. As the GitHub tools are largely free, this chapter prescribes the use of the following tools:

- GitHub public repository—For version control and file management
- GitHub's Projects—For JIRA-like task management and feature tracking
- GitHub Wiki—For Confluence-like documentation and notes

As for continuous integration (CI), many companies do not have CI tools in place, so you and your team may find yourselves needing to implement them with minimal support from IT. For that reason, the book's GitHub wiki pages[1] provide step-by-step instructions for setting up Jenkins to build GM6000 example code. However, the GM6000 only uses a single repository, and it has simple CI requirements. Consequently, the Jenkins setup is basic and does not take advantage of features like pipeline builds and automated deployments.

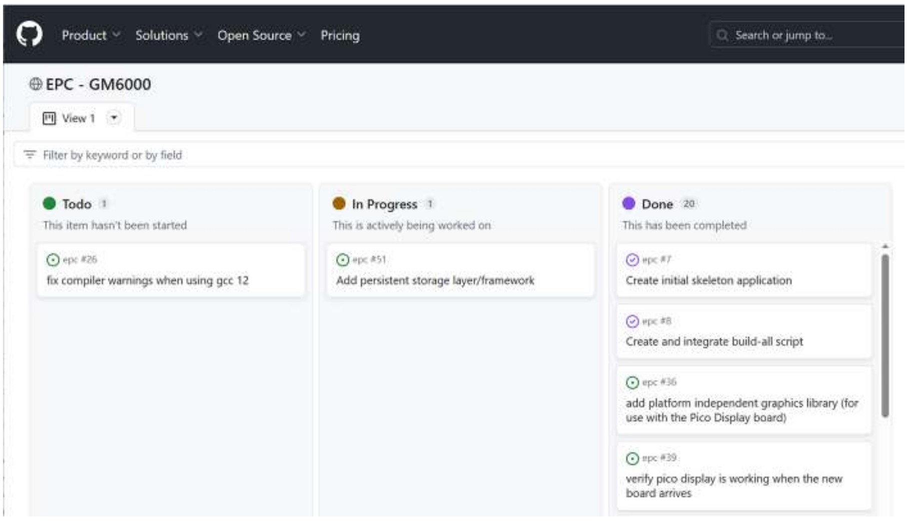
GitHub Projects

GitHub Projects is an adaptable, flexible tool for planning and tracking your work.[2] It's a free tool that allows you to track issues such as bug reports or tasks or feature requests, and it provides various views to facilitate prioritization and tracking. For example, Figure 5-1 is an example of the Kanban-style view of the GM6000 project. For a given project, the

[1] https://github.com/johnttaylor/epc/wiki

[2] https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects

issue cards can span multiple repositories, and branches can be created directly from the issue cards themselves. For the details of setting up GitHub Projects, I recommend the “Creating a Project” documentation provided by GitHub.[3]



Todo 1
This item hasn't been started
epc #26
fix compiler warnings when using gcc 12

In Progress 1
This is actively being worked on
epc #51
Add persistent storage layer/framework

Done 20
This has been completed
epc #7
Create initial skeleton application
epc #8
Create and integrate build-all script
epc #36
add platform independent graphics library (for use with the Pico Display board)
epc #39
verify pico display is working when the new board arrives

Figure 5-1. *Kanban-style view of the GM6000 project*

GitHub Wiki

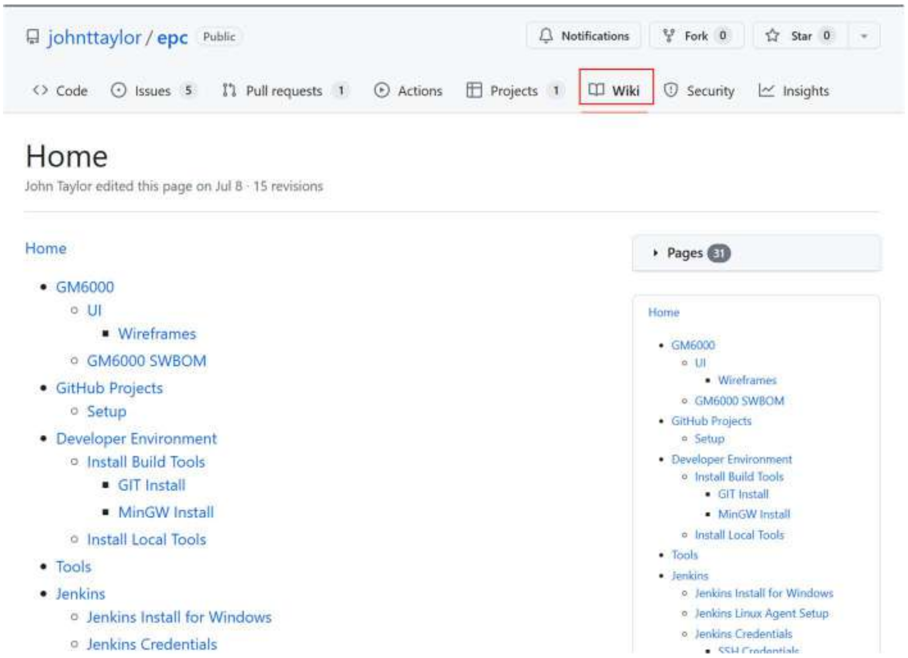
GitHub provides one free wiki per public repository. The following list provides examples of documents you should consider capturing on a Wiki:

- Developer and Build Server setup instructions
- CI Platform setup and maintenance instructions
- Software development tools list

[3] https://docs.github.com/en/issues/planning-and-tracking-with-projects/creating-projects/creating-a-project

- Software "Bill of Materials" (or some other list of all third-party packages)
- Coding standards
- Work instructions (e.g., instructions for creating releases, shipping hardware to an offshore vendor, etc.)

Figure 5-2 shows an example of a GitHub wiki page for the GM6000 project.



Home

John Taylor edited this page on Jul 8 · 15 revisions

Home

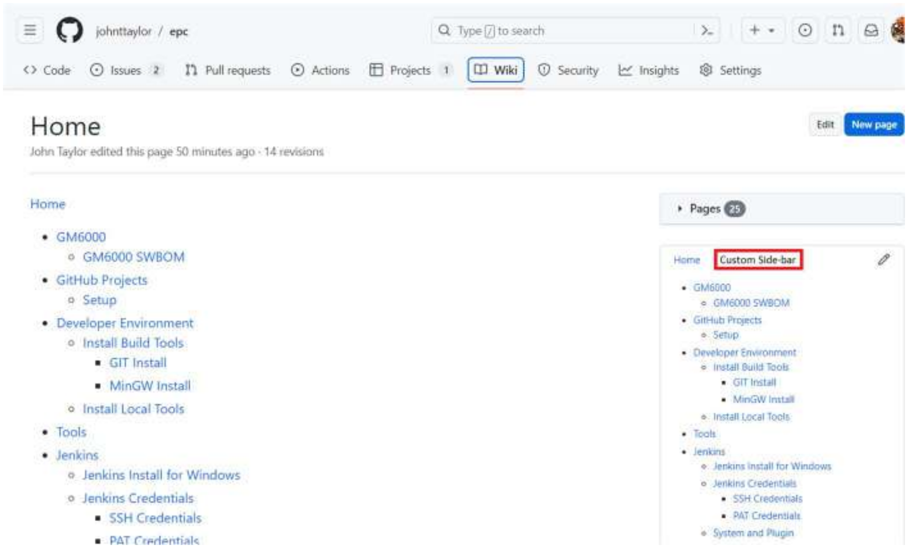
- GM6000
  - UI
    - Wireframes
  - GM6000 SWBOM
- GitHub Projects
  - Setup
- Developer Environment
  - Install Build Tools
    - GIT Install
    - MinGW Install
  - Install Local Tools
- Tools
- Jenkins
  - Jenkins Install for Windows
  - Jenkins Credentials

Home

- GM6000
  - UI
    - Wireframes
  - GM6000 SWBOM
- GitHub Projects
  - Setup
- Developer Environment
  - Install Build Tools
    - GIT Install
    - MinGW Install
  - Install Local Tools
- Tools
- Jenkins
  - Jenkins Install for Windows
  - Jenkins Linux Agent Setup
  - Jenkins Credentials
    - SSH Credentials

Figure 5-2. *GitHub wiki page for the GM6000 project*

The wiki uses GitHub's markdown language for formatting page content, and it supports storing images as part of a wiki page. The organization of the wiki is flat—there is no direct support for hierarchy of pages—but you can create your own "Custom Side-bar" where you can apply a hierarchy, or table of contents, as shown in Figure 5-3.



Home

John Taylor edited this page 50 minutes ago · 14 revisions

Home

- GM6000
  - GM6000 SWBOM
- GitHub Projects
  - Setup
- Developer Environment
  - Install Build Tools
    - GIT Install
    - MinGW Install
  - Install Local Tools
- Tools
- Jenkins
  - Jenkins Install for Windows
  - Jenkins Credentials
    - SSH Credentials
    - PAT Credentials

Home

- GM6000
  - GM6000 SWBOM
- GitHub Projects
  - Setup
- Developer Environment
  - Install Build Tools
    - GIT Install
    - MinGW Install
  - Install Local Tools
- Tools
- Jenkins
  - Jenkins Install for Windows
  - Jenkins Credentials
    - SSH Credentials
    - PAT Credentials
  - System and Plugin

Figure 5-3. *GitHub wiki Custom Side-bar*

The Custom Side-bar is essentially just another wiki page of markdown links that you can indent to indicate hierarchy. The following markdown snippet shows an example of how the side-bar for the GM6000 wiki was defined. Additional help for constructing wiki pages can be found on GitHub's website.[4]

```
[Home](https://github.com/johnttaylor/epc/wiki/Home )
  * [GM6000](https://github.com/johnttaylor/epc/wiki/GM6000)
    * [GM6000 SWBOM](https://github.com/johnttaylor/epc/wiki/
      GM6000---Software-Bill-of-Materials-(SWBOM))
  * [GitHub Projects](https://github.com/johnttaylor/epc/wiki/
    GitHub---Projects)
```

[4] https://docs.github.com/en/communities/documenting-your-project-with-wikis/creating-a-footer-or-sidebar-for-your-wiki#creating-a-sidebar

```
    * [Setup](https://github.com/johnttaylor/epc/wiki/GitHub-
      Projects---Setup)
  * [Developer Environment](https://github.com/johnttaylor/epc/
    wiki/Development-Environment)
    * [Install Build Tools](https://github.com/johnttaylor/epc/
      wiki/Developer---Install-Build-Tools)
      * [GIT Install](https://github.com/johnttaylor/epc/wiki/
        Developer---GIT-Install)
    * [Install Local Tools](https://github.com/johnttaylor/epc/
      wiki/Developer---Install-Local-Tools)
  * [Tools](https://github.com/johnttaylor/epc/wiki/Tools)
```

Continuous Integration Requirements

In a perfect world, there would be a person or team in charge of designing, creating, running, and maintaining the continuous integration hardware and software. If you happen to be in this situation, consult with the CI lead about incorporating some of the requirements listed here into your CI. They are probably doing many of these things already. But if not, collaborate with them on incorporating some of these features:

- Ensure that all individual developer code that gets committed to the Software Configuration Management (SCM) system compiles and passes all automated unit tests. This is considered a basic CI build. In practice, this means CI builds are performed on all pull requests.
- Ensure that a basic CI build completes successfully before the source code for that build is merged into mainline or stable branches.

- Ensure that every CI build builds all unit tests—both manual and automated—along with all the final application executables.
- Ensure that the simulator or simulators—if you have them—build every time.
- Ensure that the build server that is used for the CI builds is the same build server used for creating formal builds from stable branches in the SCM repository.
- Ensure that formal builds, or releases, are done from mainline or some other stable branch. Formal builds are required to pass all automated unit tests just like CI builds.
- Ensure that the formal builds are tagged or labeled in the SCM repository.
- With both CI and formal builds, ensure that all permutations of the final application and simulators and release variants and engineering-only variants get built every time.
- Ensure that build artifacts (e.g., .hex files, .exe files, Doxygen output, etc.) for formal releases are archived so they are retrievable by nondevelopers.
- Incorporate, at your discretion, any automated "smoke tests" or "sanity tests" into the build process. See Chapter 13, "Integration Testing," for additional discussion about automated smoke tests.

Soapbox: CI is not a simple or free addition to a project. So why do it? The reason to do it is to detect integration errors as quickly as possible. On the surface, this may not sound like a huge win, but CI is a significant net positive when it comes to maintaining stable branches in your SCM. Avoiding the pain of broken builds, and the misery of being bogged down in "merge hell," is more than enough compensation for the effort you put into setting up and maintaining a CI server.

Jenkins

Installing and configuring Jenkins for your specific CI needs is nontrivial. But here is the good news:

- The Jenkins website provides detailed instructions for installing Jenkins on multiple host platforms.[5]
- Jenkins is widely used. This means that there are plenty of plug-ins available, and information about how to do things with Jenkins can often be found with simple Internet searches.
- The book's GitHub Wiki page, `https://github.com/johnttaylor/epc/wiki/All-Things-Jenkins`, provides step-by-step instructions on how to stand up Jenkins.

[5] `www.jenkins.io/doc/book/installing/`

Here are the high-level steps for installing and configuring CI on Jenkins:

1. Install Jenkins—This involves downloading and getting the Jenkins services running. For the GM6000 project, the primary Jenkins controller runs on a Windows PC.

2. Set up Credentials for accessing GitHub—Jenkins accesses GitHub repositories using GitHub APIs.

3. Configure the system, the tools, and the plug-ins.

4. Set up a Linux build agent—A Linux build agent, running on a physical or virtual machine, is required to compile the platform-independent code for a Linux host.

5. Create automation projects—This involves creating jobs that are automatically triggered when pull requests are created and when code is merged to stable branches such as develop and main. These jobs are responsible for building the code, executing the unit tests, generating reports, and archiving build artifacts.

Having two build platforms (Windows and Linux) adds to the complexity of the Jenkins configuration, but in order to keep the GM6000 example code honest, I needed my CI build server to perform native Linux builds.

Summary

The preparation step in the planning stage is all about enabling software development. Many of the activities are either mostly completed, because the tools are already in place from a previous project, or can be started in parallel with the creation of the Software Development Plan (SDP). So there should be ample time to complete this step before the construction stage begins. But I can tell you from experience, if you don't complete this step before construction begins, you can expect to experience major headaches and rework. Without a working CI server, you may have broken builds or failing unit tests without even knowing it. And the more code that is written before these issues are discovered, the more the work (and pain) needed to correct them increases.

INPUTS

- The initial draft of the Software Development Plan

OUTPUTS

- A Software Configuration Management (SCM) server (e.g., GitHub) for managing the project's source code is up and running.
- User accounts with appropriate permissions have been created for all the software developers on the SCM server. Also, testers should have been given access to the repositories.
- A planning and tracking tool (e.g., JIRA or GitHub Projects) for tracking is up and running, and a project has been created.

- User accounts have been created for all of the team members (including nonsoftware developers) in the tracking tool.
- A wiki server (e.g., Confluence, or a GitHub wiki) to host lightweight documentation is up and running, and a home page has been created.
- User accounts have been created for all team members (including nonsoftware developers) on the wiki server.
- A continuous integration (CI) server is up and running. This includes creating minimal automation jobs that can check out repository code and execute a script from the repository that returns pass/fail.

CHAPTER 6

Foundation

The foundation step in the planning phase is about getting the day-to-day environment set up for developers so they can start the construction phase with a production-ready workflow. This includes performing tasks, for example:

- Setting up the SCM repositories (e.g., setting up the repository in GitHub)
- Defining the top-level source code organization for the project
- Creating build scripts that can build unit tests and application images
- Creating skeleton applications for the project, including one for the functional simulator
- Creating CI build scripts that are fully integrated and that build all the unit tests and application projects using the CI build server
- Creating the outline for the Software Detailed Design (SDD) document. Because the SDD is very much a living document, as detailed design is done on a just-in-time basis, nothing is set in stone at this point, and you can expect things to change throughout the course of the project. (See Chapter 11, "Just-in-Time Detailed Design," for more details.)

J. T. Taylor and W. T. Taylor, *The Embedded Project Cookbook*,
https://doi.org/10.1007/979-8-8688-0327-7_6

SCM Repositories

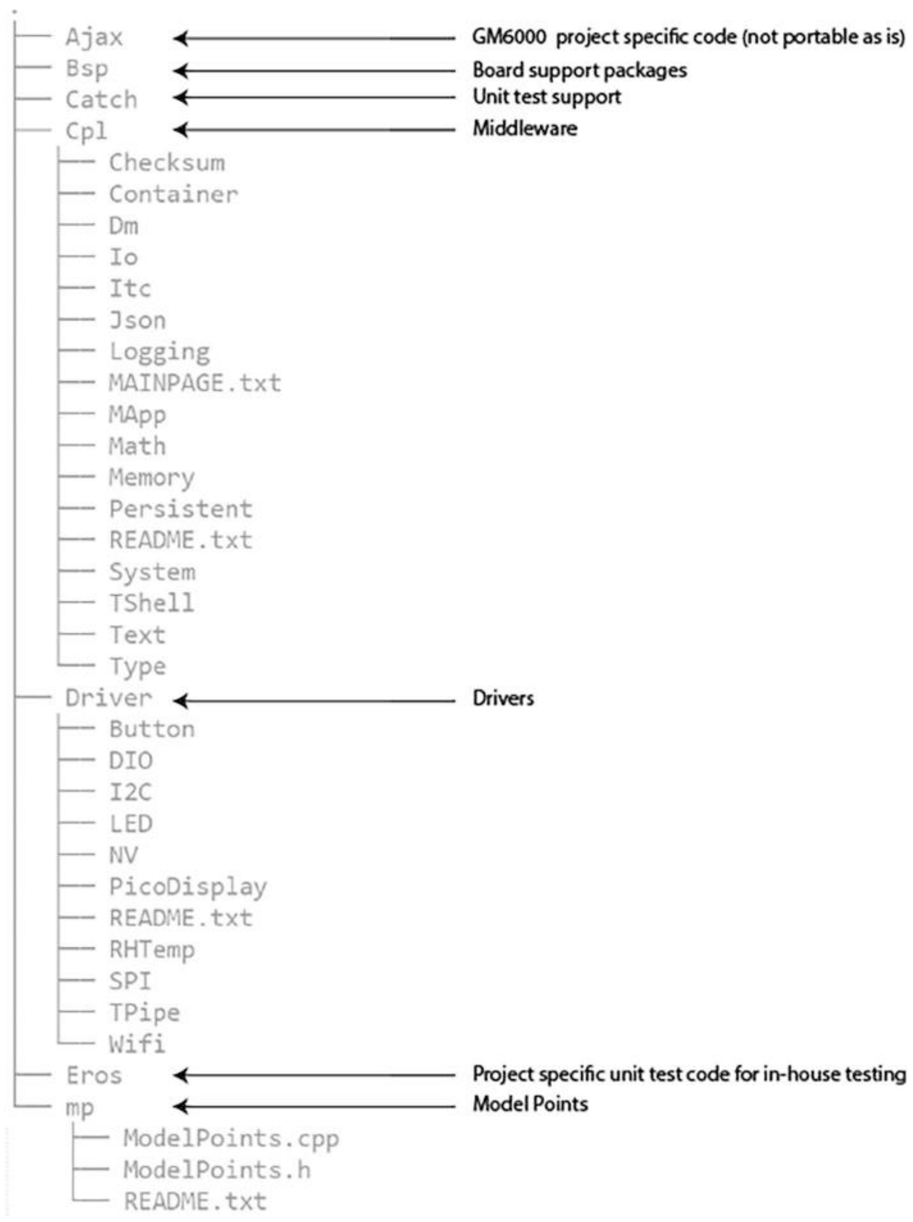
The SCM tool should have been "stood up" as part of the preparation step. So for the foundation step, it is simply a matter of creating the repository and setting up the branch structure according to the strategy you defined in the SDP.

Source Code Organization

You will need to make several decisions about the layout of your source code tree, and these decisions can have far-ranging consequences. A well-thought-out, consistent file organization facilitates code reuse as well as the creation and maintenance of unit tests. It is also critical to your ability to build multiple images and executables from your code base. These decisions should be captured—actually written down—in the SDD.

Unfortunately, you never get the source code organization quite right on the first pass. However, I have learned from experience that it is best to organize your files by component dependencies—not by project. That is, do not create your directory structure to reflect a top-down decomposition of the project. Rather, organize your code so that the directories reflect the namespaces of your code. By doing this, you can simply inspect the directory structure as a quick visual check that there are no undesired cyclical dependencies. As an example, Figure 6-1 shows an annotated view of the selectively expanded, top-level directory structure for the GM6000 project.

The GM6000 directory structure is more fully documented in Appendix L, "GM6000 Software Detailed Design (Initial Draft)." Additionally, Appendix O, "Software C/C++ Embedded Coding Standard," contains specifics about naming conventions in the source code organization.



```
.
├── Ajax
├── Bsp
├── Catch
├── Cpl
│   ├── Checksum
│   ├── Container
│   ├── Dm
│   ├── Io
│   ├── Itc
│   ├── Json
│   ├── Logging
│   ├── MAINPAGE.txt
│   ├── MApp
│   ├── Math
│   ├── Memory
│   ├── Persistent
│   ├── README.txt
│   ├── System
│   ├── TShell
│   ├── Text
│   └── Type
├── Driver
│   ├── Button
│   ├── DIO
│   ├── I2C
│   ├── LED
│   ├── NV
│   ├── PicoDisplay
│   ├── README.txt
│   ├── RHTemp
│   ├── SPI
│   ├── TPipe
│   └── Wifi
├── Eros
└── mp
    ├── ModelPoints.cpp
    ├── ModelPoints.h
    └── README.txt
```

GM6000 project specific code (not portable as is)
Board support packages
Unit test support
Middleware
Drivers
Project specific unit test code for in-house testing
Model Points

Figure 6-1. *Annotated top-level directory structure for GM6000*

Build System and Scripts

Selecting build tools is a decision that impacts the entire life of the project. That is, it is extremely painful to change build systems after you have applications in place and have created a bunch of unit tests. And sometimes, in real life, you just get stuck with something. For example, there may be a component of your project that requires you to use `cmake`.[1] However, if you do get to choose your own tools, here are some best practices to consider:

- Builds need to be command line based to support automation. This requirement is not absolute, but it is a rare use case where building within a developer's IDE and building from a fully integrated CI tool set will yield the same results. Depending on the IDE, you may be able to launch it from the command line with enough parameters to have it build and exit unassisted. This is okay as long as you always use it in this way.
- The continuous integration build machine and the developers both use the same build scripts. This is an absolute requirement.
- The build tools should create the application (including all variants) and support an unlimited number of unit tests. The build scripts must scale because the number of unit tests or application variants increases over time.
- Adding a new file to the build scripts should require low-to-no effort for the developer. Adding a new directory should also require low effort.

[1] https://cmake.org/

- For any build script file that is located in the same directory as the source code, the script should not contain any "unique to the final output" details like compiler flags, linker directives, -D symbols, etc. The reason is that any given directory may be built for different output images. For example, it could be built as part of the application and additionally it could be built as part of a unit test image.
- The build scripts should have the ability to selectively build directories. That is, the build scripts should never assume that it is okay to recursively build everything in a directory including all of its subdirectories. Once again, the reason is that, depending on whether the build is creating the application, the functional simulator, or a unit test, the subdirectories that need to be used can differ.
- Build scripts must be constructed such that path information about where a source code file is located is not lost when generating and referencing derived objects. In other words, the build scripts cannot assume or require globally unique file names.
- Build scripts must be constructed such that the generated object files are not placed in the same directory as the source code. This is especially important when building the source code for multiple platforms with different compiler options.
- The build engine should be host, compiler, and target independent. The unit test, functional simulator, and application builds will be built across multiple host environments for multiple target platforms.

After you have selected your build tools, you need to create scripts that will build the skeleton applications and unit tests (both manual and automated).

The art of constructing build scripts and using makefiles is out of the scope of this book, simply because of the number of build tools available and the nuances and variations that come with them. Suffice it to say, though, that over the years I have been variously frustrated by many of the build tools that I was either required to use or tried to use. Consequently, I built my own tool—the NQBP2 build engine—which eschews makefiles in favor of a list of directories to build. This tool is freely available and documented on GitHub.

The GM6000 example uses the NQBP2 build engine, and more details about NQBP2 are provided in Appendix F, "NQBP2 Build System."

Skeleton Applications

The skeleton projects provide the structure for all the code developed in the construction phase. There should be a skeleton application for all released images, including target hardware builds and functional simulator builds for the applications. Chapter 7, "Building Applications with the Main Pattern," provides details about creating the skeleton application for the target hardware and the functional simulator.

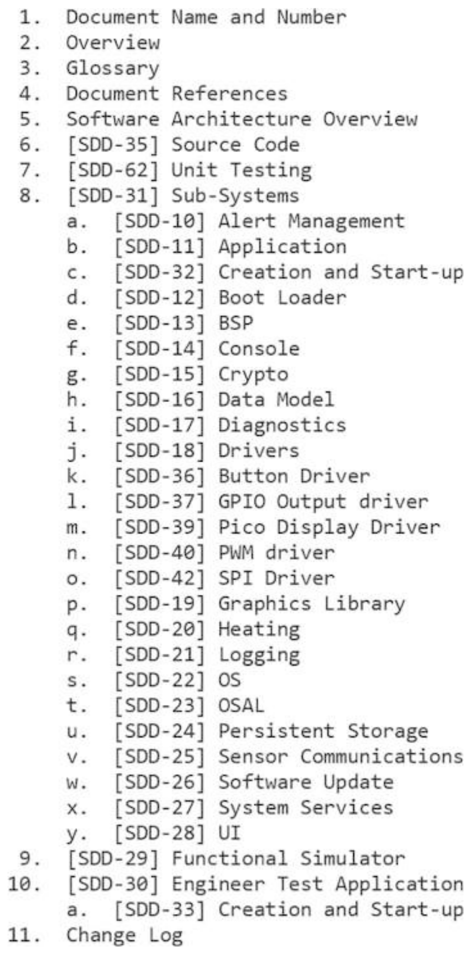
CI "Build-All" Script

After you have selected your build engine and have created build scripts that can build skeleton applications and unit tests, you will want to create a "build-all" script that builds all the applications and unit tests in the project. Chapter 8, "Continuous Integration Builds," goes into detail about how to create a build-all script. After the build-all script is written, the final step is to integrate it with the CI build server. After this work is done, your CI workflow should be complete (except for maintenance).

The build-all script should be architected such that it does not have to be updated when new applications and unit tests are added to the project.

Software Detailed Design

Creating the outline for the Software Detailed Design (SDD) document is pretty simple after all of the top-level subsystems in the software architecture have been identified. The bulk of the outline for the SDD consists of some boilerplate sections and the subsystems section of the software architecture. Figure 6-2 shows the outline for the GM6000's SDD.



```
 1.  Document Name and Number
 2.  Overview
 3.  Glossary
 4.  Document References
 5.  Software Architecture Overview
 6.  [SDD-35] Source Code
 7.  [SDD-62] Unit Testing
 8.  [SDD-31] Sub-Systems
     a.  [SDD-10] Alert Management
     b.  [SDD-11] Application
     c.  [SDD-32] Creation and Start-up
     d.  [SDD-12] Boot Loader
     e.  [SDD-13] BSP
     f.  [SDD-14] Console
     g.  [SDD-15] Crypto
     h.  [SDD-16] Data Model
     i.  [SDD-17] Diagnostics
     j.  [SDD-18] Drivers
     k.  [SDD-36] Button Driver
     l.  [SDD-37] GPIO Output driver
     m.  [SDD-39] Pico Display Driver
     n.  [SDD-40] PWM driver
     o.  [SDD-42] SPI Driver
     p.  [SDD-19] Graphics Library
     q.  [SDD-20] Heating
     r.  [SDD-21] Logging
     s.  [SDD-22] OS
     t.  [SDD-23] OSAL
     u.  [SDD-24] Persistent Storage
     v.  [SDD-25] Sensor Communications
     w.  [SDD-26] Software Update
     x.  [SDD-27] System Services
     y.  [SDD-28] UI
 9.  [SDD-29] Functional Simulator
10.  [SDD-30] Engineer Test Application
     a.  [SDD-33] Creation and Start-up
11.  Change Log
```

Figure 6-2. *SDD outline for the GM6000*

The *Overview* and *Software Architecture Overview* sections are included to provide some basic context for the design details. Do not make these sections too verbose because the SDD is not the canonical source for the material presented in these sections.

You will also note that there are additional sections that are neither boilerplate nor subsystems. This is okay because the SDD is not restricted to just the top-level subsystems. It contains all of the Software Detailed

Design that the project needs. However, all sections (except housekeeping sections) must be traceable back to a section in the software architecture document. If a detailed design section can't be traced back to the software architecture document, you should stop immediately and resolve the disconnect.

The primary advantage of organizing the SDD according to the subsystems identified in the software architecture document is that you get requirements traceability for free. This is because you have created a one-to-one mapping between the software architecture subsystems and the SDD sections.

The other sections for the SDD outline for the GM6000 are as follows:

- Source code—This section finalizes and documents the top-level source code organization for the entire project. It traces back to *SWA-38 File and Directory Organization*.
- Unit testing—This section defines the naming conventions for unit tests. It traces back to *SWA-42 Unit Testing*.
- Functional simulator—This section captures design details specific to the simulator. It traces back to *SWA-34 Simulator*.
- Engineering test application—This section covers the design details specific to the special test software used for engineering validation and manufacturing testing. It traces back to *SWA-41 Engineering and Manufacturing Testing*.

Summary

The foundation step is the final, gating work that needs to be completed before moving on to the construction phase. The foundation step includes a small amount of design work and some implementation work to get the build scripts to successfully build the skeleton applications and to integrate these scripts with the CI server.

You may be tempted to skip the foundation steps, or to do them later after you started coding the interesting stuff. Don't! The value of the CI server is to detect broken builds immediately. Until your CI server is fully stood up, you won't know if your `develop` or `main` branches are broken. Additionally, skipping building the skeleton applications is simply creating technical debt. The creation and startup code will exist at some point, so doing it first, while planning for known permutations, is a much more efficient and cost-effective solution than organically evolving it over time.

INPUTS

- The initial draft of the Software Development Plan (SDP)
- The initial draft of the software architecture (SWA) document
- The continuous integration (CI) server is running.
- The Software Configuration Management (SCM) server is running.

OUTPUTS

- Skeleton applications for all planned released images. This includes both target hardware and simulator builds.
- Source code repository created and populated with the skeleton applications
- The CI server is able to build all applications and unit tests, tag and label formal builds, and archive build artifacts.
- The first draft of the outline for the Software Detailed Design document. In addition to the outline, the SDD should also include the design for
    - The organization of the source code tree
    - Application creation and startup (i.e., the `main` pattern)
- An updated requirements trace matrix that shows the back trace from SDD sections to the software architecture sections

CHAPTER 7

Building Applications with the Main Pattern

At this point in the process, your principal objective is to set up a process to build your application to run on your hardware. However, you may also want to build variations of your application to provide different features and functionality. Additionally, you may want to create builds that will allow your application to run on different hardware. Consequently, the goal of this chapter is to put a structure in place that lets you build these different versions in an automated and scalable way.

Even if you have only a single primary application running on a single hardware platform, I recommend that you still build a functional simulator for your application. A simulator provides the benefits of being able to run and test large sections of your application code even before the hardware is available. It also provides a much richer debug environment for members of the development team.[1]

The additional effort to create a functional simulator is mostly planning. The key piece of that planning is starting with a decoupled design that can effectively isolate compiler-specific or platform-specific code. The trick, here, is not to do this with IF-DEFs in your function calls.

[1] For an in-depth discussion of the value proposition for a functional simulator, I recommend reading *Patterns in the Machine: A Software Engineering Guide to Embedded Development*.

J. T. Taylor and W. T. Taylor, *The Embedded Project Cookbook*,
https://doi.org/10.1007/979-8-8688-0327-7_7

The IF-DEF approach can be used to manage the platform-specific details at compile time, but the better way to accomplish this is to build your application using the Main pattern, which manages the platform-specific and application-variant-specific details at link time.

About the Main Pattern

By using the Main pattern, the difference between constructing the target application and constructing the function simulator is isolated to the top-level creator. This means that a functional simulator uses over 80% of the same modules as the actual application. But, again, for this wiring magic to work, there must be explicit interface boundaries for all platform-specific functionality. In other words, you've designed your application and drivers to be decoupled from the underlying hardware platform (see Chapter 15, "Drivers," for details). At an implementation level, this generally means that you've separated your hardware-specific, compiler-specific, and application-variant-specific code into different files.

Figure 7-1 illustrates the dependencies of a hypothetical embedded application on its target platform.

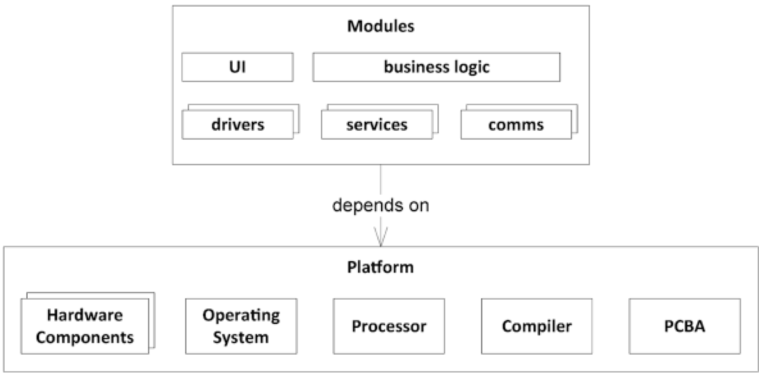


Figure 7-1. *Application platform dependencies*

By using Hardware Abstraction Layer (HAL) interfaces and Operating System Abstraction Layer (OSAL) interfaces, you can decouple the core application from the target platform. Note, however, that the separation of the platform-specific entities can be done with any abstract interface—not just at an HAL or OSAL interface. The concepts of HAL and OSAL interfaces are used here because they are more familiar and conceptually easier to understand. Figure 7-2 shows the same hypothetical embedded application with its hardware dependencies decoupled from the application.

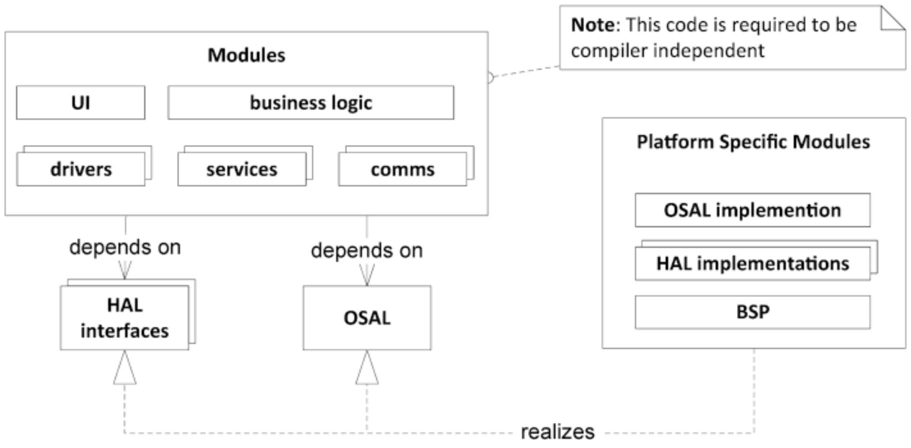


Figure 7-2. *Decoupled application platform dependencies*

Operating System Abstraction Layer

An OSAL provides interfaces for all functionality provided by the underlying operating system. For an embedded system that is using a basic thread scheduler, it would typically include interfaces such as

- Elapsed time (a replacement for the standard C library `clock()` function)
- Non-busy-wait delay (a replacement for the standard C library `sleep()` function)
- Thread management (create, delete, suspend, etc.)
- Mutexes
- Semaphores

You can create your own OSAL or use an open source OSAL such as CMSIS-RTOS2 for ARM Cortex processors. In addition, many microcontroller vendors (e.g. Microchip, TI, etc.) provide an OSAL with their SDKs. The example code for the GM6000 uses the OSAL provided by the CPL C++ class library.

Hardware Abstraction Layer

A HAL interface is an interface with no implementation. That is, the interface is abstract; it defines behavior for a hardware-specific or platform-specific device. At some point in the build process, or at runtime, there will be a binding of a device-specific implementation. A HAL interface is no different than any other abstract interface; prefacing it with HAL is just an attempt to categorize what the interface is abstracting. For example, one could view an OSAL as just a collection of abstract interfaces that define operating system behavior. (See Chapter 15, "Drivers," for more details of HAL interfaces.)

In general, HAL interfaces should be defined or created on an as-needed basis, and they should be created at the level that will provide the most return for the effort. This means that you should not try to create an all-encompassing HAL interface.

More About Main

After defining your OSAL and HAL interfaces, the next step is to add in the Main pattern. The Main pattern encapsulates all of the platform-specific knowledge and dependencies about how to interconnect the platform-independent modules with the platform-specific modules. The Main pattern consists of

- The resolution of interface references with concrete implementations
- The initialization and shutdown sequencing

In Figure 7-3, the differences between the target application and the functional simulator are contained in the blocks labeled.

- Main
- Platform
- Platform Specific Modules

The core of the application is everything contained in the block labeled Modules, and those modules can stay the same regardless of which Main instance uses it. While the platform and platform-specific blocks, or entities, seem to take up half of the diagram shown in Figure 7-3, in practice, the common entities in Modules constitute over 80% of the code base.

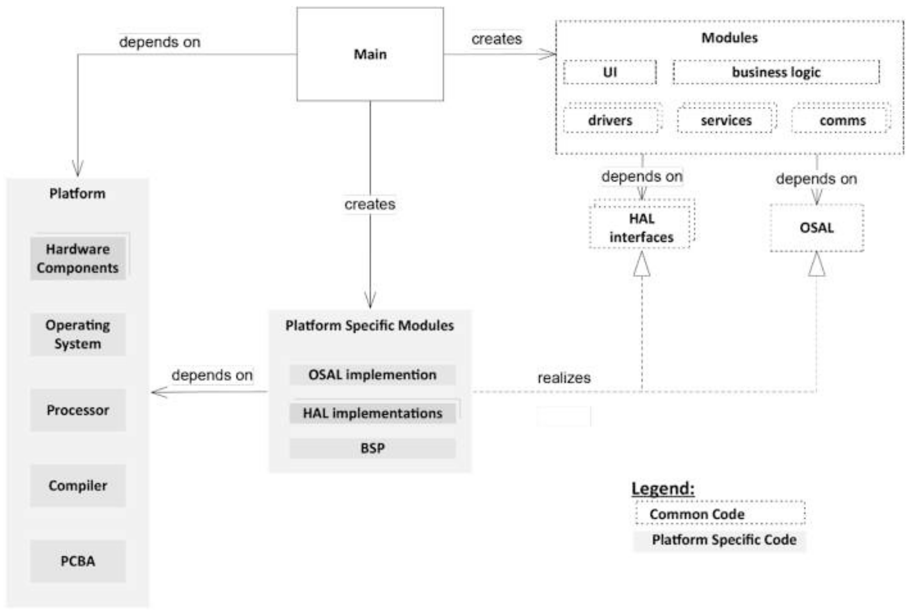


Figure 7-3. *Application with Main pattern*

Implementing Main

The Main pattern only addresses runtime creation and initialization. Of course, in practice, some of the platform-specific bindings will be done at compile or link time. These bindings are not explicitly part of the Main pattern.

It might be helpful at this point to briefly review how an embedded application gets executed. The following sequence assumes the application is running on a microcontroller:

1. The interrupt service routine for the Reset Vector is executed.

2. Then the compiler included C/C++ runtime code executes. This includes items such as initialization of static variables, execution of constructors for statically allocated C++ objects, and potentially some minimal board initialization.

3. The `main()` function is called. When and what happens in the `main()` function is obviously application specific. However, in general terms, the following actions occur:

    a. All remaining board, or BSP, initialization is completed.

    b. The OS/RTOS is initialized. The thread scheduler may or may not be started at this time.

    c. Control is turned over to the application.

    d. The various drivers, subsystems, components, and modules are initialized and started.

    e. The application runs. At this point, the application is fully constructed and initialized, interrupts are enabled, and the thread scheduler is running (if there is OS/RTOS).

The Main pattern comes into play in step 3. Here's the annotated version of that step.

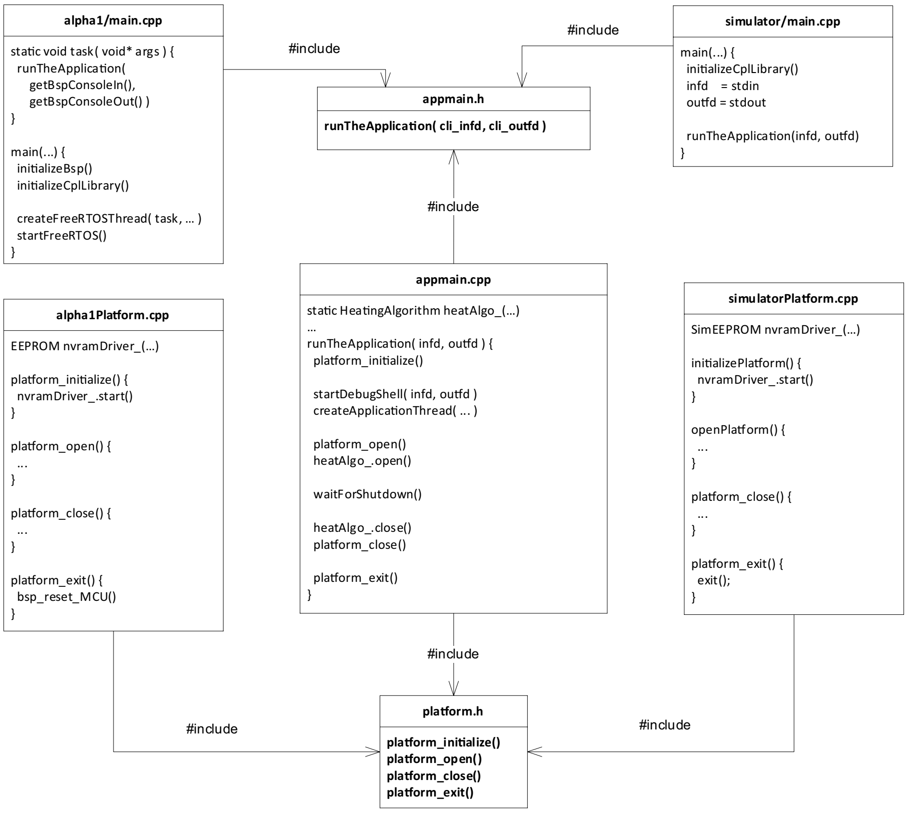
3a. All remaining board, or BSP, initialization is completed.	Highly platform-dependent code and should be physically segregated into separate files.
3b. The OS/RTOS is initialized. The thread scheduler may or may not be started at this time.	
3c. Control is turned over to the application.	Common startup source code across different platforms.
3d. The various drivers, subsystems, components, and modules are initialized and started.	
3e. The application runs. At this point, the application is fully constructed and initialized, interrupts are enabled, and the thread scheduler is running (if there is OS/RTOS)	

For steps 3a and 3b, the source code is platform dependent. One approach is to just have a single chunk of code and use an `#ifdef` whenever there are platform deltas. The problem is that `#ifdefs` within function calls do not scale, and your code will quickly become unreadable and difficult to maintain. The alternative is to separate—by platform—the source code into individual files and use link time bindings to select the platform-specific files. This means the logic of determining what hardware-specific and application-specific pieces make up a stand-alone module is moved out of your source code and into your build scripts. This approach declutters the individual source code files and is scalable to many platform variants.

For the remaining steps, 3c, 3d, and 3e, there is value in having a common application startup across different platforms, especially as the amount and complexity of the application startup code increase.

My claim is that the Main pattern is a scalable way to manage the different combinations without resorting to `#ifdefs`. And the key decision that you need to make is whether the benefits of having common code across the different common code combinations outweigh the overhead of managing independent applications. In my experience, the answer is usually yes. You want to have as much common code as possible and explicitly (or, manually) manage the combinations.

Figure 7-4 shows high-level pseudocode for how the Main pattern is implemented for the GM6000, the hypothetical heating application that runs both on the target hardware running FreeRTOS and as a functional simulator on a PC.



alpha1/main.cpp
```
static void task( void* args ) {
  runTheApplication(
     getBspConsoleIn(),
     getBspConsoleOut() )
}

main(...) {
  initializeBsp()
  initializeCplLibrary()

  createFreeRTOSThread( task, ... )
  startFreeRTOS()
}
```

appmain.h
```
runTheApplication( cli_infd, cli_outfd )
```

simulator/main.cpp
```
main(...) {
  initializeCplLibrary()
  infd   = stdin
  outfd = stdout

  runTheApplication(infd, outfd)
}
```

alpha1Platform.cpp
```
EEPROM nvramDriver_(...)

platform_initialize() {
  nvramDriver_.start()
}

platform_open() {
  ...
}

platform_close() {
  ...
}

platform_exit() {
  bsp_reset_MCU()
}
```

appmain.cpp
```
static HeatingAlgorithm heatAlgo_(...)
...
runTheApplication( infd, outfd ) {
  platform_initialize()

  startDebugShell( infd, outfd )
  createApplicationThread( ... )

  platform_open()
  heatAlgo_.open()

  waitForShutdown()

  heatAlgo_.close()
  platform_close()

  platform_exit()
}
```

simulatorPlatform.cpp
```
SimEEPROM nvramDriver_(...)

initializePlatform() {
  nvramDriver_.start()
}

openPlatform() {
  ...
}

platform_close() {
  ...
}

platform_exit() {
  exit();
}
```

platform.h
```
platform_initialize()
platform_open()
platform_close()
platform_exit()
```

Figure 7-4. *Pseudocode for a Main pattern implementation*

The GM6000 example extends the basic Main pattern to support multiple applications and target hardware variants. However, a concrete example of the basic Main pattern can be found in the `main-pattern_initial-skeleton-projects` branch in the GitHub repository. The directories of interest are

- `src/Ajax/Main`
- `src/Ajax/Main/_app`
- `src/Ajax/Main/_plat_alpah1`
- `src/Ajax/Main/_plat_simulator`

Application Variant

It may be that you have a marketing requirement for good, better, and best versions of your application, or it may be that you just need a test-enabled application for a certification process. But whatever the reason for creating additional applications, each separate application will be built from a large amount of common code (see Figure 7-5).

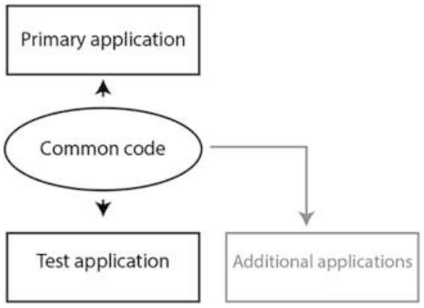


***Figure 7-5.** Building multiple applications from common code*

Additionally, in order for your different applications to execute on your target hardware or simulator, you will need to write additional platform-specific code. But even here there will be a certain amount of common platform code (see Figure 7-6).

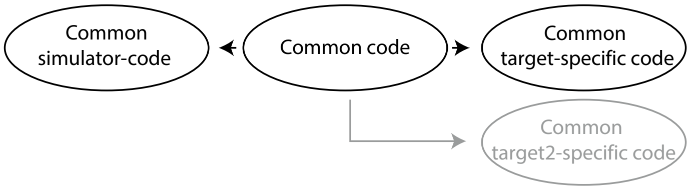


Figure 7-6. *Target-specific common code*

Essentially there are two axes of variance: the Y axis for different application variants and the X axis for different platforms (target hardware, simulator, etc.). Starting with two application variants (a main application and an engineering test application) and two platforms (target hardware and simulator), this yields nine distinct code combinations (see Figure 7-7).

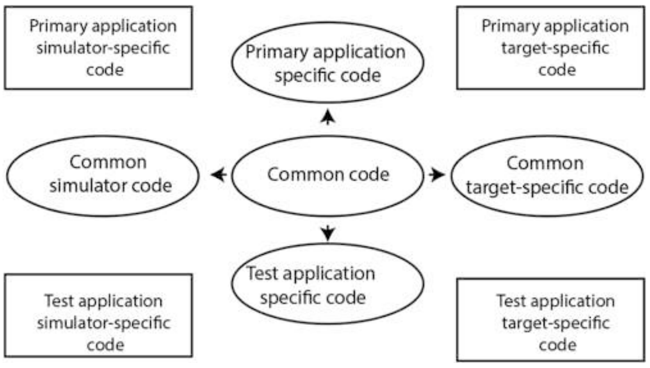

Primary application simulator-specific code
Primary application specific code
Primary application target-specific code
Common simulator code
Common code
Common target-specific code
Test application simulator-specific code
Test application specific code
Test application target-specific code

Figure 7-7. *Structure for multiple applications and multiple targets*

Admittedly, these combinations can get a bit overwhelming. As discussed earlier, it is tempting to resort to using `#ifdefs` to manage the platform, application, or platform-application specific deltas. However, it is better to extend the Main pattern to handle these new combinations. For the GM6000 example, the Main pattern has been extended to build two application variants and three platform variants.

When your project includes multiple hardware platforms and multiple application variants, the number of different combinations or permutations exists regardless of how you manage the combinations in the source code. Failing to plan for these combinations results in either a rat's nest of `#ifdefs` or duplicate code. However, at the start of the project, you may only have a single hardware platform or single application variant, so it's logical to ask, "How can I plan for the unknown?" My recommendation is to always start with the basic Main pattern: a single application running on the target hardware platform and the simulator. It can be extended later to include new application and hardware variants. The only downside to this just-in-time approach is that the directory and file naming can get a little messy.

Marketing Abstraction Layer

The marketing folks are always changing the names of products and projects, so if I use the product name *du jour* when naming directories, files, and variables, it is unlikely that they will match the final product names. Consequently, to avoid the problems associated with evolving names, I prefer to start with an arbitrary names—in fact, dramatically different names. In the GM6000 project, I chose `Ajax` for the primary application and `Eros` for the test application.

Ajax Main and Eros Main

Because the `Main` pattern facilitates the reuse of common components across different application builds and isolates platform dependencies, both applications in the GM6000 project use the `Main` architectural pattern to build. Additionally, the `Main` pattern is extended so that much of the creation logic and sequencing business logic can be shared across multiple platforms and application variants. The GM6000 implementation of the `Main` pattern supports two application variants (product and engineering test) and two[2] platform variants (target hardware and simulator), yielding four possible permutations that need to be considered.

- Ajax (primary application), target build
- Ajax (primary application), simulator build
- Eros (test application), target build
- Eros (test application), simulator build

From past experience, I have found that each of these variants is necessary, even though at the beginning of the project, some of the variants are identical. For example, at the start of the project, there will not be any application differences between the simulator builds for `Ajax` and `Eros`.

The details of how to support common `Main` code across the four permutations are captured in the Software Detailed Design (SDD) document. And, yes, I created the design first, before writing the code. Here is a snippet of Section SDD-32 from the SDD for the Ajax application. Additionally, Figure 7-8 gives you an idea of how the application- and target-specific code is organized in the source tree.

[2] The GM6000 example actually supports three different platforms: the simulator and two different hardware boards. To simplify the discussion for this chapter, only two platforms–the simulator and STM32 hardware target–are used. Scaling up to more target platforms or application variants only requires adding "new" code, that is, no refactoring of the existing Main pattern code.

The implementation of the Main pattern uses internal interfaces so that the core creation and sequencing logic can be common across platform and application variants. There are at least two platforms: the initial NUCLEO-F413ZH board and the function simulator. There are at least two application variants: the Ajax application for the production release of the GM6000 and the Eros Engineering Test software application. The design decision to share the startup code across the different variant was made to avoid maintaining N separate startup sequences.

Per the coding standard, the use of `#ifdefs` *within functions is not allowed. The different permutations are done by defining functions (i.e., internal interfaces) and then static linking the appropriate implementations for each variant.*

The following directory structure and header files (for the internal interfaces) shall be used. The directory structure assumes that the build scripts for the different variants build different directories vs. cherry-picking files within a directory.

```
src/Ajax/Main/             // Platform/Application independent implementation
+--- platform.h            // Interface for Platform dependencies
+--- application.h         // Interface for App dependencies
+--- _app/                 // Ajax specific startup implementation
+--- _plat_xxxx/           // Platform variant 1 start-up implementation
|    +--- app_platform.h   // Interface for Platform specific App dependencies
|    +--- _app_platform/   // Ajax-Platform specific startup implementation
+--- _plat_yyyy/           // Platform variant 2 start-up implementation
|    +--- app_platform.h   // Interface for Platform specific App dependencies
|    +--- _app_platform/   // Ajax-Platform specific startup implementation
```

Here is a snippet from SDD-33 from the SDD for the Eros application.

The Eros application shares, or extends, the Ajax Main pattern (see the "[SDD-32] Creation and Startup (Application)" section). The following directory structure shall be used for the Eros-specific code and extensions to the Ajax startup/shutdown logic.

```
src/Eros/Main/              // Platform/Application Specific implementation
+--- app.cpp                // Eros Application (non-platform) implementation
+--- _plat_xxxx/            // Platform variant 1 start-up implementation
|    +--- app_platform.cpp  // Eros app + specific startup implementation
+--- _plat_yyyy/            // Platform variant 2 start-up implementation
```

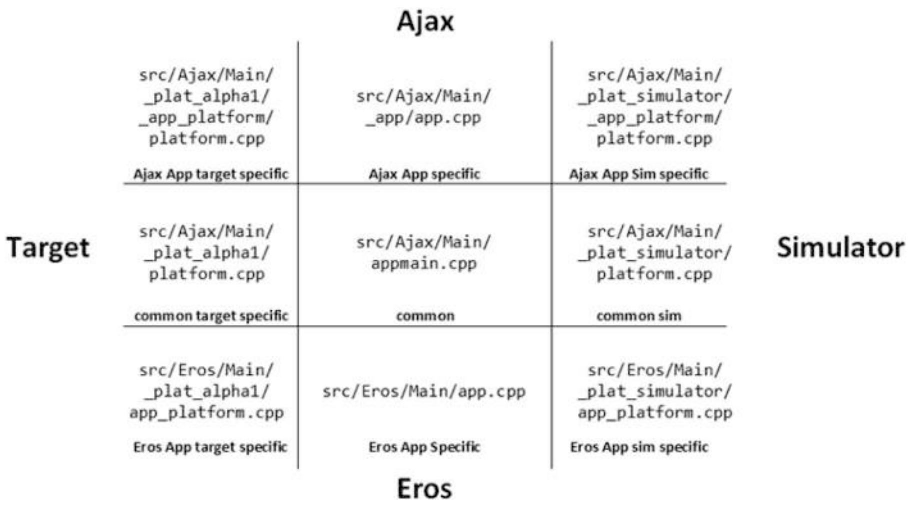

Ajax
src/Ajax/Main/_plat_alpha1/_app_platform/platform.cpp
Ajax App target specific
src/Ajax/Main/_app/app.cpp
Ajax App specific
src/Ajax/Main/_plat_simulator/_app_platform/platform.cpp
Ajax App Sim specific
Target
src/Ajax/Main/_plat_alpha1/platform.cpp
common target specific
src/Ajax/Main/appmain.cpp
common
src/Ajax/Main/_plat_simulator/platform.cpp
common sim
Simulator
src/Eros/Main/_plat_alpha1/app_platform.cpp
Eros App target specific
src/Eros/Main/app.cpp
Eros App Specific
src/Eros/Main/_plat_simulator/app_platform.cpp
Eros App sim specific
Eros

Figure 7-8. *Source code organization for application and targets*

Build Scripts

After creating the code for the largely empty `Main` pattern, the next step is to set up the build scripts for the skeleton projects. For the GM6000 project, I use the `nqbp2` build system. In fact, all the build examples in the book assume that `nqbp2` is being used. However, you can use whatever collection of makefiles or scripts that you are comfortable using. Appendix F, "NQBP2 Build System," provides information on how to use `nqbp2` to extract the compiler and link options that are used when building the GM6000 code.

My favorite thing about `nqbp2` is the feature that—after a project has been set up—I only have to maintain a list of directories to build. That list is contained in a file called `libdirs.b`. For the `Main` pattern this translates to having common `libdirs.b` files that call out common source code directories.

With `nqbp2`, a build directory is defined as a directory under the root directories `projects/` or `tests/`. These are the directories where the code is compiled; they contain the build scripts and typically the C/C++ `main()` entry function. Here are the four build directories for building the Ajax and Eros applications:

- `projects/GM6000/Ajax/alpha1/windows/gcc-arm`
- `projects/GM6000/Ajax/simulator/windows/vc12`
- `projects/GM6000/Eros/alpha1/windows/gcc-arm`
- `projects/GM6000/Eros/simulator/windows/vc12`

In practice, there are actually more build directories than this because the simulator is built using both the MinGW Windows compiler and the GCC compiler for a Linux host.

Shared `libdirs.b` files are created for each possible combination of common code. These files are located under the `/project/GM6000` tree in various subdirectories.

File	Contents
`common_libdirs.b`	Directories (i.e., code) that are common to all variants.
`sim_common_libdirs.b`	Directories that are common to all simulator variants.
`target_common_libdirs.b`	Directories that are common to all target variants.
`ajax_common_libdirs.b`	Directories that are platform independent and common to the Ajax application.
`ajax_target_common_libdirs.b`	Directories that are target dependent and common to the Ajax application.
`ajax_sim_libdirs.b`	Directories that are simulator dependent and common to the Ajax application.
`Eros_common_libdirs.b`	Directories that are platform independent and common to the Eros application.
`Eros_target_common_libdirs.b`	Directories that are target dependent and common to the Eros application.
`eros_sim_libdirs.b`	Directories that are simulator dependent and common to the Eros application.

Figures 7-9 and 7-10 are listings of the master `libdirs.b` files for the Ajax target (`alpha1` HW) build and the simulator (Win32-VC) build.

```
# Common stuffs
../../../ajax_common_libdirs.b
../../../ajax_target_common_libdirs.b
../../../../common_libdirs.b
../../../../target_common_libdirs.b

# target specific stuffs
src/Ajax/Main/_plat_alpha1
src/Ajax/Main/_plat_alpha1/_app_platform

# BSP
src/Cpl/Io/Serial/ST/M32F4
src/Bsp/Initech/alpha1/trace
src/Bsp/Initech/alpha1
src/Bsp/Initech/alpha1/MX
src/Bsp/Initech/alpha1/MX/Core/Src > freertos.c
src/Bsp/Initech/alpha1/console

# SDK
xsrc/stm32F4-SDK/Drivers/STM32F4xx_HAL_Driver/Src >
   stm32f4xx_hal_timebase_rtc_alarm_template.c
   stm32f4xx_hal_timebase_rtc_wakeup_template.c
   stm32f4xx_hal_timebase_tim_template.c

# FreeRTOS
xsrc/freertos
xsrc/freertos/portable/MemMang
xsrc/freertos/portable/GCC/ARM_CM4F

# SEGGER SysVIEW
src/Bsp/Initech/alpha1/SeggerSysView
```

Figure 7-9. *Contents of file projects/GM6000/Ajax/alpha1/windows/gcc-arm/libdirs.b*

```
# Use common libdirs.b
../win32_libdirs.b
../../ajax_sim_libdirs.b
../../../ajax_common_libdirs.b
../../../../common_libdirs.b
../../../../sim_common_libdirs.b
```

Figure 7-10. *Contents of file projects/GM6000/Ajax/simulator/windows/vc12/libdirs.b*

Preprocessor

One of the best practices that is used in the development of the GM6000 application is the LConfig pattern.[3] The LConfig pattern is used to provide project-specific configuration (i.e., magic constants and preprocessor symbols). Each build directory has its own header file—in this case, `colony_config.h`—that provides the project specifics. For example, the `LConfig` pattern is used to set the buffer sizes for the debug console and the human-readable major-minor-patch version information.

As with the build script, there are common configuration settings across the application and platform variants. Shared `colony_config.h` files are created for each possible combination.

Simulator

When initially creating the skeleton applications, there are very few differences between the target build and the simulator build. The actual differences at this point are as follows:

- Thread console command—This command is platform specific.
- How the board is initialized—The BSP has to be initialized for the target platform.
- How an application terminates—On the target platform, terminating the application reboots the microcontroller; on the simulator, the executable simply exits.

[3] A detailed discussion of the LConfig pattern can be found in *Patterns in the Machine: A Software Engineering Guide to Embedded Development.*

Because there are very few differences, it is tempting to skip including the simulator builds as part of creating the skeleton applications. But don't. Supporting a functional simulator has minimal overhead if implemented from the beginning of the project, and it is very painful to retrofit the simulator into the development process later. Not only should you not skip the simulator skeleton builds, but it should also be the first skeleton build you create because typically no target hardware is available yet. Even if the initial target hardware is an off-the-shelf evaluation board, it is still worth your time to build a simulator.

Functionality will be incrementally added to the simulator during the construction stage. As hardware drivers are needed, part of driver design includes specifying how the driver will be simulated.

The Fine Print

Standing up a simulator skeleton application is almost trivial when using the CPL C++ class library (see Appendix E, "CPL C++ Framework," for more details). The CPL class library was specifically designed and built to decouple code from the underlying platform. That is, the class library provides an Operating System Abstraction Layer (OSAL), inter-thread communications, stream interfaces, logging, persistent storage, etc., that have no direct platform dependencies. If you do not use the CPL library, then you either have to build a functional equivalent or use a third-party package for each of these things. I recommend that you do not build it from scratch. The OSAL and other middleware functionality are not particularly rocket science, but it does have a lot of details and nuances that take time to get right.

Summary

Creating the skeleton applications that include the functional simulator variants is the primary deliverable for the foundation step. The `Main` pattern is an architectural pattern that is used to allow reuse of platform-independent code across the different platforms and application variants. Constructing the skeleton applications is mostly a boilerplate activity, assuming you have an existing middleware package (such as the CPL C++ class library) that provides interfaces and abstractions that decouple the underlying platform. Avoid the temptation to create the skeleton applications in combination with features, drivers, etc. From my experience, creating the skeletons upfront helps identify all the possible build variants, which helps the implementation of the `Main` pattern to be cleaner and significantly easier to maintain.

For the GM6000, an example of the initial skeleton applications using the Main pattern with support for multiple applications and target hardware variants can be found on the `main-pattern_initial-skeleton-projects` branch in the Git repository.

INPUTS

- The initial draft of the Software Development Plan
- The initial draft of the software architecture document
- The initial draft of the Software Detailed Design document

OUTPUTS

- Skeleton applications that provide the creation and top-level sequencing structure for the various applications
- The functional simulator framework in place
- Updated Software Detailed Design (SDD) document with the `Main` pattern design

CHAPTER 8

Continuous Integration Builds

This chapter walks through the construction of the "build-all" scripts for the GM6000 example code. The continuous integration (CI) server invokes the build-all scripts when performing pull requests and merging code to stable branches such as `develop` and `main`. At a minimum, the build-all scripts should do the following things:

- Build all unit tests, both manual and automated unit tests
- Execute all automated unit tests and reports pass/fail
- Generate code coverage metrics
- Build all applications. This includes board variants, functional simulator, debug and nondebug versions, etc.
- Run Doxygen (assuming you are using Doxygen)
- Help with collecting build artifacts. What this entails depends on what functionality is or is not provided by your CI server. Some examples of artifacts are
    - Application images
    - Doxygen output
    - Code coverage data

J. T. Taylor and W. T. Taylor, *The Embedded Project Cookbook*,
https://doi.org/10.1007/979-8-8688-0327-7_8

Additionally, your scripts should have the following characteristics:

- The script accepts a unique build-number that is then passed to the application builds. The build-number is used as the canonical version identifier for the applications. By having the CI server generate the canonical build-number, it eliminates the possibility of human errors in setting the canonical identifier (e.g., the developer forgot to increment the version number before checking in their code).
- Adding (or removing) a unit test should not require edits to the build-all script since the number of unit tests will always be increasing as progress is made on the project.
- Adding (or removing) an application image build should not require edits to the build-all script.
- Any and all build, test, or script errors are reported as failures to the CI server.

Unfortunately, the construction of the build-all script is left as an exercise to the reader. I can't provide a universal script because there are just too many dependencies to take into account. For example, the build-all script will be dependent on the host platform, what build engine you are using (traditional makefiles, cmake, nqbp2, etc.), your source code organization, your repository organization (single vs. multiple repositories), etc. What I can do, however, is walk you through the details of the GM6000 build-all scripts to illustrate what is involved.

Example Build-All Scripts for GM6000

The remainder of the chapter discusses the build-all scripts for the GM6000 example. This includes the GM6000's build system, file organization, naming conventions, projects, unit tests, and the build artifacts as they relate to the build-all scripts.

The CI Server

At this point, the CI server should be up and running and should be able to invoke scripts stored in the code repository (see Chapter 5, "Preparation"). In addition, the skeleton applications should have been constructed (see Chapter 6, "Foundation"), and there should be a certain number of existing unit tests that are part of the CPL C++ class library.

The build system used for examples in this book is nqbp2 (see Appendix F, "nqbp2 Build System"). However, the code itself has no direct dependencies on nqbp2, which means you can use a different build system, but you will need to create the build makefiles and scripts.

Directory Organization

The key directories at the root of the source code tree are as follows:

- `src/`—The primary source code directory for the example project
- `xsrc/`—The location of source code for third-party packages

- `top/`—The location of the build-all scripts
- `projects/`—The directory where the applications are built and the released images are created
- `tests/`—The directory where the unit tests are built. This directory should only contain manual or automated unit tests.

To reduce the complexity and maintenance required for the build-all script, two tools from the nqbp2 build system are used: `bob.py` and `chuck.py`. The `bob.py` script recursively finds and executes all `nqbp.py` scripts in a specified directory tree. The `chuck.py` script finds and executes unit tests in a specified directory tree. The list of unit tests to execute is passed as a command-line argument to the `chuck.py` script. These scripts—along with some naming conventions—eliminate the need to enumerate each individual project and unit test in the build-all script because what gets built and executed is discovered when the script runs.

Naming Conventions

You can use whatever naming convention you would like. The important thing is that you should be able to differentiate scripts for different use cases by simply looking at the file name. For example, here are some key use cases to differentiate:

- Manual vs. automated unit tests
- Tests that can, or cannot, execute in parallel with other unit tests
- Tests that can be executed directly vs. tests that need to be executed with a script

For the GM6000 project, there are no mandated naming conventions for executables under the `projects/` tree. However, there is a naming convention for unit test executables to simplify executing the unit tests in a CI environment. Admittedly, some of my conventions aren't as intuitive or descriptive as I'd like. My conventions have evolved over time, and some of my unfortunate early naming choices have now become solidified in and across my code bases. Consequently, consider the following conventions as examples of how I solved the problem, not necessarily the only or best way to name things.

Table 8-1 provides a shorthand summary of the naming conventions.

Table 8-1. *Summary of build script naming conventions for unit tests*

Parallel	Not Parallel
a.exe, a.out	aa.exe, aa.out
a.py [runs] b.exe, b.out	aa.py [runs] bb.exe, bb.out

Here is a snippet from the Software Detailed Design document (Appendix M, "GM6000 Software Detailed Design (Final Draft)") that spells out the naming conventions used.

Per the Software Development Plan, all namespaces shall have at least one unit test. The unit tests can be manual or automated (the preference is for automated unit tests). To simplify the CI build process, the following naming conventions are imposed for the unit text executables.

Name		***Description***
`a.exe`, `a.out`	*Single lowercase "a" prefix to denote parallel*	*Scripts used for all automated units that can be run in parallel with other units.*
`aa.exe`, `aa.out`	*Dual lowercase "aa" prefix to denote not parallel*	*Scripts used for all automated units that cannot be run in parallel with other units. An example here would be a test that uses a hard-wired TCP port number.*
`b.exe`, `b.out`	*Single lowercase "b" prefix to denote parallel and require a script*	*Scripts used for all automated units that can be run in parallel and that require an external Python script to run. An example here would be piping a golden input file to stdin of the test executable.*
`bb.exe`, `bb.out`	*Dual lowercase "bb" prefix to denote not parallel and require a script*	*Scripts used for all automated units that cannot be run in parallel and that require an external Python script to run.*
`a.py`	*Single lowercase "a" prefix to script to denote it will be running only tests prefixed with single lowercase "b"*	*Python program used to execute the b.exe and b.out executables.*
`aa.py`	*Dual lowercase "aa" prefix to script to denote it will be running only tests prefixed with dual lowercase "bb"*	*Python program used to execute the bb.exe and bb.out executables.*
`<all others>`		*Manual units can use any name for the executable except for the one listed previously.*

Windows build_all Script

The Windows `build_all` script is located in the `top/` directory. The script takes two command-line arguments:

- buildNumber—This is the Jenkins project BUILD_NUMBER environment variable.
- branch—This is the branch name. It is only set when building the develop or main branches.

The script performs the following actions:

- It creates the temporary `_artifacts/` directory and ensures that it is empty. This is where Jenkins collects build artifacts.
- It runs Doxygen and copies the output to the `_artifacts/` directory.
- It builds all automated unit tests for the Windows platforms. There are two platforms:
  - Visual Studio compiler
  - MinGW_W64 compiler toolchain. The code coverage metrics are generated from these units.

    Running the same unit test built with different compilers will bring out errors that are masked by a particular compiler. For example, the Visual Studio compiler is notoriously forgiving when it comes to writing non-ISO standard C/C++ code.
- It runs all the unit tests that have been built.
- It builds all target unit tests (e.g., builds that use the GCC-ARM cross compiler). Typically these are manual unit tests.

- It builds all applications under the `projects/` directory for all Windows platforms (Windows+MinGW_64, Windows+VC12).
- It builds all applications under the `projects/` directory for the hardware target (e.g., projects that use the GCC-ARM cross compiler).
- It copies application images to the `_artifacts/` directory. This includes:
  - All target application images (`.elf, .hex, .map, .list,` etc.)
  - Windows+VC12 simulator images (`.exe, .pdb,` etc.)
- It builds *debug* versions of all applications under the `projects/` directory for the hardware target.

If any of these actions fail, then the build fails.

The script is built so that adding new unit tests (automated or manual) or new applications under the `projects/` directory does not require the script to be modified.

If you are adding a new hardware target—for example, changing from the STM32 `alpha1` board to the STM32 `dev1` board—then a single edit to script is required to specify the new hardware target.

If new build artifacts are needed, then the script has to be edited to generate the new artifacts and to copy the new items to the `_artifacts/` directory.

Table 8-2 summarizes where changes need to be made to accommodate changes to the build script.

Table 8-2. *Summary of build script modifications*

Event	**Required Change**
New Unit Test	Create a new unit test build directory under the `tests/` directory.
New Build Artifact	Edit the build-all scripts to generate the new build artifact and copy it to the `_artifacts/` directory.
New Target Hardware	Edit the build-all scripts as needed. If the new target is a "next board spin," then only the "`_TARGET`" and "`_TARGET2`" variables need to be updated.
New Application	Create the application build directory under the `projects/` directory.

None of the aforementioned change scenarios require any modifications to the Jenkins projects that invoke the `build_all` script. The following are snippets from the `top/build_all_windows.bat` script file:

```
@echo on
:: This script is used by the CI\Build machine to build the Windows Host
:: projects
::
:: usage: build_all_windows.bat <buildNumber> [branch]

set _TOPDIR=%~dp0
set _TOOLS=%_TOPDIR%..\xsrc\nqbp2\other
set _ROOT=%_TOPDIR%..
set _TARGET=alpha1
set _TARGET2=alpha1-atmel

:: Set Build info (and force build number to zero for "non-official" builds)
set BUILD_NUMBER=%1
set BUILD_BRANCH=none
IF NOT "/%2"=="/" set BUILD_BRANCH=%2
IF "%BUILD_BRANCH%"=="none"" set BUILD_NUMBER=0
echo:
echo:BUILD: BUILD_NUMBER=%BUILD_NUMBER%, BRANCH=%BUILD_BRANCH%
```

```
:: Make sure the _artifacts directory exists and is empty
cd %_ROOT%
rmdir /s /q _artifacts
mkdir _artifacts

:: Run Doxygen first (and copy the output to artifacts dir)
cd %_TOPDIR%
run_doxygen.py %BUILD_NUMBER% %BUILD_BRANCH%
IF ERRORLEVEL 1 EXIT /b 1
...

:: Build Mingw projects (just the Win32 builds)
call %_ROOT%\env.bat 3

:: Build NON-unit-test projects
cd %_ROOT%\projects
%_TOOLS%\bob.py -v4 mingw_w64 -c --bldtime -b win32 --bldnum %BUILD_NUMBER%
IF ERRORLEVEL 1 EXIT /b 1

:: Build unit test projects (debug builds for more accurate code coverage)
cd %_ROOT%\tests
%_TOOLS%\bob.py -v4 mingw_w64 -cg --bldtime -b win32  --bldnum %BUILD_NUMBER%
IF ERRORLEVEL 1 EXIT /b 1

:: Run Mingw unit tests
cd %_ROOT%\tests
%_TOOLS%\chuck.py -v --match a.exe --dir mingw_w64
IF ERRORLEVEL 1 EXIT /b 1
%_TOOLS%\chuck.py -v --match aa.exe --dir mingw_w64
IF ERRORLEVEL 1 EXIT /b 1
%_TOOLS%\chuck.py -v --match a.py --dir mingw_w64
IF ERRORLEVEL 1 EXIT /b 1
%_TOOLS%\chuck.py -v --match aa.py --dir mingw_w64
IF ERRORLEVEL 1 EXIT /b 1

:: Generate code coverage metrics
%_TOOLS%\chuck.py -v --dir mingw_w64 --match tca.py rpt --xml jenkins-gcovr.xml
IF ERRORLEVEL 1 EXIT /b 1

...
```

```
:: Skip additional project builds when NOT building develop or main
IF "%BUILD_BRANCH%"=="none" GOTO :builds_done

:: Zip up (NON-debug) 'release' builds
cd %_ROOT%\projects\GM6000\Ajax\%_TARGET%\windows\gcc-arm\_stm32
7z a ajax-%_TARGET%-%BUILD_NUMBER%.zip ajax.*
IF ERRORLEVEL 1 EXIT /b 1
copy *.zip %_ROOT%\_artifacts
IF ERRORLEVEL 1 EXIT /b 1

...

:: Everything worked!
:builds_done
echo:EVERTHING WORKED
exit /b 0
```

Linux build_all Script

The Linux `build_all` script is located in the `top/` directory. The script takes one command-line argument:

- buildNumber—This is the Jenkins project BUILD_ NUMBER environment variable.

The script performs the following actions.

- It builds all automated unit tests for the Linux platforms using the GCC compiler.
- It runs all the unit tests that have been built.
- It builds all applications under the `projects/` directory for Linux platform.

If any of the actions fail, then the build fails.

The script is designed the same as the Windows `build_all` script so that adding new unit tests or application projects does not require the script to be modified. The Linux `build_all` script is much shorter because only one compiler is used and things like running Doxygen, code coverage, and publishing build artifacts are only done once (in the Windows `build_all` script).

```
# This script is used by the CI/Build machine to build the Linux projects
#
# The script ASSUMES that the working directory is the package root
#
# usage: build_linux.sh <bldnum>
#
set -e

# setup the environment
source ./env.sh default

# Build all test linux projects (only 64bit versions)
pushd tests
$NQBP_BIN/other/bob.py -v4 linux  -gb posix64 --bldtime --bldnum $1

# Run unit tests
$NQBP_BIN/other/chuck.py -v --match a.out --dir _posix64
$NQBP_BIN/other/chuck.py -v --match aa.out --dir _posix64
$NQBP_BIN/other/chuck.py -v --match a.py --dir _posix64
$NQBP_BIN/other/chuck.py -v --match aa.py --dir _posix64

# Build all "projects/" linux projects (only 64bit versions)
popd
pushd projects
$NQBP_BIN/other/bob.py -v4 linux  -gb posix64 --bldtime --bldnum $1
```

Summary

It is critical that the CI server and the CI build scripts are fully up and running before the construction stage begins. By not having the CI process in place, you risk starting the project with broken builds and a potential descent into initial merge hell.

INPUTS

- The CI server is up and running and able to invoke build scripts stored in the GitHub repository.
- The skeleton applications have been created and can successfully build locally on a developer's computer. This includes all of the functional simulator builds, target builds, and all unit tests.

OUTPUTS

- Completed repository-based CI build scripts
- The CI build scripts in the repository fully integrated with the CI server and its automation scripts/projects
- The CI builds are full featured. That is, they support and produce the following outputs on every build.
    - They create Doxygen output.
    - They execute automated unit tests on every build.
    - They gather code coverage metrics from the automated unit tests.

- They tag the develop and main branches in GitHub with the CI build number.
- They archive the build images and other desired outputs for access at a later date. Archiving could involve using sftp or scp, calling RESTful APIs for cloud storage, or even standing up, and writing to, an Amazon S3 file manager.

CHAPTER 9

Requirements Revisited

It is extremely rare that all of the requirements have been defined before the construction phase. In fact, it is not even reasonable to expect that the requirements will be 100% complete before starting implementation. The reason is because unless all of the details are known—and nailing down the details is what you do in the construction phase—it is impossible to have all of the requirements carved in stone. So in addition to design, implementation, and testing, there will inevitably be a continuing amount of requirement work that needs to be addressed. This includes refactoring and clarifying existing requirements as well as developing new requirements. Common sources for new requirements that show up in the construction phase are as follows:

- Feedback from Failure Mode and Effects Analysis (FMEA) or risk control measures
- Feedback from detailed cybersecurity analysis
- Feedback from the detailed designs for software, hardware, mechanical, etc. This is where the wish list of requirements gets reconciled with the real world
- Feedback from voice of the customer (VOC), as well as Alpha and Beta trials

J. T. Taylor and W. T. Taylor, *The Embedded Project Cookbook*,
https://doi.org/10.1007/979-8-8688-0327-7_9

Analysis

There are many types of analysis (e.g., FMEA) that are typically done for an embedded product to ensure it is safe, reliable, compliant with regulatory agencies, and secure. These tests and analysis cover all aspects of the product—mechanical, electrical, packaging, and manufacturing—not just software. This is where many of the edge cases are uncovered and possible solutions are proposed. Often, these solutions take the form of new requirements, which are also sometimes called risk control measures. Typically, these new requirements are product (PRS) or engineering detailed requirements (SRS)—although sometimes they feed back into the marketing (MRS) requirements.

Failure analysis is a system engineering role, not a software role. As such, a detailed discussion of analysis methods, best practices, etc., is outside the scope of this book. However, it is not unusual for a hardware engineer to fill the role of the system engineer when it comes to failure analysis.

It is important to start these analysis activities as soon as possible and to iterate through them as the detailed design evolves. This means some of the analysis occurs before the construction phase. Here is a hypothetical example of a requirement being added late in the process.

RISK CONTROL MEASURE EXAMPLE

The initial PRS (PR-103) and system architecture required a hardware-based protection circuit so that the heating element does not exceed temperature and safety limits. However, while evaluating possible failure modes, it was observed that when the heater safety circuit shut off the heating element, the software would be unaware that there is no active heating. This would have the following impact:

- The software control algorithm would have the potential to "run away" (e.g., integral wind-up[1]) because there is no actual heat.
- An end user would not have any indication that something is wrong with the unit other than they are unable to control space temperature and declare the unit defective.

To mitigate this risk, the following requirement was added as a risk control measure to the PRS:

PR-206: Heater Safety. The Heating Element (HE) sub-assembly shall provide an indication to the Control Board (CB) when it has been shut off due to a safety limit.

Requirements vs. Design Statements

Requirements should be simple to define, but, unfortunately, they are not. The problem is that one stakeholder's "what" is another stakeholder's "how." Consequently, there will always be gaps and overlaps between where requirements end and where detailed designs begin. Nevertheless, it is better to just acknowledge the gaps, rather than go back and forth in an attempt to get the absolute right level of requirements from the stakeholders. The way I acknowledge these gaps is through the use of design statements. They are also useful in situations where high fidelity to details will be needed in the implementation. For example, design statements are perfect for describing all the gory details of the user interface: colors, fonts, layout, text strings, etc.

[1] See https://en.wikipedia.org/wiki/Integral_windup

Design Statement for Control Algorithm

The term *design statement* is simply a placeholder for detailed specifications that must be implemented but which are not part of—but rather are derived from—the formal requirements. For example, the definitions of control algorithms are typically specified separately as design statements. The reason is because the stakeholder providing the requirements (e.g., the marketing guy) is not the author (e.g., the guy with a PhD in control theory). In addition, most of the time, the software developer will not be the author of the control algorithm either, so detailed design statements about the algorithm are needed. Here is a hypothetical example of design statements for a control algorithm.

GM6000 HEATING DESIGN STATEMENT EXAMPLE

There is a single PRS requirement for the heating algorithm for the GM6000 example.

PR-207: Temperature Control*. The control algorithm shall maintain the sensed space temperature within ± one degree Celsius of the setpoint under the following conditions:*

- *A constant thermal load on the space.*
- *At least five minutes after the sensed space temperature reached the setpoint temperature.*
- *The sensed space temperature is the measured value (after any SW/HW filtering). That is, it's not the actual, independently measured space temperature.*

A choice was made to use a fuzzy logic controller (FLC) for the core temperature control algorithm instead of simple error or proportional-integral-derivative (PID) algorithms. The details and nuisances of the FLC are nontrivial, so the algorithm design was captured as a separate document (see Appendix N, "GM6000 Fuzzy Logic Temperature Control"). Here is a snippet from the FLC document.

Fuzzification

There are two input membership functions: one for absolute temperature error and one for differential temperature error. Both membership functions use triangle membership sets as shown in Figure 9-1.

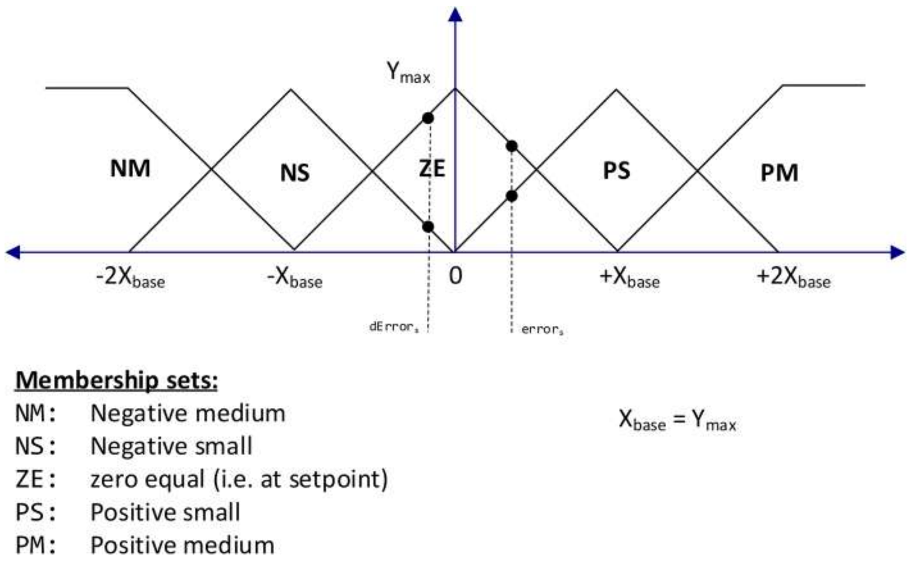


Figure 9-1. *Membership function*

Absolute error is calculated as

$$error = temperatureSetpoint - currentSpaceTemperature$$
$$error_s = error * J_{error}$$

Differential error is calculated as

$$dError = error - lastError$$
$$dError_s = dError * J_{derror}$$

The values for Ymax, J_{error}, and J_{derror} are configurable.

The units for the X axis are hundredths degrees Celsius.

The Y axis is unitless. It represents a logical 0.0 to 1.0 weighted membership value.

An input (e.g., `error` or `dError`) is fuzzified by determining its weighted membership in each of the membership sets.

Design Statement for User Interface

As I mentioned previously, user interface details are also candidates for design statements. Of course, one option would be to put all of the UI details into a formal requirement. However, formal requirements have a heavy change control process associated with them, and everyone knows the UI is always being tweaked up until the moment of release. Detailed design statement documents typically have a much lighter change control process and require fewer approvals to update. In my experience, design statements for the UI take the form of wireframe documents and style guides for supporting look and feel.

The formal UI requirements should call out

- Behavior (what controls are available to the user)
- Content (what information needs to be displayed)
- Basic navigation (how to go from one screen to another)

The remaining details (layout, colors, text, fonts, types of controls, etc.) are left to the design statements. With respect to UI requirements, the GM6000 example has four MRS requirements, one PRS requirement, and eight SRS requirements. None of the requirements get into the details of layout, colors, text strings, button assignments, etc.

For example, the epc repository's Wiki contains a link to view the wireframes for the UI. Figures 9-2, 9-3, and 9-4 show examples of those design statements.

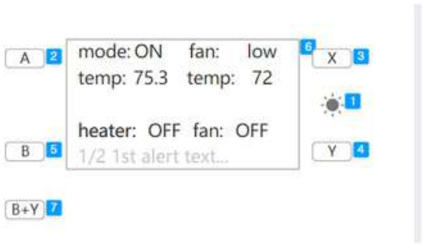
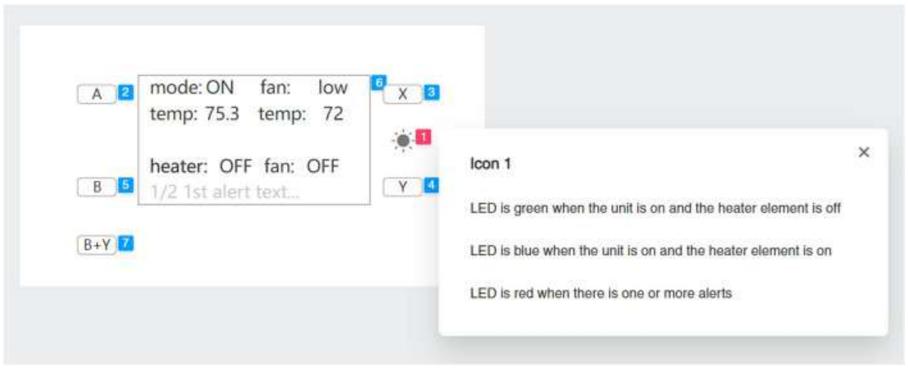


Figure 9-2. *Home Screen wireframe*

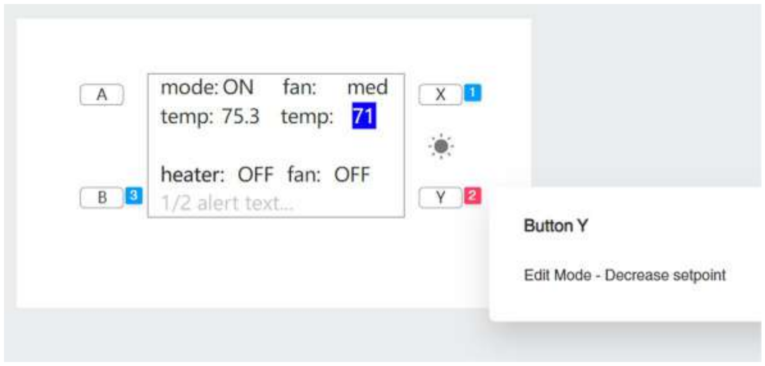


LED is green when the unit is on and the heater element is off

LED is blue when the unit is on and the heater element is on

LED is red when there is one or more alerts

Figure 9-3. *Home Screen wireframe with LED details*



Edit Mode - Decrease setpoint

Figure 9-4. *Edit Setpoint Screen*

As a final note about design statements and testing, in most companies, it is assumed that the software test team will be responsible for verifying that the software meets the formal requirements. However, the same cannot be said about design statements. The principal reasons for this are as follows:

1. Schedule pressure—All of the available test resources and time are taken up completing verification of the formal requirements.
2. Lack of pass/fail criteria—Since design statements do not have the rigor (e.g., a unique identifier, or the formalized "shall" and "should" wording) of formal requirements, it is not always clear what needs to be tested and how.
3. Ambiguity over responsibility—It is not always clear who is responsible for testing design statements.

Nevertheless, in the end, it is critical that the overall project schedule accounts for testing design statements. The schedule should identify the individuals responsible for performing the verification as well as provide time for this work to occur. Do not skip testing design statements!

Missing Formal Requirements

Yet another nuisance with requirements is there can be requirements that you are held accountable for that are not part of formal requirements. Typically, these requirements derive from implicit, nonfunctional requirements, or they materialize as the result of a project-level decision.

A nonfunctional requirement (NFR)[2] is a requirement that describes how you should do something instead of what the product does. In my experience, these missing NFRs come from your Quality System Management (QMS) and software development life cycle (SDLC) processes. Here are some examples:

- No dynamic memory allocation is allowed—This is a policy or best practice imposed by your C/C++ coding standard. And the need for a coding standard is dictated by the QMS team. A no-dynamic-memory requirement can also be an explicit requirement. For example, it could be a risk control measure for memory fragmentation leading to uncontrolled execution delays.
- Facilitating automated unit tests—A source code tree organization that facilitates automated unit tests is an NFR dictated by a software development life cycle (SDLC) process.
- Project decisions—Sometimes stakeholders make decisions that impact the design of the software. For example:
    - "The software shall be written in C/C++." This is a command decision made by the software architect or software manager.
    - "The project shall have a functional simulator." Again, this is a decision made by the software architect so that the team can work smarter, not harder, while waiting for hardware to be delivered.

[2] See Chapter 2 for discussion of functional vs. nonfunctional requirements.

In a perfect world, all the requirements mentioned previously could be included in the formal requirements. In my experience, though, this just adds another level of noise to the formal requirements with a low return on investment. Unless you are in a regulated industry that imposes a high level of rigor, the recommendation is to leave out these NFRs or "missing requirements" and then document the gaps that will inevitably show up in requirements tracing. For the GM6000 example, when backward tracing the software architecture sections to requirements, there are ten sections that trace back to process-related NFRs and two sections that trace back to project decisions that are not captured in the formal requirements.

Requirements Tracing

All functional requirements should be forward traceable to one or more test cases in the verification test plan.[3] If there are test cases that can't be backward traced to requirements, there is disconnect between what was asked for and what is expected for the release. The disconnect needs to be reconciled, not just for the verification testing, but for the software implementation (and the other engineering disciplines) as well. *Reconciled* means that there are no orphans when forward tracing functional requirements to test cases or when backward tracing test cases to requirements. NFRs are not typically included in the verification tracing.

Requirements should also be traced to design outputs. Design outputs include the software architecture and detailed design documents as well as the source code. The importance or usefulness of tracing to design outputs is the same as it is for verification testing; it ensures that you are building everything that was asked for and not building stuff that is not needed. Unlike verification tracing, NFRs should be included in the tracing to design outputs.

[3] See Chapter 3, "Analysis," for a discussion of forward and backward requirements tracing.

Requirements tracing to design outputs is not the hard and fast rule that it is for verification testing. However, with some up-front planning, and following the SDLC process steps in order, tracing requirements to design outputs is a straightforward exercise.

Requirements tracing can be done manually, or it can be done using a requirement management tool (e.g., Doors or Jama). The advantage of using requirement management tools is that they are good at handling the many-to-many relationships that occur when tracing requirements and provide both forward and backward traces. The downside to these tools is their cost and learning curve. For the GM6000 project, I manually trace requirements to and from design outputs using a spreadsheet.

The following processes simplify the work needed to forward trace software-related requirements down to source code files:

- Software Detailed Design (SDD) document
    - When creating the SDD, ensure that each content section is backward traceable to a content section in the software architecture document.[4] See Chapter 6, "Foundation," for details on how to do this. When a section doesn't trace, stop and reconcile the disconnect.
    - If a content section has source code, explicitly document the source code directories. See Chapter 11, "Just-in-Time Detailed Design," for information on how to do this.

[4] A content section is any section that is not a housekeeping section. For example, the Introduction, Glossary, and Change Log are considered housekeeping sections.

- Software architecture (SWA) document
    - Backward trace all subsystem sections to at least one SRS, PRS, or MRS requirement. See Chapter 3, "Analysis," for details on how to do this. If a section doesn't trace, then stop and reconcile the disconnect.
    - Backward trace the remaining content sections (which are not subsystems) to at least one SRS, PRS, MRS, or a "missing requirement" (e.g., an implicit NFR). If a section doesn't trace, then stop and reconcile the disconnect. Note that when a section traces back to a "missing requirement," document the source of the missing requirement in the trace matrix.
    - Forward trace all software architecture content sections to content sections in the Software Detailed Design document. If a section doesn't trace, then
        i. When the SWA section back traces to a functional requirement, stop and reconcile the disconnect.
        ii. When the SWA section back traces to an NFR, a decision needs to be made if a forward trace to a Software Detailed Design section is needed.
        iii. When the SWA section back traces to a "missing requirement," then it is acceptable to not have a forward trace to a Software Detailed Design section.

After you have completed these steps, you have effectively forward traced the software architecture through detailed design to source code files. The remaining steps are to forward trace requirements (i.e., MRS, PRS, SRS) to sections in the software architecture document.

1. For all software-related MRS requirements, forward trace the MRS requirements to one of the following:
    a. One or more PRS requirements
    b. One or more SRS requirements
    c. One or more content sections in the software architecture document
2. For all software-related PRS requirements, forward trace the PRS requirements to either
    a. One or more SRS requirements
    b. One or more content sections in the software architecture document
3. For all SRS requirements, forward trace to one or more content sections in the software architecture document.

Summary

The formal requirements should avoid excessive design details or specifics whenever possible. The use of design statements should be used to bridge the gap between formal requirements and the details needed by the software team to design and implement the solutions.

Not everything needs to be a formal requirement. There are advantages to having fewer rather than more requirements, and this is where design statements are a great advantage. Nevertheless, the software testing effort must include the verification of formal requirements as well as design statements.

Forward tracing requirements from the MRS and PRS to both formal test cases and design outputs is a tool that ensures all requirements are implemented and verified. It also keeps the team focused on required deliverables and prevents the team from building or testing features that are not essential to the end product.

INPUTS

- All existing requirements as documented at the start of the construction phase
- System architecture (SA) document including ongoing updates to the document
- Software architecture document (SWA) including ongoing updates to the document
- Software Detailed Design (SDD) document including ongoing updates to the document
- Software verification test plan
- Feedback from analysis activities: failure, fault, security, etc.
- Feedback from design validation activities
- Feedback from voice-of-the-customer activities

OUTPUTS

- A final, approved MRS document
- A final, approved PRS document
- A final, approved SRS document
- A final, approved other engineering disciplines detailed requirements documents
- Documented design statements (if used)
- All functional software requirements are forward traceable to one or more test cases in the formal software verification test plan
- (Optional, but recommended) Forward requirements tracing from the MRS and PRS to
    - The detailed engineering discipline requirements (e.g., SRS)
    - Architecture and design documentation
    - Design outputs (e.g., source code)

CHAPTER 10

Tasks

This is the start of the construction phase, and this phase is where the bulk of the development work happens. Assuming that all of the software deliverables from the planning phase were completed, there should now be very few interdependencies when it comes to software work. That is, the simulator has removed dependencies on hardware availability, the UI has been decoupled from the application's business logic, the control algorithm has been decoupled from its inputs and outputs, etc. And since there are now minimal interdependencies, most of the software development can be done in parallel; you can have many developers working on the same project without getting bogged down in merge hell.

The software work, then—the work involved in producing working code—can be broken down into tasks, where each task has the following elements:

1. Requirements
2. A detailed design that has had a peer review
3. Source code and unit tests
4. A code review
5. A source code merge

A single task does not have to contain all elements; however, the order of the elements must be honored. For example, there are three types of tasks:

- Requirement task—A requirement task involves identifying, creating, and documenting requirements or design statements. A requirement task only has one element: #1 requirements.
- Design task—A design task involves creating, documenting, and reviewing the detailed design for a set of requirements. A design task has two elements: #1 requirements and #2 design.
- Coding task—A coding task involves writing, testing, and merging code. A coding task always has all five elements.

The tasks as discussed in this chapter are very narrow in scope, and they are limited primarily to generating source code. But there are many other tasks—pieces of work or project activities such as failure analysis, CI maintenance, bug fixing, bug tracking, integration testing, etc.—that still need to be planned, tracked, and executed for the project to be completed.

1) Requirements

The requirements for a task obviously scope the amount of work that has to be done. However, if there are no requirements, then stop design and coding activities and go pull the requirements from the appropriate stakeholders. Also remember that with respect to tasks, design statements are considered requirements.

If there are partial or incomplete requirements for the task, either split the current task into additional tasks: some where you do have sufficient requirements and others where the task is still undefined and ambiguous. Save the undefined and ambiguous tasks for later when they are better and more fully specified. Proceeding without clear requirements puts all of the downstream work at risk.

2) Detailed Design

The detailed design is the "how," and it fills in the gaps from the software architecture to the source code. The detailed design needs to be formally documented and reviewed. The documentation can range from a single sentence to pages of diagrams (e.g., class, sequence, state machine, etc.) and supporting text. For me, it is axiomatic that detailed design is always done before coding. From experience, every time I am tempted to cheat on this rule, nine times out of ten, it turns out to be a mistake. The task eventually required rework or took longer than it would have if I had just done the up-front design. Chapter 11, "Just-in-Time Detailed Design," describes how to perform detailed design.

3) Source Code and Unit Tests

Write your code and test it. Or if you are a proponent of Test-Driven Development (TDD),[1] write the unit tests and then write the code. Or use some combinations of both approaches. Regardless of what you chose, do not skip writing the unit test.

[1] Test-Driven Development: https://en.wikipedia.org/wiki/Test-driven_development

Unit tests, whether manual or automated, demonstrate that your code works and meets the requirements of the task. Without unit tests, the program manager, the scrum master, the stakeholders, and the entire company simply have to trust that their software developers are perfect and never make mistakes. Sarcasm aside, requiring unit tests actually makes the timeline shorter because there will be a fewer integration errors downstream. And the later in the process errors are found, the more costly they are in terms of time and effort to correct.

4) Code Review

All code should be reviewed before it is merged into the mainline, release, or stable branches. The Git Pull Request paradigm provides a straightforward and enforceable process for performing code reviews.

5) Merge

Never break the build.

The Definition of Done

While it may seem “over the top” for a team or organization to take the time to explicitly provide a definition of what constitutes “done” for a software task, not having a clear and common definition of “done” is the gateway to technical debt.

For example, consider this scenario. Jim is assigned to write an EEPROM driver. Jim completes the work and tests the driver, but the target hardware is not available, so he writes the code and tests it on an MCU evaluation board, which has a slightly older EEPROM chip (e.g., same family, different storage size). All the tests pass, so Jim claims the EEPROM driver is done. However, there is still more work needed to update (e.g., account for different pin assignments) and verify the driver after the target hardware is received. Consequently, a new task or card should be put into the project backlog to account for the additional work. If this isn't done, bad things happen when the target board is finally available and incorporated into testing. Jim is now fully booked with other tasks and is not "really available" to finish the EEPROM driver. A minor fire drill ensues over priorities for when and who will finish the EEPROM driver.

While this example is admittedly a bit contrived, most software tasks have nuances where there is still some amount of future work left when the code is merged. You will save a lot of time and angst if the development team (including the program manager) clearly defines what done means. My recommendation for a definition of done is as follows:

- All five elements of a coding task are completed.
- New tasks or cards have been created to address and resolve any discrepancies found. Here are some examples of situation where new tasks or cards should be created before signaling that the original task is done.
    - There is a requirement that calls for a user to acknowledge alarms. However, because the UI framework is only partially implemented, the acknowledgment part of the UI will not be implemented at this time. Tasks should be created for implementing the acknowledgment behavior.
    - The current UI wireframes do not support the display of more than two concurrent alarms, but it is possible to have three concurrent alarms. Tasks should be created to update the UI wireframes and to make the UI changes.

    - A reviewer observes that certain alarms need to be persistent. Tasks should be created to have a review of the overall system design and define behavior requirements for persistent alarms.

- Nothing more needs to be done to the merged code, unless a bug is uncovered or a requirement changes.

Task Granularity

What is the appropriate granularity of a task? Or, rather, how much time should the task take to complete? Generally, taking into account the five elements of a coding task discussed previously, a task should not take longer than a typical Agile sprint, or approximately two weeks. That said, tasks can be as short as a couple of hours or as long as several weeks. Table 10-1 shows some hypothetical examples of tasks for the GM6000 project.

***Table 10-1.** Examples of task types from the GM6000 project*

Task	Task Type	Description
SPI Driver	Coding	The requirement for the task is that the physical interface to the LCD display be an SPI bus. 1) The detailed design is created. The design is captured as one or more sections in the SDD. These sections formalize the driver structure, including how the SPI interface will be abstracted to be platform independent. 2) The new sections in the SDD are reviewed. 3) Before the code is written, a ticket is created, if there is not one already, and a corresponding branch in the repository is created for the code. 4) The code is written. 5) As a coding task requires a manual unit test to verify the SPI operation on a hardware platform, the manual unit test or tests are created and run. 6) After the tests pass, a pull request is generated, the code is reviewed, and any action items arising from the review are resolved. 7) The code is then merged, and if the CI build is successful, the code is merged into its parent branch.
Control Algorithm Definition	Requirement	There is only one formal requirement (PR-207) with respect to the heater control algorithm. The scope of the task is to define the actual control algorithm and document the algorithm as design statements. The documented algorithm design is then reviewed by the appropriate SMEs.

(*continued*)

Table 10-1. *(continued)*

Task	Task Type	Description
UI Design	Design	The scope of this task is to design the UI subsystem. It involves collecting numerous UI-specific formal requirements as well as creating (or collecting) the set of wireframe diagrams that comprise the UI. The design is captured as one or more sections in the SDD. These sections define the UI threading and event model, determine what is a "screen" with respect to code, define how unit testing will be done, identify (and design) common widgets that are needed (but are not provided by the graphic library), etc. These new sections are then reviewed.

Tasks, Tickets, and Agile

Tasks are schedulable entities. They can be represented by cards in your Agile tool, line items in a Microsoft Project schedule, or issues in GitHub. Each of these tasks should map to a repository branch where the code is checked in prior to being merged into the code base. The specifics of how tasks and tickets are mapped into your workflow are left up to you. However, here are some examples that I have encountered for coding tasks.

JIRA1–Parent card with child cards for task elements

- A story card is created for a task. (This story card is sometimes referred to as an epic.) Then the following subtask cards are created. The story card is completed after all subtasks cards are completed.
    - Requirements
    - Design

- Design Review
- Unit Test Implementation
- Code Implementation–This is the ticket card; the name or number of this card is used to create the branch in the repository. This task is completed when the pull request is merged

JIRA2–Single card with a JIRA checklist for task elements

- A single JIRA task or story card is created for the task. A checklist with the following items is added to the card. This is the ticket card; the name or number of this card is used to create the branch in the repository.
    - Requirements
    - Design
    - Design Review
    - Unit Test
    - Code
    - Pull Request and Code Review
    - Merge

Azure DevOps–Parent card with child "ticket" cards

- A product backlog item card is created for the task. This card is used to track the progress of the requirements and design aspects of the task, which may or may not involve creating individual child cards.
- A child task card is created to track the coding. This is the *ticket* card; the name or number of this card is used to create the branch in the repository. This task card is completed when the pull request is merged.

Summary

Software tasks are a tool and process for decomposing the software effort into small, well-defined units that can be managed using Agile tools or more traditional Gantt chart–based schedulers. Within a task, there is a waterfall process that starts with requirements, which is followed by detailed design, which is then followed by creating the unit tests and coding.

INPUTS

- All documented requirements including ongoing updates
- Software Development Plan (SDP) including ongoing updates
- System architecture (SA) document including ongoing updates
- Software architecture (SWA) document including ongoing updates
- Software Detailed Design (SDD) document including ongoing updates

OUTPUTS

- Working code
- Unit tests
- Design reviews
- Code reviews

- Potential updates to requirements, which may be in MRS, PRS, or SRS documents
- Potential updates to the system architecture (SA) document
- Potential updates to the software architecture (SWA) document
- Updates to the Software Detailed Design (SDD) document
- Updates to design statements when used

CHAPTER 11

Just-in-Time Detailed Design

While defining the overall software architecture is a waterfall process, the planning phase activities, or detailed software design, are made up of many individual activities that are done on a just-in-time basis. As work progresses during the construction phase, the Software Detailed Design is updated along the way. At the start of the construction phase, the state of the Software Detailed Design document should consist of at least

- An overall outline (see Chapter 6, "Foundation")
- The detailed design of the creation, startup, and shutdown of the application (see Chapter 7, "Building Applications with the Main Pattern")

This chapter is not about how to design components; it is about the design process. It is about doing the design in real time and documenting the design such that it does not go stale as the code is written. The purpose of the detailed design is to

- Figure stuff out—If you can't write a description on paper about what it is your coding, then you are not ready to code. The act of coding should be the work of instantiating a solution (i.e., the detailed design) in a programming language, not crafting a solution.

https://doi.org/10.1007/979-8-8688-0327-7_11

- Provide context that is not explicit in the source code
- Document decisions and any deviations from the software architecture
- Fill in missing requirements and design statements
- Provide a higher level of abstraction for reviews, training, and end-user documentation. No one should have to read your code and reverse-engineer it to figure out how it's designed.

As you put together the detailed design components, you can write for a knowledgeable reader who is

- Familiar with the product and its functionality
- Familiar and conversant with the software and system architecture documents
- Has some experience in embedded software development

Beyond that, there is no one-size-fits-all recipe for doing detailed design. The detailed design for a component can range from a couple of sentences to class diagrams or sequence and state machine diagrams. However, there are few common items that all detailed design components share:

- Every component that needs to be coded can be found in the Software Detailed Design document. This is important because the outline of the SDD plays a key role in tracing requirements (see Chapter 9, "Requirements Revisited"). In other words, if there does not seem to be an appropriate section within the SDD for the component, something is wrong; you need to revisit the purpose of the component, the requirements, the software architecture, etc., to resolve the disconnect. Do not proceed with the component's detailed design until the disconnect has been resolved.

- All components have a name and a file location. This sounds simple and obvious, but in my experience, this is a good early sanity check for determining the granularity of the component and its dependencies. For example, this step requires you to answer questions like is the component dependent on the MCU or an off-board IC? Is the component specific to the application or is it middleware?
- The simulation approach for each component is specified. For components that directly involve hardware, a decision needs to be made about how the component will be simulated or mocked when building the functional simulator. If a component will not be (or does not need to be) simulated, document that decision.
- No header file details are included in component designs. Avoid putting code fragments or header file details in the SDD because they will quickly be outdated after coding starts. For example, do not include class data members in your class diagram unless they are key to understanding the diagram. For code-level details (e.g., typedefs, enums, structs, etc.), use a documentation tool such as Doxygen that extracts comments from your code base as a companion document to the SDD.

- Duplicate information should not be included in design components. It is always a best practice to ensure there is only one source of information. External information can certainly be referenced but doesn't include details that are already captured in the software architecture document or elsewhere.

Examples

I can't tell you how to do detailed design. There is no one-size-fits-all recipe for doing it, and at the end of the day, it will always be a function of your experience, abilities, and domain knowledge. But I can give examples of what I do. Here are some examples from this design process that I used when creating the GM6000 project.

Subsystem Design

As defined in the software architecture of the GM6000, persistent storage is one of the top-level subsystems identified. It is responsible for "framework, data integrity checks, etc., for storing and retrieving data that is stored in local persistent storage." The snippets that follow illustrate how the subsystem design was documented in the SDD (Appendix M, "GM6000 Software Detailed Design (Final Draft)").

[SDD-24] Persistent Storage

For the most part, this section is simply a copy and paste from the SW architecture document, which of course contradicts the guideline "Duplicate information should not be included in design components." Nevertheless, I included it in the SDD because it provides nontrivial context without requiring the reader to switch to another document.

The Persistent Storage subsystem provides the interfaces, framework, data integrity checks, etc., for storing and retrieving data that is stored in persistent storage. The persistent storage paradigm is a RAM cached model. The RAM cached model has the following behaviors:

1. *On startup, persistent record entries are read from NVRAM, validated, and loaded into RAM.*
    a. *If the loaded data is invalid, then the associated RAM values are set to default/factory values, and the NVRAM storage is updated with new defaulted values.*
2. *The application updates the entries stored in RAM via an API. When an update request is made, the RAM value is updated, and then a background task is initiated to update the values stored in NVRAM.*

Each subsystem or component is responsible for defining the data to be stored as well as providing the initial/factory default values for the data.

[SDD-54] Frequency of Updates

Here is where the analysis and the decision that no wear leveling is required for the application are captured.

The GM6000 uses an off-board EEPROM for its persistent storage. The EEPROM is specified to have at least 1,000,000 write cycles of endurance. The DHC is required to operate for at least five years. This translates to at most 22.8 EEPROM write cycles per hour for the life of the product.

The highest frequency of application data updates to EEPROM is the metrics data, which is every 15 minutes, or 4 writes per hour. Since 4 writes per hour is significantly less than 22.8 writes per hour endurance limit, no wear leveling of the EEPROM will be implemented.

[SDD-55] Records

Here is the design for how persistent storage records are structured and how data integrity and power-failure use cases will be handled.

The application data is persistently stored using records. A record is the unit of atomic read/write operations from or to persistent storage. The CPL C++ class library's persistent storage framework is used. In addition, the library's Model Point records are used, where each record contains one or more model points.

All records are mirrored—two copies of the data are stored—to ensure that no data is lost if there is power failure during a write operation.

All of the records use a 32-bit CRC for detecting data corruption.

Instead of a single record, separate records are used to insulate the data against data corruption. For example, as the Metrics record is updated multiple times per hour, it has a higher probability for data corruption than the Personality record that is written once.

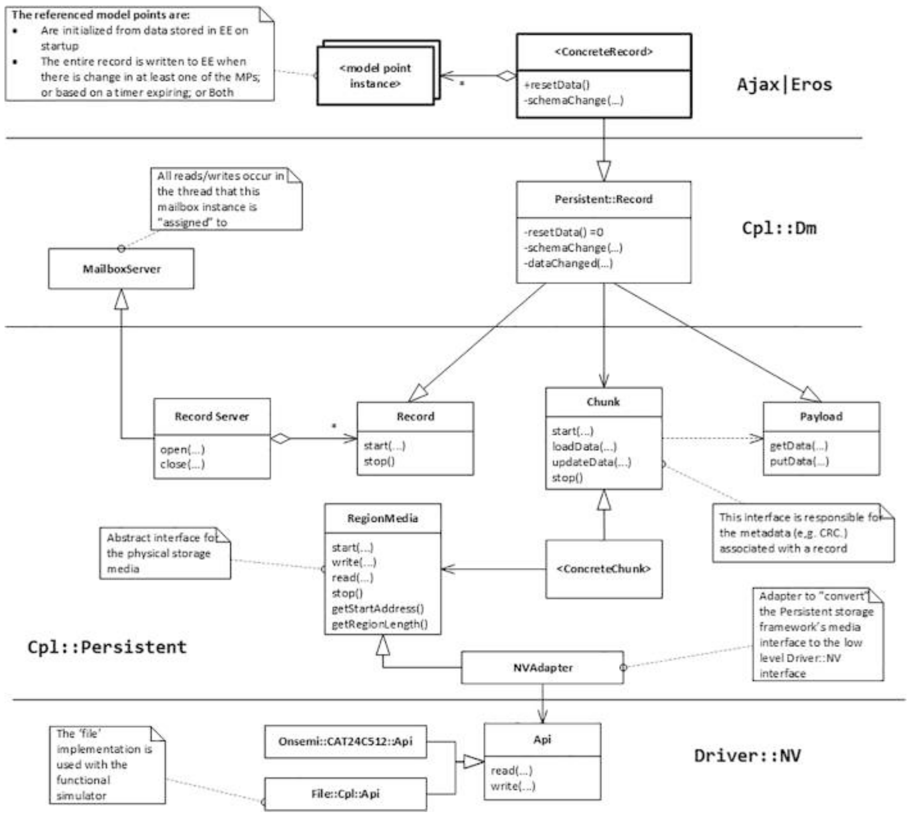
The application data is broken down into the following records:

- *User Settings*
- *Personality (contains customization to the algorithms, model and serial number, unique console password, etc.)*
- *Metrics*

[SDD-55] Records: Class Diagram

This snippet outlines the CPL library design that provides the persistent framework. Capturing the design of third-party code is not something that is frequently done; however, in this case, the decision was made to include it to illustrate exactly what application code is needed and to specify what objects need to be created at startup.

The following diagram illustrates the CPL framework and delineates what components the application provides. Only the classes with bolded outlines need to be implemented; all the other classes are provided by the CPL library.


The referenced model points are:
Are initialized from data stored in EE on startup
The entire record is written to EE when there is change in at least one of the MPs; or based on a timer expiring; or Both
<model point instance>
<ConcreteRecord>
+resetData()
-schemaChange(...)
Ajax|Eros
All reads/writes occur in the thread that this mailbox instance is "assigned" to
MailboxServer
Persistent::Record
-resetData() =0
-schemaChange(...)
-dataChanged(...)
Cpl::Dm
Record Server
open(...)
close(...)
Record
start(...)
stop()
Chunk
start(...)
loadData(...)
updateData(...)
stop()
Payload
getData(...)
putData(...)
This interface is responsible for the metadata (e.g. CRC.) associated with a record
Abstract interface for the physical storage media
RegionMedia
start(...)
write(...)
read(...)
stop()
getStartAddress()
getRegionLength()
<ConcreteChunk>
Adapter to "convert" the Persistent storage framework's media interface to the low level Driver::NV interface
Cpl::Persistent
NVAdapter
The 'file' implementation is used with the functional simulator
Onsemi::CAT24C512::Api
File::Cpl::Api
Api
read(...)
write(...)
Driver::NV

[SDD-55] Records: Location

The content of records is very application specific. Here is how I documented the decision to locate them inside the application `Main/` directories.

The concrete Record instances are defined per project variant (e.g., Ajax vs. Eros), and the source code files are located at

`src/Ajax/Main`
`src/Eros/Main`

[SDD-55] Records: Simulator

As the simulator also needs to have working persistent storage, I decided to leverage the existing CPL support to use the local host's file system as the storage media.

The hardware targets have an off-board I2C-based EEPROM IC. The CPL class library `Driver::NV::Onsemi::CAT24C512::Api` is used to read and write to the EEPROM.

The functional simulator uses the provided CPL library `Driver::NV::File::Cpl::Api` class, which uses the local host's file system as the physical persistent storage.

[SDD-56] Memory Map

It's important to do the math to ensure that everything will fit in the EEPROM. This needs to be done early in the project in case a hardware change is needed.

The following table details the offset locations in the EEPROM of the various persistently stored records and data.

Record/Data	Region Start	Region Length	Data Len	Chunk Overhd	Reserved/ Expansion
Personality-A	0	273	193	16	64
Personality-B	273	273	193	16	64
UserSettings-A	546	94	14	16	64
UserSettings-B	640	94	14	16	64

(*continued*)

Record/Data	*Region Start*	*Region Length*	*Data Len*	*Chunk Overhd*	*Reserved/ Expansion*
Metrics-A	*734*	*124*	*44*	*16*	*64*
Metrics-B	*858*	*124*	*44*	*16*	*64*
Log Entries	*982*	*40704*	*40704*	*0*	*0*
LogIndex-A	*41686*	*60*	*12*	*16*	*32*
LogIndex-B	*41746*	*60*	*12*	*16*	*32*

Allocated	*41806*	*40.8K*
Capacity	*65536*	*64.0K*
Available	*23730*	*23.2K*

I2C Driver Design

The GM6000 has an off-board EEPROM with an I2C interface.

[SDD-53] I2C Driver

This is an example of how only a few sentences are required for the design since the driver is provided by third-party code. The design details also address how the simulator is not supported, and it provides the location of the source code files.

The `Driver::I2C` abstract interface supplied by the CPL C++ class library will be used for the I2C bus driver. The I2C driver is used to communicate with the following device(s):

- *I2C EEPROM, address 0x50*

The concrete driver for the STM32 microcontroller family uses the ST HAL I2C interfaces for the underlying I2C bus implementation.

For the functional simulator, the EEPROM functionality is simulated at a higher layer (i.e., the `Cpl::Persistent::RegionMedia` layer).

The abstract I2C driver interface is located at

`src/Driver/I2C`

The target specific I2C implementation is located at

`src/Driver/I2C/STM32`

Button Driver Design

The initial release of the GM6000 is required to have a graphic display and four discrete momentary buttons for the UX.

[SDD-36] Button Driver

As was the case with the I2C driver, not much is needed for detailed design of the button driver since the driver is supplied by third-party code. However, because the design of the third-party button driver deviates from the GM6000 software architecture, the deviation is documented along with why it is considered acceptable.

The `Driver::Button::PolledDebounced` driver supplied by the CPL C++ class library will be used for the display's board discrete input buttons. The driver will execute in the driver thread.

The debounce sampling time is 100 Hz and requires two consecutive samples to match to declare a new button state.

The existing button driver only requires the low-level HAL interface to be implemented for the target platform.

For the functional simulator, the buttons on the display board are simulated using the `TPipe` implementation of the `Driver::PicoDisplay` driver.

The driver is located at

`src/Driver/Button`

The target-specific implementation is located at

`src/Driver/Button/STM32`

Note: The driver deviates from the SWA recommended implementation in that it does not use interrupts to detect the initial button edge. The proposed design meets the SWA requirements of debouncing a button within 50 msec (worse case for the current design is 30 msec), and the continual polling is not overly burdensome for the application. The tradeoff was deemed acceptable because of the simplicity of leveraging existing code instead of writing a more complex driver from scratch.

Fuzzy Logic Controller Design

The GM6000 uses a fuzzy logic controller (FLC) for the core of the heating algorithms. The FLC is documented under the "Heating" subsystem in the SDD.

[SDD-34] Fuzzy Logic Controller

This is an example of where the detailed design of an algorithm was done by a subject matter expert (SME), and the algorithm design is captured outside of the SDD. What the SDD needs is a reference to the external document and the software context for implementing the algorithm.

A fuzzy logic controller (FLC) is used to calculate updates to the requested output capacity. It is the responsibility of the control algorithm to call the FLC on a periodic basis. The FLC takes current temperature and temperature setpoint as inputs and generates a delta-change-to-capacity output value every time it is called.

The FLC is implemented per the SWA-1330 GM6000 Fuzzy Logic Temperature Control document. The implementation has the following features:

- *An algorithm supervisor is required to call the FLC and provide the current temperature and heating setpoint on a fixed, periodic basis.*
- *The FLC does not assume any particular unit of measure for temperature and setpoints. It is the responsibility of the algorithm supervisor to ensure consistency of input/output values with respect to units of measure.*
- *The geometry of the membership functions cannot be changed. However, there is a Y axis scaling parameter that is runtime configurable.*
- *All math operations are performed using integer math.*
- *Clients provide a scaling factor that is used during the defuzzification process to compensate for lack of floating arithmetic.*
- *The algorithm supports a single input membership function where the sets are defined by triangles.*

- *The algorithm supports a single centroid output membership function whose output sets have configurable weights.*
- *The algorithm has fixed predefined inference rules.*
- *The algorithm's configurable parameters are provided via a Model Point.*

The source code files are located at `src/Ajax/Heating/Flc`.

Graphics Library

The Graphics Library is a top-level subsystem identified in the software architecture document. The software architecture calls out the Graphics Library subsystem as third-party code.

[SDD-19] Graphics Library

This is an example of documenting a design decision. Specifically it is a decision that is less than optimal, and it should be re-evaluated in future releases.

The SWA requires that the Graphics Library be platform independent and run without an RTOS. There are several commercial libraries that meet these requirements. The decision was made to use the Pimoroni Graphics Library for the following reasons:

- *It is free. The library is open source with an MIT license.*
- *The CPL C++ class library provides a platform-independent port of the Pimoroni Graphics Library.*
- *The Pimoroni library comes with drivers for the ST7789 LCD controller and an RGB LED.*

Note: This decision should be revisited for future GM6000 derivatives that have a different display or a more sophisticated UI. The Pimoroni library has the following limitations that do not recommend it as a "foundational" component:

- *No support for Window management*
- *Requires C++14 (or newer) and uses C++ features (e.g., the Standard Template Library) that use dynamic memory*
- *Larger than desired footprint because it pulls in the entire standard C++ library code*

The library is located at

```
xsrc/pimoroni
```

Screen Manager Design

The Screen Manager is part of the UI subsystem.

[SDD-44] Screen Manager

This snippet details what the navigation model is and how the application is responsible for defining and changing the home screen.

The UI consists of a set of screens. At any given time, only one screen is active. Each screen is responsible for displaying content and reacting to UI events (e.g., button presses) or model point change notifications from the application.

Navigation between screens uses a stack model. Transitioning to a new screen pushes the current screen onto the navigation stack and then makes the new screen the active screen. Navigating backward is done by popping one or more screens from the navigation stack or clearing the entire navigation stack and making the home screen the active screen.

Which screen instance is the home screen is determined by the contents of a model point instance. This allows the application to change what the home screen is based on the overall state of the application.

There are three special screens that do not follow this paradigm. They are the following:

- *The splash screen, which is displayed when the system is starting up*
- *The shut-down screen, which is displayed when the system is shutting down*
- *The UI Halted screen, which is displayed when an error occurs that prevents continued operation*

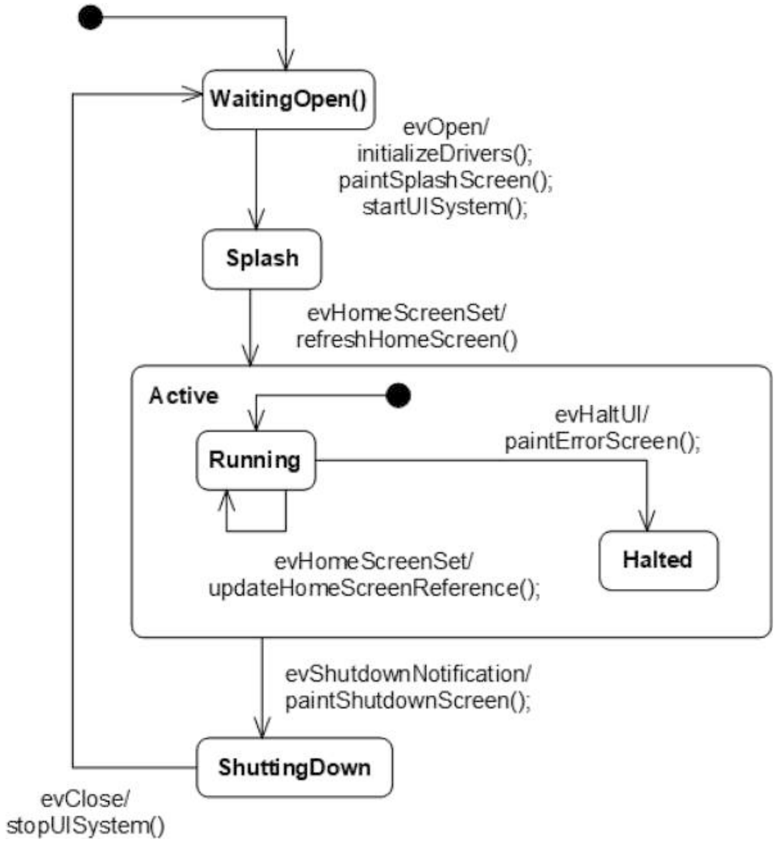
[SDD-44] Screen Manager: State Machine

A State Machine diagram is used to model the dynamic nature of the splash, home, halt, and shutdown screens.

*The following state machine diagram shows the life cycle of the Screen Manager. **Note**: The diagram describes the behavior; it is not intended to be an implementation.*

The Screen Manager should be opened as soon as possible during the startup sequence so the Splash screen is displayed instead of a blank or unknown screen contents.

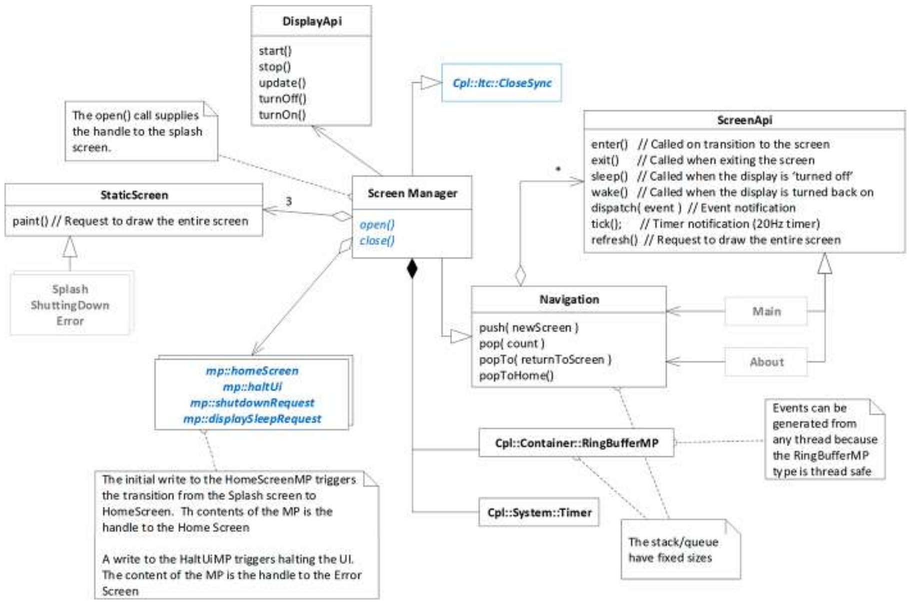
During an orderly shutdown, the application should trigger the "UI-shutdown-Request" as the first step in the shutdown sequence and then close the Screen Manager as late as possible in the shutdown sequence.



[SDD-44] Screen Manager Class Diagram

The design complexity of the Screen Manager necessitates a class diagram.

The following class diagram identifies the classes and functionality of the Screen Manager. The methods in italics are the public interfaces exposed to the application. Classes in gray/dashed outline are not part of the Screen Manager namespace.


DisplayApi
start()
stop()
update()
turnOff()
turnOn()
Cpl::Itc::CloseSync
The open() call supplies the handle to the splash screen.
ScreenApi
enter() // Called on transition to the screen
exit() // Called when exiting the screen
sleep() // Called when the display is 'turned off'
wake() // Called when the display is turned back on
dispatch(event) // Event notification
tick(); // Timer notification (20Hz timer)
refresh() // Request to draw the entire screen
StaticScreen
paint() // Request to draw the entire screen
3
Screen Manager
open()
close()
*
Splash
ShuttingDown
Error
Navigation
push(newScreen)
pop(count)
popTo(returnToScreen)
popToHome()
Main
About
mp::homeScreen
mp::haltUi
mp::shutdownRequest
mp::displaySleepRequest
Events can be generated from any thread because the RingBufferMP type is thread safe
Cpl::Container::RingBufferMP
The initial write to the HomeScreenMP triggers the transition from the Splash screen to HomeScreen. Th contents of the MP is the handle to the Home Screen
A write to the HaltUiMP triggers halting the UI. The content of the MP is the handle to the Error Screen
Cpl::System::Timer
The stack/queue have fixed sizes

[SDD-44] Screen Manager: Graphics Library

The Screen Manager is platform and hardware independent. No design considerations were needed with respect to the functional simulator, so a simple statement suffices for the design.

The Screen Manager is independent of the graphics library that is used to draw and update the physical display.

[SDD-44] Screen Manager: Location

This statement identifies the Screen Manager's namespace and source code location.

The UI events are application specific. The Screen Manager uses the LHeader pattern for the Event type to decouple itself from the application.

The Screen Manager namespace is `Ajax::ScreenMgr`. The source code is located at

`src/Ajax/ScreenMgr`

Design Reviews

Always perform design reviews and always perform them before coding the component. That is, there should be a design review for each task (see Chapter 10, "Tasks"), and you shouldn't start coding until the design review has been completed. In my experience, design reviews have a bigger return on investment (for the effort spent) than code reviews. The perennial problem with code reviews is that they miss the forest for trees. That is, code reviews tend to focus on the syntax of the code and don't really look at the semantics or the overall design of the software. Code reviews can catch implementation errors, but they don't often expose design flaws. And design flaws are much harder and more expensive to fix than implementation errors.

Do not skip the design review step when the design content is just a few sentences or paragraphs. Holding design reviews is a great early feedback loop between the developers and the software lead.

Review Artifacts

Design reviews can be very informal. For example, the software lead reviews the updated SDD and gives real-time feedback to the developer. However, depending on your QMS or regulatory environment, you may be required to have review artifacts (i.e., a paper trail) that evidence that reviews were held and that action items were identified and resolved. When formal review artifacts are required, it is tempting to hold one or two design reviews at the end of projects. Don't do this. This is analogous to having only a single code review of all the code after you have released the software. Reviews of any kind only add value when performed in real time before coding begins. And, of course, action items coming out of the review should be resolved before coding begins as well.

Here is a description of a lightweight design review process that generates tangible review artifacts. The process is done offline. That is, the reviewers independently review the SDD and then provide written feedback. The process assumes that the Software Detailed Design (SDD) is some sort of shared document (e.g., Google Doc or Microsoft Word).

1. The author, or person responsible for developing the code, updates the SDD document with the design content.

2. The author clears the change history in the document. That is, the author "accepts all changes" and then reenables change tracking.

3. The author sends or publishes the SDD to the reviewers via a shared document link identifying what section or sections should be reviewed. This notice should also contain a timeframe for completing the review.

4. The reviewers independently review the document. Feedback is provided in the shared document. Reviewers should be empowered to correct obvious typos.

5. The reviewers notify the author when they have completed their review.

6. The author reviews the feedback and makes updates as appropriate. All review comments must be addressed by the author, even if the comment is just "no change." The author then notifies the individual reviewers that their comments have been addressed. If the reviewer is satisfied with the resolution, the reviewer marks the comment as resolved.

7. Repeat steps 4–6 until all review comments have been marked as resolved. Marking comments as resolved is absolutely required.

8. After the review has been completed, make a copy of the current SDD and store it in a design review directory. This snapshot copy of the SDD constitutes the review artifact. When archiving the SDD snapshot, it is helpful to identify when the review occurred and what sections were reviewed. This can be as simple as encoding the information in the file name of the archive document.

I have used the aforementioned process on numerous projects, and it works well, but it is not perfect. For example, sometimes the design is sufficiently complex that an in-person meeting is required. Sometimes the reviewers and the author can't agree on how to resolve action items. When these situations occur, remember that the goal is to ensure that the proposed design satisfies the problem statement. It shouldn't be about "I would have solved the problem this way."

Summary

The software detail design process is about decoupling problem-solving from the act of writing source code. When detailed design creates solutions, then "coding" becomes a translation activity of transforming the design into source code. The detailed design process should include the following:

- Design first; then code and test.
- Create an entry for all components in the SDD—not just the complicated ones. This is a necessary aspect for tracing requirements to design elements and source code (see Chapter 9, "Requirements Revisited").
- At minimum view documenting your design as "rubber duck designing",[1] in that by explaining a problem to someone else—or by writing it down—you will often find edge cases, inconsistent logic, missing dependencies, etc.
- Incrementally add to the detailed design in a just-in-time manner. Avoid "big-bang" design efforts.
- Review each individual design increment in real time before coding begins.

INPUTS

- All documented requirements including ongoing updates
- Software Development Plan including ongoing updates
- System architecture document including ongoing updates
- Software architecture document including ongoing updates
- *Software Detailed Design document* including ongoing updates

[1] https://en.wikipedia.org/wiki/Rubber_duck_debugging

OUTPUTS

- Potential updates to the MRS, PRS, and SRS documents
- Potential updates to the system architecture document
- Potential updates to the software architecture document
- Updates and reviews to the design statements (if used)
- Updates and reviews to the Software Detailed Design document

CHAPTER 12

Coding, Unit Tests, and Pull Requests

The source code and unit tests are the most visible outputs of the construction phase. However, this chapter is not about how to write source code; rather, it is about the process for writing code, unit testing it, and then merging it into a stable branch. The key thing about this process is that, just like the Software Detailed Design activities, the coding and unit tests are done on a just-in-time basis.

If you followed the steps in Chapter 10, "Tasks," you have already completed these activities:

- A ticket has been created that calls for the source module to be written.
- Requirements for the to-be-written source code have been identified.
- The Software Detailed Design has been created, documented, and reviewed.

https://doi.org/10.1007/979-8-8688-0327-7_12

You can now begin the coding and unit testing process. This involves the following:

1. Creating a short-lived branch in the repository for the source code. The branch should have the same name as the ticket.
2. Writing the source code.
3. Writing the unit test code. This step in the process doesn't change whether you are writing platform-dependent or platform-independent code. In either case, unit tests must be written for each component. However, for hardware-specific components, you will need to build a stand-alone test image that you can manually exercise the unit tests against.
4. Creating the unit test project.
5. Iterating through multiple executions of the unit test until all the uncovered bugs are fixed and the desired test coverage has been achieved.

Don't cheat on this process. In my experience, every time I rationalized it was okay to skip or shortcut steps in this process, it always came back to haunt me. I thought I could save myself time, but in the end, it always ended up taking longer than if I had just followed my own rules.

The order that you perform the steps in is entirely up to you. For example, the traditional approach would be to perform the steps in the order listed. But if you're on the Test-Driven Development (TDD) side of the spectrum, the unit test project is created first, then the unit test code is written, and then the code is written. This would then be followed by iterations where the implementation and the test coverage are extended.

My personal approach is somewhere in the middle. That is, I typically write the shell or outline of the code, and then I create the unit test project and write minimum unit test code to make sure that the code I have written compiles and executes. Then I flesh out the remaining functionality incrementally: I iterate through writing code, adding unit test code, and executing the unit tests. In the end, the best suggestion I can make is: test early, test often. Avoid a big-bang approach of writing all the code and then executing all the unit tests.

Check-In Strategies

After the implementation is completed, and the desired test coverage has been achieved, the next step is to check everything into the repository. One advantage of doing the work on a branch is that the developer can check in their work in progress without impacting others on the team. I highly recommend checking in (and pushing) source code multiple times a day. This provides the advantage of being able to revert or back out changes if your latest work goes south. It also ensures that your code is stored on more than one hard drive.

Pull Requests

Use pull requests (PRs) to merge the source on the temporary branch to its parent branch (e.g., `develop` or `main`). Other source control tools provide similar functionality, but for this discussion, my examples will be GitHub specific.[1] The pull request mechanism has the following advantages:

[1] If you are not using Git, you may have to augment your SCM's merge functionality with external processes for code reviews and CI integration.

- It identifies any merge conflicts. That is, a PR cannot be completed until all merge conflicts have been resolved.
- It makes the code available to other developers to review and provide feedback in a collaborative environment. The feedback and subsequent discussions are captured and persistently stored, providing artifacts and an audit trail of the code review.
- It notifies code reviewers when the PR is created.
- It allows code updates to be checked in and added to the PR.
- It triggers a continuous integration (CI) build of the submitted code, which includes executing all unit tests (not just the newly submitted tests).
- It enforces approvals and other checks. For example, it can ensure that code is not merged in the parent branch unless the CI build is successful or that a different person approves or signs off on a code review before the code is merged.

The recommended minimum process for a pull request is as follows:

- Require at least one reviewer. If required, specify who that reviewer should be.
- Require that all review comments be successfully resolved before the PR can be merged.
- Require that the CI build has successfully completed (for the latest commit) before the PR can be merged.
- Do not allow any merge conflicts. (The Git server enforce this.)

After the PR has been merged, the ticket can be closed or moved to a ready-for-test-team-validation state depending on your workflow process.

Granularity

I strongly recommend the best practice of implementing new components in isolation without integrating the component into the application. That is, I recommend that you first go through the coding process—steps 1 through 5—and then create a second ticket to integrate the new component into the actual application or applications. Things are simpler if the scope of the changes is restricted to the integration changes only. Additionally, these are the following benefits:

- It reduces merge conflicts.
- The integration can be deferred until additional required components have been completed. For example, integrating a I2C driver into the application is not required until there is a consumer (e.g., an EEPROM driver) of the I2C driver.
- It allows you to address the multiple permutations inherent in the Main pattern (e.g., common creation, startup, and shutdown).

Examples

The sections that follow provide examples from the GM6000 project using the coding and unit testing process.

I2C Driver

The I2C driver is needed for serial communications to the off-board EEPROM IC used for the persistent storage in the GM6000 application. The class diagram in Figure 12-1 illustrates that there are different implementations for each target platform.

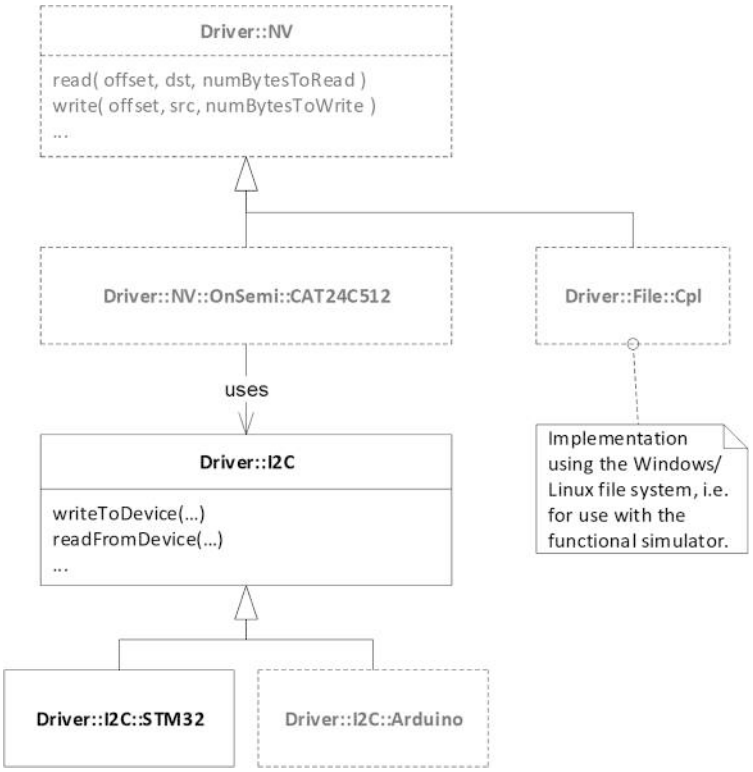

Driver::NV
read(offset, dst, numBytesToRead)
write(offset, src, numBytesToWrite)
...
Driver::NV::OnSemi::CAT24C512
Driver::File::Cpl
uses
Implementation using the Windows/ Linux file system, i.e. for use with the functional simulator.
Driver::I2C
writeToDevice(...)
readFromDevice(...)
...
Driver::I2C::STM32
Driver::I2C::Arduino

Figure 12-1. *Class diagram for the I2C driver*

Before the coding and unit testing began, the following steps were completed:

- A ticket was created for the driver.
- The requirements were identified. In this case, the requirements were for the persistent storage of the user options and for the heating algorithm customizations where the target hardware has an off-board I2C EEPROM.

- The detailed design was completed and reviewed. For the I2C driver, there was minimal design because I made the decision to leverage the ST HAL interfaces (see Appendix M, "GM6000 Software Detailed Design (Final Draft)").

At this point, I was ready to start the process. Table 12-1 summarizes the work.

***Table 12-1.** Process and work summary for the I2C driver*

Step	Work
Branch	1. Create a branch off of `develop` for the work. The branch name should contain the ticket card number in its name.
	2. Create the `src/Driver/I2C/STM32` directory in the source code tree. The location of the driver was identified during the detailed design step.
Source Code	1. Implement the I2C driver.
	a. Edit the ST Cube MX file to enable and configure the I2C bus. The BSP for the target is located at `src/Bsp/Initech/alpha1.`
	b. Update the BSP initialization code to call the newly generated MX code for configuring and initializing the I2C bus.
	c. Implement the I2C driver API using the ST HAL APIs.
Unit Test	1. The CPL library contains an existing unit test for I2C drivers that can be used to verify the driver's operation. The unit test code is located at `src/Driver/I2C/_0test.`

(*continued*)

Table 12-1. (*continued*)

Step	Work
Test Project	1. Create the unit test project and build scripts that are specific to target hardware. The new unit test project is located under the existing `tests/Driver` directory tree at `tests/Driver/I2C/_0test/master-eeprom/NUCLEO-F413ZH-alpah1/windws/gcc-arm.` a. Typically, I clone a similar test project directory to start. For the I2C driver's test project, I cloned the `tests/Driver/DIO/_0test/out_pwm/NUCLEO-F413ZH-alpah1/windows/gcc-arm`. Then I modified the copied files to create a I2C unit test using these steps: • Update the `libdirs.b` files to include the appropriate I2C directories and to remove the directories specific to the PWM output driver. • Modify `main.cpp` to call the top-level test function for the driver's unit test. The file was also modified to create a dedicated thread for the unit test to execute in. 2. Compile, link, and download the unit tests. Then verify that the test code passes. Iterate to fix any bug found or to extend the test coverage.

(*continued*)

Table 12-1. (*continued*)

Step	Work
Pull Request	1. Run the `top/run_doxgyen.py` script to verify that there are no Doxygen errors in the new file. This step is needed because the CI builds will fail when there are Doxygen errors present in the code base.
	2. Commit and push the source code.
	3. Generate a PR for the branch. The PR triggers a CI build.
	a. Notify the code reviewers that the code is ready for review. This is done automatically by the Git server when the PR owner selects or assigns reviewers as part of creating the PR.
	4. If there are CI build failures, commit and push code fixes to the PR (which triggers a new CI build).
	5. Resolve all code review comments and action items. And again, any changes you make to files in the PR trigger a new CI build.
	6. After all review comments have been resolved, and the last pushed commit has built successfully, the branch can be merged to `develop`. The merge will trigger a CI build for the `develop` branch.
	7. Delete the PR branch as it is no longer needed.

At this point, the I2C driver exists and has been verified, but it has not been integrated into the actual GM6000 application (i.e., Ajax or Eros). Additional tickets or tasks are needed to integrate the driver into the Application build.

The I2C driver is intermediate driver in the GM6000 architecture in that it is used by the `Driver::NV::Onsemi::CAT24C512` driver. In turn, the NV driver is used by the `Cpl::Dm` persistent storage framework for reading and writing persistent application records. The remaining tasks would be the following:

1. Create the STM32 specific unit test project for the CAT24C512 driver and verify that the unit test passes. The CAT24C512 driver exists and has a unit test, but there is no unit test for the STM32 platform. This unit test project works out the kinks in how to create the CAT24C512 driver for the STM32 platform.
2. Integrate the driver into the application. The integration is deferred until task 1 has been completed and there is at least one persistent application record defined (i.e., the first usage of the EEPROM).

Screen Manager

The UI consists of a set of screens. At any given time, only one screen is active. Each screen is responsible for displaying content and reacting to UI events (e.g., button presses) or model point change notifications from the application. The Screen Manager component is responsible for managing the navigation between screens as well as handling the special use cases such as the splash and UI halted screens.

The Screen Manager itself does not perform screen draw operation so it is independent of the Graphics library as well as being hardware independent. Before the coding and unit testing began, the following steps should have been completed:

- A ticket was created for the driver.
- The requirements were identified. In this case, it is the UI wireframes (see the EPC wiki).
- The detailed design was completed and reviewed. From the detailed design, the Screen Manager is responsible for

- Screen navigation
- Supporting dynamic home screen semantics
- Displaying a splash screen
- Displaying a terminal error screen

At this point, I was ready to start the process of creating the Screen Manager. Table 12-2 summarizes the work.

Table 12-2. *Process and work summary for the Screen Manager*

Step	Work
Branch	1. Create a GIT branch (off of `develop`) for the work. The branch name should contain the ticket number in its name.
	2. Create the `src/Ajax/ScreenMgr` and `src/Ajax/ScreenMgr/_0test` directories. The location of the Screen Manager was identified during the detailed design step.
Source Code	1. Implement the Screen Manager.
Unit Test	1. Implement the unit test. The unit test is an automated unit test since the Screen Manager has no direct hardware dependencies.

(*continued*)

Table 12-2. (*continued*)

Step	Work

Test Project	1. Create the unit test projects. Since it is an automated unit test, the test needs to be built with three different compilers to eliminate issues that could potentially occur when using the target cross compiler. These are the three compilers I use:
	• Microsoft Visual Studio compiler—Used because it provides the richest debugging environment
	• MinGW compiler—Used to generate code coverage metrics
	• GCC for Linux—Used to ensure the code is truly platform independent
	2. I recommend that you build and test with the compiler toolchain that is the easiest to debug with. After all tests pass, then build and verify them with the other compilers. The unit test project directories are as follows:
	`tests/Ajax/ScreenMgr/_0test/windows/vc12`
	`tests/Ajax/ScreenMgr/_0test/windows/mingw_w64`
	`tests/Ajax/ScreenMgr/_0test/linux/mingw_gcc`
	a. When creating the test project, typically I start by cloning a similar test project directory. For the Screen Manager test projects, I cloned the projects under the `tests/Ajax/Heating/Flc/_0test/` directory tree. I then modified the copied files for the Screen Manager unit test. This included the following changes per compiler variant:
	• Updating the `mytoolchain.py` file so that the `unit_test_objects` variable points to the Screen Manager driver's unit test directory. The `unit_test_objects` variable is needed when linking Catch2 test applications with NQBP2.

(*continued*)

Table 12-2. (*continued*)

Step	Work
	• Updating the `libdirs.b` files to include the appropriate Screen Manager directories and to remove the directories specific to the fuzzy logic controller (FLC) directories. 3. Compile, link, and execute the unit tests and verify that the test code passes with the targeted code coverage. 4. Iterate through the process to extend the test coverage and fix any bugs.
Pull Request	1. Run the `top/run_doxgyen.py` script to verify that there are no Doxygen errors in the new file that was added. This step is needed because the CI builds will fail if there are Doxygen errors present in the code base. 2. Commit and push the source code. 3. Generate a PR for the branch. The PR triggers a CI build. a. Notify the code reviewers that the code is ready for review. This is done automatically by the Git server when the PR author selects or assigns reviewers as part of creating the PR. 4. If there are CI build failures, commit and push code fixes to the PR (which will trigger new CI builds). 5. Resolve all code review comments and action items. And again, any changes you make to files in the PR triggers a new CI build. 6. After all review comments have been resolved and the last pushed commit has built successfully, the branch can be merged to `develop`. The merge will trigger a CI build for the `develop` branch. 7. Delete the PR branch as it is no longer needed.

At this point, the Screen Manager code exists and has been verified, but it has not been integrated into the actual GM6000 application (i.e., Ajax or Eros). A second ticket is needed to actually integrate the Screen Manager into the application build. For the GM6000 code, I created a second ticket that included the following:

- Integrating the `Driver::PicoDisplay` into the Ajax and Eros applications. It also included integrating the external "Display Simulator" application for the simulator builds of Ajax and Eros.
- Implementing the Splash and Shutdown screens for the Ajax and Eros applications.
- Creating a stub for a Home screen for the Ajax and Eros applications.

Summary

If you wait until after the Software Detailed Design has been completed before writing source code and unit tests, you effectively decouple problem-solving from the act of writing source code. The coding, unit test, and pull request process should include the following:

- Using short-lived branches to isolate the work-in-progress from the rest of the application. The branch should map to the project's ticket workflow. That is, the branch name contains the ticket name.
- Translating the detailed design into source code.
- Verifying the code works by running unit tests. Automate the unit tests wherever there are no direct hardware dependencies.

- Holding code reviews and documenting them for each ticket. You can use pull requests to enforce code reviews and approvals.
- Requiring all newly committed code to pass the CI build before merging.
- Only allowing trivial merges—merges where there are no merge conflicts—when updating the parent branch (e.g., `develop`).

INPUTS

- All documented requirements including ongoing updates
- Software Development Plan including ongoing updates
- System architecture document including ongoing updates
- Software architecture document including ongoing updates
- Software Detailed Design document including ongoing updates

OUTPUTS

- Source code with unit tests
- Pull requests, which include documented code reviews and successful CI builds
- Merged pull requests

CHAPTER 13

Integration Testing

Integration testing is where multiple components are combined into a single entity and tested. In theory, all of the components have been previously tested in isolation with unit tests, and the goal of the integration testing is to verify that the combined behavior meets the specified requirements. Integration testing also serves as incremental validation of the product as more and more functionality becomes available.

Integration testing is one of those things that is going to happen whether you explicitly plan for it or not. The scope and effort of the integration testing effort can range from small to large, and the testing artifacts that are generated can be formal or informal. Here are some examples of integration testing:

- Testing between system-level components. For example, testing between an IoT device and the cloud
- Testing the combined software being developed by multiple teams. For example, testing the software developed by one team to run on a main CPU board in conjunction with the real-time firmware running on an off-board MCU being developed by another team
- Testing the integration of in-house developed software with software developed by external service organizations

- Testing the integration of various control components into a single control application
- Testing new hardware components and software drivers

Integration testing is not a one-time event or effort. A typical project will have many integration testing efforts as components are continually integrated into increasingly larger components, culminating with the final testing of the entire product or system. At one end of the spectrum, your final project verification may be one giant integration test. At the other end, you may be working in a highly regulated environment that imposes strict integration steps as part of the software development life cycle process.

Regardless of the scope, complexity, or degree of formality, all integration testing follows the following steps:

1. Plan
    a. Define the goal for the test effort.
    b. Define a timeline for the test effort.
2. Develop the code
    a. This includes developing "code under test," potential test harnesses, test scripts, etc.
3. Define use cases or test cases
    a. This includes defining pass/fail criteria.
4. Execute the integration tests
5. Report results

How formal each of these steps is depends on the project, what is being integrated, and the company's Quality Management System (QMS) or regulatory requirements. In my experience, even when it is not dictated by regulatory requirements, some formality is needed if the project spans multiple teams, especially if the teams are geographically diverse or span organizations. The value of a documented integration test plan is that

- It clearly defines what is and isn't in scope.
- It sets a timeline.
- It sets pass/fail criteria. Or, said another way, it establishes when the integration testing effort ends.
- It specifies who is responsible for defining and executing the test cases.

As mentioned earlier, integration testing will happen organically if it is not explicitly formalized. A principal advantage of formalizing integration testing is that it avoids or reduces the amount drama that naturally occurs between teams and the amount of tension between developers and program managers. In addition, integration events are often viewed as key milestones that have visibility with stakeholders and senior leadership. Taking the time to properly set expectations beforehand more than pays for itself in not having to spend time on damage control.

For the GM6000 project, there is no example integration test plan because it is a single device with a single small development team—namely, me. I performed all of the integration testing either informally or as part of automated integration tests that I wrote to be executed during CI builds. However, a future release of the GM6000 calls for a wireless remote temperature sensor that would likely require more structured integration testing. In this case, I could envision at least three integration test efforts:

1. An RF integration test plan where both the base unit and the wireless sensor are able to pair, transfer data, and unpair. The goal of this test plan would be to evaluate and characterize the RF performance of the two devices.

2. A user integration test plan where the procedures for pairing and unpairing are performed. The goal of this test is to verify the procedures and evaluate them for ease of use with respect to the end user. Items such as RF range, temperature control, etc., would be outside the scope of this test effort as would testing the entire UI.

3. A temperature control test plan where the control algorithm uses the remote temperature sensor at various locations throughout the room. The goal of this test is to verify algorithm performance and to identify potential modifications to the control algorithm occasioned by temperature sensor placement.

Continuing with this hypothetical scenario, a minimal formal RF integration testing effort might look like this:

1. The software lead sends an email to the HW lead, System Engineer, Program Manager, etc., that defines the specific RF attributes to be tested: for example, signal to noise, range, bit-error rates, etc. The email also contains the target date for when the testing will begin and how long the test execution will take. While the software lead is responsible for the overall test effort, in this particular scenario, goal definition and test criteria are assigned as a joint effort with the hardware lead.

2. The software team meets to determine the minimum functionality that needs to be implemented on both devices to achieve the stated goals. Work tickets are created for functionality not already completed.

3. The software and hardware leads work together to select an engineer to write the test procedures and define the individual pass/fail requirements. They also identify who will execute the tests. Work tickets are created for the assigned work. The test procedures are defined in a spreadsheet as a high-level list of steps or use cases and include the pass/fail criteria. Note that the documented test procedures provide for a higher degree of confidence in the repeatability of the tests.

4. After the necessary software has been developed, a formal build[1] is created, and the test cases from step 3 are executed. The pass/fail results along with any notes and commentary are recorded in the spreadsheet. Any bugs found are triaged to determine if they need to be fixed in order to continue testing. Blocker bugs are fixed and retested.

Based on the current test results, the software and hardware leads determine if the goals of the testing have been reached. If yes, the software lead sends out an email with test results. Note that the testing goals can be met without all the test cases passing. For example, if the RF range tests fail, and a new antenna design is needed to resolve the issue, testing can be paused until new hardware is received, and then the integration test plan can be re-executed.

[1] A formal build is defined as a build of a "stable" branch (e.g., `develop` or `main`) performed on the CI build server where the source code has been tagged and labeled. It is imperative that all non-unit testing be performed on formal builds because the provenance of a formal build is known and labelled in the SCM repository (as opposed to a private build performed on a developer's computer).

Smoke Tests

Smoke or sanity tests are essentially integration tests that are continually executed. Depending on the development organization, smoke tests are defined and performed by the software test team, the development team, or both. In addition, smoke tests can be automated or manual. The automated tests can be executed as part of the CI build or run on demand. If the tests are manual, it is essential that the test cases and steps be sufficiently documented so that the actual test coverage is repeatable over time even when different engineers perform the testing.

One downside to smoke tests is that they can be easily broken as requirements and implementations evolve over time. This means that

- From a project planning perspective, there needs to be time and resources allocated for the care and feeding of the test cases.
- Do not implement the smoke test too early in the project when the code base is relatively immature. When to start implementing smoke testing will always be a project-specific decision. However, always start by testing functionality that is unlikely to change and keep the initial test coverage minimal. You can add more test coverage as the requirements and code mature.

Simulator

A functional simulator can be a no-cost platform for performing automated smoke tests that can be run as part of the CI builds. These automated simulator tests can be simple or complex. In my experience, the only limitation for simulator-based tests that run as part of the CI build is the amount of time they add to the overall CI build time.

The GM6000 project has an automated smoke test and a heating simulation test that run as part of the CI builds. These tests leverage the debug console and the Python `pexpect` library,[2] which provides control of interactive console applications. An additional Python library, RATT, is used as a thin wrapper on top of `pexpect`. See Appendix G, "RATT," for an in-depth discussion of the RATT library.

The automated smoke test runs in real time and performs only basic operations (i.e., only a very high-level verification that the build works). The smoke test scripts are located at the `src/Ajax/Main/_0test/` directory. The smoke test runs on the Windows simulator and is explicitly called out in the `build_all.bat` script (see Chapter 8, "Continuous Integration Builds").

The heating simulation test runs in simulated time and verifies the functional behavior of the heating algorithm that is made up of several individual components. The test does not validate the quality or effectiveness of the heating algorithm. As mentioned earlier, this test executes in simulated time, where the test cases advance time per algorithm tick (which is two seconds), allowing you to run several minutes of algorithm time in just a few seconds. The CPL library provides a simulated time feature that is transparent to the application code. The simulated time feature is enabled at link time by selecting the simulated time implementation for the underlying "timing" interfaces in the `Cpl::System` namespace. At run time, the `Cpl::TShell::Cmd::Tick` command is used to advance the simulated time. The heating simulator test scripts are located at `src/Ajax/Heating/Simulate/_0test`. The test executable is built and executed as an automated "unit test" (see Chapter 8, "Continuous Integration Builds").

[2] https://pypi.org/project/pexpect/

Summary

Integration testing performed by the software team occurs throughout a project. How formal or informal the integration testing should be depends on what is being integrated. Generally, a more formal integration testing effort is required when integrating components across teams. However, a minimum level of formality should be that the pass/fail criteria is written down.

Continual execution of integration tests, for example, smoke tests or sanity tests, provides an initial quality check of a build. Ideally, these tests would be incorporated in the CI build process.

INPUTS

- All documented requirements including ongoing updates
- Software Development Plan including ongoing updates
- System architecture document including ongoing updates
- Software architecture document including ongoing updates
- Software Detailed Design document including ongoing updates
- Source code

OUTPUTS

- Defined scope and pass/fail criteria for the testing:
    - (Optional) Documented test plan
    - (Optional) Written test cases
    - (Optional) Integration code
- Evaluation and decision regarding the value of automating the testing. If automated testing is determined to be worthwhile, this would be followed by the creation of an automated test suite for regression testing or CI builds.
- Test reports that document the results of the integration tests
- List of executed tests including the backlog tickets or bug cards created for each issue uncovered

CHAPTER 14

Board Support Package

The Board Support Package (BSP) is a layer of software that allows applications and operating systems to run on the hardware platform. Exactly what is included in a BSP depends on the hardware platform, the targeted operating system (if one is being used), and potential third-party packages that may be available for the hardware (e.g., graphics libraries). The BSP for a Raspberry PI running Linux is much more complex than a BSP for a stand-alone microcontroller. In fact, I have worked on numerous microcontroller-based projects that had no explicit concept of a BSP in their design. So while there is no one-size-fits-all definition or template for BSPs in the microcontroller hardware space, ideally a microcontroller BSP encapsulates the following:

- The compiler toolchain, as it relates to the specific MCU
- The MCU's datasheet
- The target board's schematic

Compiler Toolchain

The compiler toolchain is all of the glue and magic that has to be in place from when the MCU's reset vector executes up until the application's `main()` function is called. This includes items, for example:

1. The vector table—The list of function pointers that are called when a hardware interrupt occurs.

2. Minimal MCU and memory configuration—Depending on the MCU, a certain amount of clock configuration and memory access—both RAM and Flash—may be needed before the C/C++ runtime code can be executed.

3. The C/C++ runtime code—The code that executes before `main()` is called. This code is responsible for setting up the C/C++ environment (e.g., initializing the Data and BSS segments in RAM, invoking static C++ constructors, etc.).

4. The platform dependencies for the C standard library are satisfied. For example, when using the GCC compiler, there needs to be a board-specific implementation for low-level functions such as `__io_putchar()` and `__io_getchar()` that read and write characters to and from a UART. These functions are called by the C library's `stdio` functions.

5. Linker script—This script defines the memory map for the compiler toolchain and the MCU.

BSPs are dependent on a specific compiler toolchain as well as MCU and board schematic. Non-ANSI standard compiler features are typically used when implementing the C code for a BSP (e.g., linker script syntax, using GCC's `__weak` attribute, assembly code, etc.).

Encapsulating the Datasheet

Most MCU vendors provide header files that contain C data structures, functions, and macros to access individual registers and bits within the registers. Ideally, the BSP will further abstract the MCU datasheet and registers by providing functionality based on behavior, not just by hardware register name, for example, the work of configuring the TX and RX pins for a UART. Depending on the MCU, the following configuration is required:

- Configuring the RX as an input pin and selecting the appropriate "pull-up/pull-down/none" options for the pin
- Selecting the RX pin to be used for the UART (i.e., selecting one of many possible pin usages)
- Configuring the TX as an output pin and selecting a drive strength and drive type for the pin
- Selecting the TX pin to be used for the UART (i.e., selecting one of many possible pin usages)

An example of a BSP abstraction for the RX and TX management would be to have single function call that is passed the physical pin assignments for the RX and TX pins, and it would be responsible for knowing which registers, bits, and values (per the datasheet) need to be written to fully configure the specified RX and TX pins.

Another example of a BSP abstraction would be setting the baud rate for a UART. Typically, there are multiple registers (clock selection, clock dividers, match registers, etc.) that need to be configured when setting the baud rate. The BSP abstraction would provide a single function call that specifies a desired baud rate, and the BSP would take care of the details.

The goal of defining BSP abstractions is to only provide logical interfaces accessing MCU peripherals instead of dealing directly with MCU's hardware registers. BSP abstractions are not for defining a generalized abstract layer across multiple BSPs. Or said another way, the BSP interfaces should be focused on a specific MCU and PCB with no eye toward potential reuse with other BSPs. In my experience, MCUs and SDKs vary so widely it would be difficult to define common BSP-level abstractions across different MCUs, and it would have very low return on the investment for the work required.

Encapsulating the Board Schematic

Microcontroller projects typically run on custom hardware that is specific to a company's devices and products. The BSP's role is to abstract away the circuit-level details of the schematic. For example:

- Provide logical names (e.g., preprocessor symbols) for the MCU's physical pins
- Provide logical assert and de-assert semantics for physical pin signals (e.g., decouple the knowledge that to assert a Chip Select to an SPI device is a physical low signal)
- Configure the MCU's hardware peripherals per the schematic

BSPs in Practice

As previously mentioned, there is no one size fits all when it comes to microcontroller BSPs. There are two BSPs for the GM6000 project: one is for the ST NUCLEO-F413ZH evaluation board, and the second is for the Adafruit Grand Central M4 Express board. The two BSPs are structured completely differently, and the only thing they have in common is each BSP has `Api.h` and `Api.cpp` files. The BSP for the Nucleo board uses ST's HAL library and Cube MX IDE for low-level hardware functionality. The BSP for the Adafruit Grand Central board leverages the Arduino framework for abstracting the hardware. To illustrate the differences, the following are summaries of the file and directory structure of the two BSPs.

ST NUCLEO-F413ZH BSP

```
src/Bsp/Initech/alpha1/     // Root directory for the BSP
+--- Api.h                  // Public BSP interface
+--- Api.cpp                // Implementation for Api.h
+--- ...
+--- stdio.cpp              // C Library support
+--- syscalls.c             // C Library support
+--- ...
+--- mini_cpp.cpp           // C++ support when not linking against stdlibc++
+--- MX/                    // Root directory for the ST Cube MX tool
|    +--- MX.ioc            // MX project file
|    +--- startup_stm32f413xx.s   // Initializes the Data & BSS segments
|    +--- STM32F413ZHTx_FLASH.ld  // Linker script
|    +--- ...
|    \--- Core/             // Contains the MX tool's auto generated code.
                            // The dir contains the vector table, clock cfg, etc.
|
+--- console/               // Support for the CPL usage of the debug UART
|    +--- Ouptut.h          // Cpl::Io stream instance for the debug UART
|    \--- Ouptut.cpp        // Cpl::Io stream instance for the debug UART
+--- SeggerSysView/         // Run time support for Segger's SysView tool
\--- trace/                 // Support for CPL tracing
     \--- Ouptut.cpp        // Tracing using C libray stdio and Cpl::Io streams
```

Adafruit Grand Central M4 Express BSP

```
src/Bsp/Initech/alpha1-atmel/  // Root directory for the BSP
+--- Api.h                     // Public BSP interface
+--- Api.cpp                   // Implementation for Api.h
+--- cortex_handlers.c         // Vector table for the MCU
+--- main.cpp                  // Contains main(), calls Arduino's setup() & loop()
+--- sdfat.cpp                 // Support FAT32 file system on the SPI dataflash
+--- wiring.c                  // Low level hardware configuration
\--- FreeRTOS/                 // Source files for FreeRTOS
```

A majority of the Adafruit BSP functionality is performed by the Arduino framework located under the `arduino/hardware/samd/` directory in the EPC repository.

Structure

The structure for BSPs that I recommend is minimal because each BSP is conceptually unique since it is compiler, MCU, and board specific. This structure is a single `Api.h` header file that uses the LHeader pattern and exposes all of the BSP's public interfaces. An in-depth discussion of the LHeader pattern can be found in Appendix D, "LHeader and LConfig Patterns." However, here is a summary of how I used the LHeader pattern in conjunction with BSPs.

1. I created a single `Api.h` header file that is not owned by a specific BSP. This header file will ultimately provide access to a concrete BSP's public interfaces. The GM6000 architecture uses the file `src/Bsp/Api.h.`

   a. This file defines a minimal set of common functions that all BSPs should support. In this case, there is a `Bsp_Api_initialize()` function declared.

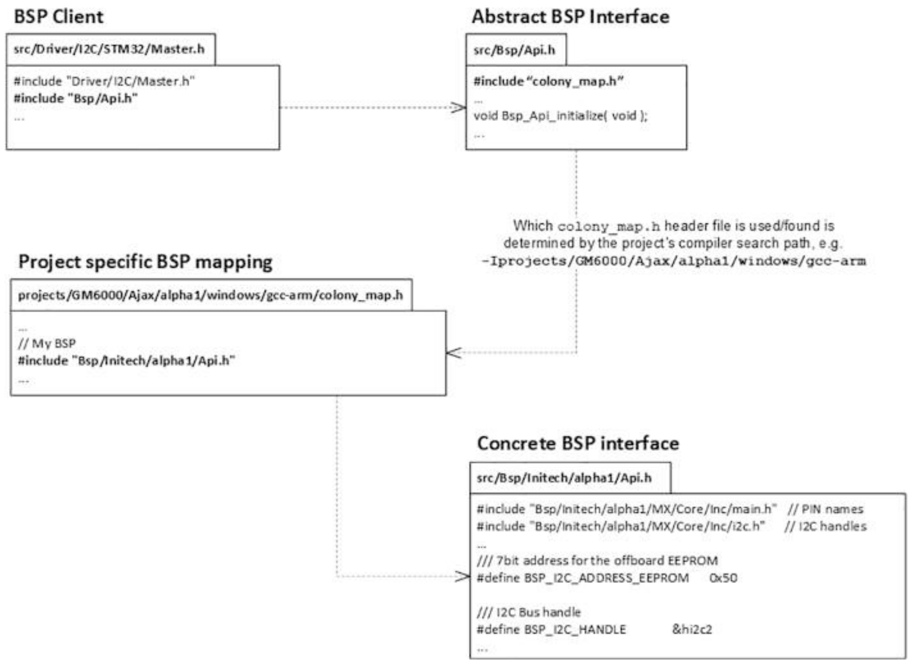
   b. This file contains a `#include "colony_map.h"` statement as dictated by the LHeader pattern. The actual header file that gets included at compile time for this statement is based on each individual project's set of header search paths specified in its make script. This allows individual projects to select a specific BSP.

   c. All symbols in the header file should be prefixed with `Bsp_` or `BSP_` to isolate the public BSP interfaces in the global compiler namespace. As an application has only one BSP, there is no need for naming conventions to distinguish between individual concrete BSPs.

2. I created a subdirectory under the `src/Bsp` directory for the new BSP. There are no recommended naming conventions for BSP directories; however, keep in mind that the BSPs are MCU, board, and compiler specific, and your naming convention should accommodate possible BSP variants.

   a. Create a single `Api.h` header file in the preceding directory. This header file contains everything that is needed to expose the public interfaces for the BSP. The public interfaces should include an implementation of the public interfaces defined in 1a.

Figure 14-1 illustrates how referencing a concrete BSP header file is deferred to when a specific project is compiled. Being able to defer which BSP is used is critical in that it allows drivers to be decoupled from a specific BSP. A driver may still be specific to an MCU or off-board IC, but not a BSP. For example, when migrating from using an evaluation board to the first in–house designed board, the existing driver source code does not have to be updated when changing to the in-house board.


BSP Client
Abstract BSP Interface
Which colony_map.h header file is used/found is determined by the project's compiler search path, e.g. -Iprojects/GM6000/Ajax/alpha1/windows/gcc-arm
Project specific BSP mapping
Concrete BSP interface

src/Driver/I2C/STM32/Master.h
```
#include "Driver/I2C/Master.h"
#include "Bsp/Api.h"
...
```

src/Bsp/Api.h
```
#include "colony_map.h"
...
void Bsp_Api_initialize( void );
...
```

projects/GM6000/Ajax/alpha1/windows/gcc-arm/colony_map.h
```
...
// My BSP
#include "Bsp/Initech/alpha1/Api.h"
...
```

src/Bsp/Initech/alpha1/Api.h
```
#include "Bsp/Initech/alpha1/MX/Core/Inc/main.h"   // PIN names
#include "Bsp/Initech/alpha1/MX/Core/Inc/i2c.h"    // I2C handles
...
/// 7bit address for the offboard EEPROM
#define BSP_I2C_ADDRESS_EEPROM     0x50

/// I2C Bus handle
#define BSP_I2C_HANDLE          &hi2c2
...
```

Figure 14-1. *LHeader deferred header include*

Dos and Don'ts

Here is my short list of dos and don'ts when creating BSPs.

- Build the BSP in a just-in-time manner. That is, only add functionality when there is active development that requires hardware support. For example, you don't need to implement I2C support in the BSP until someone is ready to work on a driver that utilizes an I2C bus.

- The initial BSP should contain support for the UART that is used for the debug console and the C library (a.k.a. `printf`).
- Do not include any C++ code in the BSP's public interface header file (i.e., in `Api.h`). This allows C modules to #include the BSP's header file and be compiled as a C file. This scenario usually occurs when compiling source code provided by the MCU vendor or third-party packages. Also it is okay to implement a BSP using C++. The only restriction is that the public interface declarations have to be fully C compatible.
- Create a unit test project to verify the basic operation on the hardware when creating a new BSP. Because the BSP deals with the C/C++ runtime code and C library support, the fact that the test project successfully compiles and links is not sufficient to claim victory. For example, if the vector table is not located at the proper memory address, the application will not start or the MCU will immediately crash on boot-up.
- BSPs are very concrete in nature so don't be bogged down with reuse across BSPs or BSP variants. It is okay for copy-paste-edit reuse across BSPs. Or said another way, there is typically very little to no business logic captured in a BSP that warrants the extra effort to make reusable BSP components.
- Keep the BSPs small since they are not reuse friendly. Typically, this means implementing drivers outside of the BSP whenever possible.

- BSPs are fairly static in nature. That is, after they are working, they require very little care and feeding. Doing a lot of refactoring or maintenance on existing BSP functionality is a potential indication that there are underlying architecture or design issues.

Bootloader

The discussion so far has omitted any discussion using a bootloader with the MCU. The reason is because designing and implementing a bootloader is outside the scope of this book. However, many microcontroller projects include a bootloader so that the firmware can be upgraded after the device has been deployed. Conceptually a bootloader does the following:

1. It executes when the reset vector is executed.
2. It makes a determination if there is new firmware to load.
    a. If yes, then the MCU application memory is programmed with the new firmware.
3. It verifies that the MCU's application memory contains a valid image.
    a. If yes, the control is transferred to the application; that is, the application's startup code is executed and `main()` is called.

While the aforementioned behavior is straightforward, the implementation details of a bootloader vary greatly. The good news is that bootloaders can be mostly encapsulated in a stand-alone executable that is separate from the BSP. This means the construction of the BSP can be done without having first implemented a bootloader. After the bootloader is in place, the BSP usually requires a few updates because the BSP is no longer the owner of the MCU's vector table. However, the changes are relatively isolated (e.g., an alternate linker script) and usually do not disrupt the existing BSP structure.

If your project requires a bootloader, it is strongly recommended that it is implemented and deployed early in the development cycle. This maximizes the runtime on the bootloader before the first release. This is especially critical if the bootloader itself is not upgradable after the product is deployed.

Summary

The goal of the Board Support Package is to encapsule the low-level details of the MCU hardware, the board schematic, and compiler hardware support into a single layer or component. The design of a BSP should decouple the concrete implementation from being directly referenced (i.e., #include statements) by the drivers and application code that consume the BSP's public interfaces. This allows the client source code to be independent of concrete BSPs. The decoupling of a BSP's public interfaces can be done by using the LHeader pattern.

INPUTS

- All documented requirements including ongoing updates
- Software Development Plan (SDP) including ongoing updates
- System architecture (SA) document including ongoing updates
- Software architecture (SWA) document including ongoing updates
- Software Detailed Design (SDD) document including ongoing updates

OUTPUTS

- Working BSP code
- BSP test projects
- Design reviews
- Code reviews
- Potential updates to requirements that may be in MRS, PRS, or SRS documents
- Potential updates to the system architecture (SWA) document
- Potential updates to the software architecture (SA) document
- Updates to the Software Detailed Design document
- Updates to design statements when used

CHAPTER 15

Drivers

This chapter is about how to design drivers that are decoupled from a specific hardware platform that can be reused on different microcontrollers. In this context, reuse means reuse across teams, departments, or your company. I am not advocating designing general-purpose drivers that work on any and all platforms; just design for the platforms you are actively using today.

Writing decouple drivers does not take more effort or time to implement than a traditional platform-specific driver. This is especially true once you have implemented a few decoupled drivers. The only extra effort needed is a mental shift in thinking about what functionality the driver needs to provide as opposed to getting bogged down in the low-level hardware details. I am not saying the hardware details don't matter; they do. But defining a driver in terms of a specific microcontroller register set only pushes hardware details into your application's business logic.

A decoupled driver requires the following:

- A public interface that is defined such that there are no direct hardware dependencies. That is, only the driver's services are defined.
- A layer or mechanism that decouples the driver implementation from a specific hardware platform, for example, a Hardware Abstraction Layer (HAL). Remember that the driver depends on the HAL declarations, not the implementation of the HAL.

https://doi.org/10.1007/979-8-8688-0327-7_15

Binding Times

A Hardware Abstraction Layer is created using late bindings. The general idea behind late binding time is that you want to wait as long as possible before binding data or functions to names. Here are the four types of name bindings. However, only the last three are considered late bindings.

- Source time—Source time bindings are made when you edit a source code file. This is reflected primarily in what your `#include` statements are and how they define data types, as well as numeric and string constants. As source code bindings are bindings that cannot be changed or undone later without editing the file, you want to minimize source time bindings in your HAL interfaces.
- Compile time—Compile-time bindings are late bindings that are made during the compilation stage. The primary mechanisms involved with compile-time bindings are the specification of preprocessor symbols when the compiler is invoked. Oftentimes this is done by setting the compiler's header file search paths. The *LHeader* and *LConfig* patterns (see Appendix D, "LHeader and LConfig Patterns") leverage the header search path mechanism to provide concrete definitions for preprocessor symbols that were declared without any definition provided.

- Link time—Link-time bindings are late bindings that are made during the link stage of the build process. The linker binds names to addresses or binds code with a specific function name. Link-time binding allows a developer to define a function (or set of functions) and then have multiple implementations for those functions. The selection of which implementation to use is specified in the project's build script. (This is why I advocate separate stand-alone build scripts for each variation of a project.)
- Run time—Run-time bindings are late bindings that occur when the application executes. An example of these types of bindings is the C++ polymorphic bindings that occur at run time when a virtual method is called.

Public Interface

Defining a public interface for a driver is a straightforward task. What makes it complicated is trying to do it without referencing any underlying hardware-specific data types. And interacting with the MCU's registers (and its SDK) always involves data types of some kind. For example, when configuring a Pulse Width Modulation (PWM) output signal using the ST HAL library for the STM32F4x, the MCU requires a pointer to a timer block (`TIM_HandleTypeDef`) and a channel number (of type `uint32_t`). When using the Arduino framework with the ATSAMD51 MCU, only a simple `int` is needed. So how do you define a handle that can be used to configure a PWM output signal that is platform independent?

One way to solve this is by using a forward declaration, for example:

```
typedef struct DriverDioPwmConfig_T* DriverDioPwmConfigPtr_T.
```

The compiler allows clients (or consumers) to pass around a pointer to a data structure (e.g., `DriverDioPwmConfigPtr_T`) without a concrete type definition because all pointers have the same known storage size. The concrete data type for `DriverDioPwmConfig_T` can then be defined in platform-specific code. The limitations of this approach are as follows:

- It requires memory management for how an instance is created using platform-specific code. For drivers, this is usually not an issue because the Board Support Package (BSP) typically provides the concrete instance.
- A pointer type is cumbersome when the underlying type is an integer or a C++ reference. For example, when using the Arduino framework, the concrete platform type for `DriverDioPwmConfig_T` is an `int`.

A better approach is to use the LHeader pattern, which uses compile-time bindings to provide a concrete and platform-specific data type for the PWM configuration type. The LHeader pattern eliminates the limitations incurred when using a forward reference. Unfortunately, this comes at the expense of some additional complexity. An in-depth discussion of the LHeader pattern can be found in Appendix D, “LHeader and LConfig Patterns”; however, here is a summary of how the LHeader pattern is used in conjunction with drivers for deferring hardware-specific data types.

1. In the driver’s public interface header file (e.g., `src/Driver/DIO/Pwm.h`), use a `#include` statement—without any path information—that references a header file that does not resolve to any header file in the baseline set of header search file paths. For example:

   ```
   #include "colony_map.h"          // Note: no path specified
   ```

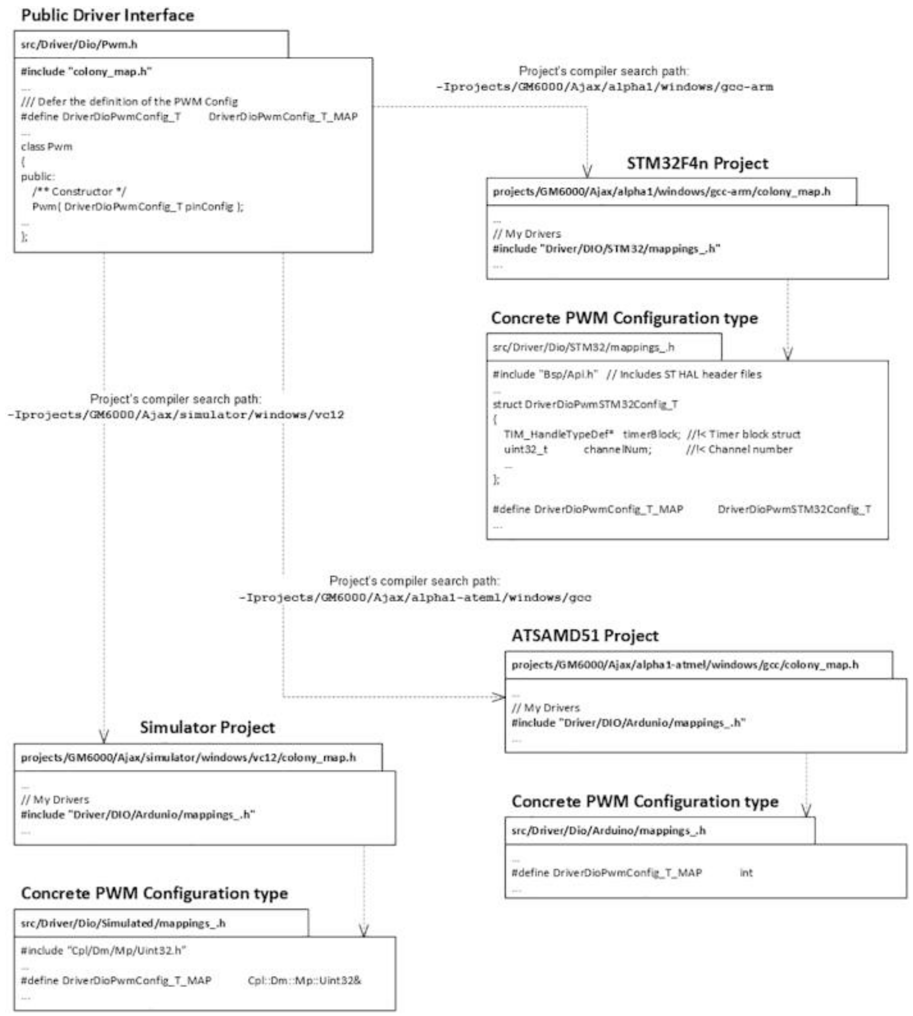
2. In the public interface header file, use the preprocessor to create a name for the PWM configuration type that maps to another, yet to be resolved, symbol name. For example:

   ```
   /// Defer the definition of the PWM configuration type
   #define DriverDioPwmConfig_T     DriverDioPwmConfig_T_MAP
   ```

3. In the project's build directory, or in a directory that is unique to the project, create a `colony_map.h` header file. This header file then includes a platform-specific header file that maps the `DriverDioPwmConfig_T_MAP` symbol to a concrete type.

4. In the project's build script, add the directory where the `colony_map.h` file (from step 3) is located into the compiler's header search path.

Figure 15-1 illustrates how using the LHeader pattern allows project-specific concrete data types for the PWM configuration data type.


Public Driver Interface
src/Driver/Dio/Pwm.h
Project's compiler search path:
-Iprojects/GM6000/Ajax/alpha1/windows/gcc-arm
STM32F4n Project
projects/GM6000/Ajax/alpha1/windows/gcc-arm/colony_map.h
Concrete PWM Configuration type
src/Driver/Dio/STM32/mappings_.h
Project's compiler search path:
-Iprojects/GM6000/Ajax/simulator/windows/vc12
Project's compiler search path:
-Iprojects/GM6000/Ajax/alpha1-ateml/windows/gcc
ATSAMD51 Project
projects/GM6000/Ajax/alpha1-atmel/windows/gcc/colony_map.h
Concrete PWM Configuration type
src/Driver/Dio/Arduino/mappings_.h
Simulator Project
projects/GM6000/Ajax/simulator/windows/vc12/colony_map.h
Concrete PWM Configuration type
src/Driver/Dio/Simulated/mappings_.h

```
#include "colony_map.h"
...
/// Defer the definition of the PWM Config
#define DriverDioPwmConfig_T        DriverDioPwmConfig_T_MAP
...
class Pwm
{
public:
   /** Constructor */
   Pwm( DriverDioPwmConfig_T pinConfig );
...
};
```

```
...
// My Drivers
#include "Driver/DIO/STM32/mappings_.h"
...
```

```
#include "Bsp/Api.h"  // Includes ST HAL header files
...
struct DriverDioPwmSTM32Config_T
{
   TIM_HandleTypeDef*   timerBlock;  //!< Timer block struct
   uint32_t          channelNum;         //!< Channel number
   ...
};

#define DriverDioPwmConfig_T_MAP          DriverDioPwmSTM32Config_T
...
```

```
...
// My Drivers
#include "Driver/DIO/Ardunio/mappings_.h"
...
```

```
...
#define DriverDioPwmConfig_T_MAP          int
...
```

```
...
// My Drivers
#include "Driver/DIO/Ardunio/mappings_.h"
...
```

```
#include "Cpl/Dm/Mp/Uint32.h"
...
#define DriverDioPwmConfig_T_MAP        Cpl::Dm::Mp::Uint32&
...
```

Figure 15-1. *PWM compile-time binding using the LHeader pattern*

Hardware Abstract Layer (HAL)

There are numerous ways to create interfaces for a Hardware Abstraction Layer (HAL). The following section covers three ways to create abstract interfaces that decouple a driver from platform-specific dependencies. They are as follows:

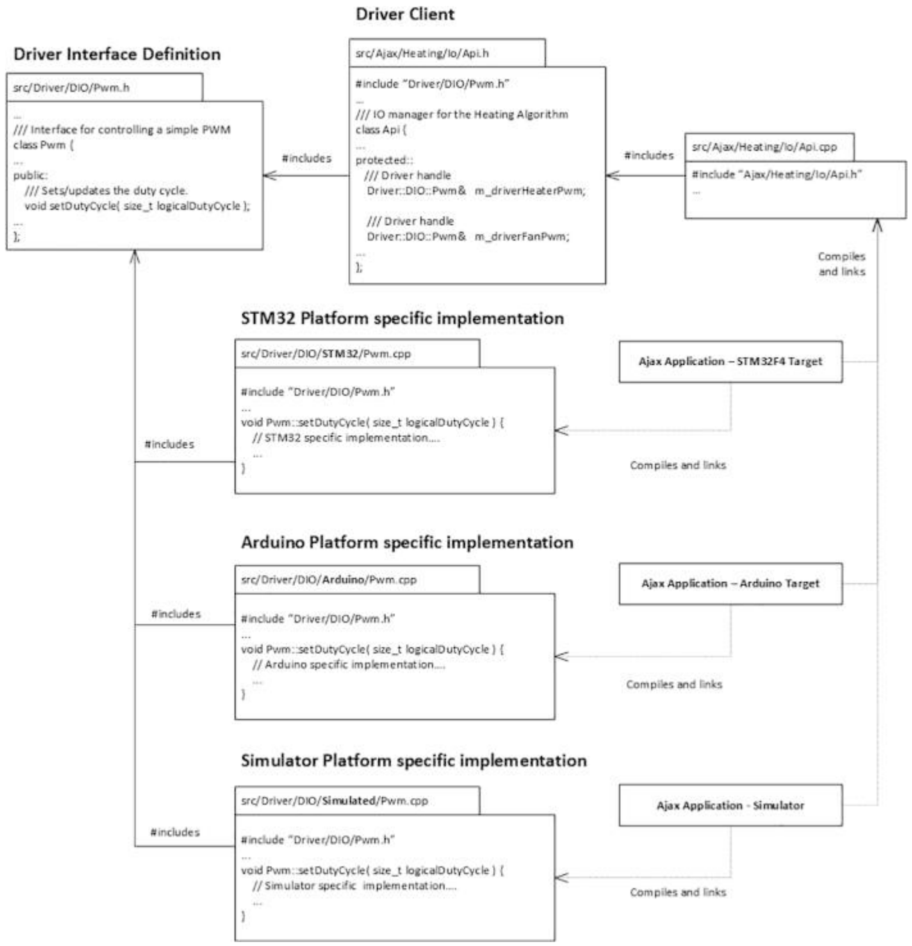
- Facade—Using link-time bindings
- Separation of concerns—Using compile-time bindings
- Polymorphic—Using run-time bindings

In this context, an “abstract interface” simply means any interface that defines a behavior that has a late binding. In fact, the same techniques can be used to decouple components throughout the application. In other words, these techniques are not unique to writing hardware-independent drivers.

Facade

A facade driver design is one where the public interface is defined and then each supported platform has its own unique implementation. That is, there is no explicit HAL defined. A simple and effective approach for a facade design is to use a link-time binding. This involves declaring a set of functions (i.e., the driver’s public interface) and then having platform-specific implementations for that set of functions. In this way, each platform gets its own implementation. The PWM driver in the CPL class library is an example of an HAL that uses link-time binding. This driver generates a PWM output signal at a fixed frequency with a variable duty cycle controlled by the application.

Figure 15-2 illustrates how a client of the PWM driver is decoupled from the target platform by using link-time bindings.



src/Driver/DIO/Pwm.h

```
...
/// Interface for controlling a simple PWM
class Pwm {
...
public:
   /// Sets/updates the duty cycle.
   void setDutyCycle( size_t logicalDutyCycle );
...
};
```

src/Ajax/Heating/Io/Api.h

```
#include "Driver/DIO/Pwm.h"
...
/// IO manager for the Heating Algorithm
class Api {
...
protected::
   /// Driver handle
   Driver::DIO::Pwm&   m_driverHeaterPwm;

   /// Driver handle
   Driver::DIO::Pwm&   m_driverFanPwm;
...
};
```

src/Ajax/Heating/Io/Api.cpp

```
#include "Ajax/Heating/Io/Api.h"
...
```

src/Driver/DIO/**STM32**/Pwm.cpp

```
#include "Driver/DIO/Pwm.h"
...
void Pwm::setDutyCycle( size_t logicalDutyCycle ) {
   // STM32 specific implementation....
   ...
}
```

src/Driver/DIO/**Arduino**/Pwm.cpp

```
#include "Driver/DIO/Pwm.h"
...
void Pwm::setDutyCycle( size_t logicalDutyCycle ) {
   // Arduino specific implementation....
   ...
}
```

src/Driver/DIO/**Simulated**/Pwm.cpp

```
#include "Driver/DIO/Pwm.h"
...
void Pwm::setDutyCycle( size_t logicalDutyCycle ) {
   // Simulator specific  implementation....
   ...
}
```

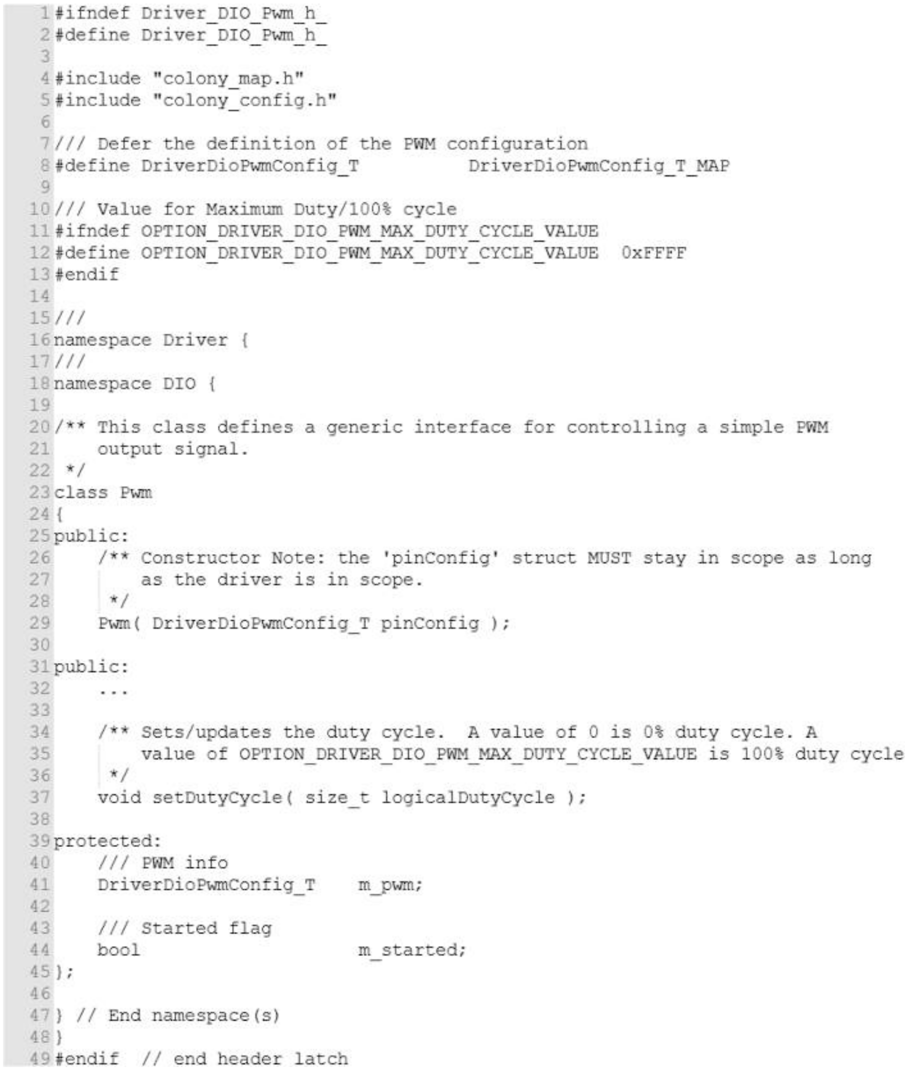
Driver Interface Definition
Driver Client
#includes
#includes
Compiles and links
STM32 Platform specific implementation
Ajax Application – STM32F4 Target
#includes
Compiles and links
Arduino Platform specific implementation
Ajax Application – Arduino Target
#includes
Compiles and links
Simulator Platform specific implementation
Ajax Application - Simulator
#includes
Compiles and links

Figure 15-2. *PWM driver design using a facade and link-time bindings*

Example PWM Driver

The interface for the PWM driver defines the concrete class `Driver::DIO::PWM` that is the public interface for the driver (in the file `src/Driver/DIO/Pwm.h`). In addition to link-time binding, the driver design uses the LHeader pattern for defining a platform-independent PWM configuration type—`DriverDioPwmConfig_T`. Figure 15-3 shows the class definition and the driver's public interface. Note the following:

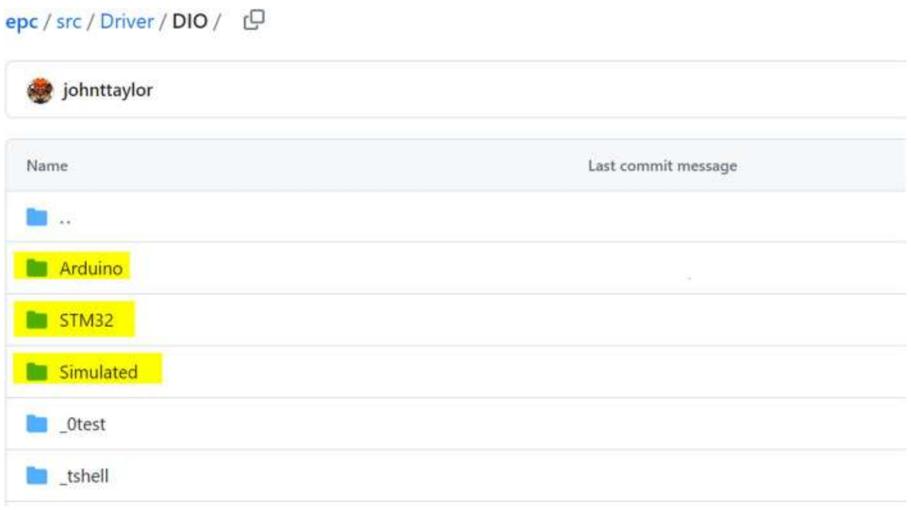
- Lines 4 and 5 do not contain path information. The build scripts specify the header path for resolving these `#include` statements.
- Line 8 uses the LHeader pattern to defer the selection of the concrete type for the PWM configuration structure to the application's build script.
- Lines 11–13 show an example of using the LConfig pattern for defining 100% PWM value.
- Lines 23–45 define a concrete class for the PWM driver. The actual implementation will be specified by the application's build script.



```
#ifndef Driver_DIO_Pwm_h_
#define Driver_DIO_Pwm_h_

#include "colony_map.h"
#include "colony_config.h"

/// Defer the definition of the PWM configuration
#define DriverDioPwmConfig_T          DriverDioPwmConfig_T_MAP

/// Value for Maximum Duty/100% cycle
#ifndef OPTION_DRIVER_DIO_PWM_MAX_DUTY_CYCLE_VALUE
#define OPTION_DRIVER_DIO_PWM_MAX_DUTY_CYCLE_VALUE  0xFFFF
#endif

///
namespace Driver {
///
namespace DIO {

/** This class defines a generic interface for controlling a simple PWM
    output signal.
 */
class Pwm
{
public:
    /** Constructor Note: the 'pinConfig' struct MUST stay in scope as long
        as the driver is in scope.
     */
    Pwm( DriverDioPwmConfig_T pinConfig );

public:
    ...

    /** Sets/updates the duty cycle.  A value of 0 is 0% duty cycle. A
        value of OPTION_DRIVER_DIO_PWM_MAX_DUTY_CYCLE_VALUE is 100% duty cycle
     */
    void setDutyCycle( size_t logicalDutyCycle );

protected:
    /// PWM info
    DriverDioPwmConfig_T    m_pwm;

    /// Started flag
    bool                    m_started;
};

} // End namespace(s)
}
#endif  // end header latch
```

Figure 15-3. *Partial file listing of epc/src/Driver/DIO/Pwm.h*

Each supported hardware platform requires its own implementation of the concrete class. It is the project (or unit test) build scripts that determine which implementation is linked into the executable image. This is why in the repository there are three different implementations of the PWM class in three different directories (see Figure 15-4).


epc / src / Driver / DIO /

johnttaylor

Name	Last commit message
..	
Arduino	
STM32	
Simulated	
_0test	
_tshell	

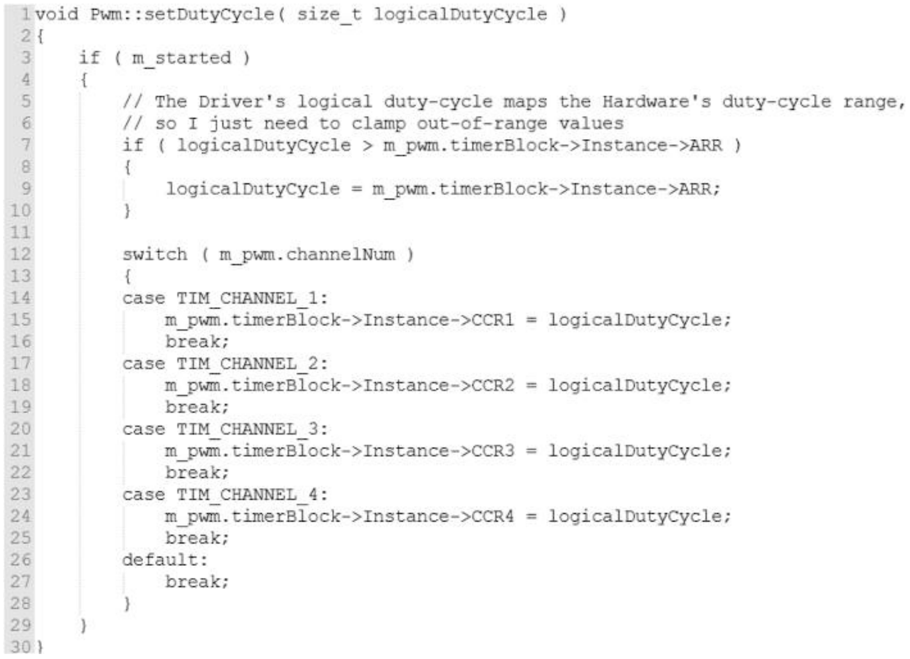
Figure 15-4. *Location of the three different implementations of the PWM class*

In the following three sections, you will see the implementation of the PWM functionality for each of these platforms:

- STM32—See Figure 15-5.
- Arduino—See Figure 15-6.
- Simulated—See Figure 15-7.

The STM32 Implementation of PWM

The implementation relies on the ST HAL library and initialization code generated by the ST Cube MX IDE. The implementation is located at `src/Driver/DIO/STM32/Pwm.cpp`. Figure 15-5 shows the implementation for the STM32 `setDutyCycle()` method.

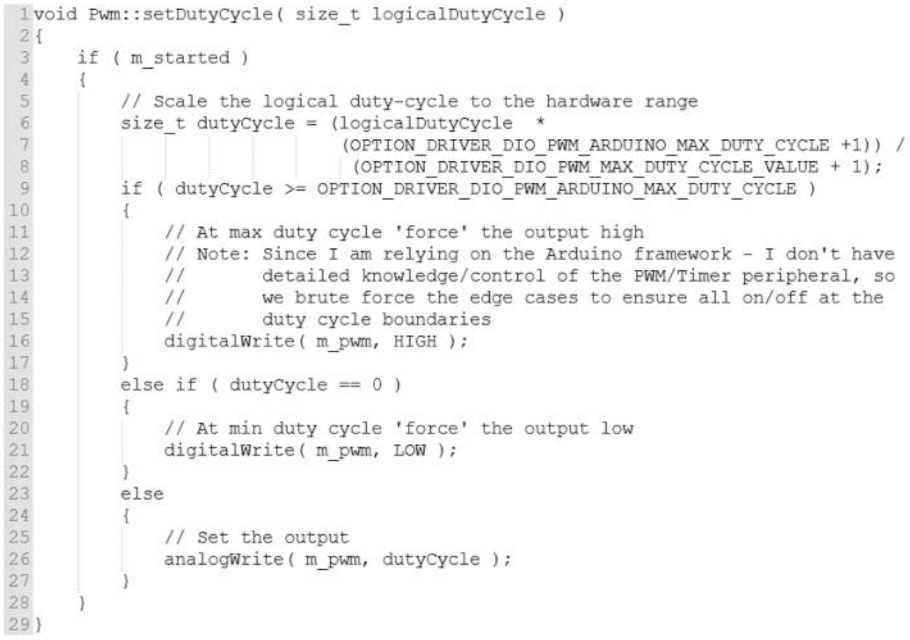


```
void Pwm::setDutyCycle( size_t logicalDutyCycle )
{
    if ( m_started )
    {
        // The Driver's logical duty-cycle maps the Hardware's duty-cycle range,
        // so I just need to clamp out-of-range values
        if ( logicalDutyCycle > m_pwm.timerBlock->Instance->ARR )
        {
            logicalDutyCycle = m_pwm.timerBlock->Instance->ARR;
        }

        switch ( m_pwm.channelNum )
        {
        case TIM_CHANNEL_1:
            m_pwm.timerBlock->Instance->CCR1 = logicalDutyCycle;
            break;
        case TIM_CHANNEL_2:
            m_pwm.timerBlock->Instance->CCR2 = logicalDutyCycle;
            break;
        case TIM_CHANNEL_3:
            m_pwm.timerBlock->Instance->CCR3 = logicalDutyCycle;
            break;
        case TIM_CHANNEL_4:
            m_pwm.timerBlock->Instance->CCR4 = logicalDutyCycle;
            break;
        default:
            break;
        }
    }
}
```

Figure 15-5. *Partial listing of src/Driver/DIO/STM32/pwm.cpp*

Arduino Framework Implementation of PWM

The implementation is built on top of the Arduino framework. The implementation is located at `src/Driver/DIO/Arduino/Pwm.cpp`. Figure 15-6 shows the Arduino implementation for the `setDutyCycle()` method.

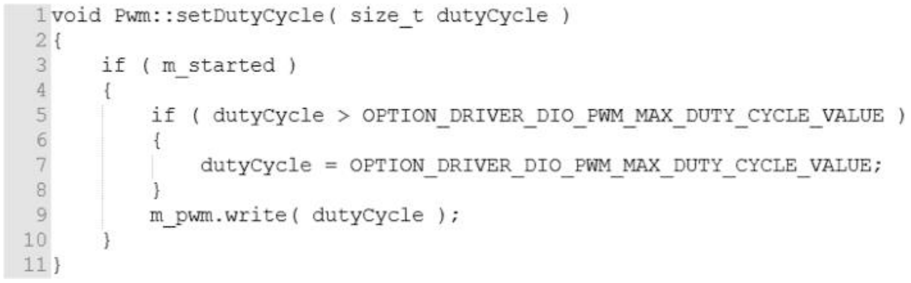


```
void Pwm::setDutyCycle( size_t logicalDutyCycle )
{
    if ( m_started )
    {
        // Scale the logical duty-cycle to the hardware range
        size_t dutyCycle = (logicalDutyCycle  *
                            (OPTION_DRIVER_DIO_PWM_ARDUINO_MAX_DUTY_CYCLE +1)) /
                             (OPTION_DRIVER_DIO_PWM_MAX_DUTY_CYCLE_VALUE + 1);
        if ( dutyCycle >= OPTION_DRIVER_DIO_PWM_ARDUINO_MAX_DUTY_CYCLE )
        {
            // At max duty cycle 'force' the output high
            // Note: Since I am relying on the Arduino framework - I don't have
            //       detailed knowledge/control of the PWM/Timer peripheral, so
            //       we brute force the edge cases to ensure all on/off at the
            //       duty cycle boundaries
            digitalWrite( m_pwm, HIGH );
        }
        else if ( dutyCycle == 0 )
        {
            // At min duty cycle 'force' the output low
            digitalWrite( m_pwm, LOW );
        }
        else
        {
            // Set the output
            analogWrite( m_pwm, dutyCycle );
        }
    }
}
```

Figure 15-6. *Partial listing of src/Driver/DIO/Arduino/Pwm.cpp*

Simulator Implementation of PWM

The functional simulator implementation uses model points to simulate the PWM output signal. That is, the duty cycle is a simple model point that can be accessed using the debug console. The implementation is located at `src/Driver/DIO/Simulated/Pwm.cpp.` Figure 15-7 shows the simulator implementation for the `setDutyCycle()` method.



```
void Pwm::setDutyCycle( size_t dutyCycle )
{
    if ( m_started )
    {
        if ( dutyCycle > OPTION_DRIVER_DIO_PWM_MAX_DUTY_CYCLE_VALUE )
        {
            dutyCycle = OPTION_DRIVER_DIO_PWM_MAX_DUTY_CYCLE_VALUE;
        }
        m_pwm.write( dutyCycle );
    }
}
```

Figure 15-7. *Partial listing of src/Driver/DIO/Simulated/Pwm.cpp*

The PWM example is a C++ example. However, the mechanism of link-time binding can easily be done in C. To do this, replace the member functions of the PWM class with C functions that take a pointer to the `DriverDioPwmConfig_T` structure as one of the function arguments. For example, in C, functions would be defined like this:

```
void pwm_initialize( DriverDioPwmConfig_T* pwmHdl );
bool pwm_start( DriverDioPwmConfig_T* pwmHdl );
void pwm_stop( DriverDioPwmConfig_T* pwmHdl );
void pwm_setDutyCycle( DriverDioPwmConfig_T* pwmHdl, size_t dutyCycle);
```

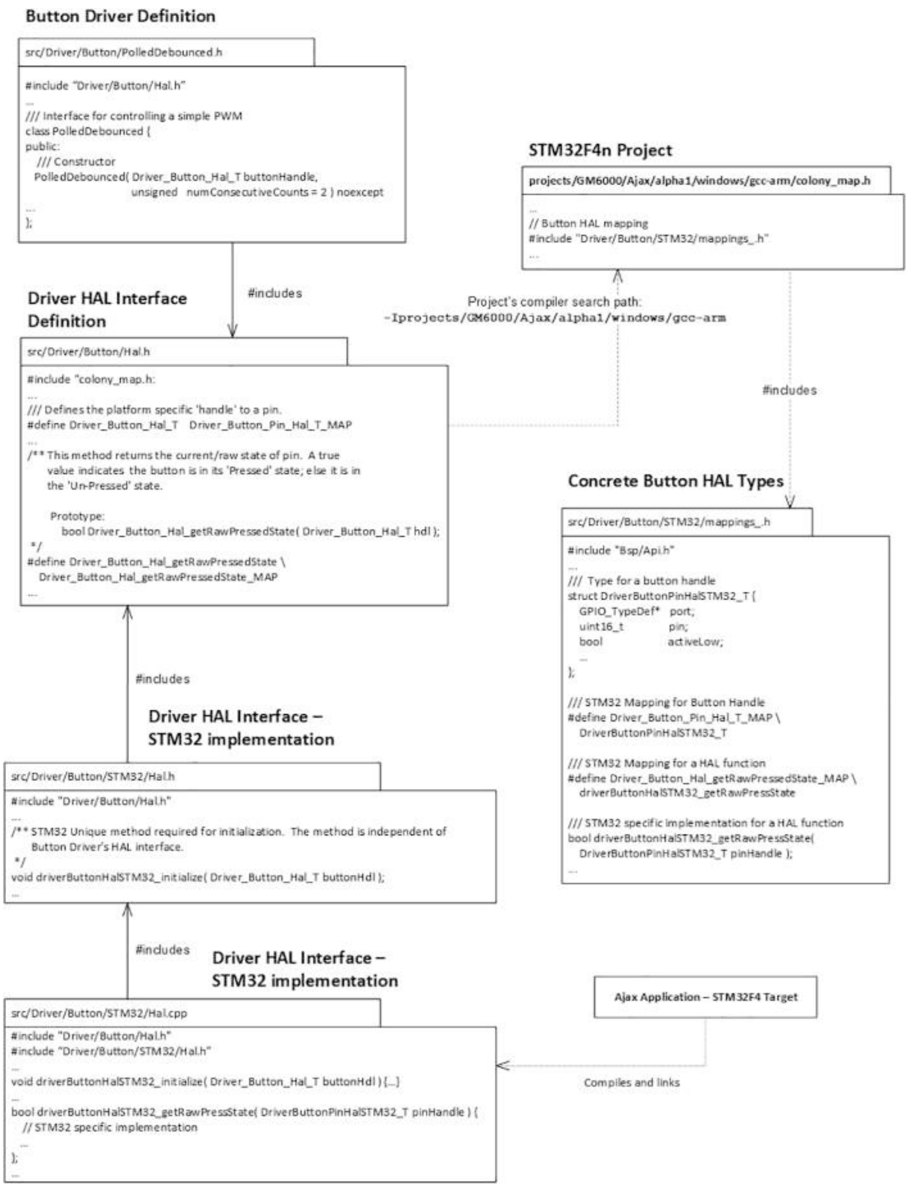
Separation of Concerns

The “separation of concerns” approach to driver design is to separate the business logic of the driver from the hardware details. This involves creating an explicit HAL interface definition in a header file that is separate from the driver’s public interface. The HAL interface specifies basic hardware actions. Or, said another way, the HAL interface should be limited to encapsulating access to the MCU’s registers or SDK function calls. The implementation of the public driver interface provides the business logic while relying on the HAL interface for interactions with the hardware.

There are several options for how to structure the HAL interface definition. One approach is to use link-time bindings as described in the "Facade" section. Another is to use compile-time binding. An advantage of using compile-time binding is that it provides the least amount of runtime overhead when implementing an HAL interface since it relies solely on the preprocessor.

The implementation for the compile-time binding is to use the same LHeader mechanism that was used in the "Facade" section to include methods not just a data type. The Polled Debounced Button driver in the CPL class library is an example of an HAL using only compile-time bindings. The PolledDebounced driver samples the raw button state at a fixed frequency (determined by the application at run time) requiring *N* consecutive samples with the same raw button state to declare a change in the logical, or debounced, button state.

Figure 15-8 illustrates how the HAL interface for the Button driver is decoupled from the target platform by using the LHeader pattern.


Button Driver Definition
#includes
Driver HAL Interface Definition
STM32F4n Project
Project's compiler search path:
-Iprojects/GM6000/Ajax/alpha1/windows/gcc-arm
#includes
Concrete Button HAL Types
#includes
Driver HAL Interface – STM32 implementation
#includes
Driver HAL Interface – STM32 implementation
Ajax Application – STM32F4 Target
Compiles and links

src/Driver/Button/PolledDebounced.h

```
#include "Driver/Button/Hal.h"
...
/// Interface for controlling a simple PWM
class PolledDebounced {
public:
  /// Constructor
  PolledDebounced( Driver_Button_Hal_T buttonHandle,
                   unsigned  numConsecutiveCounts = 2 ) noexcept
...
};
```

src/Driver/Button/Hal.h

```
#include "colony_map.h:
...
/// Defines the platform specific 'handle' to a pin.
#define Driver_Button_Hal_T   Driver_Button_Pin_Hal_T_MAP
...
/** This method returns the current/raw state of pin.  A true
    value indicates the button is in its 'Pressed' state; else it is in
    the 'Un-Pressed' state.

     Prototype:
       bool Driver_Button_Hal_getRawPressedState( Driver_Button_Hal_T hdl );
 */
#define Driver_Button_Hal_getRawPressedState \
  Driver_Button_Hal_getRawPressedState_MAP
...
```

projects/GM6000/Ajax/alpha1/windows/gcc-arm/colony_map.h

```
...
// Button HAL mapping
#include "Driver/Button/STM32/mappings_.h"
...
```

src/Driver/Button/STM32/mappings_.h

```
#include "Bsp/Api.h"
...
/// Type for a button handle
struct DriverButtonPinHalSTM32_T {
  GPIO_TypeDef*  port;
  uint16_t       pin;
  bool           activeLow;
  ...
};

/// STM32 Mapping for Button Handle
#define Driver_Button_Pin_Hal_T_MAP \
  DriverButtonPinHalSTM32_T

/// STM32 Mapping for a HAL function
#define Driver_Button_Hal_getRawPressedState_MAP \
  driverButtonHalSTM32_getRawPressState

/// STM32 specific implementation for a HAL function
bool driverButtonHalSTM32_getRawPressState(
  DriverButtonPinHalSTM32_T pinHandle );
...
```

src/Driver/Button/STM32/Hal.h

```
#include "Driver/Button/Hal.h"
...
/** STM32 Unique method required for initialization.  The method is independent of
    Button Driver's HAL interface.
 */
void driverButtonHalSTM32_initialize( Driver_Button_Hal_T buttonHdl );
...
```

src/Driver/Button/STM32/Hal.cpp

```
#include "Driver/Button/Hal.h"
#include "Driver/Button/STM32/Hal.h"
...
void driverButtonHalSTM32_initialize( Driver_Button_Hal_T buttonHdl ) {...}
...
bool driverButtonHalSTM32_getRawPressState( DriverButtonPinHalSTM32_T pinHandle ) {
  // STM32 specific implementation
  ...
};
...
```

Figure 15-8. *Button driver design using separation of concerns and the LHeader pattern*

Example Button Driver

The interface for the Polled Debounced Button driver defines a hardware-independent concrete class `Driver::Button::PolledDebounced` and a separate HAL header file. The HAL header file uses the LHeader pattern to define a hardware-independent data type and behavior (e.g., "get raw button pressed state"). The `PolledDebounced` class implements the driver logic using the abstract interface declared in the HAL header file. Figure 15-9 shows a partial listing for the `PolledDebounced` button class definition in `src/Driver/Button/PolledDebounced.h`. Note the following:

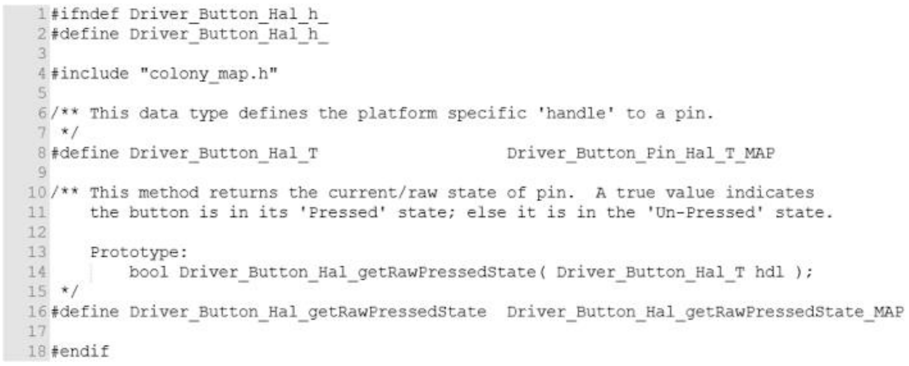
- Line 4 includes the HAL interface. This header file defines the `Driver_Button_Hal_T` structure (using the LHeader pattern).
- Lines 19–33 define the PolledDebounced driver's public interface with no dependencies on the target platform.

```
#ifndef Driver_Button_PolledDebounced_h_
#define Driver_Button_PolledDebounced_h_

#include "Driver/Button/Hal.h"

namespace Driver { namespace Button {

/** This concrete class implements a button driver where a single button
    is polled and its raw button state is de-bounced.
 */
class PolledDebounced
{
public:
    /** Constructor. The physical button being is sampled is specified by
        'buttonHandle'.  The 'numConsecutiveCounts' specifies the number of
        consecutive sample periods - without the raw button state changing -
        to declare a new de-bounced state.
    */
    PolledDebounced( Driver_Button_Hal_T buttonHandle,
                     unsigned            numConsecutiveCounts = 2 ) noexcept;
    ...

public:
    /// This method returns the de-bounced pressed button state.
    inline bool isPressed() noexcept { return m_pressed; }

    /** The application is required to call this method on fixed periodic
        intervals.  The raw button state is sampled during this call.
     */
    void sample() noexcept;

    /// Returns the driver's button handle
    inline Driver_Button_Hal_T getHandle() { return m_buttonHdl; }

protected:
    /// Handle to the button
    Driver_Button_Hal_T m_buttonHdl;

    /// Required number of consecutive counts
    unsigned            m_requiredCount;

    /// Consecutive counts
    unsigned            m_counts;

    /// Previous raw state
    bool                m_previousRawPressed;

    /// De-bounced state
    bool                m_pressed;
};

} // End namespace(s)
}
#endif  // end header latch
```

Figure 15-9. *Partial file listing of src/Driver/Button/PolledDebounced.h*

Figure 15-10 shows a snippet from the HAL header file (located at `src/Driver/Button/Hal.h`). Note the following:

- Line 4 does not contain path information. The build scripts specify the header path for resolving this `#include` statement.
- Line 8 uses the LHeader pattern to defer the selection of the concrete type for the Button structure to the application's build script.
- Line 16 defines the HAL interface method as a macro, per the LHeader pattern, to defer the implementation selection to the application's build script.
- There are no create or initialize functions declared. This is intentional since the creation is very platform specific (unless you add additional abstractions such as a factory pattern).[1]



```
 1 #ifndef Driver_Button_Hal_h_
 2 #define Driver_Button_Hal_h_
 3 
 4 #include "colony_map.h"
 5 
 6 /** This data type defines the platform specific 'handle' to a pin.
 7  */
 8 #define Driver_Button_Hal_T                   Driver_Button_Pin_Hal_T_MAP
 9 
10 /** This method returns the current/raw state of pin.  A true value indicates
11     the button is in its 'Pressed' state; else it is in the 'Un-Pressed' state.
12 
13     Prototype:
14         bool Driver_Button_Hal_getRawPressedState( Driver_Button_Hal_T hdl );
15  */
16 #define Driver_Button_Hal_getRawPressedState  Driver_Button_Hal_getRawPressedState_MAP
17 
18 #endif
```

Figure 15-10. *Partial file listing of `src/Driver/Button/Hal.h`*

[1] https://en.wikipedia.org/wiki/Factory_method_pattern

By only including the usage behavior in your HAL interfaces, you can delegate the creation and initialization to platform-specific code on startup (see Chapter 7, "Building Applications with the Main Pattern") without burdening consumers of the driver with platform dependencies. The next step is to implement the abstractions declared in the HAL header file. To accomplish this, you will need to perform the following tasks for each hardware platform:

1. Implement the abstract functions declared in the HAL interface header file. I recommend that you create a pair of files: a header file and a `.c|.cpp` file. The header file will be used to declare public platform-specific functions that provide functionality that was not defined in the driver's HAL interface (e.g., initialize methods). The `.c|.cpp` file will contain the platform-specific HAL implementation.

2. Create a mapping header file. This file contains the mapping for the abstract data types and functions.

In the following three sections, you will see the implementation of the button driver functionality for each of these platforms:

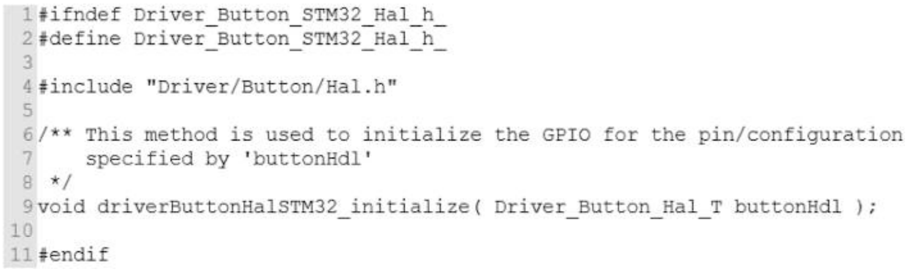
- STM32—See Figures 15-11, 15-12, and 15-13.
- Arduino—See Figures 15-14, 15-15, and 15-16.
- Simulated—See Figures 15-17, 15-18, and 15-19.

STM32 MCU Implementation of the Button Driver

The implementation relies on the ST HAL library and initialization code generated by the ST Cube MX IDE. Figures 15-11 through 15-13 show the files that comprise the implementation.

The `src/Driver/Button/STM32/Hal.h` is an STM32-specific header file that provides hardware-specific initialization for the button driver. Note the following lines in Figure 15-11.

- Line 4 includes the HAL interface. This header file defines the `Driver_Button_Hal_T` structure (using the LHeader pattern).
- Line 9 is the STM32-specific driver initialize method. The expectation is that on startup, STM32-specific code would call this method to complete initializing the button driver.



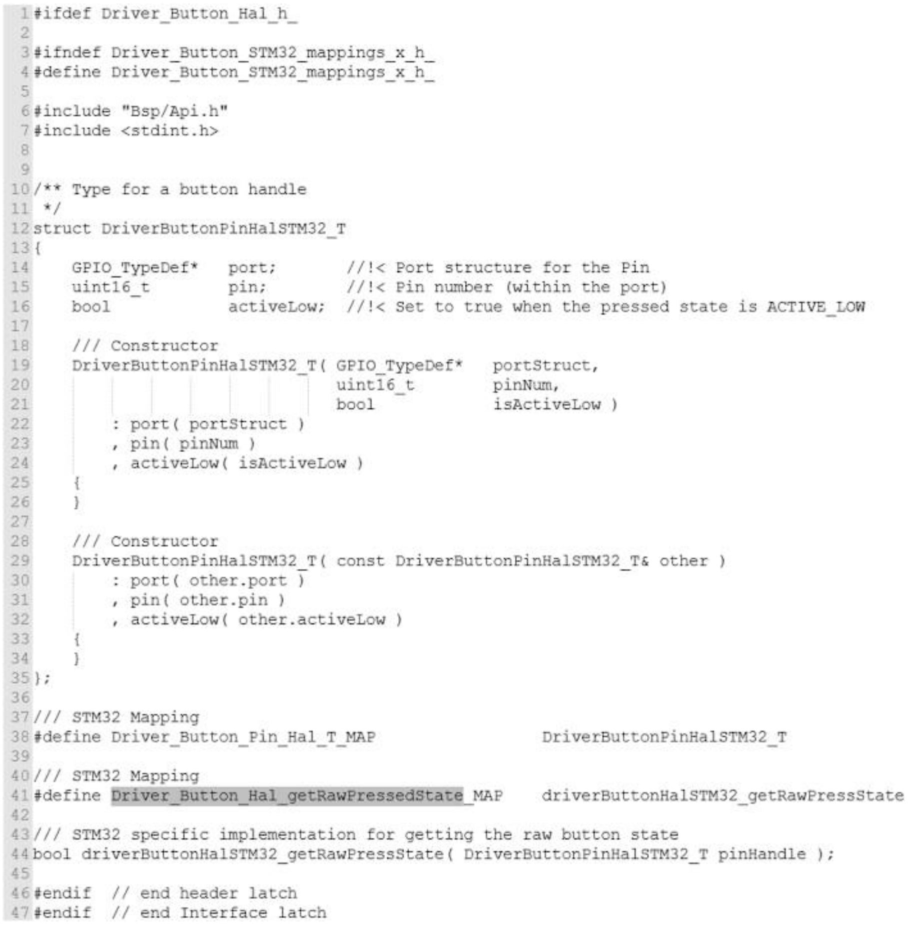
```
 1 #ifndef Driver_Button_STM32_Hal_h_
 2 #define Driver_Button_STM32_Hal_h_
 3
 4 #include "Driver/Button/Hal.h"
 5
 6 /** This method is used to initialize the GPIO for the pin/configuration
 7     specified by 'buttonHdl'
 8  */
 9 void driverButtonHalSTM32_initialize( Driver_Button_Hal_T buttonHdl );
10
11 #endif
```

Figure 15-11. *STM32-specific interface for a button driver*

The `src/Driver/Button/STM32/mappings_.h` is used to resolve the LHeader deferred methods and types to an STM32 hardware platform. Note the following lines in Figure 15-12.

- Lines 1 and 47 comprise an additional header latch to break cyclical header file includes when using the LHeader pattern. See the "LHeader Caveats" section for an explanation of why this is needed.
- Lines 3, 4, and 46 are the traditional header latch so that the header is only included once when compiling files.

- Line 6 is another LHeader pattern usage that allows the STM32 button driver to work with any STM32-based BSP.
- Lines 12–35 define the STM32-specific concrete data type for the instance.
- Line 38 provides the LHeader mapping for the abstract HAL interface to the STM32-specific data structure defined at lines 12–35.
- Line 44 provides the function prototype for the STM32-specific implementation of the abstract HAL i nterface's `Driver_Button_Hal_getRawPressedState()` method.
- Line 41 provides the LHeader mapping for the abstract HAL interface's `Driver_Button_Hal_getRawPressedState()` method.



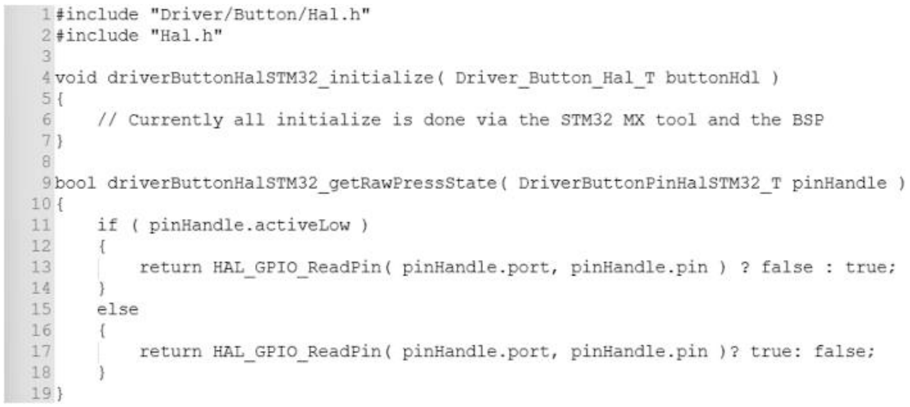
```
#ifdef Driver_Button_Hal_h_

#ifndef Driver_Button_STM32_mappings_x_h_
#define Driver_Button_STM32_mappings_x_h_

#include "Bsp/Api.h"
#include <stdint.h>


/** Type for a button handle
 */
struct DriverButtonPinHalSTM32_T
{
    GPIO_TypeDef*   port;       //!< Port structure for the Pin
    uint16_t        pin;        //!< Pin number (within the port)
    bool            activeLow;  //!< Set to true when the pressed state is ACTIVE_LOW

    /// Constructor
    DriverButtonPinHalSTM32_T( GPIO_TypeDef*   portStruct,
                               uint16_t        pinNum,
                               bool            isActiveLow )
        : port( portStruct )
        , pin( pinNum )
        , activeLow( isActiveLow )
    {
    }

    /// Constructor
    DriverButtonPinHalSTM32_T( const DriverButtonPinHalSTM32_T& other )
        : port( other.port )
        , pin( other.pin )
        , activeLow( other.activeLow )
    {
    }
};

/// STM32 Mapping
#define Driver_Button_Pin_Hal_T_MAP                 DriverButtonPinHalSTM32_T

/// STM32 Mapping
#define Driver_Button_Hal_getRawPressedState_MAP    driverButtonHalSTM32_getRawPressState

/// STM32 specific implementation for getting the raw button state
bool driverButtonHalSTM32_getRawPressState( DriverButtonPinHalSTM32_T pinHandle );

#endif  // end header latch
#endif  // end Interface latch
```

Figure 15-12. *STM32 LHeader mappings for the button driver*

The `src/Driver/Button/STM32/Hal.cpp` file implements the HAL interface for the STM32 platform. Note the following lines in Figure 15-13.

- Line 1 includes the abstract HAL interface definition.
- Line 2 includes the STM32 button driver interface definition.

- Lines 4–7 are the STM32 implementation of the method declared in the header file from line 2.
- Lines 9–19 are the implementation of the abstract HAL interface.



```
 1 #include "Driver/Button/Hal.h"
 2 #include "Hal.h"
 3
 4 void driverButtonHalSTM32_initialize( Driver_Button_Hal_T buttonHdl )
 5 {
 6     // Currently all initialize is done via the STM32 MX tool and the BSP
 7 }
 8
 9 bool driverButtonHalSTM32_getRawPressState( DriverButtonPinHalSTM32_T pinHandle )
10 {
11     if ( pinHandle.activeLow )
12     {
13         return HAL_GPIO_ReadPin( pinHandle.port, pinHandle.pin ) ? false : true;
14     }
15     else
16     {
17         return HAL_GPIO_ReadPin( pinHandle.port, pinHandle.pin )? true: false;
18     }
19 }
```

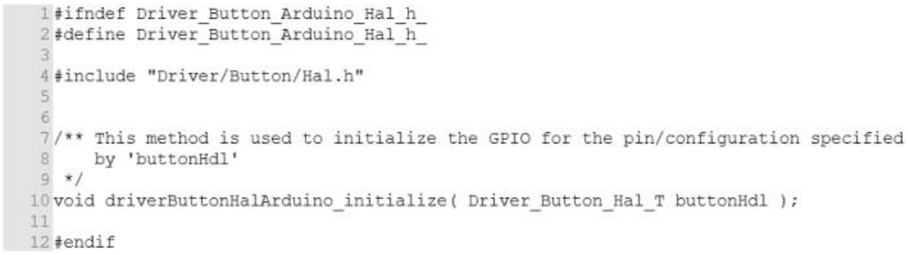
Figure 15-13. *STM32 HAL button driver implementation*

Arduino Framework Implementation of the Button Driver

The implementation is built on top of the Arduino framework. Figures 15-14 through 15-16 show the files that comprise the implementation.

The `src/Driver/Button/Arduino/Hal.h` is an Arduino-specific header file that provides hardware-specific initialization for the button driver. Note the following lines in Figure 15-14.

- Line 4 includes the HAL interface. This header file defines the `Driver_Button_Hal_T` structure using the LHeader pattern.
- Line 10 is the Arduino-specific driver initialization method. The expectation is that on startup, Arduino-specific code would call this method to complete the initialization of the button driver.



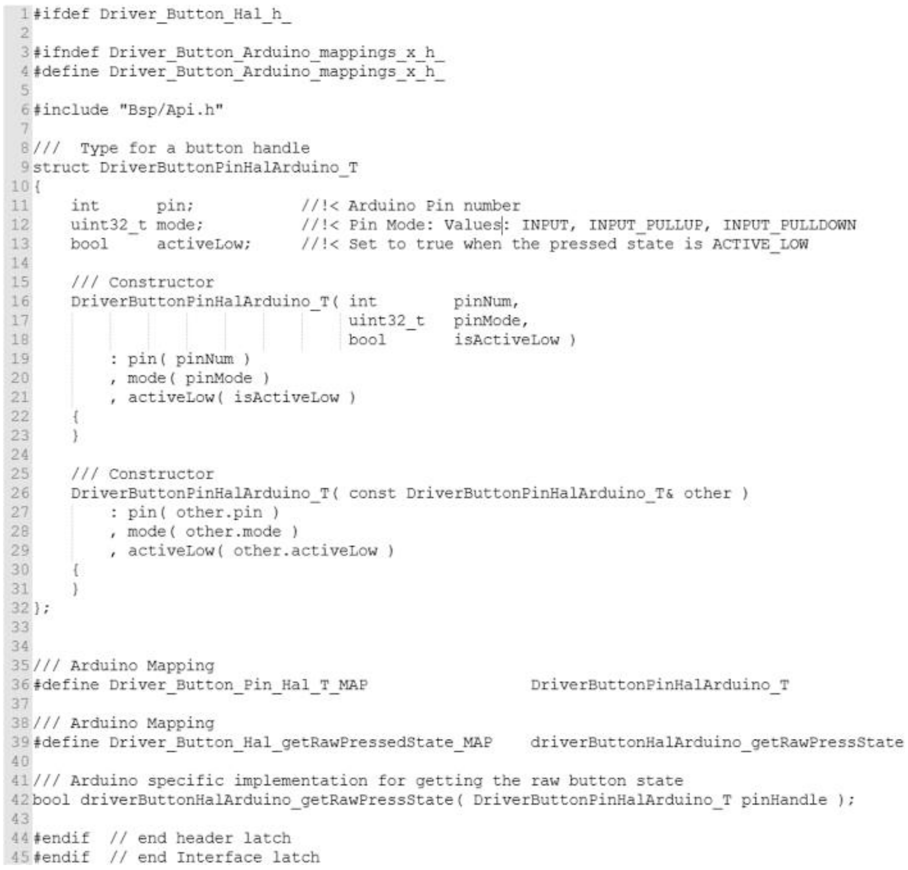
```
#ifndef Driver_Button_Arduino_Hal_h_
#define Driver_Button_Arduino_Hal_h_

#include "Driver/Button/Hal.h"


/** This method is used to initialize the GPIO for the pin/configuration specified
    by 'buttonHdl'
 */
void driverButtonHalArduino_initialize( Driver_Button_Hal_T buttonHdl );

#endif
```

Figure 15-14. *Arduino framework–specific interface for a button driver*

The `src/Driver/Button/Ardunio/mappings_.h` is used to resolve the LHeader deferred methods and types to the Arduino framework. Note the following lines in Figure 15-15.

- Lines 1 and 45 comprise an additional header latch to break cyclical header file includes when using the LHeader pattern. See the “LHeader Caveats” section on why this is needed.
- Lines 3, 4, and 44 are the traditional header latch so that the header is only included once when compiling files.
- Line 6 is another LHeader pattern usage that allows the Arduino button driver to work with an Arduino framework–based BSP.
- Lines 9–32 define the Arduino-specific concrete data type for the button instance.
- Line 36 provides the LHeader mapping for the abstract HAL interface to the Arduino framework-specific data structure defined at lines 9–32.

- Line 42 provides the function prototype for the Arduino framework–specific implementation of the abstract method for `Driver_Button_Hal _getRawPressedState()`.
- Line 39 provides the LHeader mapping for the abstract method for `Driver_Button_Hal _getRawPressedState()`.

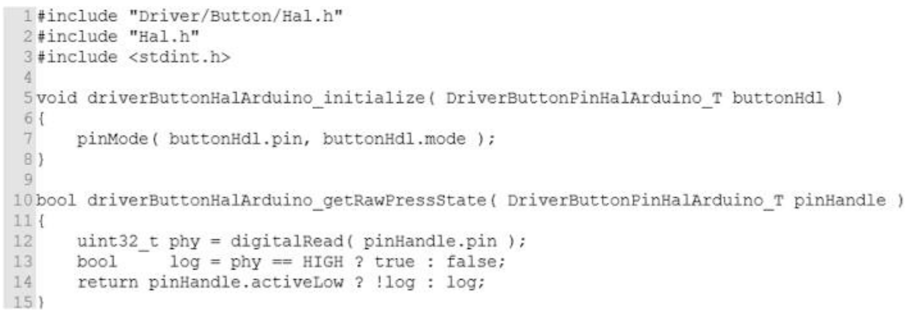


```
 1 #ifdef Driver_Button_Hal_h_
 2 
 3 #ifndef Driver_Button_Arduino_mappings_x_h_
 4 #define Driver_Button_Arduino_mappings_x_h_
 5 
 6 #include "Bsp/Api.h"
 7 
 8 ///  Type for a button handle
 9 struct DriverButtonPinHalArduino_T
10 {
11     int      pin;           //!< Arduino Pin number
12     uint32_t mode;          //!< Pin Mode: Values: INPUT, INPUT_PULLUP, INPUT_PULLDOWN
13     bool     activeLow;     //!< Set to true when the pressed state is ACTIVE_LOW
14 
15     /// Constructor
16     DriverButtonPinHalArduino_T( int        pinNum,
17                                  uint32_t   pinMode,
18                                  bool       isActiveLow )
19         : pin( pinNum )
20         , mode( pinMode )
21         , activeLow( isActiveLow )
22     {
23     }
24 
25     /// Constructor
26     DriverButtonPinHalArduino_T( const DriverButtonPinHalArduino_T& other )
27         : pin( other.pin )
28         , mode( other.mode )
29         , activeLow( other.activeLow )
30     {
31     }
32 };
33 
34 
35 /// Arduino Mapping
36 #define Driver_Button_Pin_Hal_T_MAP                 DriverButtonPinHalArduino_T
37 
38 /// Arduino Mapping
39 #define Driver_Button_Hal_getRawPressedState_MAP    driverButtonHalArduino_getRawPressState
40 
41 /// Arduino specific implementation for getting the raw button state
42 bool driverButtonHalArduino_getRawPressState( DriverButtonPinHalArduino_T pinHandle );
43 
44 #endif  // end header latch
45 #endif  // end Interface latch
```

Figure 15-15. *Arduino framework LHeader mappings for the button driver*

The `src/Driver/Button/Arduino/Hal.cpp` file implements the HAL interface for the Arduino framework. Note the following lines in Figure 15-16.

- Line 1 includes the abstract HAL interface definition.
- Line 2 includes the Arduino framework button driver interface definition.
- Lines 5–8 comprise the implementation of the Arduino framework for the method declared in the header file from line 2.
- Lines 10–15 comprise the implementation of the abstract HAL interface.



```
 1 #include "Driver/Button/Hal.h"
 2 #include "Hal.h"
 3 #include <stdint.h>
 4
 5 void driverButtonHalArduino_initialize( DriverButtonPinHalArduino_T buttonHdl )
 6 {
 7     pinMode( buttonHdl.pin, buttonHdl.mode );
 8 }
 9
10 bool driverButtonHalArduino_getRawPressState( DriverButtonPinHalArduino_T pinHandle )
11 {
12     uint32_t phy = digitalRead( pinHandle.pin );
13     bool     log = phy == HIGH ? true : false;
14     return pinHandle.activeLow ? !log : log;
15 }
```

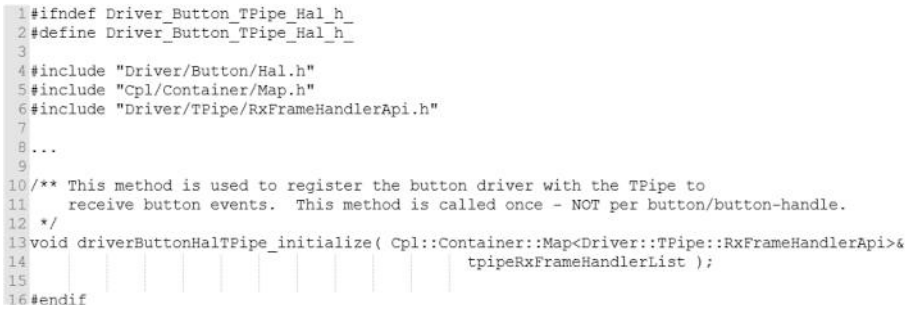
Figure 15-16. *Arduino framework HAL button driver implementation*

Simulator Implementation of the Button Driver

The GM6000 functional simulator uses an external C# executable to emulate the LCD display and its associated buttons. The simulator uses a TPipe driver to communicate with the external executable. TPipe is a platform-independent driver in the CPL class library for point-to-point, full-duplex, text-based stream, which is used to pass commands between two end points (see `src/Driver/TPipe`). As the TPipe driver only supplies the communication tunnel, a TPipe-aware button driver is also needed. The implementation of this button driver consists of the files shown in Figures 15-17 through 15-19.

The `src/Driver/Button/TPipe/Hal.h` is a TPipe-specific header file that provides hardware-specific initialization for the button driver. Note the following lines in Figure 15-17.

- Line 4 includes the HAL interface. This header file defines the `Driver_Button_Hal_T` structure (using the LHeader pattern).
- Line 13 is the TPipe-specific driver initialize method. The expectation is that on startup, this method is called to complete initializing the button driver.



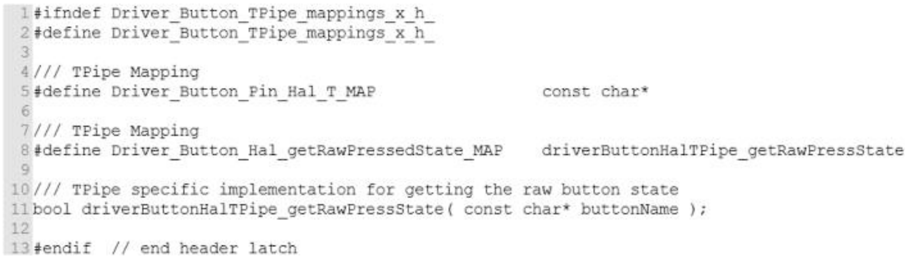
```
 1 #ifndef Driver_Button_TPipe_Hal_h_
 2 #define Driver_Button_TPipe_Hal_h_
 3
 4 #include "Driver/Button/Hal.h"
 5 #include "Cpl/Container/Map.h"
 6 #include "Driver/TPipe/RxFrameHandlerApi.h"
 7
 8 ...
 9
10 /** This method is used to register the button driver with the TPipe to
11     receive button events.  This method is called once - NOT per button/button-handle.
12  */
13 void driverButtonHalTPipe_initialize( Cpl::Container::Map<Driver::TPipe::RxFrameHandlerApi>&
14                                        tpipeRxFrameHandlerList );
15
16 #endif
```

***Figure 15-17.** TPipe-specific interface for a button driver*

The `src/Driver/Button/TPipe/mappings_.h` is used to resolve the LHeader deferred methods and types on the simulator platform. Note the following lines in Figure 15-18.

- Lines 1, 2, and 13 are the traditional header latch so that the header is only included once when compiling files.
- Line 5 provides the LHeader mapping for the abstract HAL interface to the TPipe-specific data type.

- Line 11 provides the function prototype for the TPipe-specific implementation of the abstract HAL interface's `Driver_Button_Hal_getRawPressedState()` method.
- Line 8 provides the LHeader mapping for the abstract method `Driver_Button_Hal_getRawPressedState()`.

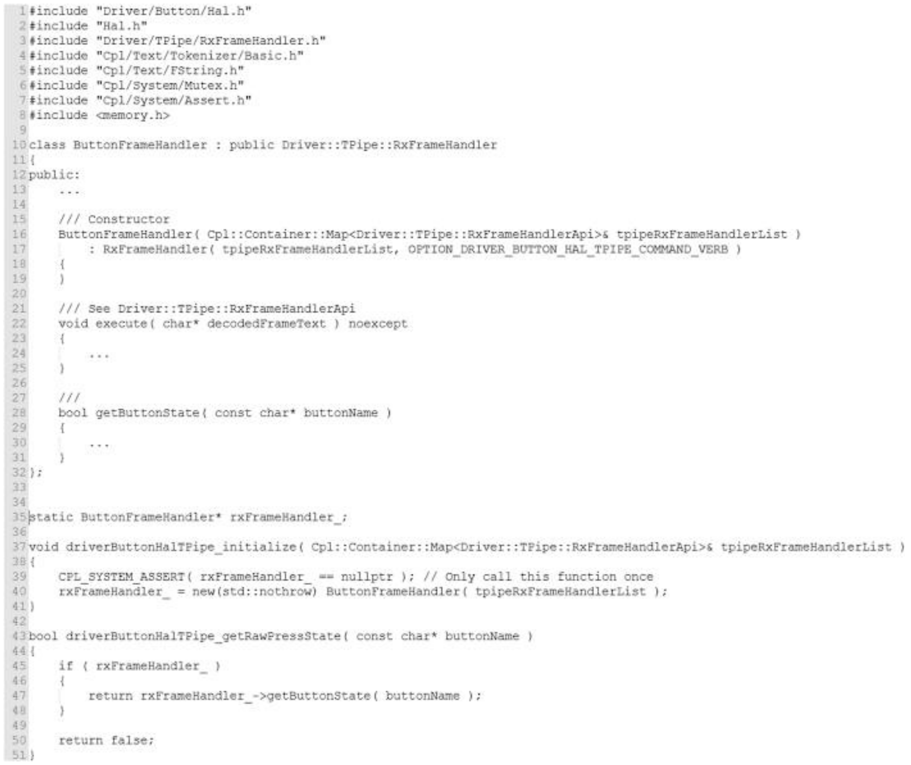


```
1 #ifndef Driver_Button_TPipe_mappings_x_h_
2 #define Driver_Button_TPipe_mappings_x_h_
3
4 /// TPipe Mapping
5 #define Driver_Button_Pin_Hal_T_MAP                const char*
6
7 /// TPipe Mapping
8 #define Driver_Button_Hal_getRawPressedState_MAP    driverButtonHalTPipe_getRawPressState
9
10 /// TPipe specific implementation for getting the raw button state
11 bool driverButtonHalTPipe_getRawPressState( const char* buttonName );
12
13 #endif  // end header latch
```

Figure 15-18. *TPipe LHeader mappings for the button driver*

The `src/Driver/Button/TPipe/Hal.cpp` file implements the HAL interface for the simulator platform. Note the following lines in Figure 15-19.

- Line 1 includes the abstract HAL interface definition.
- Line 2 includes the TPipe button driver interface definition.
- Lines 10–32 implement the required TPipe child class for the `Driver::TPipe::RxFrameHandlerApi` interface.
- Line 35 is used to store a pointer to the single instance of class defined in lines 10–32.
- Lines 37–41 are the implementation of the TPipe method defined in the header file from line 2.
- Lines 43–51 comprise the implementation of the abstract HAL interface.



```
#include "Driver/Button/Hal.h"
#include "Hal.h"
#include "Driver/TPipe/RxFrameHandler.h"
#include "Cpl/Text/Tokenizer/Basic.h"
#include "Cpl/Text/FString.h"
#include "Cpl/System/Mutex.h"
#include "Cpl/System/Assert.h"
#include <memory.h>

class ButtonFrameHandler : public Driver::TPipe::RxFrameHandler
{
public:
    ...

    /// Constructor
    ButtonFrameHandler( Cpl::Container::Map<Driver::TPipe::RxFrameHandlerApi>& tpipeRxFrameHandlerList )
        : RxFrameHandler( tpipeRxFrameHandlerList, OPTION_DRIVER_BUTTON_HAL_TPIPE_COMMAND_VERB )
    {
    }

    /// See Driver::TPipe::RxFrameHandlerApi
    void execute( char* decodedFrameText ) noexcept
    {
        ...
    }

    ///
    bool getButtonState( const char* buttonName )
    {
        ...
    }
};


static ButtonFrameHandler* rxFrameHandler_;

void driverButtonHalTPipe_initialize( Cpl::Container::Map<Driver::TPipe::RxFrameHandlerApi>& tpipeRxFrameHandlerList )
{
    CPL_SYSTEM_ASSERT( rxFrameHandler_ == nullptr ); // Only call this function once
    rxFrameHandler_ = new(std::nothrow) ButtonFrameHandler( tpipeRxFrameHandlerList );
}

bool driverButtonHalTPipe_getRawPressState( const char* buttonName )
{
    if ( rxFrameHandler_ )
    {
        return rxFrameHandler_->getButtonState( buttonName );
    }

    return false;
}
```

Figure 15-19. *TPipe HAL button driver implementation*

LHeader Caveats

At this point, I should note that the LHeader pattern has an implementation weakness. It breaks down in situations where interface A defers a type definition using the LHeader pattern, and interface B also defers a type definition using the LHeader pattern, and interface B has a dependency on interface A. This use case results in a “cyclic header include scenario,” and the compile will fail. This problem can be solved by adding an additional header latch using the header latch symbol defined in the HAL header file to the platform-specific mappings header file. Figure 15-12 shows an example of the additional header latch from STM32 mapping header file for the Button driver, and Appendix D, "LHeader and LConfig Patterns," provides additional details.

Unit Testing

All drivers should have unit tests. These unit tests are typically manual unit tests because they are built and executed on a hardware target.[2] One advantage of using the separation of concerns paradigm for a driver is that you can write automated unit tests that run as part of CI builds. For example, with the Button driver, there are two separate unit tests:

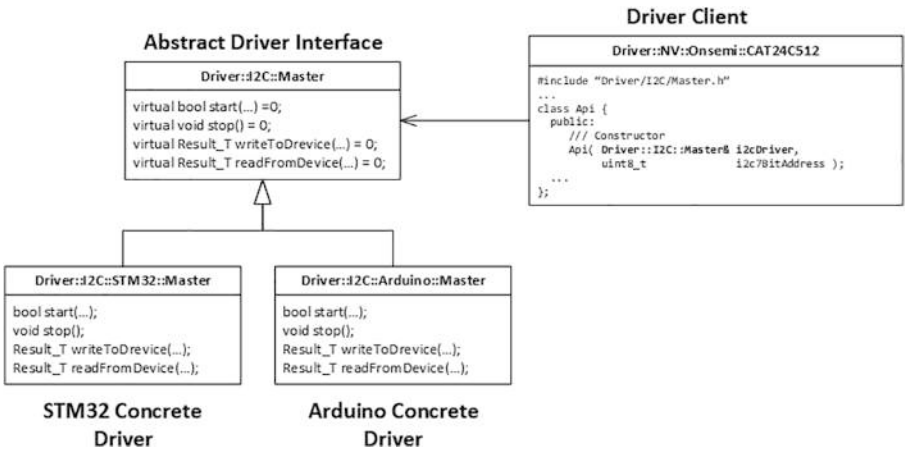
- One test is designed as a manual test and runs on target hardware (see `src/driver/Button/_0test/_hw/test.cpp`). In addition, the same manual unit test can run on multiple hardware targets. That is, each hardware target used for testing has its own unit test project that shares common unit test code.
- The second test is designed as an automated unit and runs as a command-line executable under Windows and Linux (see `src/driver/Button/_0test/polleddebounced.cpp`). The automated unit test mocks the hardware by having a Boolean array hold the button state.

[2] In my experience, test automation involving target hardware is an exception because of the effort and expenses involved.

Polymorphism

A polymorphic design is similar to a facade design, except that it uses run-time bindings for selecting the concrete hardware-specific implementation. A polymorphic design is best suited when using C++.[3] The I2C driver is an example of a polymorphic Hardware Abstraction Layer. It encapsulates an I2C data transfer from an I2C master to an I2C slave device.

Figure 15-20 is a class diagram of I2C driver abstract interface and concrete child classes.


Abstract Driver Interface
Driver::I2C::Master
virtual bool start(...) =0;
virtual void stop() = 0;
virtual Result_T writeToDrevice(...) = 0;
virtual Result_T readFromDevice(...) = 0;
Driver Client
Driver::NV::Onsemi::CAT24C512
Driver::I2C::STM32::Master
bool start(...);
void stop();
Result_T writeToDrevice(...);
Result_T readFromDevice(...);
Driver::I2C::Arduino::Master
bool start(...);
void stop();
Result_T writeToDrevice(...);
Result_T readFromDevice(...);
STM32 Concrete Driver
Arduino Concrete Driver

```
#include "Driver/I2C/Master.h"
...
class Api {
  public:
     /// Constructor
     Api( Driver::I2C::Master& i2cDriver,
          uint8_t              i2c7BitAddress );
  ...
};
```

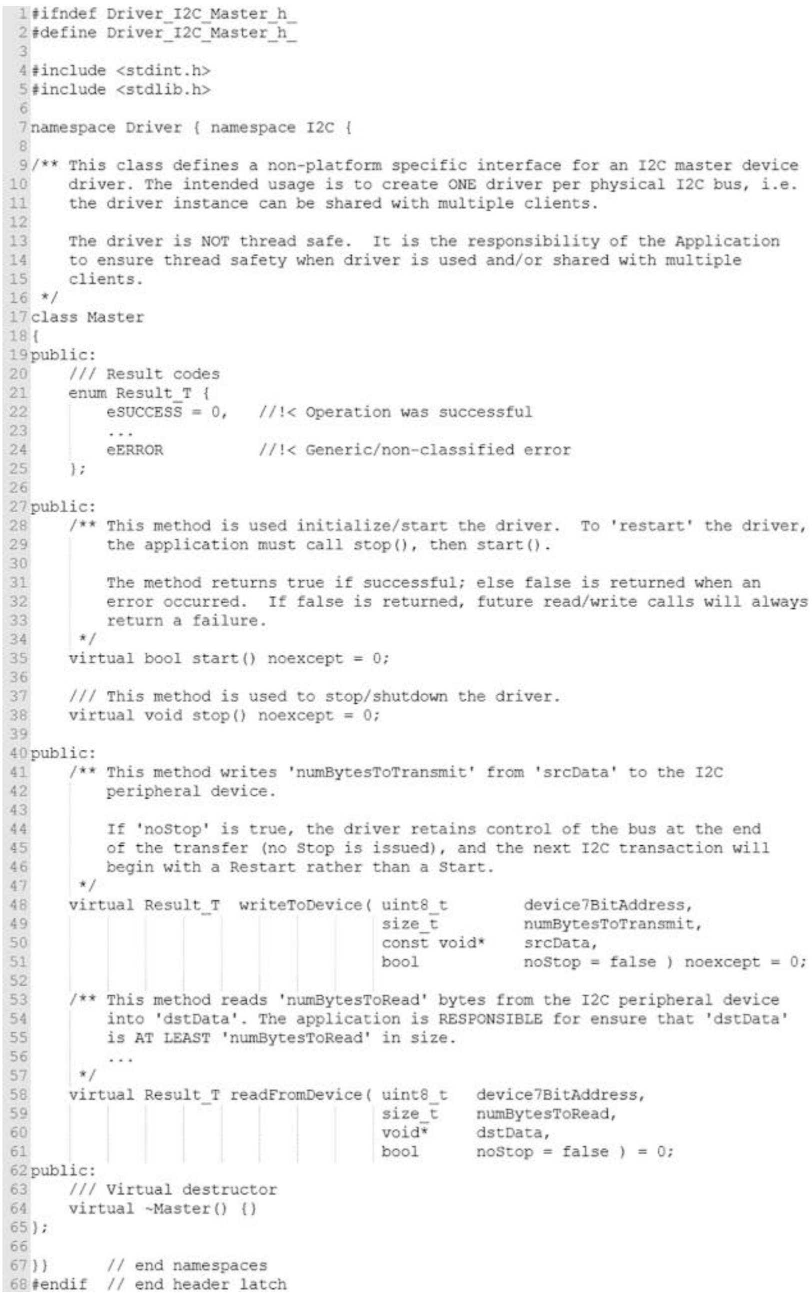
Figure 15-20. *I2C driver class diagram*

[3] It is possible to implement run-time polymorphism in C. For example, the original CFront C++ compiler translated the C++ source code into C code and then passed it to the C compiler. I recommend that you use C++ instead of hand-crafting the equivalent of the vtables in C.

There is no simulated implementation of the I2C driver in the CPL class library. The reason is because the I2C driver and the SPI driver are intermediary drivers. That is, the other drivers (like the EEPROM driver) are the immediate clients of the I2C/SPI drivers. The client drivers are what need to be simulated.

Example I2C Driver

The public interface for the I2C driver is the abstract C++ class `Driver::I2C::Master` in file `src/Driver/I2C/Master.h`. Figure 15-21 shows the abstract class definition.



```
#ifndef Driver_I2C_Master_h_
#define Driver_I2C_Master_h_

#include <stdint.h>
#include <stdlib.h>

namespace Driver { namespace I2C {

/** This class defines a non-platform specific interface for an I2C master device
    driver. The intended usage is to create ONE driver per physical I2C bus, i.e.
    the driver instance can be shared with multiple clients.

    The driver is NOT thread safe.  It is the responsibility of the Application
    to ensure thread safety when driver is used and/or shared with multiple
    clients.
 */
class Master
{
public:
    /// Result codes
    enum Result_T {
        eSUCCESS = 0,   //!< Operation was successful
        ...
        eERROR          //!< Generic/non-classified error
    };

public:
    /** This method is used initialize/start the driver.  To 'restart' the driver,
        the application must call stop(), then start().

        The method returns true if successful; else false is returned when an
        error occurred.  If false is returned, future read/write calls will always
        return a failure.
     */
    virtual bool start() noexcept = 0;

    /// This method is used to stop/shutdown the driver.
    virtual void stop() noexcept = 0;

public:
    /** This method writes 'numBytesToTransmit' from 'srcData' to the I2C
        peripheral device.

        If 'noStop' is true, the driver retains control of the bus at the end
        of the transfer (no Stop is issued), and the next I2C transaction will
        begin with a Restart rather than a Start.
     */
    virtual Result_T  writeToDevice( uint8_t        device7BitAddress,
                                     size_t         numBytesToTransmit,
                                     const void*    srcData,
                                     bool           noStop = false ) noexcept = 0;

    /** This method reads 'numBytesToRead' bytes from the I2C peripheral device
        into 'dstData'. The application is RESPONSIBLE for ensure that 'dstData'
        is AT LEAST 'numBytesToRead' in size.
        ...
     */
    virtual Result_T readFromDevice( uint8_t   device7BitAddress,
                                     size_t    numBytesToRead,
                                     void*     dstData,
                                     bool      noStop = false ) = 0;
public:
    /// Virtual destructor
    virtual ~Master() {}
};

})      // end namespaces
#endif  // end header latch
```

Figure 15-21. *Partial file listing of src/Driver/I2C/Master.h*

Each supported hardware platform requires its own concrete child class of the `Driver::I2C::Master` class shown in Figure 15-21. In addition to providing the implementation for the pure virtual methods, the child class contains platform-specific methods, data types, etc. This means that instances of the driver are created in platform-specific code on startup (see Chapter 7, "Building Applications with the Main Pattern").

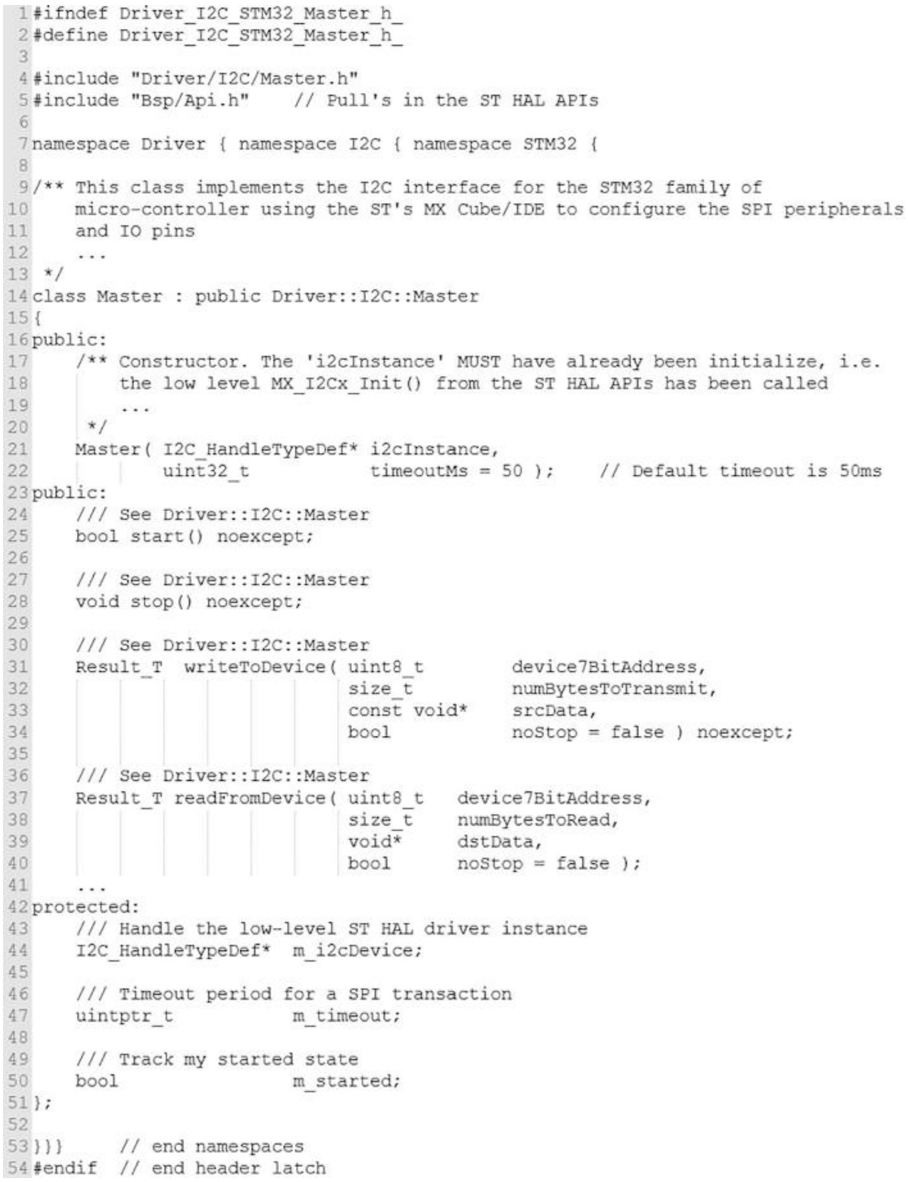
In the following two sections, you will see the implementation of the I2C driver functionality for each of these platforms:

- STM32—See Figure 15-22.
- Arduino—See Figure 15-23.

STM32 Implementation of the I2C Driver

The implementation relies on the ST HAL library and initialization code generated by the ST Cube MX IDE. The implementation is found in the files `src/Driver/I2C/STM32/Master.h` and `src/Driver/I2C/STM32/Master.cpp`. Figure 15-22 shows a snippet from the `Master.h` file. Note the following:

- Line 4 includes the abstract I2C class to be implemented.
- Line 5 is another LHeader pattern usage that allows the STM32 I2C driver to work with any STM32-based BSP.



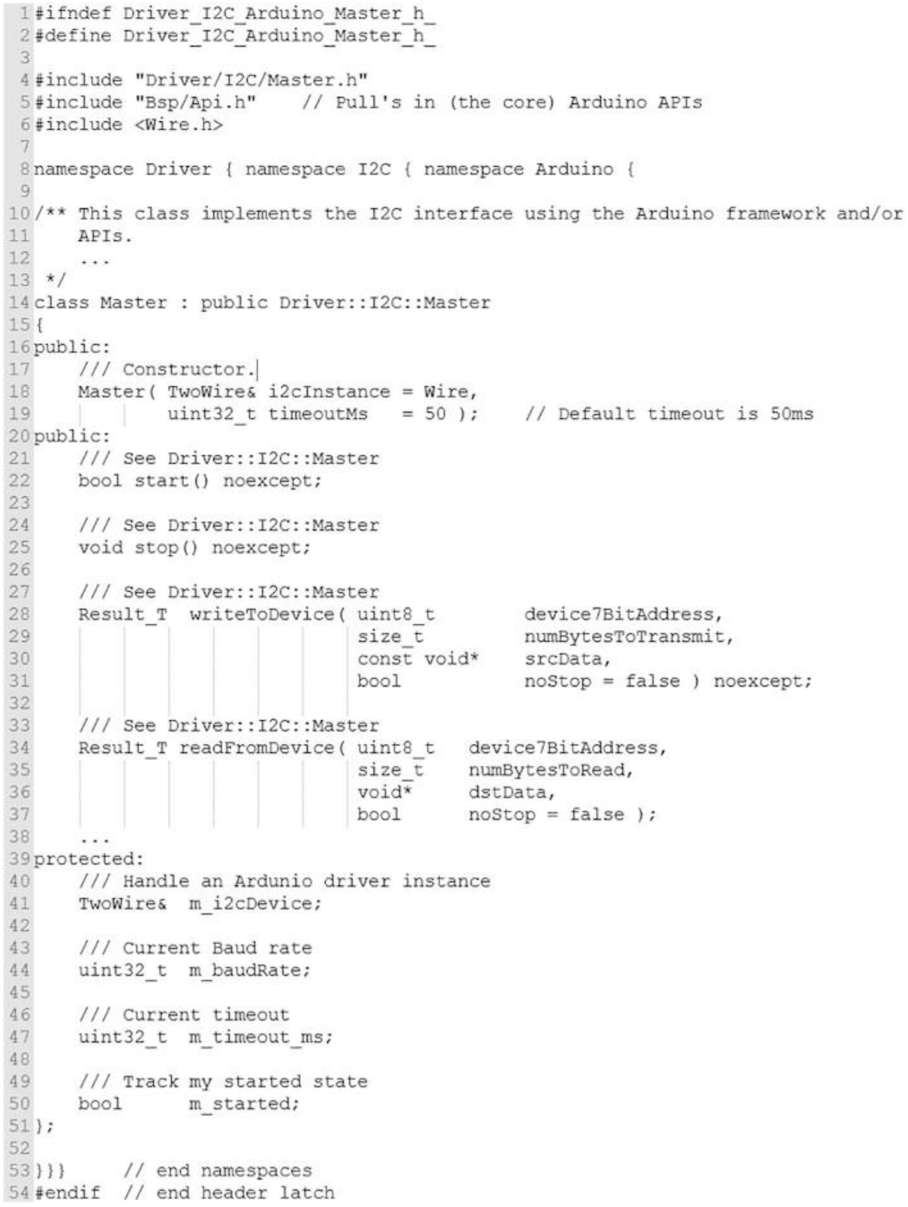
```
#ifndef Driver_I2C_STM32_Master_h_
#define Driver_I2C_STM32_Master_h_

#include "Driver/I2C/Master.h"
#include "Bsp/Api.h"    // Pull's in the ST HAL APIs

namespace Driver { namespace I2C { namespace STM32 {

/** This class implements the I2C interface for the STM32 family of
    micro-controller using the ST's MX Cube/IDE to configure the SPI peripherals
    and IO pins
    ...
 */
class Master : public Driver::I2C::Master
{
public:
    /** Constructor. The 'i2cInstance' MUST have already been initialize, i.e.
        the low level MX_I2Cx_Init() from the ST HAL APIs has been called
        ...
     */
    Master( I2C_HandleTypeDef* i2cInstance,
            uint32_t           timeoutMs = 50 );    // Default timeout is 50ms
public:
    /// See Driver::I2C::Master
    bool start() noexcept;

    /// See Driver::I2C::Master
    void stop() noexcept;

    /// See Driver::I2C::Master
    Result_T  writeToDevice( uint8_t        device7BitAddress,
                             size_t         numBytesToTransmit,
                             const void*    srcData,
                             bool           noStop = false ) noexcept;

    /// See Driver::I2C::Master
    Result_T readFromDevice( uint8_t   device7BitAddress,
                             size_t    numBytesToRead,
                             void*     dstData,
                             bool      noStop = false );
    ...
protected:
    /// Handle the low-level ST HAL driver instance
    I2C_HandleTypeDef*  m_i2cDevice;

    /// Timeout period for a SPI transaction
    uintptr_t           m_timeout;

    /// Track my started state
    bool                m_started;
};

}}}     // end namespaces
#endif  // end header latch
```

Figure 15-22. *Partial file listing of src/Driver/I2C/STM32/Master.h*

Arduino Framework Implementation of the I2C Driver

The implementation is built on top of the Arduino framework. The implementation is located in `src/Driver/I2C/Arduino/Master.h` and `src/Driver/I2C/Arduino/Master.cpp`. Figure 15-23 shows a snippet from the `Master.h` file. Note the following:

- Line 4 includes the abstract I2C class to be implemented.
- Line 5 is another LHeader pattern usage that allows the Arduino framework I2C driver to work with any Arduino-based BSP.
- Line 6 includes the Arduino I2C driver.



```
#ifndef Driver_I2C_Arduino_Master_h_
#define Driver_I2C_Arduino_Master_h_

#include "Driver/I2C/Master.h"
#include "Bsp/Api.h"    // Pull's in (the core) Arduino APIs
#include <Wire.h>

namespace Driver { namespace I2C { namespace Arduino {

/** This class implements the I2C interface using the Arduino framework and/or
    APIs.
    ...
 */
class Master : public Driver::I2C::Master
{
public:
    /// Constructor.
    Master( TwoWire& i2cInstance = Wire,
            uint32_t timeoutMs   = 50 );    // Default timeout is 50ms
public:
    /// See Driver::I2C::Master
    bool start() noexcept;

    /// See Driver::I2C::Master
    void stop() noexcept;

    /// See Driver::I2C::Master
    Result_T  writeToDevice( uint8_t        device7BitAddress,
                             size_t         numBytesToTransmit,
                             const void*    srcData,
                             bool           noStop = false ) noexcept;

    /// See Driver::I2C::Master
    Result_T readFromDevice( uint8_t   device7BitAddress,
                             size_t    numBytesToRead,
                             void*     dstData,
                             bool      noStop = false );
    ...
protected:
    /// Handle an Ardunio driver instance
    TwoWire&  m_i2cDevice;

    /// Current Baud rate
    uint32_t  m_baudRate;

    /// Current timeout
    uint32_t  m_timeout_ms;

    /// Track my started state
    bool      m_started;
};

}}}     // end namespaces
#endif  // end header latch
```

Figure 15-23. *Partial listing of src/Driver/I2C/Arduino/Master.h*

Dos and Don’ts

There is no silver bullet when it comes to driver design. That said, I recommend the following best practices:

- When designing a new driver, base your design on your current hardware platform, BSP, and application needs. This minimizes the amount of glue logic that is needed for the current project. It also avoids having to “guesstimate” the nuances of future hardware platforms and requirements. Don’t spend a lot of effort in coming up with an all-encompassing interface or HAL definition. Don’t overdesign your driver, and don’t include functionality that is not needed by the application. For example, if your application needs a UART driver that never needs to send or receive a break character,[4] do not include logic for supporting break characters in your UART driver.
- Always include a `start()` or `initialize()` function even if your initial implementation does not require it. In my experience, as a driver and the application mature, you typically end up needing some initialization logic. I also recommend always including a `stop()` or `shutdown()` function as well.
- If you have experience writing the driver on other platforms, leverage that knowledge into current driver design.

[4] A break character consists of all zeros and must persist for a minimum of 11bit times before the next character is received. A break character can be used as an out-of-band signal, for example, to signal the beginning of a data packet.

- Don't optimize until you have to. A product with a simpler driver that has good-enough performance and ships on time is a very good thing. For example, in the GM6000 example, the space or room temperature changes slowly (e.g., seconds to minutes). A polled ADC driver, then, is sufficient; there is no need to build an interrupt-driven ADC driver with KHz sampling.
- Separate creation from usage when it comes to driver interfaces. That is, have one header file (or interface) for creating the driver and a second header file that contains the runtime functionality of the driver. Creating a driver instance is almost always platform specific, which means the code that creates the driver cannot be reused on other platforms. Nevertheless, in embedded applications, drivers are only created once on startup, so reuse is not as important. However, there is a lot of value in being able to reuse the run-time behavior of a driver (e.g., sending/receiving data over an I2C bus). The return on investment for decoupling the run-time behavior of a driver from the underlying hardware platform is high; decoupling the creation of driver is typically low. Separate interfaces make it possible to skip the effort of implementing decoupled driver creation.
- Do not create a single, unified, all-encompassing HAL interface. HAL interfaces should be created on a per-driver basis. A fat HAL interface definition imposes an all or nothing restriction when attempting to reuse the drivers that rely on the fat HAL interface.

- It is better to have many different drivers that do similar tasks, instead of a single all-purpose, all-encompassing driver. For example, I created the `PolledDebounced` button driver for the example code because that is all the application required, and it was simple. However, when presented with more complex button requirements like a 3 x 4 button array, the `PolledDebounced` button driver should not be used. Instead, a new button driver should be created that performs row and column GPIO multiplexing for the button array. Also, by creating a separate driver, you don't have to worry about breaking the existing products that use the `PolledDebounced` driver.

Summary

A decoupled driver design allows reuse of drivers across multiple hardware platforms. A driver is decoupled from the hardware by creating a Hardware Abstraction Layer (HAL) for its run-time usage. The following late binding strategies can be used for the construction of the HAL interfaces:

- Facade—Using link-time bindings
- Separation of concerns—Using compile-time bindings
- Polymorphic—Using run-time bindings

Additional benefits of decouple driver design are as follows:

- The HAL interface facilitates simulating or mocking drivers when building a functional simulator.
- It allows the construction of automated unit tests that can be run as part of the CI build without having hardware automation.

INPUTS

- All documented requirements including ongoing updates
- Software Development Plan (SDP) including ongoing updates
- System architecture (SA) document including ongoing updates
- Software architecture (SWA) document including ongoing updates
- Software Detailed Design (SDD) document including ongoing updates

OUTPUTS

- Working driver code
- Unit tests
- Design reviews
- Code reviews
- Potential updates to requirements that may be in MRS, PRS, or SRS documents
- Potential updates to the system architecture (SWA) document
- Potential updates to the software architecture (SA) document
- Updates to the Software Detailed Design document
- Updates to design statements when used

CHAPTER 16

Release

For most of the construction stage, you are focused on writing and testing and optimizing your software. And as the product gets better and better and the bugs get fewer and fewer, you start to have the sense that you're almost done. However, even when you finally reach the point when you can say "the software is done," there are still some last-mile tasks that need to be completed in order to get the software to your customers. These mostly not-coding-related tasks comprise the release activities of the project.

The release stage of the project overlaps the end-of-construction stage. That is, about the time you start thinking about creating a release candidate, you should be starting your release activities. Ideally, if you've followed the steps in this cookbook, your release activities will simply involve collecting, reviewing, and finalizing reports and documentation that you already have. If not, you'll get to experience the angst and drama of a poorly planned release: fighting feature creep, trying to locate licenses, fighting through installation and update issues, etc. If you find yourself in the position of struggling with the logistics of releasing your software, it means you probably "cheated" during the planning and construction stages and built up technical debt that now has to be retired before shipping the software. My recommendation is if you are working with an Agile methodology, you practice "releasing" the software at the end of each sprint. That is, when you estimate you have about three sprints left to finish the project, go through all the release activities as if you were going to release the product.

The work items for the release stage are as follows:

- Tightening up change management—Your change management process needs to prevent unnecessary changes from being made as you work toward the gold release. The goal is to get to a "code freeze," where there are no more changes because every change requires more testing and has the potential to introduce new bugs.
- Generating a Software Bill of Materials—The Software Bill of Materials (SBOM) is a list of all third-party packages, including licensing and version information, that are contained with the released software. The SBOM is used to verify that software licensing of all the packages is "good to go." The SBOM is also used to track Common Vulnerabilities and Exposures[1] (CVEs) for the packages after the release.
- Creating an anomalies list—The anomalies list is an exhaustive list of known bugs and issues for a release. Generally, this is a development document, and it is the tool that the team and product stakeholders use to make the ship-don't-ship decision.
- Generating release notes—These may take the form of internal release notes or customer release notes. Release notes may use the anomalies list as input but usually add additional details—especially about workarounds that may be available. Internal release notes are created on a regular basis. For example, every

[1] The US National Institute of Standards and Technology (NIST) department maintains a database of known CVEs. See https://nvd.nist.gov/vuln

formal CI build that is handed off to QA for testing benefits from internal release notes. Customer release notes, however, only need to be generated when a release is deployed as an early-access release (e.g., Alpha or Beta release) or as a general availability (GA) release to the end customer. Customer release notes are oftentimes derived from internal release notes, but they are usually edited for simplicity as well as to "properly position sensitive limitations of the software." In other words, you generally want to edit your dirty laundry out of the customer release notes.

- Deploying the application—The deployment step includes making the software images available to manufacturing who will use them when physically assembling the product. Additionally, these images may be placed on servers that are responsible for providing over-the-air (OTA) updates to previously sold products.
- Completing QMS deliverables—Depending on your QMS process, there can be many non-software artifacts that must be completed with each gold release (e.g., requirements tracing documentation, design documentation, design and code review artifacts, etc.)
- Archiving the build tools—This involves archiving all the components that were used to build the release software and archiving them so that, if necessary, an identical version of the release can be built at a later time.

While the work involved in the release stage of a project is not particularly complex or difficult, it can still take a substantial amount of time to perform it. Make sure you have allocated resources and time for all the release activities in your project schedule—especially the tasks that occur after the product has technically shipped.

About Builds and Releases

Not every build gets released, but every release is a build—it just gets gussied up a bit. That being said, it is helpful to establish a common understanding with stakeholders about what constitutes different builds and releases.

Generally, there are three types of builds:

- Private build—A build that is performed on a developer's build machine. The build scripts for a developer build are responsible for setting the build number to zero to identify that it is not a formal build. In addition, the provenance of actual source code used for a private build is assumed to be unknown in that it may or may not have been locally edited prior to the actual build. Consequently, private builds should never be used for any formal testing or design verification. There can be hundreds of private builds floating around in different development environments and on different team member's machines.

- CI build—A build performed on the build server for each pull request. Because there can be many CI builds per single pull request (e.g., a build needs to be fixed or a test error needs to be corrected or a change is required because of code review feedback), CI builds should not be consumed outside of the CI environment. That is, if someone needs to use a CI build, even for a quick and dirty test, the CI build should be turned into a formal build. The build script or build process for a CI build is also responsible for setting the build number to zero to identify that it is not a formal build.

- Formal build—A build performed on the build server of a stable branch that has a canonical build number. Furthermore, an SCM tag or label should be created to identify all of the source code used for the build. Formal builds are deployable in the sense that they have known (and documented) provenance, which is critical for logging bugs against the build. Without this formality, any bugs identified could be false positives in that the bugs could be a result of changes that were rejected during the pull request and code review process. Whether or not a formal build can be a release candidate is dependent on your branching strategy for stable branches.

When discussing releases, all release builds must be formal builds and have a human-friendly version identifier. Generally, there are four different types of releases:

- Working release—A formal build that is deployed to the test team during the construction stage for test case development and early verification. Working releases can also be deployed externally for activities such EMC (Electromagnetic Compatibility) and regulatory testing. Which SCM branch a working release is built from is dependent on your branching strategy. For the GM6000 example, working releases are built from the `develop` branch.

- Early-access release (Alpha, Beta Release)—A formal build that is deployed to a select, and restricted, number of customers or field sites. These builds are typically not feature complete and have undergone less change control rigor than a candidate release. What SCM branch early-access releases are built from is dependent on your branching strategy. For the GM6000 example, early-access releases are built from the `main` branch.
- Candidate release—A formal build that is feature complete. Additionally, there is an expectation that the release can successfully pass all formal testing and verification. Which SCM branch candidate releases are built from is dependent on your branching strategy. For the GM6000 example, candidate releases are built from the `main` branch.
- Gold Release—A gold release is the candidate release that has been declared ready to ship by the stakeholders.

A human-readable version number (e.g., 1.4.0) is not used as the canonical identifier for releases. This is because at build time, it is unknown whether or not the release can be deployed to a customer either as an early-access release or a gold release. This is why the build number is used as the canonical version identifier. The human-readable version number should only be changed after early-access or GA releases are deployed.

Tightening Up the Change Control Process

During the construction stage, the software team is largely left alone to determine when and what work, features, and changes are made to the code base. However, as the project enters the release stage of the project, unbounded changes become counterproductive since every change has the potential to introduce new bugs and trigger additional regression testing. To limit the number of changes and to focus the work on "only what is needed" for release, you need to define a formal, well-advertised process for how all forthcoming code changes will be approved. Typically, this takes the form of establishing a Change Control Board (CCB) that is made up of a limited number of stakeholders. CCB meets regularly to review the state of the software along with the anomalies list (i.e., the known bug list) in order to approve code changes.

The CCB is also responsible for reviewing and approving the scope of any regression testing activities that are needed as a result of source code changes. Or said another way, when approving changes, the CCB needs to consider what the impact every change will have on the testing and verification efforts.

In my experience, how strict the CCB is with respect to approving changes varies over time. When the CCB is first put in place, the board typically accepts all of the recommendations from the software lead and the project manager on what should be changed. As the project approaches its first candidate release, the CCB becomes more conservative with respect to approving recommendations. Eventually, the CCB evolves into a single stakeholder who approves the changes and who ultimately will designate a candidate release as a gold release.

Obviously, it is best to define the CCB process before your project gets to the release stage because de facto CCB processes form organically. There are always competing must-have priorities, bug fixes, feature creep, etc., across all the stakeholders, which results in there always being one more software change request. The CCB process (formal or informal) provides the discipline to stop changes so the software can finally ship.

There is an assumption that when the project enters the release stage, any work for the next release or next product variant is isolated from the code that is getting ready to ship. Typically, this is done by having separate branches in the SCM repository for the upcoming release and the follow-on release. After the current release is completed, the source code from the gold release is reconciled with the follow-on release.

Software Bill of Materials (SBOM)

When I first started as an embedded developer many decades ago, it was rare to have any third-party software items included in a product's code base. The exception was usually a minimal amount of software provided by the microcontroller vendor. In today's environment, it is rare to not include third-party packages. In fact, I was surprised to see, for example, that for the GM6000 project there were nine third-party packages used in the GM6000 code base.

The Software Bill of Materials (SBOM), then, is used to identify all third-party software included in the released product. For each item in the SBOM, the item's software licensing and version information is captured. The software licensing information is needed to ensure that all conditions for the item's license have been met (e.g., complying with GPL requirements or having purchased the required licenses). The version information from the SBOM is used when monitoring the item for bugs and CVEs. Additionally, there can be other potential concerns with respect to third-party software such as export restrictions on encryption technology.

It should be obvious that the SBOM should be created as you go. That is, the process should start when external packages are first incorporated into the code base, not as a "paperwork" item during the release stage. For non-open-source licensing, purchasing software licenses takes time and money, and you need to make sure that money for those purchases is budgeted. Your company's legal department needs to weigh in on what is or is not acceptable with respect to proprietary and open source licenses. Or, said another way, make sure you know, and preferably have some documentation on, your company's software licensing policies. Don't assume something is okay because it is widely used in or around your company because nothing is as frustrating as the last-minute scramble to redesign (and retest) your application because one of your packages has unacceptable licensing terms.

The SBOM is not difficult to create since it is essentially just a table that identifies which third-party packages are used along with their pertinent information. My recommendations for creating and maintaining the SBOM are as follows:

- Initially create the SBOM as a living document (e.g., as a Wiki page) and update it as third-party packages are added to the code base.
- As entries are added to the SBOM, have the new items reviewed by the appropriate stakeholders (e.g., legal team, your manager, the project manager, etc.) to ensure that the licensing terms are acceptable, and that money is budgeted for purchasing the license.
- Depending on your QMS process, you may need to convert the working SBOM into a formal document prior to completing the release stage. See Appendix Q, "GM6000 Software Bill of Materials," for an example of a formal SBOM document.

Anomalies List

By definition, your bug tracking tool is the canonical source for all known defects. The anomalies list is simply a snapshot in time of the known defects and issues for a given release. In theory, the anomalies list for any given release could be extracted from the bug tracking tool, but having an explicit document that enumerates all of the known defects for a given release simplifies communications within the cross-functional team and with the leadership team. When an anomalies list needs to be generated should be defined by your QMS process; however, as it is a key tool in determining if a release is ready for early access or GA, you may find yourself generating the list more frequently at the end of the project.

Release Notes

Internal release notes should be generated every time a formal build is deployed to anyone outside of the immediate software team. This means you need release notes for formal builds that go to testers, hardware engineers, QA, etc. Simply put, the internal release summarizes what is and what is not the release. It is important to remember that when writing the internal release notes, the target audience is everyone on the team—not just the software developers. The software lead or the "build master" is typically responsible for generating the internal release notes.

There should be a line or bullet item for every change (from the previous release) that is included in the current release. There should be an item for every pull request that was merged into the release branch. Internal release notes can be as simple as enumerating all of the work item tickets and bug fixes—preferably with titles and hyperlinks—that went into the release. Remember, if you are following the cookbook, there will be a work item or bug tickets created for each pull request. One advantage of referencing work or bug tickets is that individual tickets will contain additional information, context, comments, steps to reproduce, etc., which means all that detail can stay there and not be included in the release notes.

In addition to the changes and bug fixes, the release notes should include a section about "known issues" that includes items such as

- Missing or incomplete functionality
- Performance issues
- Workarounds for existing bugs

A comprehensive and verbose list of known issues makes for better team communication, and it heads off any premature bug reporting about missing functionality.

Customer-facing release notes are only required when a release (alpha, beta, or gold) is deployed to an end customer. These release notes are usually derived from the internal release notes, but they are simplified and sanitized. However, depending on your relationship with your customers or partners, the customer release notes for alpha and beta releases may just be the internal release notes.

Deployment

In most cases, the embedded software you create is sold with your company's custom hardware. This means that deploying your software requires making the images available to your company's manufacturing process. Companies that manufacture physical products typically track customer-facing releases in a Product Lifecycle Management (PLM) tool such as Windchill or SAP that is used to manage all of the drawings and bill of materials for the hardware. With respect to embedded software, the software images and their respective source files are bill-of-material line items or sub-assemblies tracked by the PLM tool. Usually, the PLM tool contains an electronic copy of the release files. As PLM tools have a very strict process for adding or updating material items, the process discourages frequent updates, so you don't want to put every working release or candidate release into the PLM system—just the alpha, beta, and gold releases.

These processes are company and PLM tool specific. For example, at one company I worked for, the PLM tool was Windchill, but since the software images were zipped snapshots of the source code (i.e., very large binary files), the electronic copies of the files were formally stored in a different system and only a cover sheet referencing the storage location was placed in Windchill.

Typically, the following information is required for each release into the PLM system:

- A brief summary or description of the release
- An electronic copy of the software images with a checksum (MD5 or SHA512) for each item
- Version identifiers for all software images
- The SCM tag or label associated with the release build
- A snapshot of all the source code files that make up the release (with an MD5 checksum for the compressed file)
- (Optional) Build instructions for release
- (Optional) Programming instructions for the manufacturing process

Over-the-Air (OTA) Updates

In addition to deployment of images to manufacturing, you may be able to update hardware in the field through a variety of methods. The methods used by field engineers will be similar to those used by engineering and

QA to test the product. However, if customers will initiate and deploy the update, you will need to have developed, tested, and documented a straightforward process for accomplishing this. Typically, anything requiring more than connecting a standard USB cable to a computer to perform the update is a bad idea for customer deployment.

With the right network connectivity (Wi-Fi or cellular), you might also deploy updates using over-the-air (OTA) strategies. For this scenario, there is the additional step of deploying the software images to a server that is responsible for delivering the OTA updates to products in the field. This includes the code that instructs the hardware how to swap in, use, and then load the new image. Regardless of how your software updates itself with a new image, the process should have been included in the definition and design of the product.

The same care and due diligence that went into updating the PLM system for manufacturing with a new release should be applied when releasing images to the OTA server. The last thing you want is a self-made crisis of releasing the wrong or bad software to the field. Since almost all embedded software interacts with the physical world, a bad release can have negative real-world consequences for a period of time before a fixed release can be deployed (e.g., no heating for an entire building for days or weeks in the middle of winter). Or, worst case, the OTA release “bricks” the hardware, requiring field service personnel to resurrect the device.

QMS Deliverables

While the Quality Management System (QMS) deliverables do not technically gate the building and testing of a release, the required processes can delay shipping the software to customers. What is involved in the QMS deliverables is obviously specific to your company's defined QMS processes. On one end of the spectrum, there are startup companies that have no QMS processes, and all that matters is shipping the software. On the other end, there are the regulated industries, such as medical devices, that will have quite verbose QMS processes. If you have no, or minimal, QMS processes defined, I recommend the following process be followed and the following artifacts be generated for each gold release.

1. Finalize and publish final drafts of the following documents:

    - Software Development Plan (SDP)
    - Formal requirements (MRS, PRS, SRS, etc.)
    - System architecture
    - Software architecture
    - Software Detailed Design
    - Software coding standards
    - Developer environment setup
    - Build server setup
    - CI setup

    In theory, if you have been following the cookbook, this should be limited to closing out a few remaining TBDs or action items and should not require a large amount of effort. It should not involve "big-bang" document reviews.

2. Archive existing quality artifacts. Do not go back and attempt to create missing artifacts. Just archive what artifacts you do have. You can note missing artifacts as lessons learned for the next project. Additional quality artifacts you might archive are as follows:
    - Design review artifacts
    - Code review artifacts, where individual documents were created per the review. You can skip this step if you are using a collaborative tool for code reviews (such as GitHub's pull request process) because you can always retrieve or review the code review comments and actions at a later date using the tool.
    - Doxygen output for the release code base (if using Doxygen)
    - Any software integration testing artifacts such as test plans, test cases, and test reports
    - Any formal software testing artifacts such as test plans, test cases, and test reports
3. Archive all items that are release deliverables as described in your Software Development Plan.
4. Create a master index of the locations of your quality documentation and artifacts. Also include links and references to any Wiki pages that were used to support the software development. Be sure to only store the master index, documents, and build artifacts in a single location on shared storage. There should never be confusion about where to find the "one true source" for the aforementioned content.

Archiving Build Tools

The product lifetimes for embedded software applications are often a decade or more. A lot can happen over that time period that could trigger the need to rebuild or patch a previously deployed release, and there is no guarantee that the same compiler, CASE tool, third-party SDK, operating system, etc., that were used for the original release will still be available *N* years later.

For example, while working at an engineering service company several years ago, the company took on a project to update a customer's existing product for hardware end-of-life issues. Part of the work involved making changes to the existing software to support the new hardware components. Because of the age of the product, this required setting up a build environment using Visual Studio 2008, Service Pack 1. Fortunately, in this case, Visual Studio 2008 was still available from Microsoft, albeit through their MSDN subscription program.

If you've followed the steps in this cookbook, archiving the build tools simply means archiving your CI build server after each GA release.[2] Another advantage of archiving the build server is that the host operating system also gets archived. For example, I have worked on projects where the build tools only ran on Windows XP—and this was well after Microsoft had deprecated support for Windows XP.

[2] Archiving a build server can take many forms. For example, if the build server is a VM, then creating a backup or snapshot of VM is sufficient. If the build server is a Docker container, then backing up the Docker file or Docker image is sufficient. For a physical box, creating images of the box (using tools like Ghost, AOMEI, or True Image) could be an option. Just make sure that the tool you use has the ability to restore your image to a machine that has different hardware than the original box.

Summary

The start of the release stage overlaps the end of the construction phase and finishes when a gold release is approved. The release stage has several deliverables in addition to the release software images. Ideally, the bulk of the effort during the release stage is focused on collecting the necessary artifacts and quality documents, and not the logistics and actual work of creating and completing them.

Ensure that you have a functioning Change Control Board in place for the end-game sprints in order to prevent feature creep and never-ending loops of just-one-more-change and to reduce your regression testing efforts.

Finally, put all end-customer release images and source code into your company's PLM system to be included as part of the assembled end product.

INPUTS

- Final source code, that is, the code base, is closed to unauthorized changes
- Current defect list
- Project documentation and artifacts (e.g., requirements documents, Software Development Plan, Software Detailed Design, integration test reports, etc.)

OUTPUTS

- Release notes (both internal and customer facing)
- Software Bill of Material (SBOM)

- Anomalies list (i.e., the list of known defects at the time of release)
- Archived source code. This includes tagging and labeling branches in your SCM repositories as well as storing them in the company's PLM system (when applicable)
- Archived images, executables, etc., that are required for manufacturing. Typically, these are released into the company's PLM system
- Final revisions of the project documentation. This includes items such as the following list. The company's QMS will define the definitive list of documentation and artifacts
    - Requirements and requirements tracing
    - Software Development Plan
    - Software architecture
    - Software Detailed Design
    - Software coding standards
    - Doxygen output
    - Code review artifacts
    - Design review artifacts
    - Integration test plans, test procedures, and test reports
    - Software test plans, test procedures, and test reports
    - Developer environment setup
    - Build server setup
    - CI setup