J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_6

6. Software Quality, Metrics, and Processes

Jacob Beningo[1]
(1) Linden, MI, USA

I suspect, if I were to ask you this question, you would respond that you write quality software. Now, you may have some reservations about some of your code, but I don't think any of you will say that you purposely write poor code or that you are a terrible programmer. On the other hand, I would expect that those of you who say you don't write quality code probably have some reason for it. Management pushes you too hard; you're inexperienced and still wet behind the ears, but you're doing the best you can.

What if I asked you to define what quality software is? Could you define it? I bet many of you would respond the way Associate Justice Potter Stewart wrote in the 1964 Supreme Court case (Jacobellis v. Ohio) about hard-core pornography, "I know it when I see it."[1] Yet, if you were to give the same code base to several dozen engineers, I bet they wouldn't be able to agree on its quality level. Without a clear definition, quality is in the eye of the beholder!

In this chapter, we will look at how we can define software quality, how quality can be more of a spectrum than a destination, and what embedded software developers can do to hit the quality levels they desire more consistently. We will find that to hit our quality targets, we need to define metrics that we can consistently track, and put in place processes that catch bugs and defects early, before they become costly and time-consuming.

Defining Software Quality

Let's explore the definition of software quality bestowed upon us by that great sage Wikipedia:

> *Software Quality refers to two related but distinct notions:*
>
> 1. *Software* **functional quality** *reflects how well it complies with or conforms to a given design based on functional requirements or specifications.*[2] *That attribute can also be described as the fitness for the purpose of a piece of software or how it compares to competitors in the marketplace as a worthwhile product.*[3] *Finally, it is the degree to which the correct software was produced.*
> 2. *Software* **structural quality** *refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability. It has much more to do with the degree to which the software works as needed.*[4]

From this definition, it's obvious why developers have a hard time defining quality; there are two distinct but related components to it. Many of us probably look at the first notion focusing on the "fitness of purpose." If it works and does what it is supposed to, it's high quality (although, to many, the need to occasionally restart a device still seems acceptable, but I digress).

When I started working in the embedded systems industry, my employers loved the software quality I produced. I was that young, hard-working engineer who consistently delivered a working product, even if it meant working 15-hour days, seven days a week. Demos, tests, and fielded units met all the functional requirements. Behind the scenes, though, structurally, the software was, well, not what I would have wanted it to be. In fact, the words big ball of spaghetti and big ball of mud come to mind. (Thankfully, I quickly matured to deliver on both sides of the quality equation.)

In many situations, the customer, our boss, and so forth view the functional behavior of our software as the measure of quality. However, I

would argue that the objective measure of software quality is structural quality. Anyone can make a product that functions, but that doesn't mean it is not buggy and will work under all the desired conditions. It is not uncommon for systems to work perfect under controlled lab conditions only to explode in a companies face when it is deployed to customers.

Genuine quality is exhibited in the structural code because it directly impacts the product's robustness, scalability, and maintainability. Many products today evolve, with new features added and new product SKUs branching off the original. To have long-term success, we must not just meet the functional software requirements but also the structural quality requirements.

> **Note** The disconnect between management and the software team is often that management thinks of functional quality, while developers think of structural quality.

Suppose the defining notion of software quality is in the structural component. In that case, developers and teams can't add quality to the product at the end of the development cycle! Quality must be built into the entire design, architecture, and implementation. Developers who want to achieve quality must live the software quality life, always focusing on building quality at every stage.

Structural Software Quality

When examining the structural quality of a code base, there are two main areas that we want to explore: architectural quality and code quality. Architectural quality examines the big picture of how the code is organized and structured. Code quality examines the fine details of the code, such as whether best practices are being followed, the code is testable, and that complexity has been minimized.

Architectural Quality

In the first part of this book, we explored how to design and architect embedded software. We saw that software quality requirements often en-

compass goals like portability, scalability, reusability, etc. Measuring the quality of these requirements can be difficult because they require us to track how successfully we can change the code base over time. Developers can make those measurements, but it could take a year of data collection before we know if we have a good software architecture or not. By that time, the business could be in a lot of trouble.

Instead of relying upon long-term measurements to determine if you have reached your architectural quality goals, you can rely on software architecture analysis techniques. Software architecture analyses are just techniques that are used to evaluate software architectures. Many tools are available to assist developers in the process, such as Understand,[5] Structure101,[6] etc. (these tools are further discussed in Chapter 15). The analysis aims to detect any noncompliances with the architecture design, such as tight coupling, low cohesion, inconsistent dependencies, and so forth.

As a team builds their software, it's not uncommon for the architecture to evolve or for ad hoc changes to break the goals of good architecture. Often, tight coupling or dependency cycles enter the code with the developer being completely unaware. The cause is often a last-minute feature request or a change in direction from management. If we aren't looking, we won't notice these subtle issues creeping into our architecture, and before we know it, the architecture will be full of tightly coupled cycles that make scaling the code a nightmare. The good news is that we can monitor our architecture and discover these issues before they become a major issue.

I recently had a project that I was working on that used FreeRTOS Kernel v10.3.1 that had a telemetry task that collected system data and then transmitted it to a server. Using Understand, I plotted a dependency graph of the telemetry task to verify I didn't have any unwanted coupling or architectural violations, as seen in Figure 6-1. I was surprised to discover a cyclical dependency within the FreeRTOS kernel, as seen by the red line between FreeRTOS_tasks.c and trcKernelPort.c in Figure 6-1.

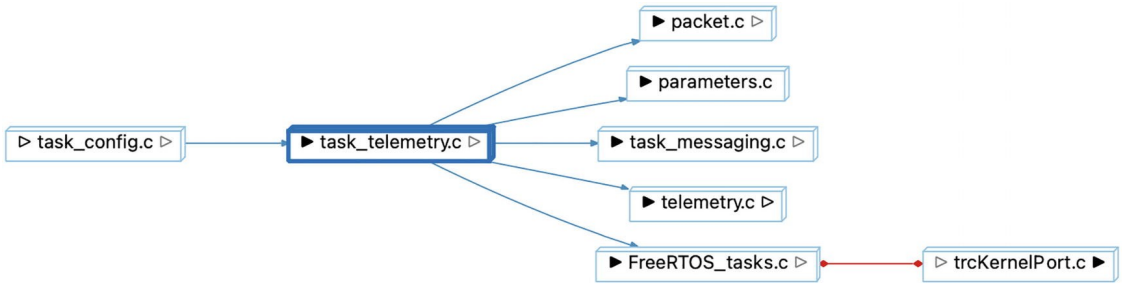

Figure 6-1 The task_telemetry.c call graph has a cyclical dependency highlighted by the red line between FreeRTOS_tasks.c and trcKernelPort.c

Since the violation occurred within a third-party library and is contained to it, I normally would ignore it. However, I was curious as to what was causing the cycle. The red line violation tells us, as architects, that the developer has code in trcKernelPort.c that is calling code in FreeRTOS_tasks.c and vice versa! Hence, a cycle has been created between these code modules. As architects, we would want to drill a bit deeper to determine what is causing this violation in the architecture. You can see in Figure 6-2 that the cyclical calls are caused by two calls within trcKernelPort.c.

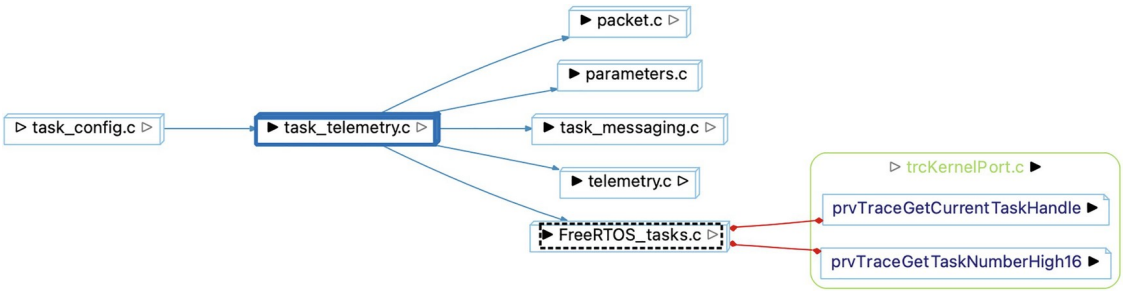

Figure 6-2 Expanding trcKernelPort.c reveals that two functions are involved in the cyclical call tree

We want to avoid cyclical calls like this because it tightly couples our software, decreases scalability, and starts to create a giant ball of mud. If this cycle had occurred in higher-level application code, there is the potential that other code would use these modules, which in turn are used by others. The result is a big ball of mud from an architectural standpoint.

Digging even deeper, we can expand FreeRTOS_tasks.c to see what functions are involved, as shown in Figure 6-3. Unfortunately, the tool that I am using at that point removes the coloring of the violations, so we must manually trace them. However, examining the expanded diagram, you can see clearly why the two violations occur.

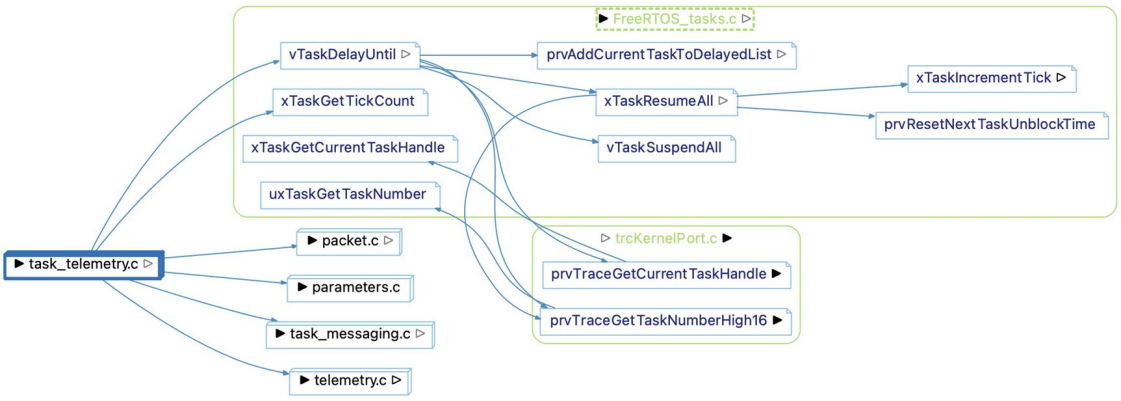

Figure 6-3 Expanding the modules to trace the cyclical violation fully reveals the functions involved: xTaskGetCurrentTaskHandle and uxTaskGetTaskNumber

First, vTaskDelayUntil (FreeRTOS_tasks.c) makes a call to prvTraceGetCurrentTaskHandle (trcKernelPort.c) which then calls xTaskGetCurrentTaskHandle (FreeRTOS_tasks.c).

Second, vTaskDelayUntil (FreeRTOS_tasks.c) makes a call to xTaskResumeAll (FreeRTOS_tasks.c) which calls prvTraceGetTaskNumberHigh16 (trcKernelPort.c) which then calls uxTaskGetTaskNumber (FreeRTOS_tasks.c).

These two modules have cyclical dependencies, which we want to avoid in our architectures! Now in FreeRTOS, we are less concerned with this small cycle because it doesn't affect our use of FreeRTOS because it is abstracted away behind the interfaces. Since the violation occurs in the third-party library and is contained to it, there is little risk of the violation impeding our application's scalability or causing other architectural issues.

> **Note** If you use FreeRTOS, the cyclical dependency we are looking at does not affect anything in our application and is no cause for any genuine concern.

If you discover a cyclical dependency in your code, it usually means that the code structure needs to be refactored to better group functionality and features. For example, looking back at the FreeRTOS example, xTaskGetCurrentTaskHandle and uxTaskGetTaskNumber should not be in FreeRTOS_tasks.c but, instead, be relocated to trcKernelPort.c. Moving these functions to trcKernelPort.c would break this cycle and improve the

code structure and architecture.

> **Caution** We are examining FreeRTOS in a very narrow scope! Based on the larger picture of the kernel, these changes might not make sense. However, the technique for finding issues like this is sound.

Architectural quality is key to improving a product's quality. The architecture is the foundation from which everything is built upon in the code base. If the architecture is faulty or evolves in a haphazard manner, the result will be lower quality, more time-consuming updates, and greater technical debt for the system. The architecture alone won't guarantee that the software product is high quality. The code itself also has to be written so that it meets the quality level you are attempting to achieve.

Code Quality

Software architecture quality is essential and helps to ensure that the technical layering and domains maintain high cohesion and low coupling. However, software architecture alone won't guarantee that the software will be of high quality. How the code is constructed can have just as much of an impact, if not more, on the software's quality.

When it comes to code quality, there's quite a bit that developers can do to ensure the result is what they desire. For example, low-hanging fruit that is obvious but I often see overlooked is making sure that your code compiles without warnings! Languages like C set low standards for compiler checking, so if the compiler is warning you about something, you should pay attention and take it seriously.

When I look at defining code quality, there are several areas that I like to look at, such as

- The code adheres to industry best practices and standards for the language.
- The code's function complexity is minimized and meets defined code metrics.
- The code compiles without warnings and passes static code analysis.

- The code unit test cases have 100% branch coverage.
- The code has gone through the code review processes.

Let's look at each of these areas and how they help to contribute to improving the quality of embedded software.

Coding Standards

Coding standards and conventions are a set of guidelines for a specific programming language that recommend programming style, practices, and methods for each aspect of a program written in that language.[7] Developers should not think of standards as rules handed down by God that every programmer must follow to avoid programmer hell, but instead as best practices to improve quality and make the code more maintainable. Sometimes, the rules make sense for what you are doing; other times, they don't and should be cast aside. The idea behind the standards is that if developers follow them, they will avoid common mistakes and issues that are hell to debug!

The C programming language, which is still the most common language used in embedded systems development, has several standards developers can use to improve their code quality, such as **GNU**,[8] **MISRA C**,[9] and **Cert C**.[10] Typically, each standard has its focus. For example, MISRA C is a standard that focuses on writing safe, portable, and reliable software. On the other hand, Cert C focuses on writing secure software. In many cases, developers may decide to follow more than one standard if the standards do not conflict.

In many cases, a static code analysis tool can check adherence to a standard. For example, I'll often use a tool like Understand to perform MISRA C checking on the code I am developing. For MISRA, a static code analysis can be used to verify many directives. However, some require a code review and must be performed manually. In many cases, static analysis can even be used to confirm that the code meets the style guide the team uses. Attempting to manually check that a code base is meeting a standard is foolish. An automated static analysis tool is the only way to do this efficiently and catch everything. Be warned though; the output from a static

analysis tool will vary by vendor. It is not uncommon for teams focused on high quality to use more than one static analysis tool.

Best Practice Use an analysis tool to verify that your software meets your following standards.

Standards help developers ensure that their code follows industry best practices. While we often think that we will catch everything ourselves, if you take a minute to browse Gimpel Software's old "The Bug of the Month"[11] archive, you'll see how easily bugs sneak through. Did you know, if you examine the C standard(s) appendix, you'll find around 200 undefined implementation-dependent behaviors? If you aren't careful, you could have code that behaviors one way with compiler A, and then the same code behaves totally different using compiler B. That's some scary (insert your preferred profanity statement)!

Standards can also help to ensure that no matter how many developers are working on the project, the code is consistent and looks like just a single developer wrote it. There are a few things more difficult than trying to review code that is poorly organized and commented and varies wildly in its look and feel. When the code looks the same no matter who wrote it, it becomes easier to perform code reviews and focus on the intent of the code rather than the style of the code.

The goal of a standard is to provide developers with best practices that will help them to avoid problems that would have otherwise bogged them down for hours or weeks of painful debugging. It's critical that developers take the time to understand the standards that are pertinent to their industry and put in place the processes to avoid costly mistakes.

Software Metrics

A software metric is a measurement that developers can perform on their code that can be used to gain insights into the behavior and quality of their software. For example, a standard metric that many teams track is lines of code (LOC). The team can use LOC for many purposes, such as

- Determining the rate at which the team generates code

- Calculating the cost to develop software per LOC
- Examining how the code base changes over time (lines added, modified, removed, etc.)
- Identifying where a module may be growing too large and refactoring may be necessary

Developers do need to be careful with their metrics, though. For example, when we talk about LOC, are we talking about logical lines, comment lines, or both? It's essential to be very clear in the definition of the metric so that everyone understands exactly what the metric means and what is being measured. Before using a metric, developers also need to determine why the metric is important and what the changes in that metric mean over time. Tracking the trend in metrics can be as critical taking the measurement. A trend will tell you if the code is heading in the right direction or slowly started to fall over a cliff! Also, don't put too much stock in any one metric. Collectively examining the trends will provide far more insights into the software quality state.

Developers can track several quality metrics that will help them understand if potential issues are hiding in their software. For example, some standard metrics that teams track include

- McCabe Cyclomatic Complexity
- Code churn (lines of code modified)
- CPU utilization
- Assertion density
- Test case code coverage
- Bug rates
- RTOS task periodicity (minimum, average, and max)

There are certainly plenty of metrics that developers can track. The key is to identify metrics that provide your team insights into the characteristics that will benefit you the most. For example, don't select metrics just to collect them! If LOC doesn't tell you anything constructive or exciting, don't track it! No matter what metrics you choose, make sure you take the time to track the trends and the changes in the metrics over time.

> **Tip** If you want to get some ideas on metrics you can track, install a program like Metrix++, Understand, etc., and see the giant list of metrics they can track. Then select what makes sense for you.

An important metric that is often looked by teams involve tracking bugs. Bug tracking can show how successful a team is at creating quality code. Capers Jones has written about this extensively in several of his books and papers. Jones states that "defect potential and defect removal efficiency are two of the most important metrics" for teams to track. It's also quite interesting to note that bugs tend to cluster in code. For example:

- Barry Boehm found that 80% of the bugs are in 20% of the modules.[12]
- IBM found 57% of their defects in 7% of their modules.[13]
- The NSA found 95% of their bugs in 2.5% of their code.[14]

The trends will often tell you more about what is happening than just a single metric snapshot. For example, if you monitor code churn and LOC, you may find that over time, code churn starts to rise while the LOC added every sprint slowly declines. Such a case could indicate issues with the software architecture or code quality issues if, most of the time, developers are changing existing code vs. adding new code. The trend can be your greatest insight into your code.

One of my favorite metrics to track, which tells us quite a bit about the risk our code presents, is McCabe's Cyclomatic Complexity. So let's take a closer look at this metric and see what it tells us about our code and how we can utilize it to produce higher-quality software.

McCabe Cyclomatic Complexity

Several years ago, I discovered a brain training application for my iPad that I absolutely loved. The application would test various aspects of the brain, such as speed, memory, agility, etc. I was convinced that my memory was a steel trap; anything entering it stayed there, and I could recall it with little to no effort. Unfortunately, from playing the games in the application, I discovered that my memory is not as good as I thought and that it could only hold 10–12 items at any given time. As hard as I might try, remembering 13 or 14 things was not always reasonable.

The memory game is a clear demonstration that despite believing we are superhuman and capable of remembering everything, we are, in fact, quite human. What does this have to do with embedded software and quality? The memory game shows that a human can easily handle and track 8–12 items at a time on average.[15] Or in software terms, developers can remember and manage approximately 8–12 linearly independent paths (branches) at any given time. Writing functions with more than ten branches means that a developer may suddenly struggle to remember everything happening in that function. This is a critical limit, and it tells developers that to minimize risk for bugs to creep into their software, the function should be limited to ten or fewer independent paths.

McCabe Cyclomatic Complexity (Cyclomatic Complexity) is a measurement that can be performed on software that measures the number of linearly independent paths within a function. Cyclomatic Complexity is a valuable measure to determine several factors about the functions in an application, which include

- The *minimum* number of test cases required to test the function
- The risk associated with modifying the function
- The likelihood that the function contains undiscovered bugs

The value assigned to the Cyclomatic Complexity represents the complexity of the function. The higher the value, the higher the risk (along with a larger number of test cases to test the function and paths through the function).

Table 6-1 provides a general rule of thumb for mapping the McCabe Cyclomatic Complexity to the functions' associated risks. Notice that the risk steadily rises from minimal risk to untestable when the Cyclomatic Complexity reaches 51! Any value over 50 is considered to be untestable! There is a simple explanation for why the risk increases with more complexity; humans can't track all that complexity in their minds at once, hence the higher the likelihood of a bug in the function.

Table 6-1 McCabe Cyclomatic Complexity value vs. the risk of bugs

Cyclomatic Complexity	Risk Evaluation
1–10	A simple function without much risk

Cyclomatic Complexity	Risk Evaluation
11–20	A more complex function with moderate risk
21–50	A complex function of high risk
51 and greater	An untestable function of very high risk

Best Practice The Cyclomatic Complexity for a function should be kept to ten or less to minimize complexity and the risk associated with the function.

As the Cyclomatic Complexity rises, the risk of injecting a new defect (bug) into the function while fixing an existing one or adding functionality increases. For example, look at Table 6-2. You can see that there is about a 1 in 20 chance of injecting a new bug into a function if its complexity is between one and ten. However, that risk is four times higher for a function with a complexity between eleven and twenty!

Lower Cyclomatic Complexity for a function has two effects:

1) The less complex the code is, the less likely the chance of a bug in the code.
2) The less complex the code is, the lower the risk of injecting a bug when modifying the function.

Wise developers looking to produce quality software while minimizing the amount of time they spend debugging will look to Cyclomatic Complexity measurements to help guide them. Managing complexity will decrease the number of bugs that are put into the software in the first place! The least expensive and least time-consuming bugs to fix are the ones that are prevented in the first place.

Table 6-2 The risk of bug injection as the Cyclomatic Complexity rises[16,17]

Cyclomatic Complexity	Risk of Bug Injection
1–10	5%

Cyclomatic Complexity	Risk of Bug Injection
11–20	20%
21–50	40%
51 and greater	60%

What happens when a developer discovers that they have a high Cyclomatic Complexity for one of their functions? Let's look at the options. First, a developer should determine the function's actual risk. For example, if the Cyclomatic Complexity value is 11–14, it's higher than desired, but it may not be worth resolving if the function is readable and well documented. That range is "up to the developer" to use their best judgment.

Next, for values higher than 14, developers have a function that is trying to do too much! The clear path forward is to refactor the code. Refactoring the code, in this case, means going through the function and breaking it up into multiple functions and simplifying it. The odds are you'll find several different blocks of logic in the function that are doing related activities but don't need to be grouped into a single function. You will discover that some code can be broken up into smaller "helper" functions that make the code more readable and maintainable. If you are worried about the overhead from extra function calls, you can inline the function or tell the compiler that those functions should be compiled for speed. With most modern microcontrollers and toolchains today, the concern for inlining a function is generally not a concern because the compiler will inline where it makes sense to optimize execution and microcontrollers are very efficient.

Code Analysis (Static vs. Dynamic)

Developers can use two types of code analysis to improve the quality of their software: static and dynamic analyses. Static code analysis is an analysis technique performed on the code without executing the code.

Dynamic analysis, on the other hand, is an analysis technique performed on the code while the code is running.

Static analysis is often performed on the code to identify semantic and language issues that the compiler would not otherwise detect. For example, a compiler is not going to detect that you are going to overflow the stack, while a static analysis tool might be able to do so. Static analysis can also verify that code meets various industry standards and coding style guides. For example, it's not uncommon for a team to use a static code analyzer to confirm that their code meets the MISRA-C or MISRA-C++ standards. Static analysis tools can vary in price from free to tens of thousands of dollars. Typically, the more expensive the tool is, the more robust the analysis.

Dynamic code analysis is performed on the code to understand the runtime characteristics of the system. For example, a developer might use a tool like **Percepio Tracealyzer**[18] to collect and analyze the runtime behavior of their RTOS application. Developers can check the periodicity of their tasks, the delay time between when the task is ready to run and when it runs, CPU utilization, memory consumption, and a bunch of other exciting and valuable metrics.

In some cases, static and dynamic analyzers can give you two different pictures of the same metric. For example, if you are interested in knowing how to set your stack size, a static analysis tool can calculate how deep your stack goes during analysis. On the other hand, developers can also use a dynamic analysis tool to monitor their stack usage during runtime while running all their worst test cases. The two results should be very similar and can then help developers properly set their stack size.

Achieving 100% Branch Coverage

Another great metric, although one that can be very misleading, is branch coverage. Branch coverage tells a developer whether their test cases cause every branch in their code to be executed or not. Branch coverage can be misleading because a developer sees the 100% coverage and assumes that there are no bugs! This is not the case. It means the tests cover

all the branches, but the code may or may not meet the system requirements, perform the function the customer wants, or cover boundary conditions related to variable values.

The nice thing about branch coverage is that it requires developers to write their unit tests for their modules! When you have unit tests and use a tool like gcov, the tools will tell you if you haven't covered all your branches. For example, Figure 6-4 shows the results of running unit tests on a code base where you can see that not every branch was executed.

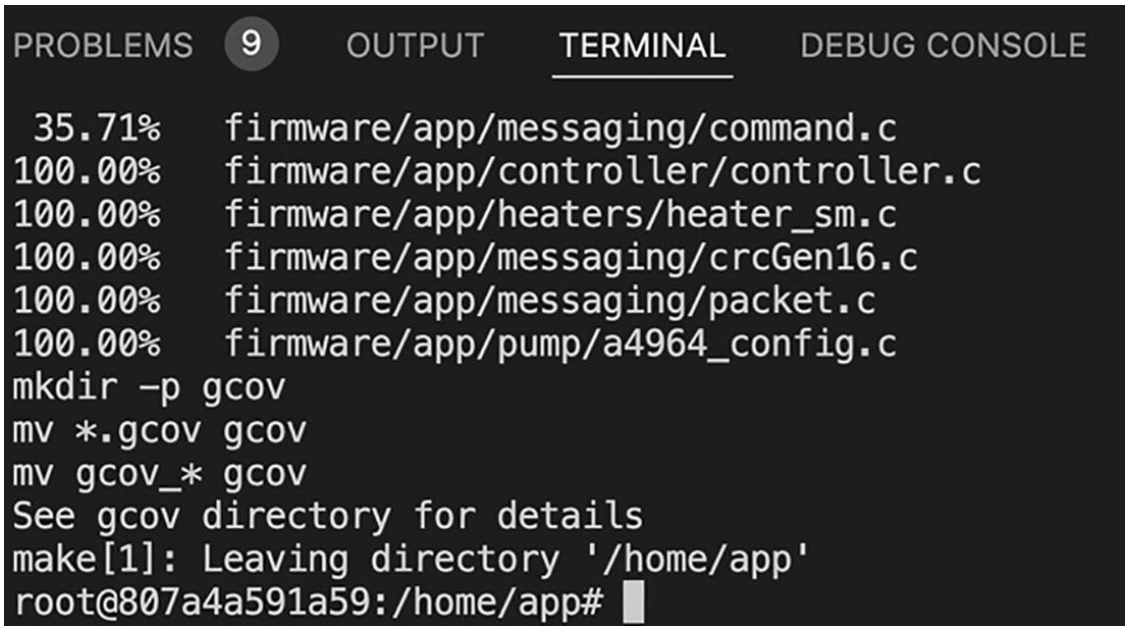

Figure 6-4 gcov reports the coverage that unit tests reach in the code

Notice from Figure 6-4 that most of the modules have 100% coverage, but there is one module named command.c that only has 35.71% coverage. The nice thing about this report is that I now know that command.c is only tested 35.71%. There very well could be all kinds of bugs hiding in the command.c module, but I wouldn't know because all the branches were not covered in my tests! The gcov tool will produce an analysis file with the extension gcov. We can go into the file and search ####, which will be prepended to the lines of code that were not executed during our tests. For example, Figure 6-5 shows that within command.c, there is a function named Command_Clear that was not executed! It looks like a series of function stubs that have not yet been implemented!

```
268        -:  264:*******************************************************************************/
269    #####:  265:static void Command_Clear(uint8_t const * const Data)
270        -:  266:{
271        -:  267:    // This is just to use the passed parameters to remove static analysis errors
272        -:  268:    // that we don't use the passed variable.
273        -:  269:    (void)Data;
274    #####:  270:}
```

Figure 6-5 gcov identifies an untested function and line of code using ####

Writing test cases that get 100% branch coverage won't remove all the bugs in your code, but it will go a long way in minimizing the bugs, making it easier to detect bugs as they are injected, and improving the quality of the code. Don't let it lure you into a false sense of security, though! 100% code coverage is not 100% bug free!

During one of my advisory/mentoring engagements with a customer, I was working with an intelligent junior engineer who disagreed with me on this topic. "If the code has 100% coverage, then the code is 100% bug free," he argued. I politely listened to his arguments and then assigned him to write a new code module with unit tests and a simulation run to show that the module met requirements. In our next meeting, he conceded that 100% test coverage does not mean 100% bug free.

Code Reviews

When one thinks about getting all the bugs out of an embedded system, you might picture a team of engineers frantically working at their benches trying to debug the system. Unfortunately, debugging is one of the least efficient methods for getting the bugs out of a system! Single-line stepping is the worst, despite being the de facto debugging technique many embedded software developers choose. A better alternative is to perform code reviews!

> A code review is a manual inspection of a code base that is designed to verify

- There are no logical or semantic errors in the software.
- The software meets the system requirements.
- The software conforms to approved style guidelines.

Code reviews can come in many forms. First, they can be large, formalized, process-heavy beasts where there are four to six people in a dimly lit room pouring over code for hours on end. These code reviews are inefficient and a waste of time, as we will soon see. Next, code reviews can be lightweight reviews where developers perform short, targeted code re-

views daily or when code is available to review. These code reviews are preferable as they find the most defects in the shortest time. Finally, code reviews fall throughout the spectrum between these two. The worst code reviews are the ones that are never performed!

Several characteristics make code reviews the most efficient mechanism to remove bugs, but only if the right processes are followed. For example, SmartBear performed a study where they found examining more than 400 LOC at a time resulted in finding fewer defects! Look at Figure 6-6 from their research. The defect density, the number of defects found per thousand lines of code (KLOC), was dramatically higher when less code was reviewed. Conversely, the more code reviewed in a session resulted in a lower defect density.

Taking a moment to think through this result does make a lot of sense. When developers start to review code, their minds are probably engaged and ready to go. At first, they spot a lot of defects, but as time goes on in the review and they look at more and more code, the mind probably starts to glaze over the code, and defects start to get missed! I do a fair number of code reviews, and I know that after looking in detail at several hundred lines of code, my mind starts to get tired and needs a break!

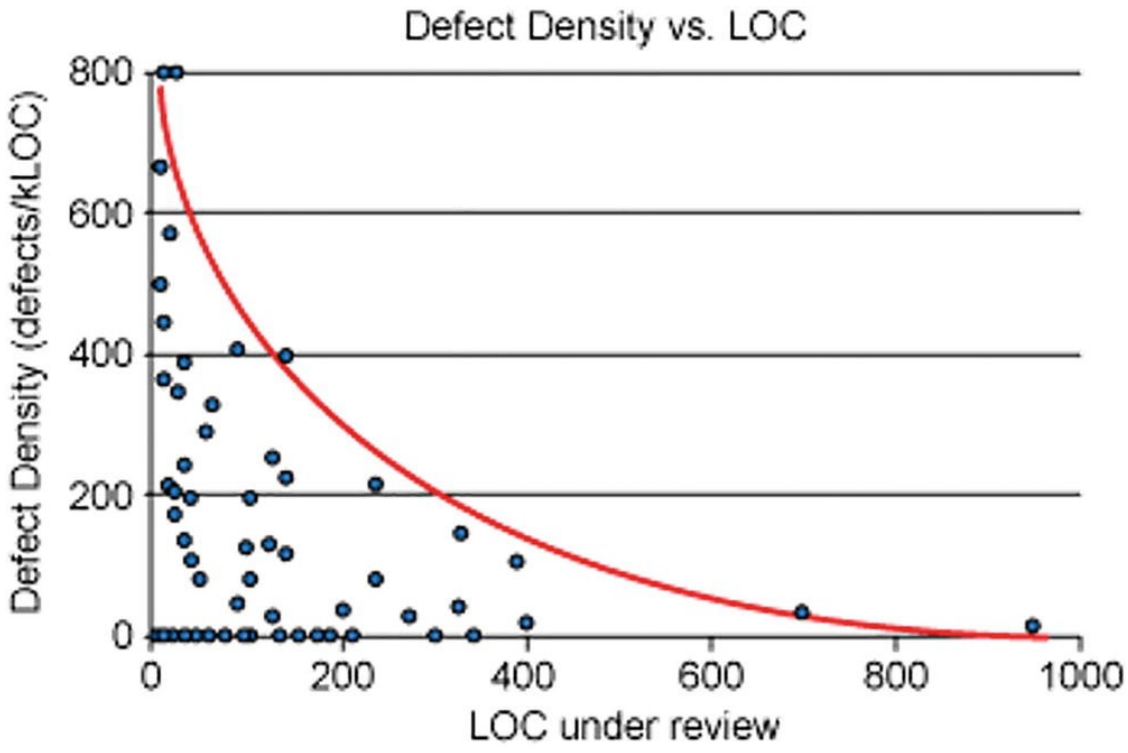

Figure 6-6 Defect density per KLOC based on the number of lines reviewed in a session[19]

A successful code review requires developers to follow several best practices to maximize the benefit. Code review best practices include

- Review no more than 400 LOC at a time.
- Review code at a rate of less than 500 LOC per hour, although the optimal pace is 150 LOC per hour.
- Perform code reviews daily.
- Perform code reviews when code is ready to be merged.
- Avoid extensive, formalized code reviews; instead, perform smaller team or individual code reviews.
- Keep a checklist of code review items to watch for.

Case Study – Capstone Propulsion Controller

Now that we've examined a few aspects of software quality, let's look at an example. I recently worked on a project that I found to be quite interesting. The project involved architecting and implementing the software for a propulsion controller used on NASA's Capstone mission. Capstone is a microwave oven–sized CubeSat weighing just 55 pounds that will serve as the first spacecraft to test a unique, elliptical lunar orbit as part of the Cislunar Autonomous Positioning System Technology Operations and Navigation Experiment (CAPSTONE)[20] mission. I worked with my customer, **Stellar Exploration**,[21] to help them design, build, and deploy the embedded software for their propulsion controller in addition to advising and mentoring their team.

From the 30,000-foot view, a propulsion system consists of a controller that receives commands and transmits telemetry back to the flight computer. The controller oversees running a pump that moves fuel through the fuel lines to valves that can be commanded to an open or closed position. (It's much more complicated than this, but this level of detail serves our purpose.) If the valve is closed, the fuel can't flow to the combustion chamber, and no thrust is produced. However, if the valve is opened, fuel can flow to interact with the catalyst and produce thrust. A simplified system diagram can be seen in Figure 6-7.

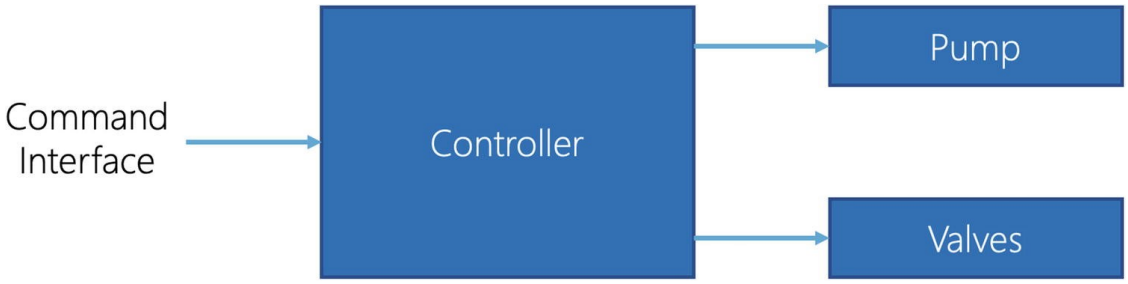

Figure 6-7 A simplified block diagram of a propulsion system

Even though the controller seems simple, it consists of ~113,000 LOC. That's a lot of code for a little embedded controller! If we were to take a closer look, we'd discover why you can't just blindly accept a metric like LOC! Figure 6-8 shows the breakdown of the code that was measured using Understand. Notice that there is only 24,603 executable LOC, a reasonable number for the complexity of the control system.

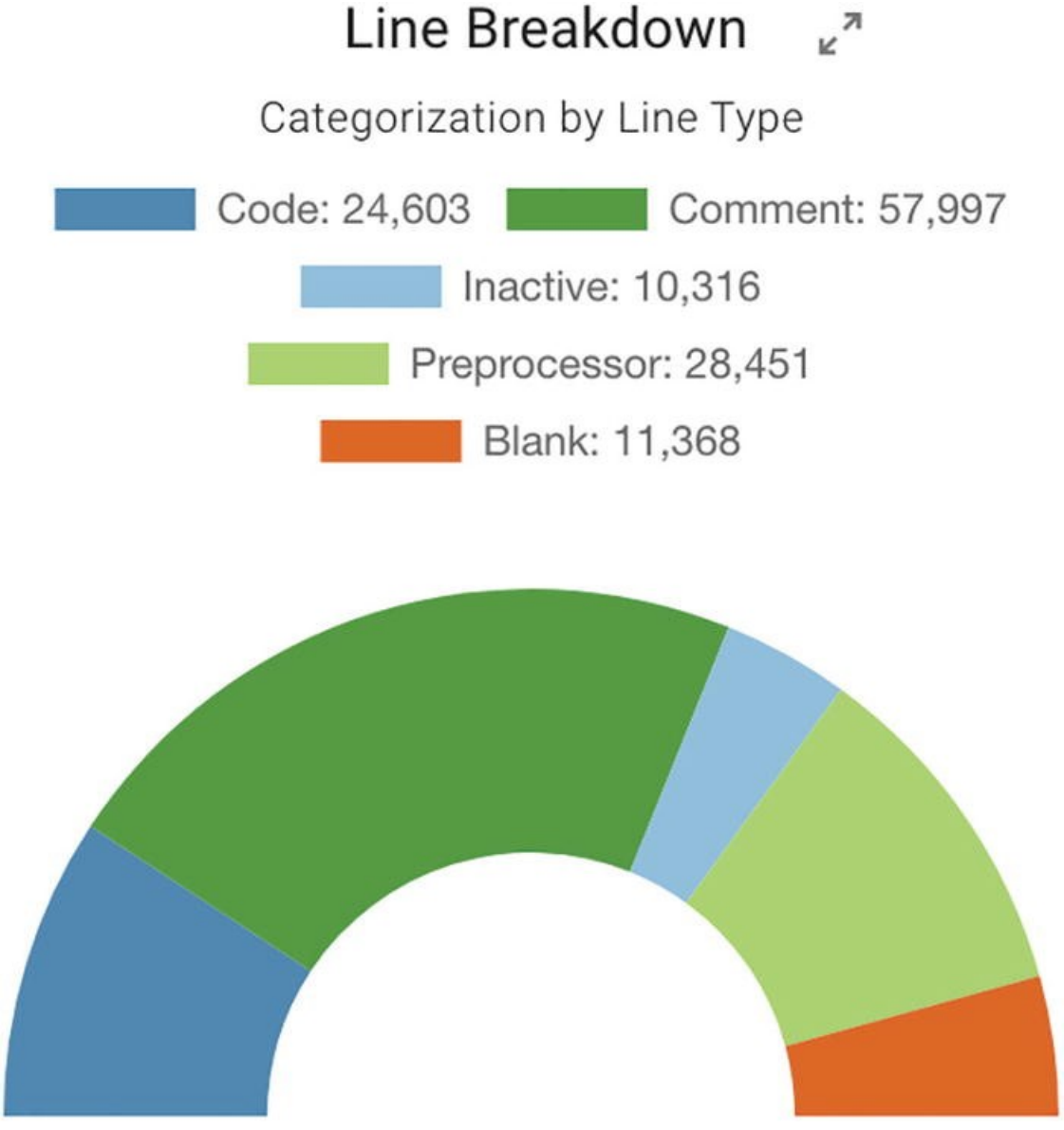

Figure 6-8 Capstone propulsion controller code breakdown for the core firmware

The LOC breakdown can help us gain a little insight into the code without looking at a single LOC! For example, we can see that the code base is most likely relatively well documented. We can see that there are ~60,000 LOC that are comments.

One metric I like to track on projects that some people would argue about is the comment-to-code ratio. Many argue that you don't need code comments because the code should be self-documenting. I agree, but I've also found that LOC does not explain how something should work, why it was chosen to be done that way, and so forth. For this reason, I prefer to see a comment-to-code ratio greater than 1.0. For this code base, you can see that the ratio is ~2.36 comments per LOC. We don't know the quality of the comments, but at least there are comments!

Best Practice Comment code with your design decisions, intended behavior, and so forth. Your future self and colleagues will be thankful later!

Someone being picky might look at Figure 6-8 and ask why there are ~10,000 lines of inactive code. The answer is quite simple. A real-time operating system was used in the project! An RTOS is often very configurable, and that configuration usually uses conditional preprocessor statements to enable and disable functionality. Another contributor to the inactive lines could be any middleware or vendor-supplied drivers that rely on configuration. Configuration tools often generate more code than is needed because they are trying to cover every possible case for how the code might be used. Just because there are inactive lines doesn't mean there is a quality issue. It may just point to a very configurable and reusable code base!

One metric we've been looking at in this chapter is Cyclomatic Complexity. So how does the propulsion controller code base look from a Cyclomatic Complexity standpoint? Figure 6-9 shows a heat map of the maximum Cyclomatic Complexity by file. The heat map is set so that the minimum is 0 and the maximum is 10. Files in green have functions well below the maximum. As functions in a file reach the maximum, they begin to turn a shade of red. Finally, when the file's function complexity exceeds the maximum, they turn red. At a quick glance, we can see what

files may have functions with a complexity issue.

> **Note** If you can't read the filenames in Figure 6-9, that is okay! I just want you to get a feel for the technique vs. seeing the details of the code base.

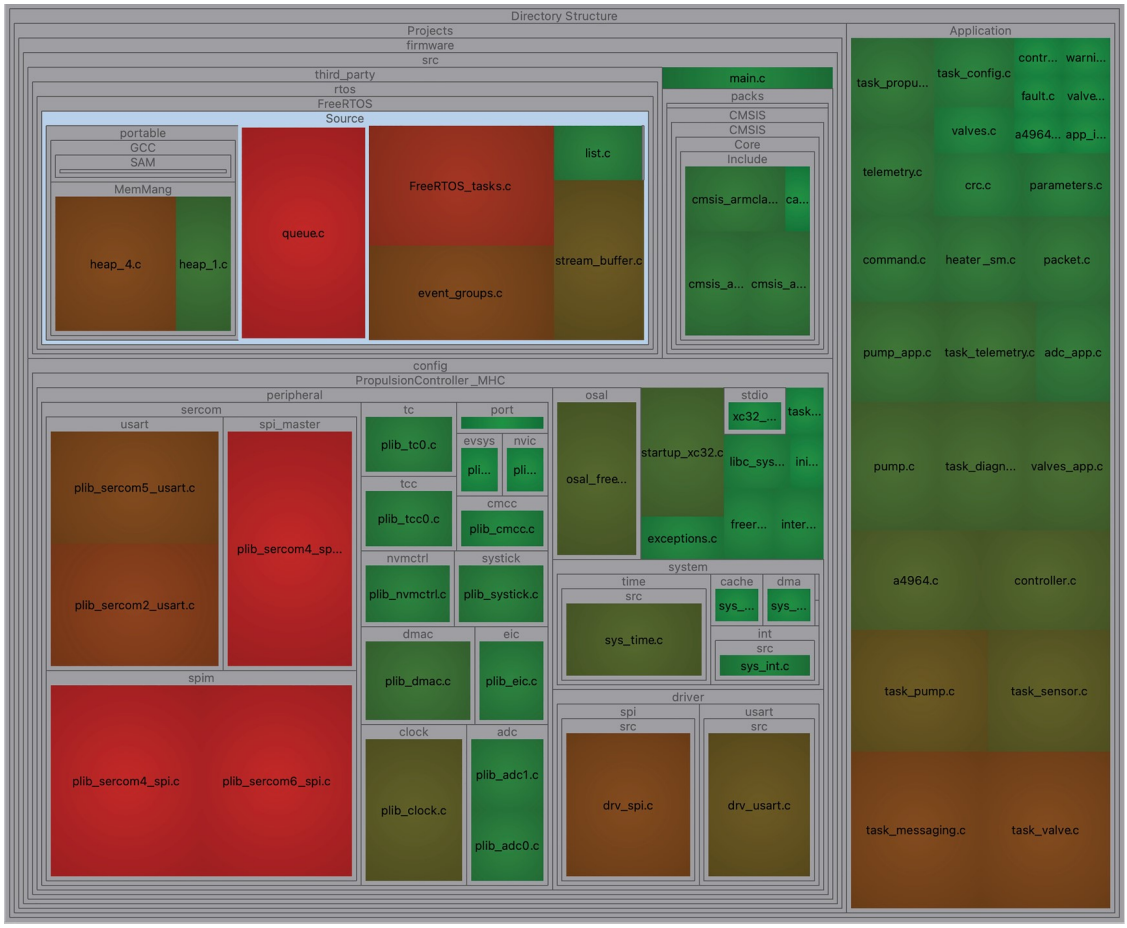

Figure 6-9 An example Cyclomatic Complexity heat map to identify complex functions

A glance at the heat map may draw a lot of attention to the red areas of the map! The attention drawn is a good thing because it immediately puts our sights on the modules that contain functions that may require additional attention! For example, five files are a deep red! Don't forget; this does not mean that the quality of these files or the functions they contain is poor; it just means that the risk of a defect in these functions may be slightly higher, and the risk of injecting a defect when changing these functions is elevated.

We can investigate the details of the functions that are catching our attention in several ways. One method is to generate another heat map, but, this time, generate the heat map on a function basis rather than a file basis. The new heat map would look something like Figure 6-10.

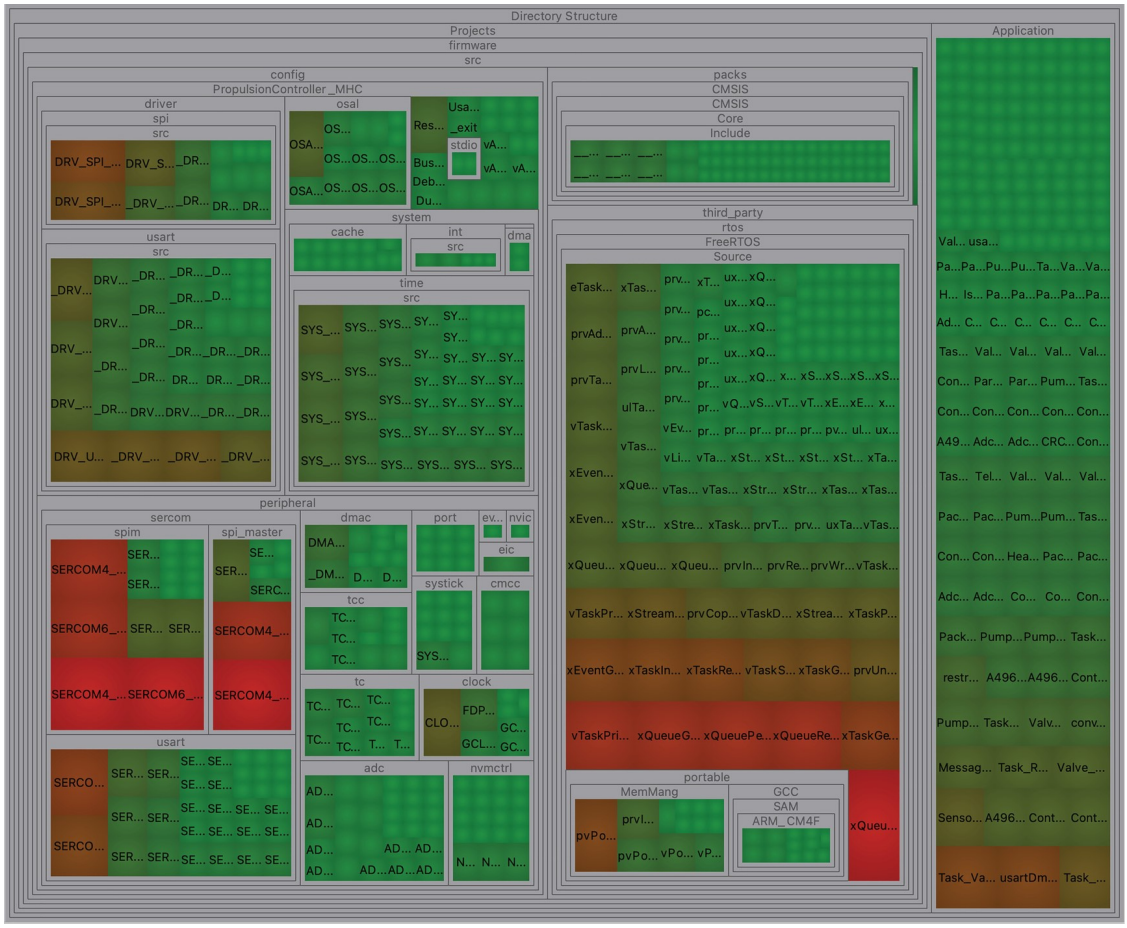

Figure 6-10 A Cyclomatic Complexity heat map displayed by function

We can see from the new heat map which functions are causing the high complexity coloring. Two areas are causing the high complexity readings. First, several serial interrupts have a Cyclomatic Complexity of 17. These serial interrupt handlers are microcontroller vendor–supplied handlers. The interrupts are generated automatically by the toolchain that was used. The second area where the complexity is higher than ten is in the RTOS! The RTOS has high complexity in the functions related to queues and task priority settings.

I think it is essential at this point to bring the reader's attention to something important. The heat maps appear to show a lot of red, which catches a human's eye. However, out of the 725 functions that make up the software, less than 15 functions have a complexity more significant than 10! Examine the breakdown shown in Figure 6-11. You'll see that there are 13 in total with not a single function with a complexity more significant than 17.

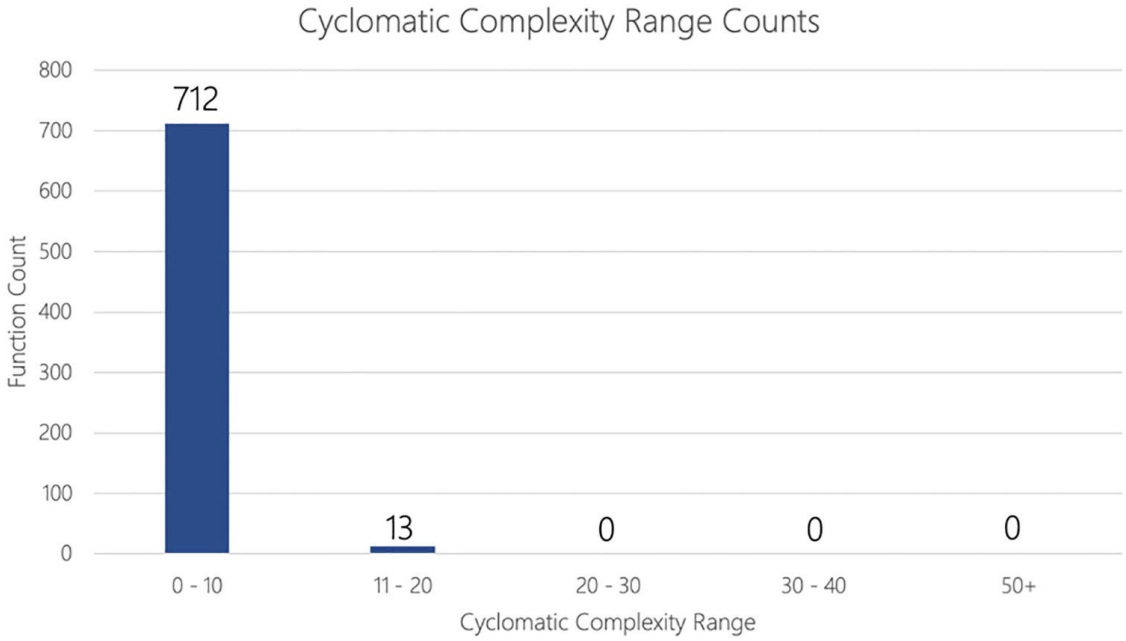

Figure 6-11 The number of functions per Cyclomatic Complexity range for the propulsion controller software

The controller software was written to minimize complexity and followed at least the quality techniques we've discussed so far in this chapter. Without context, it can be easy to look at the complexity and say, "Yea, so it should be that way!". I've been blessed with the opportunity to analyze a lot of embedded software in my career. An average representation of what I typically see can be seen in Table 6-3. This table averages projects in the same size range for a total number of functions.

Table 6-3 An average project representation for several projects with their average function Cyclomatic Complexity ranges

-

Project	*Function Total*	1 - 10	11 - 20	21 - 50	51+
A	*530*	494	26	8	2
B	*1058*	1016	31	7	4
C	*3684*	3378	205	72	29
D	*4190*	4086	61	32	11
E	*4973*	*4884*	*53*	*28*	*8*

When I first put this table together, I was expecting to see the worst results in the largest projects; however, I would have been wrong. The projects that I've analyzed to date seem to suggest that mid-sized projects struggle the most with function complexity. All it takes though is one poor project to skew the results. In any event, notice on average even the smallest projects can end up with functions that are untestable and may hold significant risk for bugs.

High complexity in third-party components brings a development team to an interesting decision that must be made: Do you live with the complexity, or do you rewrite those functions? First, I wouldn't want to modify an RTOS kernel that I had not written. To me, that carries a much higher risk than just living with a few functions with higher complexity. After all, if they have been tested, who cares if the risk of injecting a bug is high because we will not modify it! If we are concerned with the functions in questions, we can also avoid using them the best we can.

The serial interrupt handlers also present themselves as an interesting conundrum. The complexity in these handlers is higher because the drivers are designed to be configurable and handle many cases. They are generic handlers. The decision is not much different. Do you go back and rewrite the drivers and handlers, or do you test them to death? The final decision often comes down to these factors:

- What is the cost of each?
- Is there time and bandwidth available?
- Can testing reduce the risk?

In this case, after examining the complex functions, it would be more efficient and less risky to perform additional testing and, when the time was available on future programs, to go back and improve the serial drivers and handlers. After all, they performed their function and purpose to the requirements! Why rewrite them and add additional risk just to appease a metric?

Best Practice Never add cost, complexity, or risk to a project just to meet some arbitrary metric.

So far, I've just walked you through the highest level of analysis performed regularly as the controller software was developed. I realize now that I would probably need several chapters to give all the details. However, I hope it has helped to provide you with some ideas about how to think about your processes, metrics, and where your defects may be hiding. The following are a few additional bullet points that give you some idea of what else was done when developing the controller software:

- Consistent architectural analysis and evolution tracking.
- Unit tests were developed to test the building blocks with Cyclomatic

Complexity being used to ensure there were enough tests.
- Regression testing using automated methodologies.
- Algorithm simulations to verify expected behavior with comparisons to the implemented algorithms.
- Runtime performance measurements using Percepio Tracealyzer to monitor task deadlines, CPU utilization, latency, and other characteristics.
- Metrics for architecture coupling and cohesion.
- System-level testing (lots and lots and lots of it).
- Test code coverage monitoring.
- Code reviews.

A lot of sweet and effort goes into building a quality software system. Even with all the effort, it doesn't mean that it is bug or defect free. Situations can come up that are unexpected or expected and reproducible like a single event upset or a cosmic ray corrupting memory. With the right understanding of what quality means, and backing it up with processes, metrics, and testing, you can decrease the risk that an adverse event will occur. Systems can be complex, and unfortunately there are never guarantees.

(As of this writing, Capstone is successfully on its path to orbit the Moon and has successfully completed several burns as scheduled. If all goes according to plan, it will enter its final orbit in November 2022.)

Final Thoughts

Defining software quality for you and your team is critical. We've explored several definitions for quality and explored the metrics and processes that are often involved in moving toward higher-quality development. Unfortunately, we've just been able to scratch the surface; however, the details we've discussed in this chapter can help put you and your team on a firm footing to begin developing quality embedded software. In the rest of this book, we will continue to explore the modern processes and techniques that will help you be successful.

Action Items

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

- How do you define software quality? First, take some time to define what software quality is to you and your team and what level is acceptable for the products you produce.
- What characteristics tell you that you have a quality architecture? Take some time this week to analyze your software architecture. What areas of improvement do you find?
- Improve your metrics tracking:
  - Make a list of code metrics that should be tracked.
  - Note which ones you currently track and which ones you don't.
  - Add another note for the period at which those metrics should be reviewed.
  - What changes can you make to your processes to make sure you track the metrics and the trends they produce? (Do you need a checklist, a check-in process, etc.?)
- How complex is your code? Install a free Cyclomatic Complexity checker like pmccabe and analyze your code base. Are there functions with a complexity greater than ten that you need to investigate for potential risk?
- What coding standards do you use? Are there any new standards you should start to follow? What can you do to improve standard automatic checking in your development processes?
- Code reviews are the most effective method to remove defects from software. How often are you performing your code reviews? Are you following best practices? What changes can you make to your code review process to eliminate more defects faster?
- What processes are you missing that could help improve your software quality? Are the processes missing or is it the discipline to follow the processes? What steps are you going to take today and in the future to improve your quality?

Footnotes

1 www.thefire.org/first-amendment-library/decision/jacobellis-

v-ohio/

2 Board (IREB), International Requirements Engineering. "**Learning from history: The case of Software Requirements Engineering – Requirements Engineering Magazine**." *Learning from history: The case of Software Requirements Engineering – Requirements Engineering Magazine*. Retrieved 2021-02-25.

3 **Pressman, Roger S.** (2005). *Software Engineering: A Practitioner's Approach* (Sixth International ed.). McGraw-Hill Education. p. 388. **ISBN** **0071267824**.

4 https://en.wikipedia.org/wiki/Software_quality

5 www.scitools.com/

6 https://structure101.com/

7 https://en.wikipedia.org/wiki/Coding_conventions

8 www.gnu.org/prep/standards/html_node/Writing-C.html

9 www.misra.org.uk/

10 https://wiki.sei.cmu.edu/confluence/display/c

11 www.gimpel.com/archive/bugs.htm

12 Barry Boehm, *Software Engineering Economics*.

13 *Facts and Fallacies of Software Engineering*, Robert Glass et al.

14 Thomas Drake, Measuring Software Quality: A Case Study.

15 Miller, G. (1956). Found the "magical" number is 7 +/– 2 things in short-term memory.

16 McCabe, Thomas Jr. Software Quality Metrics to Identify Risk. Presentation to the Department of Homeland Security Software Assurance Working Group, 2008.

17 (www.mccabe.com/ppt/SoftwareQualityMetricsToIdentifyRisk.ppt#36) and Laird, Linda and Brennan, M. Carol. Software Measurement and Estimation:

A Practical Approach. Los Alamitos, CA: IEEE Computer Society, 2006.

18 https://percepio.com/

19 https://smartbear.com/learn/code-review/best-practices-for-peer-code-review/

20 www.nasa.gov/directorates/spacetech/small_spacecraft/capstone

21 www.stellar-exploration.com/
