J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_15

# 15. The Right Tools for the Job

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

When I teach a class, speak at a conference, or work with a client, one of the most asked questions I get is related to the tools I use to get the job done. Of course, engineers love their tools, and they should! Tools help them get their job done; if you aren't using the right tools, it will most likely take longer and cost more to get the job done. However, I often find that engineers and companies overlook the benefits and value of tools. This chapter will explore the benefits of using the right tools, how to justify getting those tools, and what tools you need to succeed. I'll also share a little about the tools I use personally, but keep in mind it changes and evolves yearly. (You might consider subscribing to my newsletter to stay up to date on new and exciting tools![1])

## The Types of Value Tools Provide

Imagine for a moment that you are building a house or a shed or finishing a basement. If you wanted to do so as fast as possible, with minimal labor, would you choose to frame the structure using a hammer and nails or nail gun and nails? Would you use a screwdriver to install the drywall or a cordless drill? The tools we choose to use can have a dramatic effect on a project's success. When it comes to embedded software engineering worldwide, it would surprise you how many teams use a "hammer and nails."

The tool's value is the first and most important consideration when selecting and purchasing tools. Value is the importance, worth, or usefulness of something.[2] Value comes in four different types:

- Monetary

- Functional
- Emotional
- Social

Monetary value can be equated to the direct revenue generated or saved from having and using the tool. Economic value can also be calculated from the time saved on a project. Development requires time which requires developers who need monetary payment for their services. If a tool can allow a developer to get more done, that tool provides economic value to the company.

Functional value is something that solves a problem. For example, a compiler tool will solve the problem of cross-compiling the application into instructions that the target processor can execute. A logic analyzer will allow the engineer to see the bus signals. Monetary and functional values are the core values that businesses look at. If a developer wants a tool to make their job easier, they must convince their company using monetary and functional values.

**Tips and Tricks**When requesting a new tool, build up a concise but powerful return on investment statement based on monetary and functional values. If done correctly, you'll rarely be denied the tools you need.

Emotional and social values are generally less important to a business but can be important to the engineer. Emotional value provides psychological value, such as a sense of well-being or a tool that decreases stress. For example, I leverage my calendar and project management software to schedule my time and understand my workload. These tools are emotionally valuable because they offload activities I would otherwise have to track and ensure I don't overwork myself mentally.

Social value connects someone to others or gives them social currency among their peers. For example, as a consultant, I often leverage techniques or tools that I use or know about to get companies or engineers interested in my work and services. I have tools and knowledge they don't know about, and I can use that to create a social connection. That interest for me will often lead to engineers taking my courses, signing up for mentoring or advisory services, or companies asking me to help them archi-

tect and review their software systems. Social value is less valuable, though, to an engineer in the trenches who are working for their boss. However, it could be essential to a company's customers.

When selecting and requesting the purchase of a new tool, developers need to think about and clearly communicate the value proposition for the tool. To do so, they must learn how to calculate the tool's value.

## Calculating the Value of a Tool

A big problem I often encounter is that companies are more than willing to buy oscilloscopes, spectrum analyzers, and other expensive hardware tools, but the moment someone wants to buy a software tool like a compiler, static analyzer, or linter, they are instantly denied the purchase. There seems to be a belief in business that software tools are supposed to be open source and free, which, to be honest, is a great way to set up for being late on a project, going over budget, and complete failure.

Part of the problem with software developers getting the tools they need to be successful is learning how to talk business to their managers and bosses. A company is only interested in investing in the business and producing a profit. Purchasing a new tool (spending money) must have a clear return on investment (profit). The hardware engineers have something physical that management can understand and wrap their heads around. For software developers, they have nebulous bits and bytes that remind managers of the streaming code of the Matrix. They don't understand software. Software to them is expensive, holds up projects, and is something they wish they didn't need (even though the software is a great product differentiator and a huge business asset).

> **Caution**Don't get caught in the trap of believing you can build your own tool faster and cheaper! Developers are notorious for underestimating the value of a tool. I've seen too many teams try to save money by building it themselves, only to spend 10x or more and still have to spend the money to bail themselves out.

Developers need to learn how to communicate the value that software tools bring to the company and how the business will profit from it. Developers need to

convert the four values discussed in the last section into business needs. For example, a developer might demonstrate that a tool purchase will

- Improve developer efficiency
- Increase system robustness
- Result in shorter development times
- Decrease costs
- Improve scalability and reuse
- Minimize rework

The list can go on and on.

The trick to ensuring you get the tools you need and want is to request the tool in a way that the business understands: return on investment (ROI). Return on investment can be calculated using the following formula:

- Return on Investment (ROI) = Value Provided – Value Invested

The ROI should be positive. The larger the positive value, the greater the investment in the tool is, and the harder it will be for the business to say no! The company may also want to look at the ROI from a percentage standpoint. For example, instead of saying that a purchase will annually save the business $2000, the investment will return 20%. The ROI as a percentage can be calculated using the following formula:

- ROI % = (Value Provided – Value Invested) / Value Invested * 100%

Let's look at an example of how to justify purchasing "expensive" tools costs. One tool that I often get pushed back on being expensive is debugging tools. So let's calculate the ROI of a professional debugger such as a SEGGER J-Link.

> **Caution**It's not uncommon for developers to look at a tool's price and decide that it is expensive based solely on a comparison with their salary or standard of living expenses. Don't get caught in this trap! The price is inconsequential for a business if it has a good return on investment! (Look at your salary and ask yourself if you would pay someone that much for what you do!)

## The ROI of a Professional Debugging Tool

Compilers and professional debuggers are software tools I often see a lot of resistance to purchasing. I say professional here because nearly every development board today comes with some sort of onboard debugger, and there are many "one-chip"[3] debuggers out there that can be purchased for less than $100. Since these low-cost debuggers are everywhere, getting permission to buy an $800 or $1500 debugging tool becomes an uphill battle.

The reader must recognize that the low-cost tools provide the minimum debugging capability possible to get the job done. The debugger typically can program and erase flash, perform basic debugging, and provide maybe two or three breakpoints. Breakpoint debugging is one of the most inefficient ways to debug an application. Developers need the ability to trace their application in real time, record the trace, and set up data watchpoints to trigger events of interest. These capabilities take a lot of time to develop and are usually absent from onboard debuggers and low-cost programmers. Low-cost programmers are useful for quick evaluation purposes, though.

> **Lesson Learned**Early in my career, I accumulated a box of low-cost programmers that probably cost me upward of $3000 over the course of just a few years. I had dozens of them for all sorts of microcontrollers. Each debugger provided the minimal capability possible, making each debug cycle exceptionally painful. One day, out of annoyance of having to buy yet another tool, I decided to invest in a SEGGER J-Link Ultra+. Upon first use, I realized how much my attempt to be frugal had cost money, time, and performance. I was able to discard my box of programmers, and nearly eight years later, I still have not had to purchase a single programming tool. (Well, I did buy a SEGGER J-Trace so I could do instruction tracing, but I don't count that.)

Consider the value a professional debugger might present to a developer and the company they work for. On average, a developer spends ~20% of their time on a project debugging software.[4] (When I speak at conferences and survey the audience, the result is closer to 50% or more.)[5] Assuming that the average work year is 2080 hours, the average developer spends 416 hours per year debugging. (Yes, that's 2.6 months yearly

when we are incredibly conservative!)

According to **glassdoor.com**, the average US salary in 2022 for an embedded software engineer is ~$94,000.[6] However, that is not the cost for an employee to the business. Businesses match the social security tax of an employee, ~$6000, and usually cover health care, which can range from $8000 to $24,000 per year (I currently pay ~$15,000/year). Therefore, we will assume health-care costs are only $5000. In fact, we will ignore all the other employee costs and expenditures. In general, it is not uncommon for a business to assume the total cost for an employee is two times their salary!

Adding all this up brings the total to around $105,000 for the average developer or ~$50 per hour. (The exact value is $50.48, but we'll perform our math in integer mathematics as our microcontrollers do.) Of course, we ignore adjustments for holidays, vacations, and the like to keep things simple.

Calculating the tool's value requires us to figure out how much time a developer can save using the new tool. For example, if the tool costs $1000, we need to save 20 hours or decrease our debugging time by ~5% to break even. Personally, I like to break out the value of the improvements the new tool will bring and assign a time value that will be saved per year. For example, the value proposition for the debugger might look something like the following:

- Adds real-time trace capability (saves ~40 hours/year)
- Unlimited breakpoints (saves ~8 hours/year)
- Increases target programming speed (saves ~8 hours/year)

We could certainly argue on the justification for the times, but from experience, they are on the conservative side. The ability to trace an application is a considerable saving. While one might argue that programming the device faster or pulling data off more quickly is not a big deal, you're wrong. When I switch from my SEGGER J-Link Ultra+ or my J-Trace, I notice a difference in the time to program and start a debug session. Given how often a developer programs their board daily, these little time frames add up and accrue to measurable time frames. (The caveat here is that if you are using disciplines like TDD, simulation, DevOps, and so forth, then

the savings may not be as high, but you'd likely need a more advanced tool anyways for network connections and the build server.)

So far, our conservative estimate would show that we can save 56 hours a year by purchasing the professional debugger. The savings for buying the tool is expected to be

- ROI = $2800 (Engineering Labor) – $1000 (Tool Cost) = $1800
- ROI % = $1800 / $1000 * 100% = 180%

It's not a considerable amount of money, but it's also freeing up the engineer by 36 hours to focus on something else during the year. For example, it might be enough time to implement a new feature or improve the CI/CD process.

An important point to realize here is that this is just the first year, and it's only for one developer! The first J-Link Ultra+ that I purchased lasted six years before it met an untimely death while connected to a piece of client's hardware. (SEGGER has a fantastic trade-in policy where even though the device was damaged, I could trade it in and purchase a new one at half price!) If we assume that the developer will use the debugger for five years, suddenly, we see a time savings of 250 hours total! Our ROI becomes

- ROI = ($50 x 250 hours (Engineering Labor)) – $1000 (Tool Cost) = $11,500
- ROI % = $11,500 / $1000 * 100% = 1150%

I don't think anyone can argue with that math. Who doesn't want a return on investment of 1150% in five years?

What's interesting is that this can still get better. For example, suppose there are five developers in the company, and each gets their own debugger, for over five years. In that case, we're suddenly talking about injecting nearly 6250 more development hours into the schedule! What could your company do with three extra person-years worth of development?

> **Tips and Tricks**Small investments and nearly imperceptible improvements can have a dramatic long-term effect on a development team and business. What minor enhancements do you need to make today to

change your long-term results?

## Embedded Software Tools

In the Introduction of the book, we discussed that the most successful teams strike a balance between their software architecture, processes, and implementation. This book is organized around the same concept, and the best approach to discuss the tools we need to develop software is to follow that approach. There are more tools and types than we can explore in this chapter; however, we can look at some of the primary tools developers should have in their toolbox. For those interested, I often post blogs about tools on my website at **www.beningo.com**. If you go to the blog, there is a category called tools that you can use to access the tool blogs. In addition, the tool reviews and tutorials I have done can also be found **here**.[7]

**Caution**Tools must provide an ROI. Some tools and processes related to project management can become tangled cobwebs that prevent rapid progress. Make sure that you dictate the tool's behavior, not the other way around!

### Architectural Tools

Architectural tools help a developer and team design the architecture for their product. We've discussed in Part 1 the importance of software architecture. The architecture is the road map, the blueprints developers use to implement their software system. While there are many architectural tools, three primary categories in this area come to mind: requirements elicitation and management, storyboarding, and UML diagramming and modeling.

#### Requirements Solicitation and Management

I often joke that requirements are the bane of a software developer's existence. Requirements are boring, tedious, and perhaps one of the most important aspects of developing a system. Requirements are a critical key to managing the customers' expectations, and if those expectations aren't managed correctly, even success could be viewed as a failure. (This is why

scope creep is so dangerous and needs to be carefully managed.)

Several different types of tools can be used to meet management requirements. The tool selected will be based on the organization's size and traceability requirements. The first tool, and the one that I still see small organizations or teams working with legacy requirements use, is just a simple Excel spreadsheet. Within Excel, teams can create columns for requirement numbers, requirements descriptions, and other status and traceability requirements needs. Excel is low cost and straightforward but does not allow other tools' linking and full design cycle management capabilities.

Commercial tools often used within the embedded systems industry include **Jama Software**[8] and **DOORS**.[9] These tools provide customizability and end-to-end traceability and integration. Thankfully, several great web pages describe many of these tools in detail. You can check out this **page** for more options.[10]

### Storyboarding

One of my favorite ways to manage requirements is to create a series of stories describing the product's use cases. I personally believe a picture is worth 1000 words and several hundred thousand lines of code. Breaking the product requirements into a series of pictures describing the use cases and user needs can dramatically help clarify and simplify requirements. A storyboard allows the entire life cycle of an idea to be expressed in one image by showing the problem, a high-level solution, and the benefit to the user.[11]

There are a lot of great tools available to help developers create storyboards. These tools can be either stand-alone or part of a larger software development life cycle process software. From a pretty picture standpoint, developers can use tools like

- Adobe Photoshop
- Boords Storyboard Creator
- Canva

- Jira
- PowerPoint
- StoryboardThat

My go-to tool for my storyboarding is to use Jira from Atlassian (and it's not the last time I will mention it). Jira allows me to create a series of user stories I can track as part of my software development life cycle processes. You might be saying yes, but that doesn't give you a pretty picture. So instead, for my storyboarding, I create UML use case diagrams.

**UML Diagramming and Modeling**

A critical tool when developing a software architecture is to utilize diagramming tools. Developers can use tools like PowerPoint, but it is far more powerful to use UML tools. UML tools provide a common language for developing the software architecture and guiding the software implementation. A really good tool will generate C/C++ code, so the developers don't need to hand-code. Of course, you must be careful with autogenerate tools. In general, the code they generate is not very human-readable, and the code could be tough to debug if there's a bug in the generator.

One popular tool that I often see used is **Lucidchart**.[12] Lucidchart is an online UML diagramming tool. Just like with many good tools, they provide professional templates that can be used to jump-start a design. The significant advantage of this tool is that it is great for distributed teams. Everything is hosted online, meaning developers worldwide can collaborate and access the same documentation. While this is great, I often consider this a disadvantage. Given the many corporate hacking cases, anything stored online must be considered vulnerable.

My personal go-to tool currently is **Visual Paradigm**.[13] Many of the UML diagrams in this book were created using it. The tool is potent, and one thing I like about it is that it links diagrams together, making navigating an architecture very easy. There are free editions of the tool along with commercial versions. The commercial versions allow the use of storyboards and can generate C++ code. There's nothing like designing a state machine or class diagram and clicking generate code. Just remember, the

generated code most likely won't meet your coding standard and could use features you would like to avoid.

There are many other tools, and I can't do them all justice in print. Guru99 has a list of the best UML tools, which you can find **here**.[14] There are lots of great tools on there. Some other favorites of mine are Dia, Draw.io, Edraw Max, and Visio. Again, try out several tools. When evaluating tools, I create a little test project to see which tool best fits my workflows and needs.

## Process Tools

I love process tools. Professionals act professionally and use the best tools to complete the job. Tools help me to plan things out and ensure that I have a consistent development process. Honestly, I think with the right tools, teams can consistently do more, deliver more value, and help more people. There are a lot of great process tools out in the world, and there are also many bad ones. In general, at a minimum, teams should have tools for managing revision control, the software development life cycle, quality control, testing, and DevOps.

### Revision Control

When it comes to revision control, obviously, the industry standard today is Git. Mercurial attempted to become the industry standard, but many services have discontinued it. Git has won (for now) and looks like it will be the revision control tool to use for the foreseeable future.

Git repository tools are everywhere. I have repositories on Bitbucket, GitHub, GitLab, and Azure DevOps. Sigh. It's quite a bit, but it also helps me understand the significant services and what they offer. The key is finding a service that provides what you need with the desired security level. For example, many of these services will allow you to set up a personal access token. Unfortunately, I set mine to 30 days, which means they expire, and I must go through the pain of creating new ones monthly. The security is well worth it, though.

I think the best tools for revision control aren't necessarily the online repositories but the client-side tools to manage the repositories. Yes, all we need is a terminal, and we can command line bang away Git commands all day long. I do this regularly. However, I find that user interface tools can be beneficial. For example, if I want to view the history and visualize the active branches and merges, I will not get that from a terminal.

> **Caution** When using a hosted repository service, I recommend enabling SSH and a personal access token. The personal access token should expire at most 90 days from creation.

One tool that I like is **Sourcetree**.[15] Sourcetree is available for Windows and Mac OS X and provides an excellent visual interface for working with repos. If you are like me and need to manage multiple accounts with several dozen active repos at any given time, a GUI-based tool is handy. I can organize my repos by customer, manage multiple repos, and track commits between employees, contractors, and clients while managing most aspects of the repos. I find that everything I need is unavailable through a button click, but that's why I keep my terminal handy. (Plus, working through the CLI[16] keeps my skills sharp.)

For Windows users, I have used **TortoiseGit**[17] for years. It provides all the tools one might need to manage a repo, all through a simple right-click of the mouse. Honestly, when I used it, I nearly forgot how to use the Git terminal commands because everything I wanted to do had a button click available. When I switched to exclusively using Mac OS and focused on terminal use only, I had to relearn many commands. I highly recommend no matter what tool you use, you occasionally go a week or two just using the terminal to keep your skills sharp if you decide to use a GUI-based tool.

### Software Development Life Cycle

Software development life cycle tools are the tools that we use to track and monitor progress on a project. They often will provide us with the concept of delivery tracking and visibility. As you might imagine, there

are hundreds of tools out there that can be used to manage projects. However, there are several that have gained the most traction.

My favorite tool for managing software projects is to use **Atlassian's Jira**.[18] A significant benefit is that it is free to use for teams up to around ten (assuming it has not changed since I wrote this). Jira can pretty much do anything that a development team needs. For example, developers can create a development road map, define their backlog, plan sprints, track issues, integrate it into their Git repo, and even manage software deployments. As I write this, Jira is the model tool for managing software development in an agile team.

Another interesting tool that is more on the commercial side is **Wrike**.[19] I've used it on several projects over the years and found that it provides most tools developers would want for an Agile-based software process.

Commercial tools aren't the only options for SDLC management. In the past, I've also used tools like **Trac**[20] and **Redmine**.[21] These tools allow a team to set up their project management server. I used to enjoy deploying and managing my own hosted servers, but over time, with all the cloud-based services that can do this for you, I've moved away from them. However, they do provide similar tools and workflows to other tools. Which one you choose comes down to your own choice and budget.

Every team needs to have an SDLC tool. Even when I operate as a one-person team, I found that trying to skip the use of a tool like these just made it harder for me to plan and get visibility into what it was I was working on. I don't care what tool you use, find the one that works for you and use it. You'll discover that over time you'll become far more efficient; just don't get stuck in the trap of spending too much time managing your tools.

> **Tip** Don't let sprint planning consume too much time. Limit your planning to 1.5 hours or less and get back to work!

### Testing

The ability to test our code and verify that it meets our customers' needs is perhaps one of the most critical aspects of developing software. Unfortunately, too many development teams and engineers I encounter don't have good testing tools (or processes). The ability to test the software is critical, and the ability to do so in an automated way is perhaps even more important. Testing allows us to verify our assumptions that the code we have written does what it is supposed to. Testing helps us to verify and trace the desired result back to our requirements.

Test harnesses provide developers with a mechanism to perform unit tests, integration tests, and even system-level testing. Testing tools can range from free, open source tools like **CppUTest**[22] to commercial tools like **Parasoft**.[23] Personally, I use CppUTest in most of the projects that I work on. CppUTest provides a test harness that can be used with C/C++ applications. It's easy to get up and run, and I often run it within a Docker container integrated as part of DevOps processes.

Following a process like Test-Driven Development, you should develop your tests before writing your production code. (We discussed this in Chapter 8.) Defining test cases doesn't necessarily mean you will create enough test cases to get full coverage. One technique that I often use to verify my test count is McCabe's Cyclomatic Complexity.[24] Cyclomatic Complexity tells a developer how many linearly independent paths there are through a function. Stated another way, Cyclomatic Complexity means the developer needs the minimum number of test cases to cover all their branches in a function. Additional test cases may be required to test boundary conditions on variable values. To measure Cyclomatic Complexity, I often use tools like **pmccabe**[25] and **Metrix++**.[26]

One additional tool that can be pretty useful when testing is **gcov**. gcov is a source code coverage analysis and statement-by-statement profiling tool.[27] Usually, a developer would enable gcov to be run as part of their test harness. The tool will then provide a file that gives the branch coverage as a percentage and a marked-up version of the source file that shows a developer which branches are covered and which ones are not. Branch coverage allows developers to go back and improve their test cases to get full coverage.

> **Caution** 100% code coverage does not mean there are no bugs in the code! It simply means that the test cases result in every branch being executed. It's still possible to be missing test cases that would reveal bugs, so don't let 100% code coverage lure you into a false sense of well-being.

### Continuous Integration/Continuous Deployment

I believe the most important tool for embedded development teams to embrace and deploy effectively are CI/CD tools. The ability to automate software's build, analysis, testing, and deployment is a game changer for nearly all teams. The potential improvement in quality and decreases in cost can be pretty profound. There are also quite a few different tools that can fit almost any team's needs.

The first tool that I would like to highlight is **Jenkins**.[28] Jenkins is an open source automation server that provides plug-ins to scale building, deploying, and automating projects. The software is a Java-based program that can be installed on any major OS like Windows, Mac OS, and Linux. It's easy to set up and use their plug-in systems; it can be integrated into many other SDLC programs. Jenkins has been quite popular due to its open source nature.

Another CI/CD tool that I've used and like quite a bit is **GitLab**.[29] GitLab integrates a project's Git repo and CI/CD pipelines that can then be used to build, test, and deploy the software. The tool also comes with some basic SDLC tools, but they are nowhere near as sophisticated as the other SDLC tools discussed in previous sections. I think the most significant advantage of the tool is how easily CI/CD is integrated into it. Developers can easily create a CI/CD pipeline like in Figure **15-1**.

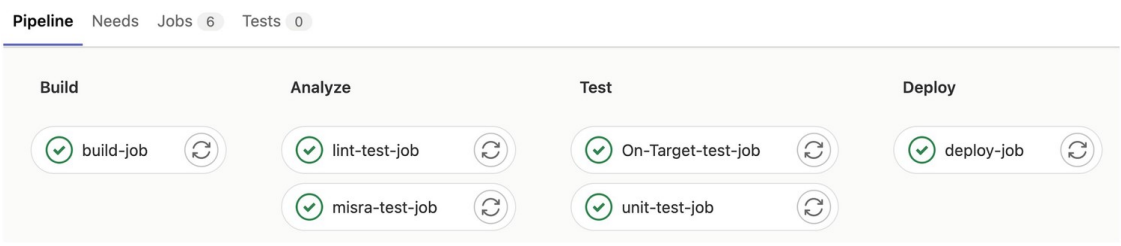


***Figure 15-1*** An example CI/CD pipeline for an embedded target that leverages the GitLab CI/CD system

## Implementation Tools

The last primary tool category we will discuss in this book, but probably not the last, is implementation tools. Developers use implementation tools to write and build their software. In addition, various tools are used to develop embedded software, such as compilers, programmers, and code analyzers. Each tool's specific purpose is instrumental in successfully developing a code base.

### IDEs and Compilers

Integrated Development Environments (IDEs) and compilers often go hand in hand. Most microcontroller vendors provide their customers with an IDE that can seamlessly interface to a compiler and do much more. For example, STMicroelectronics provides **STM32CubeIDE**,[30] which integrates the ability to configure an embedded project, set up peripherals and pin configurations, write code, debug, and deploy it to the target processor. The ecosystem that STMicroelectronics has set up also includes plug-ins with libraries for extended features such as

- Secure Boot and Secure Firmware Update (SBSFU)
- Machine Learning (STM32CubeAI and X-CUBE-AI)
- Motor Control (X-CUBE-MCSDK)
- Sensor and motion algorithms (X-CUBE-MEMS1)
- Display and graphics (X-CUBE-DISPLAY)
- IoT connectivity (X-CUBE-IOTA1)

The value that these tools can provide is immense! As a junior engineer, I remember the pain I had to go through to set up the clock trees on a microcontroller. It seems that it would take 40 hours or more to review the clock tree properly, identify the registers that had to be modified, and then configure them for the correct frequency. The first attempt often failed, which led to debugging and frustration from just wanting to get the CAN baud rate to 500 kbps, not 423.33 kbps. In addition, each peripheral had to be set up with the correct dividers, so it was not a process that was just done once.

Modern IDEs had taken what took me a week or so when I started my career to about 15 minutes today. Most IDEs now contain a clock tree configurator that allows us to go in, select our clock source, set the frequency, and then magic happens, and our clock tree and peripherals are all configured! An example of this

fantastic tool can be seen in Figure 15-2.

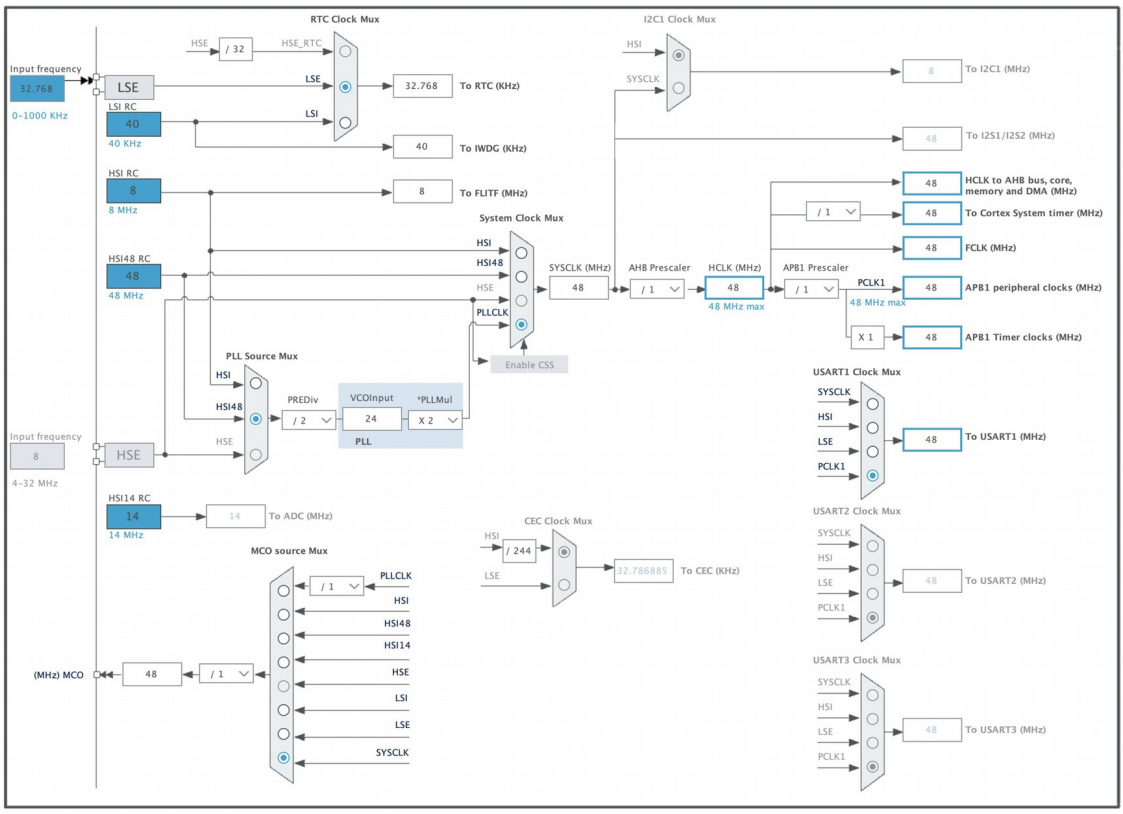


***Figure 15-2*** A clock tree configurator example from STM32CubeIDE for an STM32F0XX. Developers can quickly set their clock source, dividers, and peripheral dividers and get to writing code

The clock tree is not the only tool in these IDEs that makes developers' lives easier; there is also the pin and peripheral configurator! The pin and peripheral configurator again allows developers to avoid digging deep into the datasheet and register map by using a high-level tool instead. In many environments, developers can click individual pins to set the pin multiplexer and even assign project-related names. They can then click a peripheral, use dropdowns to assign peripheral functions and baud rates, and configure many other features. When I started, this entire process could easily take a month, but today, if you spend more than a day, you're probably doing something wrong.

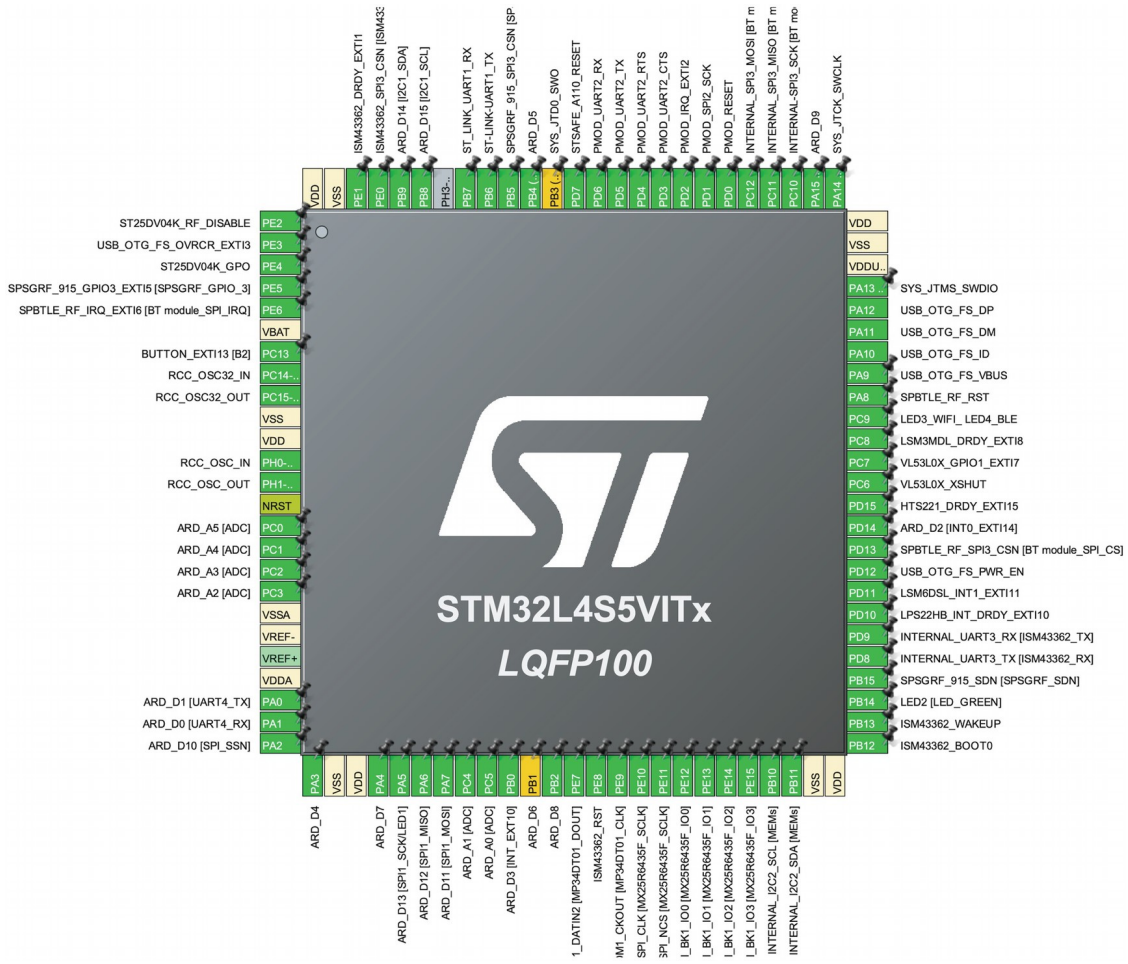


***Figure 15-3*** A pin configuration example from STM32CubeIDE for an STM32L4SX. Developers can quickly configure their pins and peripheral functions. In addition, if a development board is used, the tools can automatically configure them for you!

When considering compiler options, many teams use free compilers like **GCC for Arm**.[31] However, there are a lot of good commercial compilers available as well. In addition, many IDEs will allow teams to easily integrate compilers such as **Keil MDK**[32] and **IAR Embedded Workbench**.[33] In fact, these compiler vendors usually also provide their IDEs. I've worked with all these compilers, and which one I choose depends on the customer I'm working with, the end goals of the project, optimizations needed, and so forth. However, if you carefully examine the data from the **Embedded Microprocessor Benchmark Consortium** (EEMBC),[34] you'll discover that the compiler chosen can impact code execution.

Before you think I'm pushing STMicroelectronics, I'll just mention that I'm using them as an example. NXP, Microchip, and many other vendors have similar ecosystems and capabilities. It is interesting to note, though, that each has its own spin, libraries, and ways of doing things that often

differentiate them from each other. So I would just encourage you to try several out and figure out which fits best for your needs.

Using a compiler-provided IDE or a microcontroller vendor IDE is not the only option for the modern embedded software developer. Microsoft's **Visual Studio Code**[35] (VSC) has also become popular over the last several years. VSC provides developers with an agnostic development environment that can be used for embedded programming or general application development. The number of plug-ins it has for Arm debugging, code reviews, pair programming, source control, code analysis, and so forth is quite dizzying! The capabilities only seem to be growing too.

There are many benefits to selecting a tool like VSC. For example, VSC provides an integrated terminal, making it very easy in a single environment to issue commands, manage a build, run Docker, and so forth. I've also found that VSC often has far better editor capabilities and runs better than most vendor-supplied IDEs. VSC is also lightweight and cross-platform. I can't tell you how many times I've found vendor IDEs to be wanting in a Mac OS or Linux environment.

One last point to leave you with is that, on occasion, I will use more than one IDE to develop embedded software. For example, I've occasionally worked with teams where I would work in VSC but maintain an STM32Cube IDE project file or a Microchip MPLab X project file. This allowed developers who are more comfortable with more traditional and noncommand-line tools to make fast progress without having to slow down too much to learn new skills. (Obviously, it's good to know and push yourself to learn new things, but sometimes the resistance is just not worth it!)

### Programmers

Programmers play an essential role in embedded systems; they allow us to push our application images onto our target hardware for production and debugging! Successfully programming a target requires a physical device that can communicate over JTAG/SWD and software that can take the image and manage the physical device. Let's start our discussion with the

physical device.

Nearly every development board in existence today comes with some sort of onboard programmer. These programmers are great for evaluating functionality but hardly serve as excellent professional debuggers. The onboard debuggers are often limited in features such as only offering two breakpoints, low clock speeds, and nearly no chance of performing real-time tracing. Advanced features like energy monitoring are also missing. To get professional-level features, developers need a professional programming tool.

Several companies provide fantastic third-party programmers. I mention third party because if you buy a tool like an STLink, you'll only be able to program STMicroelectronics parts. Perhaps only programming one vendor's microcontrollers is acceptable for you, but I've always found it's worth the extra money to be flexible enough to program nearly any microcontroller. The price for programmers can, of course, vary widely depending on the space that you work in. For example, you might pay upward of five figures for **Green Hills Softwares' TimeMachine Debugging Suite**[36] while paying $700–$3000 for an Arm Cortex-M programmer.

There are several programmers that I've found to be good tools. First, I really like SEGGER's J-Link and J-Trace products. They have become my default programmers for any development that I do. I use a J-Link Ultra+ for most applications unless the ETM port is provided, in which case I then use the J-Trace. Beyond the quality aspect, I also like that SEGGER has a no-questions-asked defect policy. For example, I once had customer hardware damage my programmer but could replace it at 50% of the cost through a trade-in program. You can find an overview of the different models **here**.[37]

The next programmer that I use is Keil's **ULINK**.[38] I've used several models, but my favorite is the ULINK Plus. The ULINK Plus comes in a small package that travels well and provides a 10 MHz clock. The feature I like the most is that it gives I/O lines for test automation and continuous integration. The ULINK Plus can also be used for power measurements. All

around, a pretty cool tool.

There are undoubtedly other programmers out there as well. Other notable programmers I like include IAR's **I-Jet**[39] and **I-Scope**,[40] which provide energy monitoring and debugging capabilities. I also like PEmicro's programmers, which can be used for debugging or production.

Beyond the hardware devices, the software is required to program a microcontroller. In many instances, developers will just use the default software built into their IDEs. However, other compelling tools can also be used. For example, if you plan to set up a CI/CD server, you'll want to examine the **OpenOCD**[41] framework. There is also a scripting framework associated with SEGGER's programming tools.

I've also found that each vendor usually provides some tools and utilities to get a bit more out of their programmers. For example, SEGGER provides a tool called **J-Flash and J-Flash Lite**,[42] which provide developers with a GUI interface to perform functions such as

- Reading chip memory
- Erasing chip memory
- Programming an application
- Securing the chip
- Modifying option/configuration bytes

These additional software tools can provide developers with a quick and easy bench or production setup that does not require the source code to be "in the loop."

### Code Generators

Code generation can be a sticky subject for developers. In theory, generating a model of your software and then using the model to generate the functional code sounds fantastic! However, generated code doesn't always deliver on that promise. I've seen code generators that create code as expected, but the code is unreadable, slow, and can't be modified by hand. For example, when I started my consulting business in 2009, I used Freescale's CodeWarrior product to generate code that helped me under-

stand how to work with various peripherals and then hand-code the drivers from scratch. Thankfully, these tools have become much better over time.

Before deciding on code generation tools that might fit your development cycle, it's helpful to recognize the pros and cons of their use. First, code generators can bring development teams out of the low-level hardware and help them to focus on high-level application code. Next, developers focus on the application, which can run code in emulators, improve testing and test cases, decrease debug cycles, and much more. Of course, the biggest complaint I often hear is that generated code is larger and runs slower than hand-coded code.


***Figure 15-4*** Leveraging code generation has its pros and cons, which may impact whether a team is willing to adopt it or not. This figure shows a summary comparison of the pros and cons of using code generation

There are several code generators that I believe embedded developers can readily leverage today to help improve their embedded software and deliver in a reasonable time frame. First, there are code generation tools that nearly every microcontroller vendor provides. For example, STMicroelectronics provides **STM32CubeMx**,[43] which also integrates into STM32CubeIDE, which can be used to configure and autogenerate drivers and some application code. NXP has **MCUExpresso SDK**,[44] and Microchip has **Harmony**.[45] These tools help configure and generate low-level driver code and some middleware. Unfortunately, they don't help with application modeling.

Earlier in the chapter, we discussed several tools that can be used to model an application in UML. These tools are oriented toward developing the software architecture. However, many UML tools will provide mechanisms for generating code in Python, Java, and C/C++. State machines, sequences, and other software constructs can be generated using these

tools. Using a modeling tool can provide a huge advantage to developers in that they only need to maintain the model and can generally not care about the generated code as much. This doesn't sit well with me unless the tool generates nice human-readable, efficient code.

Probably the most well-known code generation tool is **Matlab**.[46] Matlab is mighty and can be used to create a high-level software model, run it in simulation, and then generate the target-specific code. So, for example, if a developer wants to make a digital finite impulse response (FIR) filter, they can model such a filter in Matlab with something like Figure 15-5.

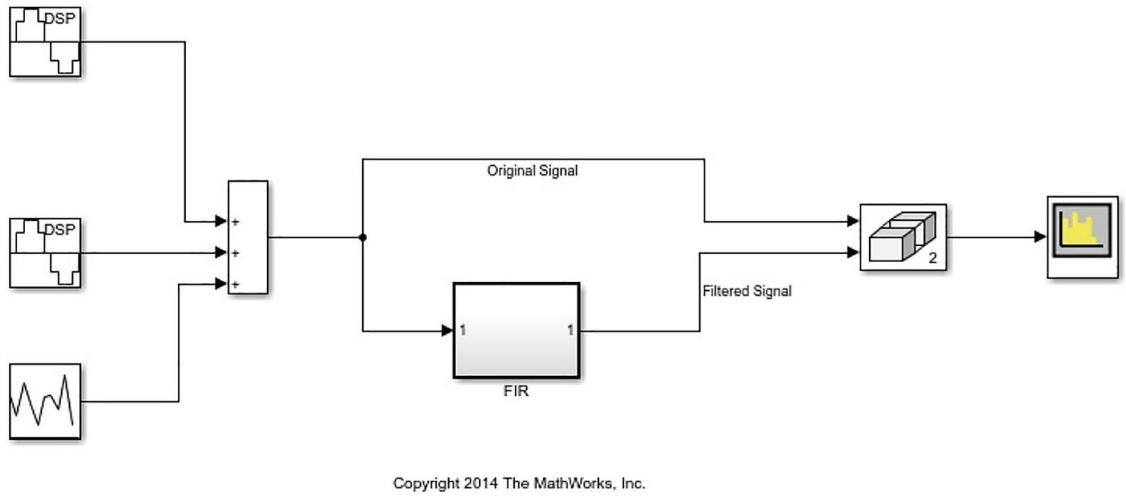


***Figure 15-5*** Modeling an FIR filter in Matlab

In the example, two frequency signals are fed into the filter and a noise signal. Next, the FIR block performs the filtering based on the FIR filter parameters in that block. Finally, the original and filtered signals are fed into a plotter. When the model is run, the simulated output looks like Figure 15-6.

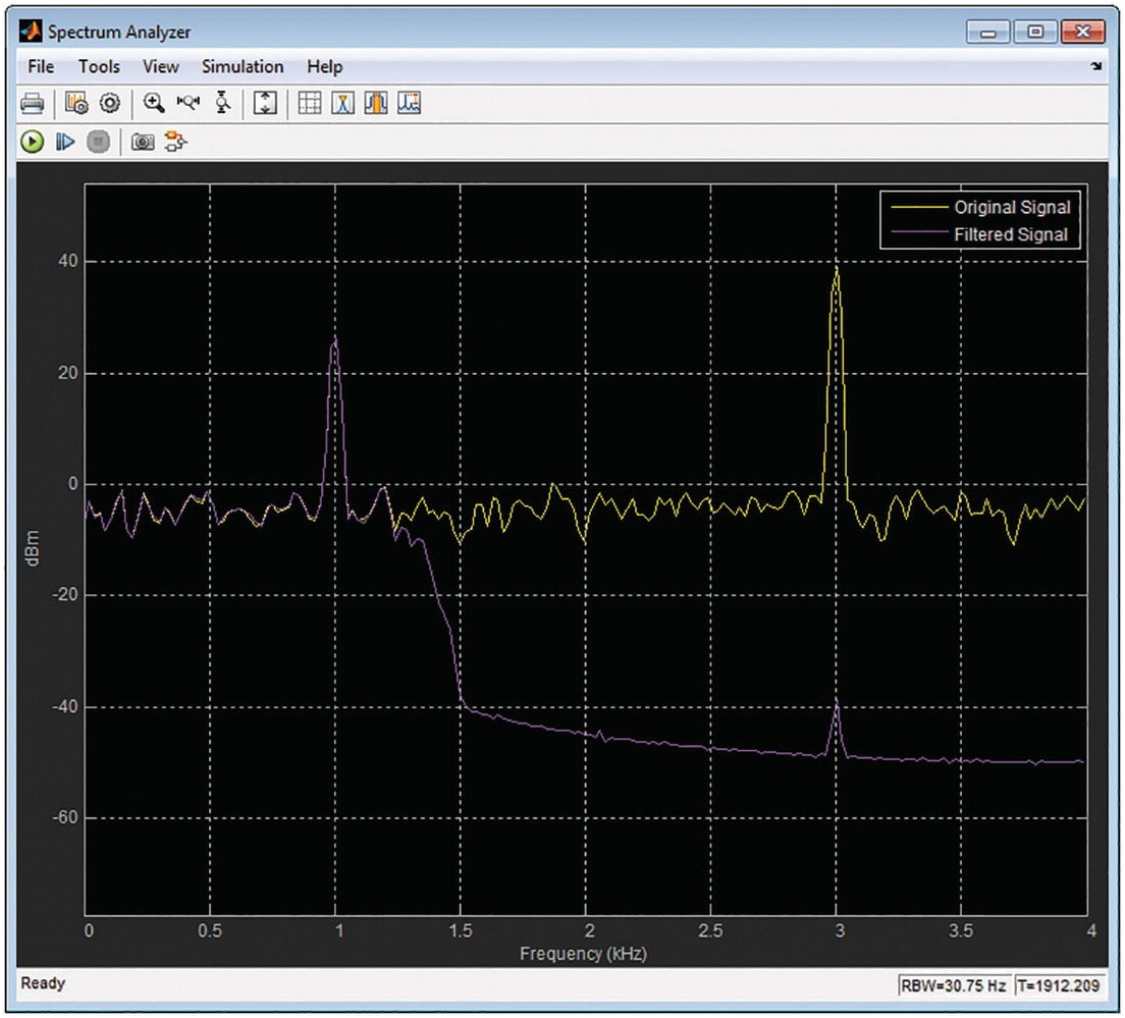


***Figure 15-6*** The frequency response of the modeled FIR filter in Matlab. The yellow line is the original, unfiltered signal. The purple line is the filtered response. Notice we get ~-40 dB attenuation at 1.5 kHz and about -80 dB on the signal at 3 kHz

If the response found in Figure 15-6 is not correct, we can simply adjust our FIR filter block instead of modifying our code and then rerun the filter until we get the response we want. Once we have the desired response, we can use tools like Embedded Coder and the Embedded Coder Support Package for ARM Cortex-M Processors. With the generated code, we can import it into whatever build system we have and integrate on target.

My opinion is that embedded software development's future is in tools that configure, simulate, and generate our software for us. I've believed this, though, since ~2008. There have been huge strides and improvements in the tools since then, but they still are not perfect and can sometimes be cost prohibitive for start-ups and small businesses that are not working in safety-critical environments or have large budgets.

### Analyzers

Analysis tools should be at the heart of every development cycle. Analysis tools provide developers with the metrics they need to ensure that the software is on track. For example, an analysis tool might tell developers how many new lines of code have been added. How many lines of code have been removed? Whether there are any violations in the static analysis.

Analysis tools can help developers remove bugs in their software before they find their way into the field. In addition, they can help developers improve the robustness of their code and software architecture. Finally, an analysis can help developers understand what is in the code base, how it is structured, and the potential problem module developers will need to tiptoe around.

How can developers analyze their software with so many different tools worldwide? There are probably nearly two dozen that I frequently use. Unfortunately, I'm not going to describe them all here, but I will tell my favorites and then try to list out a few for you to further investigate on your own.

The first analysis tool I love and am unsure how I would ever develop without is **Percepio's Tracealyzer**.[47] Tracealyzer is a runtime analysis tool often used with real-time operating systems like FreeRTOS, ThreadX, Azure RTOS, Zephyr, and others. The tool allows developers to record events in their system like giving or taking a semaphore, context switching to a different task, an interrupt firing, and much more. These events can be captured in either a snapshot mode or a real-time trace for debugging and code performance monitoring.

I was teaching a class one day on real-time operating systems and introducing Tracealyzer to the class. One of my colleagues, Jim Lee, was in the audience. Jim primarily focuses on hardware design but has also been known to write low-level code. Upon explaining Tracealyzer, he best summarized the tool by saying, "So it's an oscilloscope for software developers." It was a simple observation, but it hits the nail right on the head.

Hardware developers can't debug and verify their hardware without seeing the signals and understanding what is happening. For software developers, that is what Tracealyzer does for us developers. It allows us to capture and visualize what our software is doing.

Developers can capture information ranging from task context switching, CPU utilization, heap usage, stack usage, data flow, state machine states, performance data, and more. I generally use the tool while developing software to ensure that the software I'm working on is behaving the way I expect it to. I'll also use it to gather runtime performance metrics, which I track with each commit. That way, if I add something that changes the runtime performance, I know immediately.

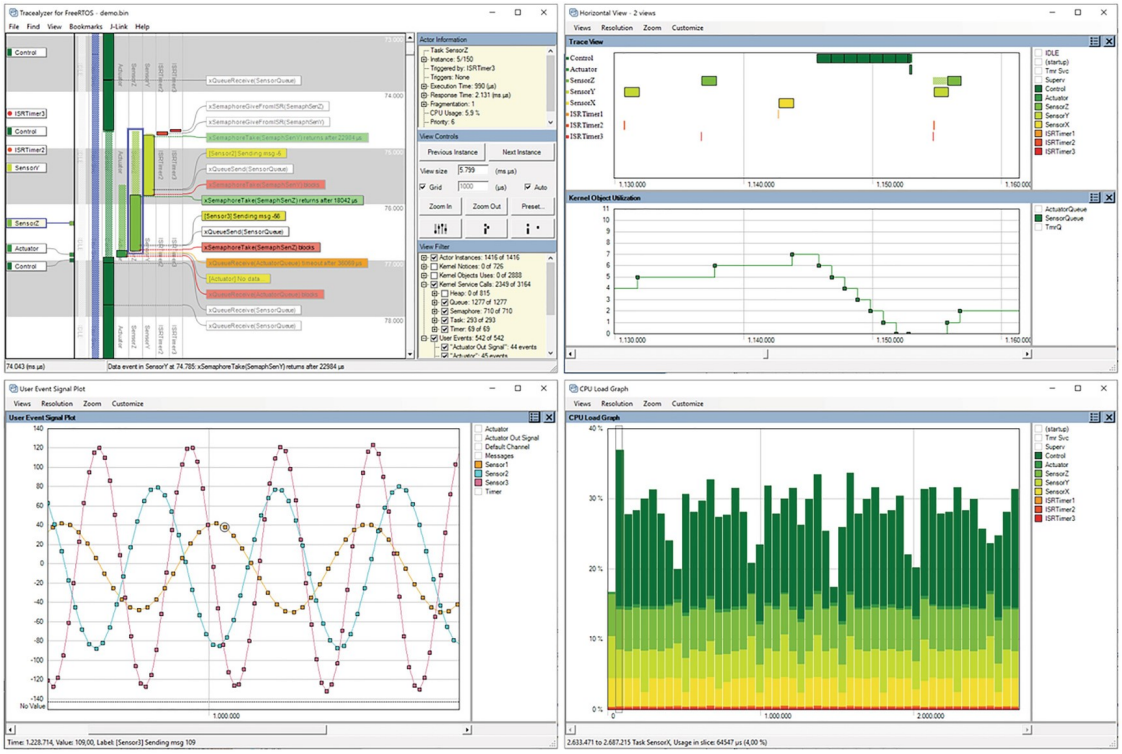

***Figure 15-7*** Tracealyzer provides developers with an event recorder and visualization tool to analyze and understand what an embedded system software is doing. This screenshot shows task execution, user event frequency, and CPU utilization. (It can do so much more, though!)

Other tools out on the market will do similar things to Tracealyzer, but I think it has the largest feature set and is the most user-friendly. There are tools out there like

- **SEGGER SystemView**[48]
- IAR Embedded Workbench (has a built-in trace capability)
- **Green Hills SuperTrace**[49]

Another helpful analysis tool category that I have found useful is tools that can monitor test coverage. I will often use **gcov**[50] to monitor my test case coverage with **CppUTest**.[51] It's important to note that just because test coverage says 100% doesn't mean there are no bugs. I had a customer I was mentoring whom I was telling this to, and they said, "yea, yea," dismissively. The next week during our call, the first thing out of his mouth was, "So I have 100% test coverage and was still finding bugs, so I see what you mean now." He had the coverage, but not the right tests!

Speaking of testing, there are also a bunch of tools from SEGGER that can be useful for developers looking to hunt bugs. One of my favorites is **Ozone**.[52] Ozone is a debugger and performance analyzer that can analyze your binary file and monitor code coverage. I've also used the tool in conjunction with my J-Trace to perform instruction tracing. However, I have found that getting instruction tracing set up can be a bit of a nightmare. Once set up, though, it's fantastic.

There are also many metric analysis tools that developers can find to analyze their software. There are all kinds of metrics that developers may want to monitor. For example, one might want to monitor lines of code (LOC), comment density (although there are arguments around this), and assertion density. The tool that I've adopted the most for metrics analysis is **Metrix++**.[53] Metrix++ can

- Monitor code trends over periods such as daily, weekly, and monthly
- Enforce trends at every code commit
- Automatically review standards in use
- Be configured for various metric classes

Figure **15-8** shows an example of the output from Metrix++ that is monitoring Cyclomatic Complexity. Notice that there is a limit of 15.0 set, and the two functions that are coming up have complexity measurements of 37 and 25. Reports like this can be built into the DevOps process, allowing developers to get results on the quality of their code base continuously. It's pretty cool and, I think, an often overlooked, low-hanging tool that can provide a lot of value.

```
./interprocess/detail/managed_open_or_create_impl.hpp:302: warning: Metric
    Metric name    : std.code.complexity:cyclomatic
    Region name    : priv_open_or_create
    Metric value   : 37
    Modified       : None
    Change trend   : None
    Limit          : 15.0
    Suppressed     : False

./interprocess/streams/vectorstream.hpp:284: warning: Metric 'std.code.com
    Metric name    : std.code.complexity:cyclomatic
    Region name    : seekoff
    Metric value   : 25
    Modified       : None
    Change trend   : None
    Limit          : 15.0
    Suppressed     : False
```

***Figure 15-8*** The Metrix++ output for Cyclomatic Complexity on a test project

Speaking of Cyclomatic Complexity, one of my favorite tools for measuring complexity is **pmccabe**.[54] The tool only measures complexity and works on the command line in Linux-based systems. It's pretty easy to install. All one needs to do is type

```
sudo apt-get update -y
sudo apt-get install -y pmccabe
```

Once installed, it can then be run on a file like packet.c by using

```
pmccabe -v packet.c
```

The tool provides several different outputs, such as traditional and modified McCabe Cyclomatic Complexity. I'll often use this tool before committing to ensure that the module(s) I'm working on is within my desired parameters (usually <=10, but if it makes sense, I'll occasionally allow a slightly higher value). Figure 15-9 shows an example output.

```
beningo@Beningos-MacBook-Pro Application % pmccabe -v packet.c
Modified McCabe Cyclomatic Complexity
|   Traditional McCabe Cyclomatic Complexity
|       |    # Statements in function
|       |        |   First line of function
|       |        |       |   # lines in function
|       |        |       |       |  filename(definition line number):function
|       |        |       |       |           |
7       7       18      233     67      packet.c(233): Packet_DecodeSm
2       2       4       335     10      packet.c(335): Packet_Sync
1       1       2       381     5       packet.c(381): Packet_Version
1       1       1       422     4       packet.c(422): Packet_SourceAddress
1       1       1       462     4       packet.c(462): Packet_DestinationAddress
1       1       1       502     4       packet.c(502): Packet_MessageID
3       3       7       542     24      packet.c(542): Packet_ProcessTwoByteField
5       5       13      602     59      packet.c(602): Packet_Data
1       1       2       697     5       packet.c(697): Packet_Checksum1
1       1       2       738     6       packet.c(738): Packet_Checksum2
2       2       6       780     20      packet.c(780): Packet_EndSync
1       1       1       834     4       packet.c(834): Packet_Get
3       3       36      878     57      packet.c(878): Packet_Validate
1       1       1       962     4       packet.c(962): Packet_ErrorGet
1       1       1       992     4       packet.c(992): Packet_ErrorClear
1       1       13      998     22      packet.c(998): Packet_Encode
1       1       3       1021    6       packet.c(1021): Packet_ResetSm
```

***Figure 15-9*** The results of running pmccabe on a test module named packet.c. Notice that the complexity measurements are all below 10

There are a lot of additional tools that you might consider helping you analyze your embedded software. I always recommend that developers use a static code analyzer. Wikipedia has a pretty good list of various static analysis tools that you can find **here**.[55] I've discovered that IAR's C-Stat is also a good one. Years ago, I used to use PC-Lint quite a bit, but their business model changed considerably, and it is not friendly toward start-ups and small teams anymore. Still a good product, though.

The tool landscape is constantly evolving, and it is a good idea to always keep your eye out for new tools that can be used to analyze software and improve it.

## Open Source vs. Commercial Tools

Before we close this chapter, I want to take the opportunity to discuss the differences between open source and commercial tools. I often come across teams who have a mentality that everything should be open source. While this is possible, I find that teams often don't look at the full picture when going this route. So let's quickly discuss the differences.

Open source tools are fantastic. They provide developers with a valuable tool at no cost! The source code is right there, so we can review it, modify it, and understand what the tool is doing. These immediately seem like wins; however, this may be shortsighted in some cases. For example, a developer often writes open source tools with a sole purpose in mind. That developer may not be considering your use cases. They also might not be interested in maintaining the tool in the long term if it's a tool that will evolve and change over time.

I've also found that open source tools tend to lack good documentation. Yes, I'm generalizing because there are some great projects out there; however, I've seen teams adopt an open source tool to regret it later. Sometimes, the tool won't entirely do what we want in the way we want to do it. We then get forced to make or request changes that may or may not ever be made. I've even seen times when teams have a question and can't find anyone to answer it.

On the flip side, commercial tools cost money. Tools are expensive to build because they require developers who want to be paid big bucks. Therefore, tools tend to be costly. The good news is that a commercially supported tool usually has better documentation. If it doesn't, usually you can call the company and get an answer for what problem you are facing. If you want a new feature, usually the company will add it because you are a paying customer, and they want you to be happy so they can keep your business.

I've also found that the quality of commercial products can be higher than open source. There is often an argument that more developers review open source software and therefore is of higher quality. I've not seen that fact come to fruition. There's a lot of harmful open source code out there. Some of it is downright scary!

Now, hear me out before you come after me with your pitchfork. There is a place for both open source and commercial tools in our development environments. I often use a mix of both. When the open source software has the right features and level of quality, I use it. When not, I use a commercial tool. I've occasionally had open source tools I love suddenly dis-

appear or the support stop because the primary backer moved on or got bored. I'm just suggesting that you be careful, analyze your options, and choose the one that best fits your needs.

## Final Thoughts

Tools are a critical component to helping embedded software teams succeed. Unfortunately, there has been a big push in the software industry that everything should be free. If you must pay for a tool, a software stack, etc., then some universal law is being broken. This mentality couldn't be further from the truth and contributes to and increases the risk of failure.

I don't sell tools, but I have found that they dramatically enrich my ability to help my clients. A good tool will provide a lot of value and a great return on investment to the developer, team, and business that use them correctly. In this chapter, we have explored how to evaluate if a tool will provide you with a return on investment, in addition to exploring several tools that I use. Unfortunately, we've only scratched the surface, and there are many more exciting and valuable tools that we have not covered.

Don't join the herd mentality that paying for a tool is terrible. Professional tools are what make you a professional and help you to deliver the quality that is expected from a professional. So don't wait; upgrade your tools if necessary and get back to coding!

**Action Items**

To put this chapter's concepts into action, here are a few activities the reader can perform to start finding and using the right tools for their job:

- Identify a tool that you use every day. Then, calculate the value and the return on investment of that tool.
- Identify a tool that you would like to have in your toolbox. Next, calculate the value and the return on investment for that tool. Then, put together a tool request for your management and present your request for the tool as discussed in this chapter.
- Look through each tool category that we have discussed. Are there

> any tools missing in your toolbox? If so, research and identify the right tool for you and pursue it!

## Footnotes

1 www.beningo.com/insights/newsletter/

2 www.merriam-webster.com/dictionary/value

3 A “one-chip” debugger is a debugger that can only program a single family of microcontrollers or just literally one from the family.

4 https://bit.ly/3AlhA8u, page 38.

5 This observation was corroborated with Jack Ganssle in addition to (thanks to Jack for the references!)

- *Software Testing in the Real World: Improving the Process*, Addison Wesley, Harlow, England, 1995.
- *Facts and Fallacies of Software Engineering*, Glass, page 90.
- www.cebase.org:444/www/defectreduction/top10/top10defects-computer-2001.pdf
- *Winning with Software*, Watts Humphrey, p. 200, and *The Art of Software Testing* (Myers, Sandler, and Badgett).
- Capers Jones (www.ifpug.org/Documents/Jones-SoftwareDefectOriginsAndRemovalMethodsDraft5.pdf).
- www.sei.cmu.edu/library/abstracts/news-at-sei/wattsnew20042.cfm

6 https://bit.ly/3qMzwpb

7 www.beningo.com/category/tools/

8 www.jamasoftware.com/

9 www.ibm.com/docs/en/ermd/9.7.0?topic=overview-doors

10 www.guru99.com/requirement-management-tools.html

11 https://devpost.com/software/storyboard-that-for-jira

12 www.lucidchart.com/

13 www.visual-paradigm.com/solution/freeumltool/

14 www.guru99.com/best-uml-tools.html

15 www.sourcetreeapp.com/

16 CLI = Command-Line Interface.

17 https://tortoisegit.org/

18 www.atlassian.com/software/jira

19 www.wrike.com/vm/

20 https://trac.edgewall.org/

21 www.redmine.org/

22 https://cpputest.github.io/

23 www.parasoft.com/

24 https://en.wikipedia.org/wiki/Cyclomatic_complexity

25 https://people.debian.org/~bame/pmccabe/pmccabe.1

26 https://metrixplusplus.github.io/metrixplusplus/

27 https://bit.ly/3wcUKjl

28 www.jenkins.io/

29 https://gitlab.com/

30 www.st.com/en/development-tools/stm32cubeide.html

31 https://bit.ly/3CRlxmL

32 www2.keil.com/mdk5

33 www.iar.com/products/architectures/arm/iar-embedded-workbench-for-arm/

34 www.eembc.org/products/

35 https://code.visualstudio.com/

36 www.ghs.com/products/timemachine.html

37 www.segger.com/products/debug-probes/j-link/models/model-overview/

38 www2.keil.com/mdk5/ulink

39 www.iar.com/ijet

40 www.iar.com/products/architectures/arm/i-scope/

41 https://openocd.org/

42 www.segger.com/products/debug-probes/j-link/tools/j-flash/about-j-flash/

43 www.st.com/en/development-tools/stm32cubemx.html

44 https://bit.ly/3qptpa2

45 www.microchip.com/en-us/tools-resources/configure/mplab-harmony

46 www.mathworks.com/products/matlab.html

47 https://percepio.com/tracealyzer/

48 https://bit.ly/3JCc5pJ

49 www.ghs.com/products/probe.html

50 https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

51 https://cpputest.github.io/

52 www.segger.com/products/development-tools/ozone-j-link-debugger/

53 https://metrixplusplus.github.io/metrixplusplus/

54 https://people.debian.org/~bame/pmccabe/pmccabe.1

55 https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

---