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.

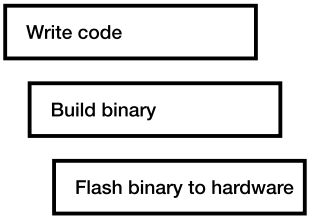

Figure 1-1. *A simple development model for embedded software*

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.

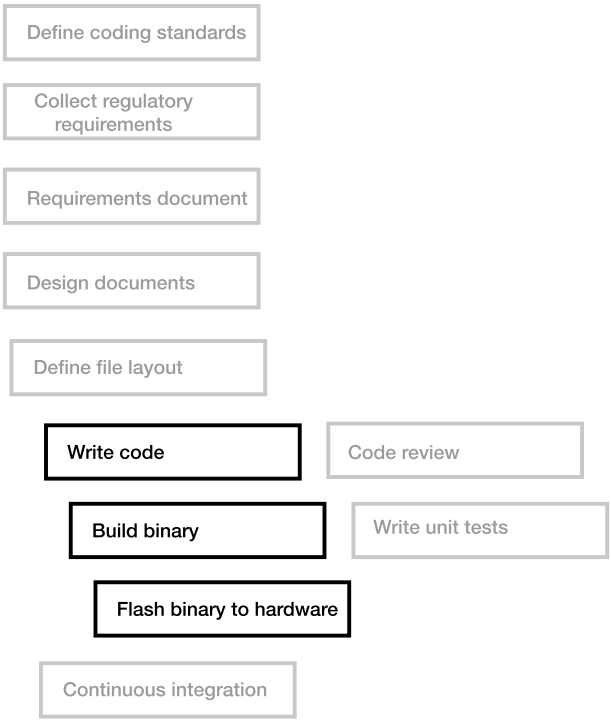

Figure 1-2. *Additional steps and processes for a simple development model*

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.

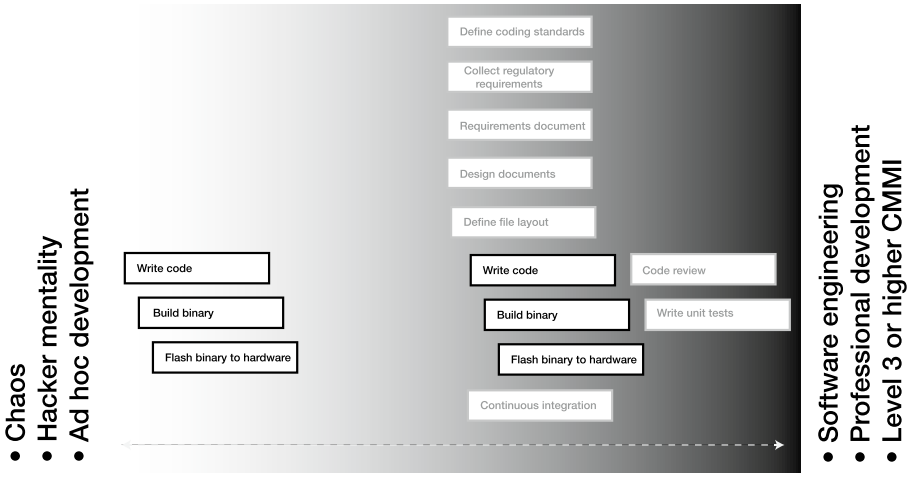

Figure 1-3. *A continuum of software development processes and practices*

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.

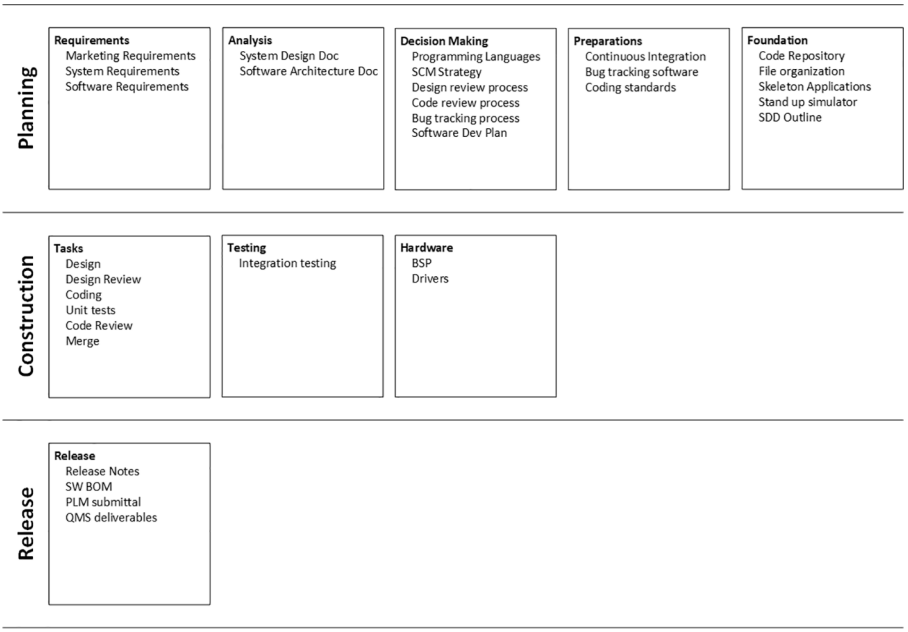

Figure 1-4. *Software development life cycle stages*

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.*

J. T. Taylor and W. T. Taylor, *The Embedded Project Cookbook*,
https://doi.org/10.1007/979-8-8688-0327-7_2

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