



Embedded Software Design

A Practical Approach to Architecture,
Processes, and Coding Techniques

—
Jacob Beningo

Apress®

Embedded Software Design

**A Practical Approach to Architecture,
Processes, and Coding Techniques**

Jacob Beningo

Apress®

Embedded Software Design: A Practical Approach to Architecture, Processes, and Coding Techniques

Jacob Beningo
Linden, MI, USA

ISBN-13 (pbk): 978-1-4842-8278-6
<https://doi.org/10.1007/978-1-4842-8279-3>

ISBN-13 (electronic): 978-1-4842-8279-3

Copyright © 2022 by Jacob Beningo

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: James Markham
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Doodlebug on Clean Public Domain (cleanpublicdomain.com/)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. 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 (<https://github.com/Apress>). For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

To my family, friends, and mentors.

Table of Contents

About the Author	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
Preface	xxi
Part I: Software Architecture and Design	1
Chapter 1: Embedded Software Design Philosophy.....	3
Challenges Facing Embedded Developers	5
7 Modern Design Philosophy Principles.....	9
Principle #1 – Data Dictates Design	10
Principle #2 – There Is No Hardware (Only Data)	11
Principle #3 – KISS the Software.....	13
Principle #4 – Practical, Not Perfect.....	14
Principle #5 – Scalable and Configurable.....	16
Principle #6 – Design Quality in from the Start	17
Principle #7 – Security Is King	18
Harnessing the Design Yin-Yang	20
Traditional Embedded Software Development	21
Modern Embedded Software Development	22
The Age of Modeling, Simulation, and Off-Chip Development.....	26
Final Thoughts.....	27

TABLE OF CONTENTS

- Chapter 2: Embedded Software Architecture Design 29**
 - A Tale of Two Architectures 32
 - Approaches to Architecture Design..... 33
 - Characteristics of a Good Architecture 36
 - Architectural Coupling 36
 - Architectural Cohesion 37
 - Architectural Design Patterns in Embedded Software 38
 - The Unstructured Monolithic Architecture 38
 - Layered Monolithic Architectures 39
 - Event-Driven Architectures 42
 - Microservice Architectures 45
 - Application Domain Decomposition 47
 - The Privilege Domain..... 48
 - The Security Domain 49
 - The Execution Domain..... 50
 - The Cloud Domain 52
 - Final Thoughts..... 52
- Chapter 3: Secure Application Design 55**
 - Platform Security Architecture (PSA) 56
 - PSA Stage 1 – Analyzing a System for Threats and Vulnerabilities 58
 - PSA Stage 2 – Architect..... 68
 - PSA Stage 3 – Implementation..... 75
 - PSA Stage 4 – Certify 81
 - Final Thoughts..... 83
- Chapter 4: RTOS Application Design 85**
 - Tasks, Threads, and Processes 87
 - Task Decomposition Techniques 90
 - Feature-Based Decomposition 90
 - The Outside-In Approach to Task Decomposition 92

Setting Task Priorities	106
Task Scheduling Algorithms	106
Verifying CPU Utilization Using Rate Monotonic Analysis (RMA).....	108
Measuring Execution Time	111
Final Thoughts.....	116
Chapter 5: Design Patterns	119
Managing Peripheral Data	119
Peripheral Polling	120
Peripheral Interrupts.....	121
Interrupt Design Patterns	123
Direct Memory Access (DMA)	129
RTOS Application Design Patterns	130
Resource Synchronization	131
Activity Synchronization	134
Publish and Subscribe Models.....	138
Low-Power Application Design Patterns.....	140
Leveraging Multicore Microcontrollers	143
AI and Real-Time Control.....	145
Real-Time Control.....	146
Security Solutions.....	147
A Plethora of Use Cases	148
Final Thoughts.....	148
Part II: Agile, DevOps, and Processes	149
Chapter 6: Software Quality, Metrics, and Processes.....	151
Defining Software Quality	152
Structural Software Quality.....	153
Architectural Quality	154
Code Quality	157
Case Study – Capstone Propulsion Controller.....	169
Final Thoughts.....	176

TABLE OF CONTENTS

- Chapter 7: Embedded DevOps..... 179**
 - A DevOps Overview 180
 - DevOps Principles in Action 182
 - Embedded DevOps Delivery Pipeline 184
 - CI/CD for Embedded Systems 187
 - Designing a CI/CD Pipeline..... 190
 - Creating Your First CI/CD Pipeline 193
 - Is DevOps Right for You? 194
 - Final Thoughts..... 194

- Chapter 8: Testing, Verification, and Test-Driven Development..... 197**
 - Embedded Software Testing Types 198
 - Testing Your Way to Success..... 199
 - What Makes a Great Test? 202
 - Regression Tests to the Rescue 204
 - How to Qualify Testing 205
 - Introduction to Test-Driven Development..... 208
 - Setting Up a Unit Test Harness for TDD 211
 - Installing CppUTest..... 212
 - Leveraging the Docker Container 214
 - Test Driving CppUTest..... 215
 - Final Thoughts..... 217

- Chapter 9: Application Modeling, Simulation, and Deployment..... 219**
 - The Role of Modeling and Simulation 219
 - Embedded Software Modeling 221
 - Software Modeling with Stand-Alone UML Tools..... 222
 - Software Modeling with Code Generation 223
 - Software Modeling with Matlab 225
 - Embedded Software Simulation..... 226
 - Simulation Using Matlab..... 226
 - Software Modeling in Python 232

Additional Thoughts on Simulation	233
Deploying Software	234
Stand-Alone Flash Tools for Manufacturing	235
CI/CD Pipeline Jobs for HIL Testing and FOTA.....	236
Final Thoughts.....	237
Chapter 10: Jump-Starting Software Development to Minimize Defects	241
A Hard Look at Bugs, Errors, and Defects	242
The Defect Minimization Process.....	244
Phase 1 – Project Setup	244
Phase 2 – Build System and DevOps Setup	246
Phase 3 – Test Harness Configuration.....	248
Phase 4 – Documentation Facility Setup.....	248
Phase 5 – Static Code Analysis	249
Phase 6 – Dynamic Code Analysis.....	250
Phase 7 – Debug Messages and Trace	252
When the Jump-Start Process Fails.....	253
Final Thoughts.....	255
Part III: Development and Coding Skills	257
Chapter 11: Selecting Microcontrollers.....	259
The Microcontroller Selection Process	259
Step #1 – Create a Hardware Block Diagram	260
Step #2 – Identify All the System Data Assets.....	263
Step #3 – Perform a TMSA	264
Step #4 – Review the Software Model and Architecture	265
Step #5 – Research Microcontroller Ecosystems	266
Step #6 – Evaluate Development Boards	267
Step #7 – Make the Final MCU Selection	269
The MCU Selection KT Matrix.....	269
Identifying Decision Categories and Criteria.....	270
Building the KT Matrix	271

TABLE OF CONTENTS

- Choosing the Microcontroller 272
- Overlooked Best Practices 273
- Final Thoughts..... 275
- Chapter 12: Interfaces, Contracts, and Assertions 277**
- Interface Design..... 277
- Design-by-Contract..... 279
 - Utilizing Design-by-Contract in C Applications 280
- Assertions 282
 - Defining Assertions..... 283
 - When and Where to Use Assertions..... 285
 - Does Assert Make a Difference? 287
 - Setting Up and Using Assertions 287
 - Three Instances Where Assertions Are Dangerous 291
- Getting Started with Real-Time Assertions 293
 - Real-Time Assertion Tip #1 – Use a Visual Aid 294
 - Real-Time Assertion Tip #2 – Create an Assertion Log..... 295
 - Real-Time Assertion Tip #3 – Notify the Application..... 296
 - Real-Time Assertion Tip #4 – Conditionally Configure Assertions 298
- A Few Concluding Thoughts..... 298
- Chapter 13: Configurable Firmware Techniques..... 301**
- Leveraging Configuration Tables..... 302
- An Extensible Task Initialization Pattern 304
 - Defining a Task Configuration Structure..... 305
 - Defining a Task Configuration Table 308
 - Initializing Tasks in a Task_CreateAll Function..... 311
- Autogenerating Task Configuration 314
 - YAML Configuration Files..... 315
 - Creating Source File Templates 317
 - Generating Code Using Python 322
- Final Thoughts..... 331

Chapter 14: Comms, Command Processing, and Telemetry Techniques	333
Designing a Lightweight Communication Protocol	334
Defining a Packet Protocol's Fields	335
A Plethora of Applications.....	337
Implementing an Efficient Packet Parser.....	338
The Packet Parsing Architecture	338
Receiving Data to Process.....	340
Packet Decoding As a State Machine	342
Validating the Packet.....	346
Command Processing and Execution.....	347
Traditional Command Parsers	347
An Introduction to Command Tables.....	350
Executing a Command from a Command Table.....	353
Managing System Telemetry.....	354
Telemetry As a "Global" Variable	355
Telemetry As a Service	357
Final Thoughts.....	358
Chapter 15: The Right Tools for the Job	361
The Types of Value Tools Provide.....	361
Calculating the Value of a Tool	363
The ROI of a Professional Debugging Tool.....	365
Embedded Software Tools.....	368
Architectural Tools	368
Process Tools.....	371
Implementation Tools	376
Open Source vs. Commercial Tools.....	390
Final Thoughts.....	391

TABLE OF CONTENTS

- Part IV: Next Steps and Appendixes 393**
- Afterword: Next Steps 395**
 - Where Are We Now?..... 395
 - Defining Your Next Steps..... 396
 - Choosing How You Grow 396
 - Final Thoughts..... 398
- Appendix A: Security Terminology Definitions 401**
 - Definitions 401
- Appendix B: 12 Agile Software Principles 405**
 - 12 Agile Software Principles 405
- Appendix C: Hands-On – CI/CD Using GitLab 407**
 - An Overview 407
 - Building STM32 Microcontroller Code in Docker 407
 - Installing GCC-arm-none-eabi in a Docker Container 408
 - Creating and Running the Arm GCC Docker Image..... 411
 - Creating an STM32 Test Project 413
 - Compiling the STM32 Makefile Project 415
 - Configuring the Build CI/CD Job in GitLab..... 415
 - Creating the Pipeline 416
 - Connecting Our Code to the Build Job..... 418
 - Build System Final Thoughts..... 421
 - Leveraging the Docker Container 421
 - Test-Driving CppUTest 422
 - Integrating CppUTest into a CI/CD Pipeline 423
 - Adding CppUTest to the Docker Image 423
 - Creating a Makefile for CppUTest 425
 - Configuring GitLab to Run Tests 429

Unit Testing Final Thoughts	430
Adding J-Link to Docker	431
Adding a Makefile Recipe	432
Deploying Through GitLab	433
Deployment Final Thoughts.....	434
Appendix D: Hands-On TDD.....	435
The Heater Module Requirements	435
Designing the Heater Module.....	435
Defining Our Tests	437
Writing Our First Test	438
Test Case – Setting to HEATER_ON	439
Heater Module Production Code	442
Heater Module Test Cases.....	449
Do We Have Enough Tests?	451
TDD Final Thoughts	453
Index.....	455

About the Author

Jacob Beningo is an embedded software consultant with around 20 years of experience in microcontroller-based, real-time embedded systems. Jacob founded Beningo Embedded Group in 2009 to help companies modernize how they design and build embedded software through software architecture and design, adopting Agile and DevOps processes and development skills. Jacob has worked with clients in over a dozen countries to dramatically transform their businesses by improving product quality, cost, and time to market in the automotive, defense, medical, and space systems industries. Jacob holds bachelor's degrees in Electrical Engineering, Physics, and Mathematics from Central Michigan University and a master's degree in Space Systems Engineering from the University of Michigan.

Jacob has demonstrated his leadership in the embedded systems industry by consulting and training at companies such as General Motors, Intel, Infineon, and Renesas, along with successfully completing over 100 embedded software consulting and development projects. Jacob is also the cofounder of the yearly Embedded Online Conference, which brings together managers and engineers worldwide to share experiences, learn, and network with industry experts.

Jacob enjoys spending time with his family, reading, writing, and playing hockey and golf in his spare time. In clear skies, he can often be found outside with his telescope, sipping a fine scotch while imaging the sky.

About the Technical Reviewer



Jack Ganssle has written over 1000 articles and six books about embedded systems, as well as a book about his sailing fiascos. He started developing embedded systems in the early 1970s using the 8008. He's started and sold three electronics companies, including one of the bigger embedded tool businesses. He's developed or managed over 100 embedded products, from deep-sea navigation gear to the White House security system... and one instrument that analyzed cow poop! He now lectures and consults about the industry and works as an expert witness in embedded litigation cases.

Acknowledgments

The book you now hold in your hand, either electronically or in paper form, is the product of 12 months of intense writing and development. The knowledge, techniques, and ideas that I share would not have been possible without over 20 years of support from family, friends, mentors, and teachers who helped me get to this point in my career. I humbly thank them for how they have enriched my life and pushed me to be the best version of me.

I would also like to thank all of you who have corresponded with me and even been my clients over the years. Your thoughts, suggestions, issues, challenges, and requests for help have helped me hone my skills and grow into a more mature embedded software consultant. I've learned how to more effectively help others, which has helped me continue to provide more value to my clients and the embedded systems industry.

I owe a big thank you to Jack Ganssle for reviewing this book. I know a lot of the review occurred during the summer while he was sailing the globe. That didn't stop him from asking great questions and giving feedback that provided me with new insights, ideas, and guidance that helped me dramatically improve the book's content.

There are also many, many others who have helped to bring this book into existence. The acknowledgments could go on indefinitely, but instead I'll simply say thank you to Apress, Steve Anglin, James Grenning, Jean Labrosse, John Little, James McClearn, Mark Powers, and Tomas Svitek.

Preface

Successful Delivery

Designing, building, and delivering an embedded product to market can be challenging. Products today have become sophisticated, complex software systems that sit upon an ever-evolving hardware landscape. At every turn, there is a challenge, whether it's optimizing a system for energy consumption, architecting for configurability and scalability, or simply getting a supply of the microcontrollers your system is built. Embedded software design requires chutzpah.

Despite the many challenges designers, developers, and teams often encounter, it is possible to successfully deliver embedded products to market. In fact, I'll go a step further and suggest that it's even possible to do so on time, on budget, and at the quality level required by the system stakeholders. It's a bold statement, and, alas, many teams fall short. Even the best teams stumble, but it is possible to reach a level of consistency and success that many teams can't reach today.

Embedded Software Design, this book, is about providing you and your team with the design philosophies, modern processes, and coding skills you need to deliver your embedded products successfully. Throughout the book, we cover the foundational concepts that will help you and your team overcome modern challenges and help you thrive. As you can imagine, the book is not all-inclusive in that every team will have different needs based on their industry, experience, and goals. *Embedded Software Design* will help get you started, and if you get stuck or need professional assistance, I'm just an email away.

Before we dive into the intricate details of *Embedded Software Design*, we need to initialize our minds, just like our embedded systems. We need to understand the lens through which we will view the material in this book and how we can apply it successfully to our unique situations. This chapter will explore how this book is organized, essential concepts on how to use the material, and how to maximize the value you receive from it.

Tip At the end of each chapter, you will find exercises to help you think through and apply the material from each chapter. “Action Items” will come in the form of questions, design experiments, and more. To maximize your value, schedule time in your calendar to carefully think through and execute the Action Items.

The Embedded Software Triad

Many teams developing embedded software struggle to deliver on time, on budget, and at a quality level that meets customer expectations. Successful development is often elusive, and software teams have all kinds of excuses for their failure.¹ However, many teams in the industry are repeatedly successful even under the direst conditions. I’ve found that these teams have mastered what I call the embedded software triad.

The embedded software triad consists of

- Software Architecture and Design
- Agile, DevOps, and Processes
- Development and Coding Skills

Software architecture and design consists of everything necessary to solicit the requirements, create user stories, and design the architecture. The software architecture defines the blueprint for the system. It’s important to note that the architecture should be evolvable and does not necessarily need to be developed up front.² How the architecture is designed follows the processes used by the team.

Agile, DevOps, and Processes are the procedures and best practices that are used to step-by-step design, build, test, and deliver the software. Processes provide the steps that allow a team to deliver successfully based on the budget, time, and quality parameters provided to them consistently. In today’s software development environment, processes

¹ Yes, I know, a bit harsh right off the bat, but it’s necessary.

² There are many arguments about this based on the processes used. We will discuss further later in the book.

are often based on various agile methodologies. Therefore, the processes help to guide the implementation under the parameters and disciplines defined by the team. I've also called out DevOps specifically because it is a type of process often overlooked by embedded software teams that offers many benefits.

Development and coding skills are required to construct and build the software. Development entails everything necessary to take the architecture and construct the system using the processes! At the end of the day, the business and our users only care about the implementation which is completed through development and coding skills. The implementation should result in a working system that meets agreed-upon features, quality, and performance characteristics. Teams will organize their software structure, develop modules, write code, and test it. Testing is a critical piece of implementation that can't be overlooked that we will be discussing throughout the book.

To be successful, embedded software designers, developers, and teams must not just master the embedded software triad but also balance them. Too much focus on one area will disrupt the development cycle and lead to late deliveries, going over budget, and even buggy, low-quality software. None of us want that outcome. This brings us to the question, "How can we balance the triad?"

Balancing the Triad

Designers, developers, and teams can't ignore any element of the embedded software triad. Each piece plays a role in guiding the team to success. Unfortunately, balancing these three elements in the real world is not always easy. In fact, in nearly every software audit I have performed from small to large companies, there is almost always some imbalance that can be found. In some teams, it's minor and adjustments can result in small efficiency improvements. In others, the imbalance is major and often crippling to the company.

The relationships between the elements in the triad can be best visualized through a Venn diagram, as shown in Figure 1. When a team masters and balances each element, marked by the number 4 in Figure 1, it is more likely that the team will deliver their software on time, on budget, and at the required quality level. Unfortunately, finding a team that is balanced is relatively rare. It is more common to find teams that are hyperfocused on one or two elements. When a team is out of balance, there are three regions of Figure 1 where the teams often fall, denoted by 1, 2, and 3. Each area has its own symptoms and prescription to solve the imbalance.

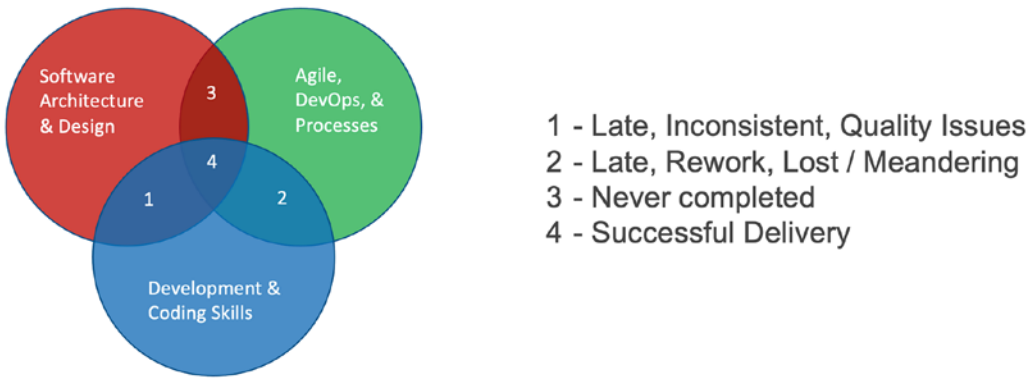


Figure 1. *The balancing act and results caused by the embedded software triad*

Caution Being out of balance doesn't ensure failure; however, it can result in the need for herculean efforts to bring a project together successfully. Herculean efforts almost always have adverse effects on teams, costs, and software quality.

The first imbalance that a team may fall into is region 1. Teams fall into region 1 when they focus more on software architecture and implementation (denoted by a 1 in Figure 1) than on development processes. In my experience, many small- to medium-sized teams fall within region 1. In smaller teams, the focus is almost always delivery. Delivery equates to implementation and getting the job done, which is good, except that many teams jump right in without giving much thought to what they are building (the software architecture) or how they will implement their software successfully (the processes) repeatedly and consistently.³

Teams that operate in region 1 can be successful, but that success will often require extra effort. Deliveries and product quality will be inconsistent and not easily reproducible. The reason, of course, is that they lack the processes that create consistency and reproducibility. Without those processes, they will also likely struggle with quality issues that could cause project delays and cause them to go over budget. Teams don't necessarily need to go all out on their processes to get back into balance, but only need to put in place the right amount of processes to ensure repeatability.

³We often aren't interested in a one-hit wonder delivery. A team needs to consistently deliver software over the long term. Flukes and accidental success don't count! It's just dumb luck.

Tip If you spend 20% or more debugging your software, you will most likely have a process problem. Don't muscle through it; fix your processes!

The second unbalanced region to consider is region 2. Teams that fall into region 2 focus on development processes and implementation while neglecting the software architecture (denoted by a 2 in Figure 1). These teams design their system on the fly without any road map or blueprint for what it is they are building. As a result, while the team's software quality and consistency may be good, they will often still deliver late because they constantly must rework their system with every new feature and requirement. I often refer to these teams as lost or meandering because they don't have the big picture to work from.

The final unbalanced region to consider is region 3. Teams in region 3 focus on their software architecture and processes with little thought given to implementation (denoted by a 3 in Figure 1). These teams will never complete their software. They either lack the implementation skills or bog down so much in the theory of software that they run out of money or customers before the project is completed.

There are several characteristics that a balanced team will exhibit to master the embedded software triad. First, a balanced team will have a software architecture that guides their implementation efforts. The architecture is used as an evolving road map that gets the team from where they are to where their software needs to be. Next, a balanced team will have the correct amount of processes and best practices to ensure quality software and consistency. These teams won't have too much process and not too little. Finally, a balanced team will have development and coding skills to construct the architecture and leverage their processes to test and verify the implementation.

Successful Embedded Software Design

Successful software delivery requires a team to balance the embedded software triad, but it also requires that teams adopt and deploy industry best practices. As we discuss how to design and build embedded software throughout this book, our focus will discuss general best practices. Best practices "are procedures shown by research and experience to produce optimal results and establish or propose a standard for widespread

adoption.”⁴ The best practices we focus on will be general to embedded software, particularly for microcontroller-based systems. Developers and teams must carefully evaluate them to determine whether they fit well within their industry and company culture.

Applying best practices within a business requires discipline and an agreement that the best practices will be adhered to no matter what happens. Too often, a team starts following best practices, but when management puts on the heat, best practices go out the window, and the software development process decays into a free for all. There are three core areas where discipline must be maintained to successfully deploy best practices throughout the development cycle, as shown in Figure 2.



Figure 2. *Best practice adoption requires discipline, agreement, and buy-in throughout a business hierarchy to be successful*

Developers form the foundation for maintaining best practices. They are the ones on the frontline writing code. If their discipline breaks, no matter what else is going on in the company, the code will descend into chaos. Developers are often the ones that also identify best practices and bring new ideas into the company culture. Therefore, they must adhere to agreed-upon best practices no matter the pressures placed upon them.

Developers may form the foundation for adhering to best practices, but the team they work on is the second layer that needs to maintain discipline. First, the team must identify the best practices they believe must be followed to succeed. Next, they need to reinforce each developer, and the team needs to act as a buffer with upper management when the pressure is on. Often, a single developer pushing back will fail, but an entire team will usually hold some sway. Remember, slow and steady wins the race, as counterintuitive as that is to us intellectually and emotionally.

⁴www.merriam-webster.com/dictionary/best%20practice

Definition Best practices “are procedures shown by research and experience to produce optimal results and establish or propose a standard for widespread adoption.”⁵

Finally, the management team needs to understand the best practices that the team has adopted. Management must also buy into the benefits and then agree to the best practices’ value. If they know why those best practices are in place, they will realize that it is to preserve product quality, minimize time to market, or some other desired benefit when the team pushes back. The push to cut corners or meet arbitrary dates will occur less but won’t completely go away, given its usually market pressure that drives unrealistic deadlines. However, if there is an understanding about what is in the best interest of the company and the customers, a short delay for a superior product can often be negotiated.

How to Use This Book

As you might have guessed from our discussions, this book focuses on embedded software design best practices in each area of the embedded software triad. There are more best practices within the industry than could probably be written into a book; however, I’ve tried to focus on the best practices in each area that is “low-hanging fruit” and should provide high value to the reader. You’ll need to consult other texts and standards for specific best practices in your industry, such as functional safety.

This book can either be read cover to cover or reviewed on a chapter-by-chapter basis, depending on the readers’ needs. The book is broken up into four parts:

- Software Architecture and Design
- Agile, DevOps, and Processes
- Development and Coding Skills
- Next Steps and Appendixes

⁵www.merriam-webster.com/dictionary/best%20practice

PREFACE

I've done this to allow the reader to dive into each area of the embedded software triad and provide them with the best practices and tools necessary to balance their development cycles. Balancing can be sometimes accomplished through internal efforts, but often a team is too close to the problem and “drinking the Kool-Aid” which makes resolution difficult. When this happens, feel free to reach out to me through www.beningo.com to get additional resources and ideas. *Embedded Software Design* is meant to help you be more successful.

A quick note on the Agile, DevOps, and Processes part. These chapters can be read independently, but there are aspects to each chapter that build on each other. Each chapter walks you through getting a fundamental piece of a CI/CD pipeline up and running. For this reason, I would recommend reading this part in order.

I hope that whether you are new to embedded software development or a seasoned professional, you'll find new or be reminded of best practices that can help improve how you design and develop embedded software. We work in an exciting industry that powers our world and society can't live without. The technologies we work with evolve rapidly and are constantly changing. Our opportunities are limitless if we can master how to design and build embedded software effectively.

ACTION ITEMS

To put this chapter's concepts into action, here are a few activities the reader can perform to start balancing their embedded software activities:

- Which area do you feel you struggle the most in?
 - Software Architecture and Design
 - Agile, DevOps, and Processes
 - Development and Coding Skills
- Review Figure 1. Which region best describes you? What about your team? (If you are not in region 4, what steps can you take to get there?)
- Take Jacob's design survey to understand better where you currently are and how you can start to improve (www.surveymonkey.com/r/7GP8ZJ8).
- Review Figure 2. Do all three groups currently agree to support best practices? If not, what can be done to get the conversation started and get everyone on the same page?

Index

A

Access control, 65, 401

Activity synchronization

 bilateral rendezvous, 136

 coordinating task execution, 134

 multiple tasks, 137, 138

 unilateral rendezvous (interrupt to task), 135, 136

 unilateral rendezvous (task to task), 135

Agile software principles, 405, 406

Analysis tools, 165, 250, 385, 387–390

Application business architecture, 33

Application domain decomposition

 cloud domain, 52

 execution domain, 50

 privilege domain, 48, 49

 security domain, 49, 50

Application programming interface (API), 277

Architect secure application, 72, 74, 75

Architectural design patterns

 Event-driven, 42–44

 layered monolithic, 39, 41, 42

 microservice, 45–47

 unstructured monolithic architecture, 38, 39

Architectural tools

 requirements solicitation and management, 369

 storyboarding, 369, 370

UML diagramming and modeling, 370, 371

Architecture

 bottom-up approach, 33, 34, 36

 characteristics, 36–38

 top-down approach, 33, 34, 36

Arm TrustZone, 71, 401

Armv8-M architecture, 77

Artificial intelligence and real-time control, 145

Assertions, 254

 definition, 282–284

 Dio_Init function, 283

 INCORRECT use, 287

 instances

 initialization, 292

 microcontroller drivers, 292, 293

 real-time hardware

 components, 293

 setting up

 assert_failed, 289–291

 assert macro definition, 288, 289

 uses, 285, 287

Assert macro, 283, 285, 288–289

Attestation, 72, 81, 401

Authenticity, 64, 80, 401

Autogenerating task configuration

 Python code, 322–331

 source file templates, 317–322

 YAML configuration files, 315, 317

Automated unit testing, 24

B

Bilateral rendezvous, 136
 Branch coverage, 166, 167, 374, 452
 Breakpoints, 254, 365, 380
 Broadcast design pattern, 137
 Budgets, 7, 47, 237, 373
 Bugs, 18, 100, 161, 186, 242, 243, 387
 Bug tracking, 161

C

Capstone propulsion controller
 (case study), 169–176
 Certification, 82
 Chain-of-Trust, 80, 401
 Checksum validation, 346
 Continuous integration, and continuous
 deployment (CI/CD), 45
 configuration Job in GitLab, 415–421
 CppUTest integration
 configuring GitLab to run tests,
 429, 430
 Docker Image, 423, 424
 Makefile creation, 425, 427, 428
 deployment, 434
 designing, 190–193
 embedded systems, 187–189
 unit testing (*see* Unit testing)
 Circular buffer design pattern, 125, 126
 Clock tree, 377, 378
 Cloud domain, 52
 Code analysis tools, 249
 Code generation, 223–224, 314, 382,
 384, 414
 Code reviews, 167, 169
 Coding standards and conventions,
 158, 159
 Cohesion, 37, 43, 154

Command processing, 347, 358
 Command table, 350, 352
 Commercial tools, 373, 390
 Communication packets, 66, 334
 Compiler-provided IDE, 379
 Confidentiality, 64, 401
 Confidentiality, integrity, and authenticity
 (CIA) model, 64
 Configuration management, 24, 93, 246
 Configuration tables, 43, 302–304, 331
 Content coupling, 37
 Core testing methodologies, 200
 Coupling, 36, 37, 43, 154, 301
 CppUTest, 192, 211–215, 374, 387, 421–430
 Cyclomatic Complexity, 162–164, 171–174,
 206, 388, 452
 Cypress PSoC 6, 71, 76, 143, 144
 Cypress PSoC 64, 71, 76, 144

D

Data assets, 10, 11, 19, 59, 60, 65, 263
 Data dictates design, 10, 11, 60, 119, 263
 Data length field, 336
 Data profiling, 254
 Defect minimization process
 build system and DevOps setup,
 246, 247
 debug messages and trace, 252, 253
 documentation facility
 setup, 248, 249
 dynamic code analysis, 250, 251
 project setup, 244
 static code analysis, 249
 test harness configuration, 248
 Delivery pipeline, 182, 184
 Denial of service, 402
 Deploying software

- CI/CD pipeline jobs for HIL testing, 236, 237
- stand-alone flash tools for manufacturing, 235
- Design-by-Contract (DbC), 279–282
- Design quality, 17, 18
- Developers challenges, 5–9
- Development costs, 7, 358
- DevOps
 - developers team, 180
 - high-level overview, 182
 - operations team, 180
 - principles, 182, 183
 - processes, principles guide, 181
 - “pure applications”, 179
 - QA team, 180
 - software development and delivery process, 179
- Diagramming tools, 370, 371
- Digital input/output (DIO), 278, 280, 302
- Direct memory access (DMA), 32, 101, 129, 130, 204
- Docker container, 189
 - GCC-arm-none-eabi installation, 408, 410, 411
 - leveraging, 421, 422
- Do-it-yourself (DIY) project, 14
- Dynamic code analysis, 165, 250

E

- Embedded DevOps delivery pipeline process, 184, 186
- Embedded DevOps process, 184–186, 194, 236, 421
- Embedded software, 26

- Embedded software triad, 396
 - Agile, DevOps, and Processes, vi
 - development and coding skills, vi
 - Software architecture and design, vi
- Emotional and social values, 362
- Errors, 192, 242, 315, 427
- Escalation of privilege, 402
- Event-driven architectures, 42–44
- Execution, command table, 353, 354
- Execution domain, 50, 51
- Extensible task initialization pattern
 - task configuration structure, 305–308
 - task configuration table, 308–311
 - Task_CreateAll function, 311, 313

F

- Feature-based decomposition, 90, 91
- Feedback pipeline, 183
- Firmware authenticity, 65, 402
- Firmware over-the-air (FOTA), 72, 234
- FreeRTOS, 88, 138, 141, 233, 304, 308, 322, 385
- Freescale S12X, 143
- Functional tests, 198
- Functional value, 362

G

- Genuine quality, 153
- Git repository tools, 371
- GPIO pin, 111, 280, 302
- Graphical user interface (GUI), 6, 277, 432
- Graphics accelerators, 32
- Graphics processing units (GPUs), 51
- GUI-based tool, 372

H

- Hardware-dependent real-time architecture, 32
- Hardware-independent business rules-based architecture, 32
- Heater Controller state machine design, 228
- Heater module, TDD
 - designing, 435–437
 - production code, 442–449
 - requirements, 435
 - setting to HEATER_ON (test case), 439–441
 - test cases, 449, 451
 - tests lists, 437
 - writing tests, 438, 439
- Heterogeneous multiprocessors, 145
- Hybrid approach, 36
- Hybrid threads, 74

I

- Impersonation, 63, 402
- Implementation tools
 - code generation, 382, 384
 - IDEs and compilers, 376, 378, 379
 - programmers, 380, 382
- Industry-wide hardware abstraction layers (HALs), 266
- Instruction Trace Macrocell (ITM), 252
- Instruction tracing, 254, 365, 387
- Integrated Development Environments (IDEs), 376
- Integration tests, 180, 193, 200, 374
- Integrity, 64, 80, 130, 335, 402
- Interface design, 277, 298
- Internet of Things (IoT) hardware, 57, 402

- Interprocessor communications (IPC), 71, 76
- Interrupt design patterns
 - bare-metal and RTOS-based systems, 123
 - circular buffer, 125, 126
 - circular buffer with notification, 126–128
 - linear data store, 123, 124
 - message queues, 128
 - ping-pong buffers, 124
- Interrupt locking, 132
- Interrupt service routine (ISR), 122, 278
- Isolation in secure application, 68, 70, 71

J

- Jump-starting software development (*see* Defect minimization process)

K

- KISS software principle, 13
- KT Matrix, 269–273

L

- Layered monolithic architectures, 39, 41, 42
- Lightweight communication protocol
 - applications, 337, 338
 - packet protocol, 335–337
- Linear data store design
 - pattern, 123, 124
- Lines of code (LOC), 160, 166, 312, 369, 387, 423, 426, 429
- Low-power application design
 - patterns, 140, 142

M

- Malware, 64, 65, 402
- Man-in-the-middle, 402
- Manual testing, 204
- Matlab, 225–231, 383
- McCabe Cyclomatic Complexity.,
 - 161–163, 165, 205, 348, 374, 389
- MCU selection KT Matrix
 - identifying decision categories and
 - criteria, 270, 271
 - microcontroller, 272, 273
- Measuring task execution time
 - manually, 111
 - trace tools, 112–115
- Memory protection unit (MPU), 32,
 - 46, 88, 402
- Message queue design pattern, 128
- Microcontroller-based embedded
 - systems, 70
- Microcontroller selection process
 - evaluate development boards, 267–269
 - hardware block diagram, 260–262
 - MCU selection, 269
 - research microcontroller ecosystems,
 - 266, 267
 - software model and architecture, 265
 - system data assets, 263, 264
 - TMSA, 264
- Microservice architectures, 45–47
- Minimum viable product (MVP), 8
- Modeling, 27
 - code generation, 223, 224
 - Matlab, 225
 - role, 219–221
 - stand-alone UML tools, 222, 223
- Modern design philosophy principles
 - data dictates design, 10, 11
 - design quality, 17, 18
 - Keep It Simple and Smart (KISS)!
 - software, 13
 - Practical, not perfect, 14, 16
 - scalable and configurable, 16
 - security is king, 18, 19
 - there is no hardware (only data), 11, 13
- Modern embedded software
 - development, 22, 24, 25, 219, 220
- Modern microcontroller selection
 - process, 260
- Monetary value, 362
- Multicore microcontrollers
 - AI and real-time control, 145
 - architectures, 144
 - execution environments, 143
 - heterogeneous multiprocessing, 144
 - homogenous multiprocessing, 144
 - security solutions, 147
 - use case, 146, 148
- Multicore processors, 71, 76, 79
- Multicore solutions, 77
- Mutex lock, 133, 134

N

- Nonsecure processing environments
 - (NSPE), 50, 71, 76, 81, 402
- Nonsecure threads, 74

O

- Off-chip development, 26
- On-target testing, 24, 193
- Open source *vs.* commercial tools,
 - 390, 391
- Operating system abstraction layer
 - (OSAL), 41, 307

INDEX

- Operational code (opcode) field, 336, 353
- Operations team, 180, 237
- Outside-in approach to task
 - decomposition
- IoT-based device, 92
- IoT thermostat
 - concurrencies, dependencies, and data flow, 100–102
 - first-tier tasks, 98, 100
 - hardware block diagram, 93
 - high-level block diagram, 94, 95
 - identifying major components, 94
 - label inputs, 95, 96
 - outputs label, 97, 98
 - second-tier tasks, 102–105
- seven-step process, 92

P

- PacketDecodeSm, 346
- Packetized communication structure, 333
- Packet parser implementation
 - architecture, 338, 340
 - packet decoding as state machine, 342, 343, 345, 346
 - receiving data to process, 340–342
 - validation, 346, 347
- Packet protocols, 335–338
- Pelion, 237
- Percepio's Tracealyzer, 113, 251, 385
- Periodic execution time (RMS), 106, 107, 109
- Peripheral interrupts, 121, 122
- Peripheral polling, 120, 121
- Peripheral protection unit (PPU), 402
- Ping-pong buffer design pattern, 124
- Platform security architecture (PSA), 402
 - architect
 - isolation in secure application, 68, 70
 - processing environments, 71
 - Root-of-Trust and trusted services, 71
 - secure application, 72, 74, 75
 - trusted applications, 72
- certified components, 58
- certify stage, 81
- description, 56
- implementation
 - multicore processors, 76, 77
 - secure boot process, 79, 81
 - single-core processors with TrustZone, 77–79
- secure solution, 57
- stages, 57
- threats and vulnerabilities analyzing
 - adversaries and threats
 - identifying, 61–64
 - identifying assets, 59, 60
 - security objectives defining, 64–66
 - security requirements, 66
 - TMSA analysis document, 67, 68
- Polling, 120, 121, 127
- Preemption lock, 133
- Printf, 252, 254, 292
- Priority-based scheduling, 85
- Process groups resources, 88
- Process tools
 - continuous integration/continuous deployment, 375, 376
 - revision control, 371, 372
 - software development life cycle, 372
 - testing, 373, 374
- Product quality, 7
- Programmers, 235, 265, 380, 382
- Project managers, 6, 8, 270

Propulsion system, 170
 PSA Certified, 57, 82
 Publish and subscribe models,
 119, 138–140
 Python controller, 232

Q

QEMU, 233, 234
 Qualify testing, 205–208
 Quality assurance (QA), 180
 Quality metrics, 160

R

Raspberry Pi Pico, 143
 Rate monotonic analysis (RMA), 108, 311
 Rate monotonic scheduling (RMS), 108
 Real-time assertions
 application, 296–298
 assertion log, 295, 296
 conditionally configure assertions, 298
 visual aid, 294, 295
 Real-time control, 146, 147
 Real-time operating system (RTOS), 74
 aware debugging, 251
 challenging, 86
 definition, 85
 design patterns
 activity synchronization
 (*see* Activity synchronization)
 resource synchronization
 (*see* Resource synchronization)
 developers with key capabilities, 86
 middleware library, 87
 semaphores, 127
 tasks, threads, and processes, 87–90

 trace capability, 112
 Real-time software architecture, 33
 Real-time systems, 85
 Regression testing, 24, 201, 204
 Repo setup, 245, 246
 Repudiation, 402
 Resource synchronization
 definition, 130
 interrupt locking, 132
 mutex lock, 133, 134
 preemption lock, 133
 Return on investment (ROI), 364–367
 Revision control, 372
 Root-of-Trust (RoT), 71, 403
 Round-robin schedulers, 85

S

Scalability, 8
 SDLC tool, 373
 Secure application design, 70
 Secure boot process, 79, 81
 Secure communication, 403
 Secure firmware in Arm Cortex-M
 processors, 81
 Secure flash programmer, 235
 Secure manufacturing process, 236
 Secure processing environment (SPE), 50,
 71, 76, 403
 Secure state, 66, 403
 Secure storage, 65, 403
 Secure threads, 74
 Security domain, 49, 50
 Semaphores, 126, 127
 Shortest job first (SJF), 106
 Shortest response time (SRT), 106
 Simulation, 26

INDEX

Simulation (*cont.*)

- Matlab, 226–231
 - in Python, 232
 - role, 219–221
- Single-core processors, 77–79
- Social value, 362
- Software **functional** quality, 152
- Software **structural** quality, 152
- Software architecture quality, 30, 31, 154–157
- Software development life cycle tools, 372
- Software metric, 160, 161
- Software quality, 152
- Source module template, 318
- Sourcetree, 372
- Stand-alone UML modeling tool, 221
- State diagrams, 221
- State machines, 221
- Static analysis, 165
- Static analyzer, 363
- Static code analysis tool, 158
- Statistical profiling, 254
- STM32CubeMx, 413
- STM32 microcontroller code in Docker
 - Arm GCC Docker image, 411, 413
 - GCC-arm-none-eabi installation, 408, 410, 411
 - Makefile project, compiling, 415
 - test project creation, 413, 414
- STM32 microcontrollers, 269
- STMicroelectronics, 376, 379
- STMicroelectronics STM32H7, 143
- Storyboarding, 370
- Structural software quality
 - architectural quality, 154–157

code quality

- branch coverage, 166, 167
 - code analysis (static *vs.* dynamic), 165
 - code reviews, 167, 169
 - coding standards and conventions, 158, 159
 - software metric, 160–163, 165
- Successful delivery pipeline, 183
- Successful embedded software design
- Symmetric multicore processing, 144
- SystemView, 113

T

- Tags, 317
- Task, 87
- Task and data tracing, 254
- Task configuration structure, 305–308
- Task configuration table, 308–311
- Task_CreateAll function, 311, 313
- Task decomposition techniques
 - feature-based decomposition, 90, 91
 - outside-in approach (*see* Outside-in approach to task decomposition)
- TaskInitParams_t configuration
 - structure, 307
- Task priorities setting
 - algorithms, 106
 - CPU utilization, RMA, 108–111
 - execution time measuring
 - manually, 111
 - trace tools, 112–115
 - task scheduling algorithms, 106–108
- Task telemetry, 357
- Telemetry
 - global variable, 355, 356

- service, 357, 358
- Temperature feedback state machine
 - design, 228
- Temperature state machine, 229
- Templates, 315
- Tensor processing units (TPUs), 51
- Test-driven development (TDD), 435
 - definition, 208
 - hardware and operating system
 - dependencies, 210
 - microcycle, 208, 209
 - unit test harness
 - CppUTest installation, 212, 213
 - CppUTest starter project, 215, 216
 - Docker container, 214
- Test-Driving CppUTest, 422, 423
- Testing
 - characteristics, 203
 - types, 198, 199
- Texas Instruments TMS1000, 3
- Thread, 88
- Threat model and security analysis
 - (TMSA), 58, 264
- Tiered architecture, 44
- Time slicing, 85
- Tool's value
 - calculation, 363
 - communicate, 364
 - ROI, 365–367
 - types, 361, 363
- Top-down approach, 34
- Tracealyzer library, 114
- Trace tools, 268
- Traditional command parsers, 347–350
- Traditional embedded software
 - development, 21, 22

- Transport layer security (TLS), 72
- Troubleshooting techniques, 253
- True random number generation
 - (TRNG), 71
- Trusted applications, 72
- Trusted execution environment (TEE), 71
- Trusted services, 71
- TrustZone, 77, 78

U, V

- Unified Markup Language (UML), 221
- Unilateral rendezvous (interrupt to task),
 - 135, 136
- Unilateral rendezvous (task to task), 135
- Unit testing, 248
 - deploying through GitLab, 433
 - J-Link to Docker, 431
 - Makefile recipe, 432, 433
- Unit tests, 200, 202
- Unstructured monolithic software
 - architecture, 38, 39
- USART communication interface, 333

W

- wxWidgets, 233

X

- xTaskCreate, 305, 313

Y, Z

- YAML configuration files, 315, 317
- Yin-Yang design, 20, 21