J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_10

10. Jump-Starting Software Development to Minimize Defects

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

How much of your time or team's time is spent debugging?

When I ask this question at conferences, I find that the average time an embedded software developer spends debugging their software is around 40%! Yes, you read that correctly! That means that, on average, developers spend 4.5 months fighting with their systems, trying to remove bugs and make them work the way they're supposed to! If I had to spend that much time debugging, I would find a different career because I loathe debugging! It's stressful, time-consuming, and did I mention stressful?

I often look at debugging as performing failure work. That's right. I didn't do it right the first time, and now I must go back and do it again, just like in school. Obviously, a lot of teams can do a lot better. So far in Part 2 of this book, we've been exploring what can be done to prevent bugs from getting into your embedded software in the first place. For example, having design reviews, performing code reviews, modeling, and simulating your software. No matter how good your team gets at preventing bugs, sometimes bugs will still get into your software. When that happens, we want to make sure that our processes allow us to discover them quickly to minimize delays and costs.

In this chapter, we will look at what developers and development teams can do to jump-start their embedded software development to minimize defects. The goal will be to provide you with some process ideas that you can use from the start of a project that should not just help you find bugs faster but prevent them in the first place!

A Hard Look at Bugs, Errors, and Defects

When you carefully consider the typical bug rates of a team and the time developers spend debugging their software, it seems a bit outlandish. Jack Ganssle once said, “No other industry on the planet accepts error rates this high!”. He’s right! Any other industry would fire a worker with failure rates of 40%! In the software industry, we record our failures as bugs and then celebrate how hard we worked to remove the bugs from the list!

To build out a successful process that minimizes “bugs” in software, we must first take a hard look at the actual definitions of bugs, defects, and errors. The very terminology we use when developing embedded software can set our perspective and potentially devastate our ability to deliver successfully. The most used term to describe an issue with software is “there is a bug,” or the code is “buggy.” The term bug implies that some external entity is fighting against us and preventing us from being successful. When things go wrong, responsibility is displaced from the engineer, who should be in control of the situation, to a phantom force, and the engineer is simply going along for the ride.

Developers, teams, and companies involved in software development and supporting toolchains need to start shifting their perspectives and mindsets away from bugs and instead creating terminology that places the responsibility where it belongs, on themselves! Now I know that this sounds uncomfortable. In fact, I wouldn’t be surprised if you are a bit squeamish now and shifting around in your chair. But, at the end of the day, what we currently call bugs are errors or defects in the software, so why not just call them what they truly are?

A defect is a software attribute or feature that fails to meet one of the desired software specifications.[1] An error is a mistake or the state of being wrong.[2] An error could also be considered a discrepancy in the execution of a program.[3]

These are some excellent official definitions, but let me present them to you in another way:

- **Errors** are mistakes made by the programmer in implementing the software design.
- **Defects** are mistakes resulting from unanticipated interactions or behaviors when implementing the software design.
- **Bugs** are fictitious scapegoats developers create to shift blame and responsibility from themselves to an unseen, unaccountable entity.

Ouch! Those are some challenging definitions to swallow!

I gravitate toward defects when considering which terminology makes the most sense for developers. It's not as harsh as saying that it's an error that gets management or nontechnical folks all worked up, but it does put the responsibility in the right place. A defect in this sense could be an unintended consequence of an expected result in the application. The defect doesn't have a mind of its own and could have resulted from many sources such as

- Improper requirements specification
- Misunderstanding hardware
- Complex system interactions
- Integration issues
- Programmer error (blasphemy!)

When I lecture on defect management (formerly known as debugging), I often ask the audience how much time they spend on average in their development cycle debugging. The most significant responses are usually around 40–50% of the development cycle, but I always have outliers at 80%! Being able to deliver code successfully requires developers to have the correct mindset. The belief that bugs oppose our every line of code leads to loosey-goosey developed software high in defects.

Isn't it time that we squash the bugs once and for all and start using terminology that is not just accurate but shifts developers' perspectives away from being software victims and instead put the responsibility and control back on the developer?

If you are game to ditch the bugs and take responsibility for the defects you and your team create, let's move forward and look at how we can jump-start development to minimize defects. The phase that we are about

to cover is the process that I follow when I am starting to implement a new software project. You can still follow the process if you already have existing legacy code.

The end goal is to decrease how much time you spend fixing software defects. If you are like the average developer, spending 40% of your time and 4.5 months per year, set a goal to decrease that time. Could you imagine if you could just reduce that time in half? What would you do with an extra 2.25 months every year? Refactor and improve that code you've wanted to get? Add new features your customers have been asking for? Or maybe, you'll stop working on weekends and evenings and start to live a more balanced life? (LOL.)

The Defect Minimization Process

The defect minimization process isn't about implementing a rigid, unchangeable process. Defect minimization is about starting a project right, using the right tools, and having the right processes in place to catch defects sooner rather than later. Remember, the later a defect is found in the development cycle, the more costly it is to find and fix. The defect minimization process is more about finding defects as soon as they occur rather than letting them stew and burn up development time and budget. The ultimate goal is to not put the bugs in the software in the first place, but when they do find their way in, catch them as quickly as possible.

The process that I follow typically has seven phases. I roughly order the phases as follows:

- Phase 1 – Project Setup
- Phase 2 – Build System Setup
- Phase 3 – Test Harness Configuration
- Phase 4 – Documentation Facility Setup
- Phase 5 – Code Analysis
- Phase 6 – RTOS-Aware Debugging
- Phase 7 – Debug Messages and Trace

Phase 1 – Project Setup

The first phase is all about getting your project set up. Setting up a project is about much more than just creating and starting to bang out code. The project setup is about getting organized and understanding how you will write your code before writing a single line of it!

The first part of a project is to set up revision control. Today, that means getting a Git repository set up. Several popular repository vendors are available such as GitHub, GitLab, Bitbucket, and so forth. A repo setup is more than just creating a single repo. Developers need to plan out how multiple repositories will come together. For example, Figure 10-1 shows a project repository with multiple subrepos that hold modular project code so it can be reused in various projects. Git organization can be as simple as using a single repo or as complex as creating a tree of dependencies. I've found it's best to decide how things will be done early to minimize history loss.

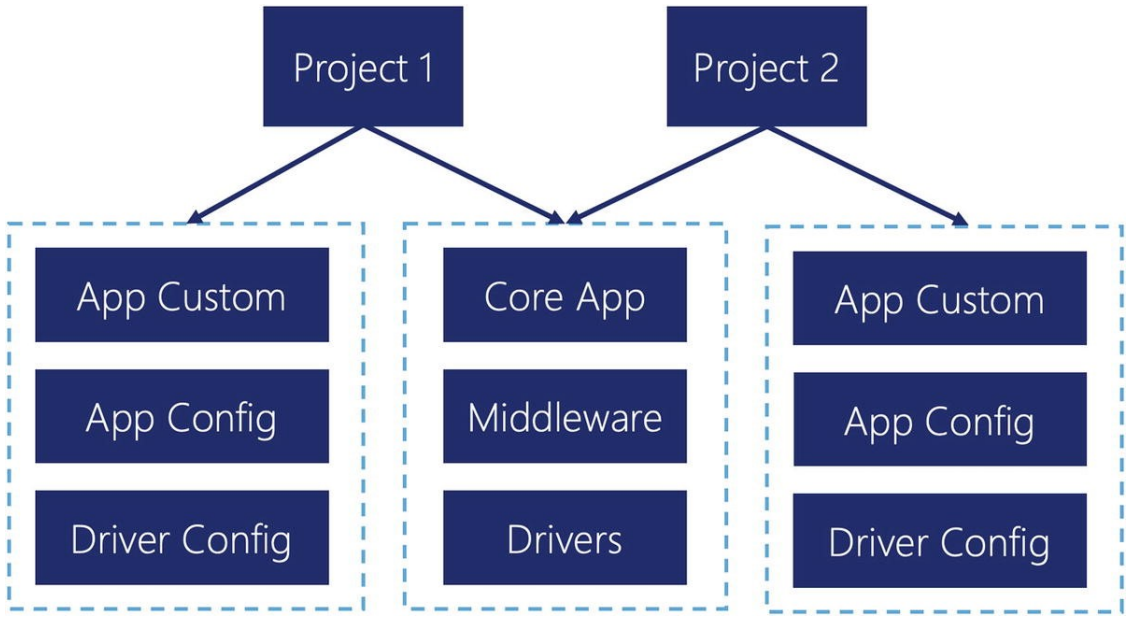


Figure 10-1 An example "tree" of Git repositories used to maintain multiple products

Best Practice While setting up your Git repos, populate your .gitignore file.

In addition to setting up the repo structure, developers in this phase will also organize their projects. Project organization involves identifying the directory structure used for the project and where specific types of files will live. For example, I often have gitignore, dockerfiles, makefiles, and readme files in the root directory. An example organization might look something like Figure 10-2.

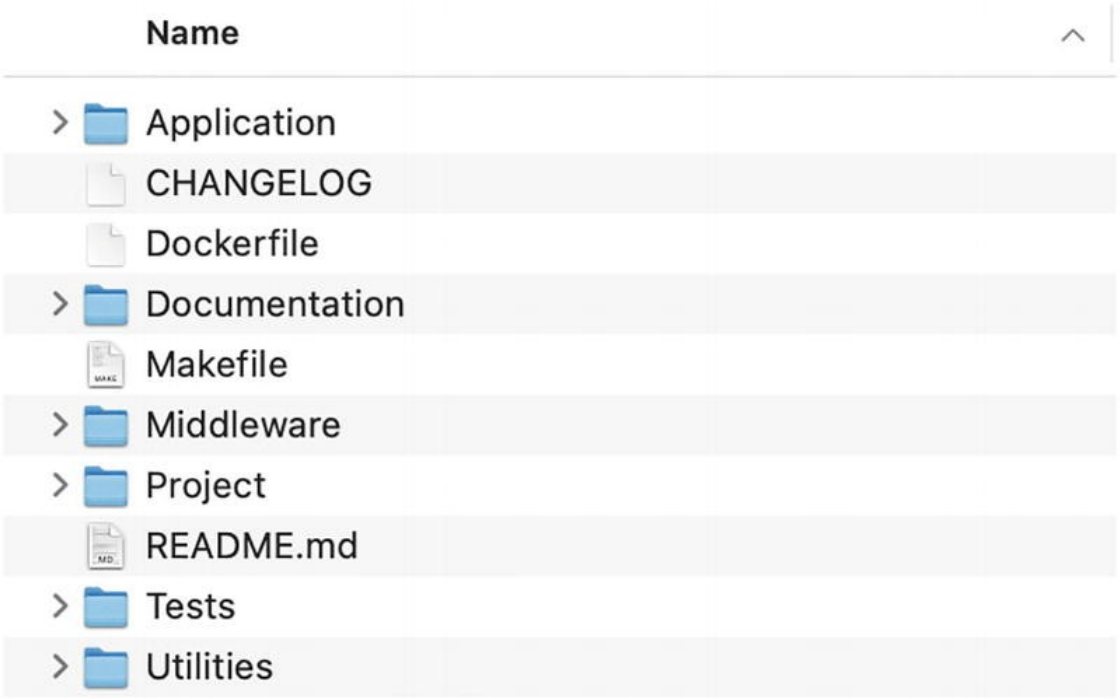


Figure 10-2 A simple project directory structure organization that contains vital elements in a project

To some degree, it doesn't matter how you organize your project if it's obvious to the developers working on it where things belong. I like having a project folder where I can have IDE-generated project files, analysis files, etc. You'll find every developer has their own opinion.

Phase 2 – Build System and DevOps Setup

Once the project is set up, the next step is configuring your build system. The build system has two parts: the local build system consisting of the makefiles necessary to build your target(s) and the build system set up to run your DevOps. If you aren't familiar with Embedded DevOps, check out Chapter 7 for details.

Rely on your IDE to build your application. You may not realize that behind the scenes, your IDE is autogenerating a makefile that tells the compiler how to take your source files, generate objects, and then link them together into a final binary image. A lot of modern development is moving away from relying on these autogenerated makefiles or at least removing them from being the center of the build process. With the popularity of Linux and the trending need for DevOps, many developers are moving away from IDEs and returning to working at the command line.

There is a significant need to customize the build process with DevOps, and if you just rely on the makefiles generated by your IDE, you will come up short.

A typical project today will require the build system to have several different makefiles. This includes

- An application makefile
- A configuration management makefile
- A makefile for the test harness
- A high-level "command" makefile

The relationship between these makefiles and the functionality they provide can be seen in Figure 10-3.

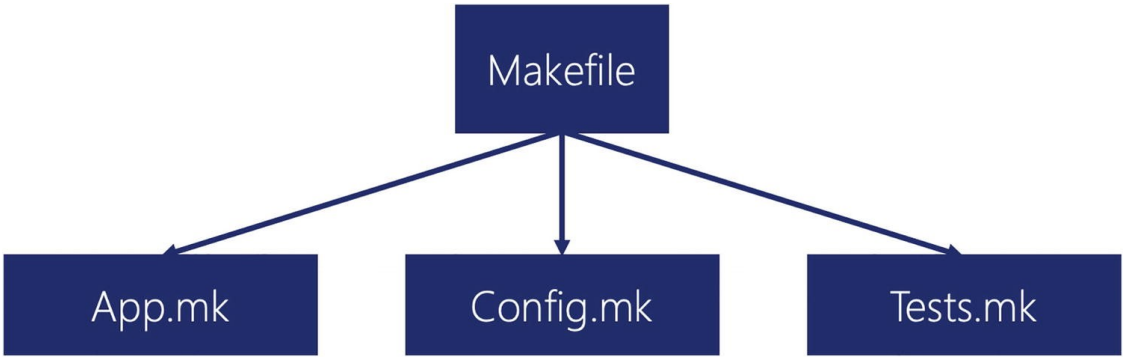


Figure 10-3 Relationships between various makefiles in a modern embedded application

Beyond the makefile, developers should also configure their DevOps processes during this phase. Again, there are plenty of tools that developers can leverage today, like GitLab and Jenkins. Besides configuring the CI/CD tool, developers will need to set up a container like Docker to hold their build system and make it easily deployable in their DevOps build pipeline. Appendix C gives a basic example on how to do this. What's excellent about Docker is that we can immediately set up our entire build environment and make it easily deployable to nearly any computer! No more setting up a slew of tools and hoping we don't have a conflict!

The details about how to set up and use DevOps processes will be covered in Chapter 7. At this point, I think it's essential for developers to realize that they don't have to set up their entire DevOps process in one fell

swoop. If you are new to DevOps, you can take a phased or agile approach to implement your DevOps. Start by just getting a basic build running in your pipeline. Then build out more sophisticated checks and deployments over time.

Having your DevOps processes set up early can help you catch build issues and bugs much faster than if you don't have one. I can't tell you how many times I've committed code that I thought worked only to have the DevOps system send me a build failed or test failed email. It would be embarrassing to commit code to a master or develop branch to break the branch for all the other developers. So getting DevOps in place early can help save a lot of time and money for defects.

Phase 3 – Test Harness Configuration

Before ever writing a single line of code, it's highly recommended that developers configure their test harnesses. There are certainly many types of testing that can be done on a system, but unit testing is the first line of defense for developers looking to decrease defects. A unit test harness provides developers with a mechanism for testing the functionality of each function and having those functions' regression tested locally before committing code to the repo and in the DevOps pipeline.

At first, using unit test harnesses for embedded systems can seem like a fool's errand. We're embedded software engineers! Much of our code interacts with hardware that can't be easily simulated in the host environment. At least, that is the excuse that many of us have used for years. If I think back to all the projects I've worked on in my career, probably 70%[4] of the code could have easily been tested in a unit test harness. The other 30% touches the hardware and probably still could have been tested with some extra effort.

Unit testing your software is crucial because it verifies that everything works as expected at the function level. You can even build up unit tests that verify function at the task level if you want to. The ability to perform regression testing is also essential. I can't tell you how many times I've worked on a project where a change in one part of the system broke a

piece in a different part. Without the ability to run regression tests, you would never know that new code added broke something already working.

We must discuss testing much, but we will go into deeper detail later. In this phase, remember that you want to get your test harness set up and ready to go. We aren't quite ready yet to start writing tests and developing our application.

Phase 4 – Documentation Facility Setup

The fourth phase in minimizing defects is configuring how the software will be documented. I have heard many arguments about how code is self-documenting and blah blah. Still, I've rarely encountered a code base where I could pick it up and understand what was going on without documentation or talking to the original developer. Perhaps I'm just dense, but good documentation goes a long way in making sure others understand design and intent. If I know that, I'll reduce the chances of injecting new defects into a code base.

Many tools and techniques are available to developers today to document their code. I may be considered a bit "old school" because I still like to use Doxygen. Today, a lot of open source projects use Sphinx. Honestly, I don't know that it matters which tool you use so long as you are using something that can pull comments from your code and that you can add high-level design details to. Everyone has their own preferences. I tend to still gravitate toward Doxygen because it does a great job of processing inline comments and making great documentation package. Developers can build templates that make it quick and easy to document code too.

During this phase, I will often configure Doxygen using the Doxygen Wizard. I'll configure my build system so that when I commit my code, the documentation is automatically generated, and the HTML version is posted to the project website. Since I reuse as much code as possible, I'll often copy over Doxygen template files for application and driver abstractions that I know I'll need in the project. I'll also set up a main page for the documentation and write any initial documentation that describes the

overarching project goals and architectural design elements. (These details will change, but it helps to start with something that can be changed as needed on the fly.)

Phase 5 – Static Code Analysis

Installing and configuring code analysis tools are key pillars to developing quality software. We want to ensure that with every commit, we are automatically analyzing the code to ensure it meets our coding standard and style guides and following industry best practices for our programming language (see Chapter 15 for practical tools). We also want to ensure that we are collecting our code metrics! The best way to do this is to install code analysis tools.

There are a couple of options for developers in handling their code analysis. First, developers should install all their code analysis tools as part of their DevOps pipelines. Utilizing DevOps in this way ensures that with every commit, the code is being analyzed, and a report is generated on anything that needs to be adjusted. Second, potentially optional, developers can have a duplicate copy of the tools on their development machines to analyze their software before committing it.

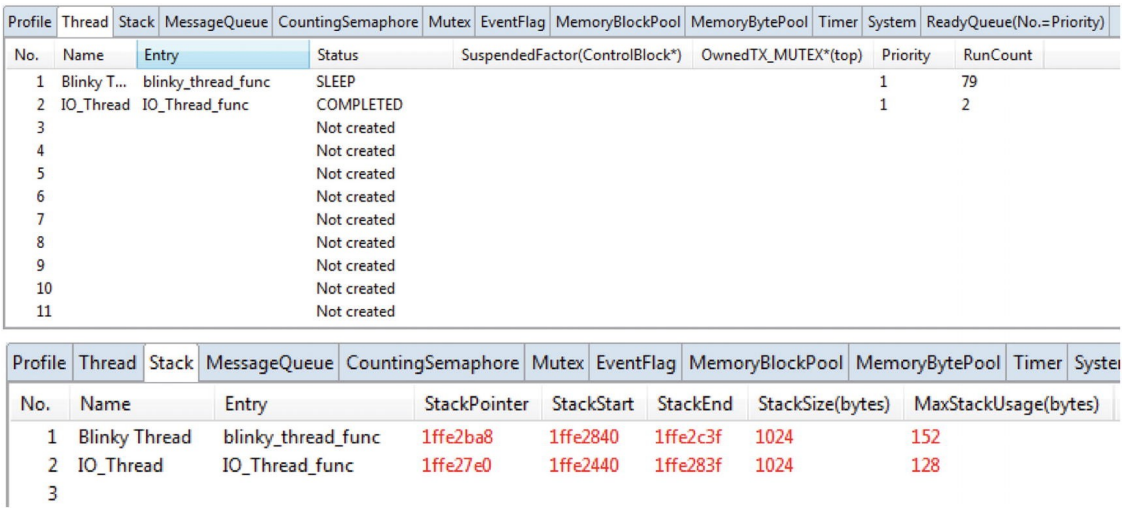
To some degree, it's again opinion. I usually like to analyze my code and ensure everything is good to go before I commit it. I don't want to commit code that has defects. However, if I am often committing in a feature branch that I have control over, why not just leverage the DevOps system to analyze my code while I'm working on other activities? For this reason, I will often push as much analysis to the build server as I can. My short cycle commits may not be pristine code, but with feedback from the DevOps tools, I can refactor those changes and have pristine code by the time I'm ready to merge my branch into the develop or master branches.

Again, we must recognize that what gets measured gets managed. This phase is about setting up our code analysis tools, whatever they are, before we start writing our code.

Phase 6 – Dynamic Code Analysis

In addition to statically analyzing code, developers must also dynamically allocate the code. The same tools developers use for this analysis will vary greatly depending on their overall system architecture and design choices. For example, suppose a team is not using an operating system. In that case, the tools they set up for dynamic code analysis may be limited to stack checkers and custom performance monitoring tools.

Teams using an RTOS may have more tools and capabilities available to them for monitoring the runtime behavior of their software. For example, Figure 10-4 shows a screenshot from Renesas' e^2 Studio where a simple, two-task application's behavior is monitored. We can see that the IO_Thread_func has run to completion, while the blinky_thread_func is currently in a sleep state. The run count for these tasks is 79:2. On the flip side, we can see the maximum stack usage for each task.



Profile | Thread | Stack | MessageQueue | CountingSemaphore | Mutex | EventFlag | MemoryBlockPool | MemoryBytePool | Timer | System | ReadyQueue(No.=Priority)

No.	Name	Entry	Status	SuspendedFactor(ControlBlock*)	OwnedTX_MUTEX*(top)	Priority	RunCount
1	Blinky T...	blinky_thread_func	SLEEP			1	79
2	IO_Thread	IO_Thread_func	COMPLETED			1	2
3			Not created				
4			Not created				
5			Not created				
6			Not created				
7			Not created				
8			Not created				
9			Not created				
10			Not created				
11			Not created				

Profile | Thread | Stack | MessageQueue | CountingSemaphore | Mutex | EventFlag | MemoryBlockPool | MemoryBytePool | Timer | Syste

No.	Name	Entry	StackPointer	StackStart	StackEnd	StackSize(bytes)	MaxStackUsage(bytes)
1	Blinky Thread	blinky_thread_func	1ffe2ba8	1ffe2840	1ffe2c3f	1024	152
2	IO_Thread	IO_Thread_func	1ffe27e0	1ffe2440	1ffe283f	1024	128
3							

Figure 10-4 RTOS-aware debugging using e^2 Studio and ThreadX

Having dynamic analysis tools early can help developers track their system's performance and runtime characteristics early. For example, if, in the ThreadX example, we had been expecting the two tasks to run in lockstep with a ratio of 1:1, the RTOS-aware debugging tab would have shown us immediately that something was wrong. Seeing that the IO_Thread_func had completed might even direct us to the culprit! Stack monitoring can provide the same help. For example, if one of the tasks only had 128 bytes of stack space allocated, but we were using 152, we would see that in the monitor and realize that we had a stack overflow situation.

RTOS-aware debugging is not the only tool developers should use at this stage.

Real-time tracing is another excellent capability that can help developers quickly gain insights into their applications. There are several different tools developers might want to set up. For example, developers might want to set up SEGGER Ozone or SystemView to analyze code coverage, memory, and RTOS performance. One of my favorite tools to set up and use early is Percepio's Tracealyzer, shown in Figure 10-5. The tool can be used to record events within an RTOS-based application and then used to visualize the application behavior.

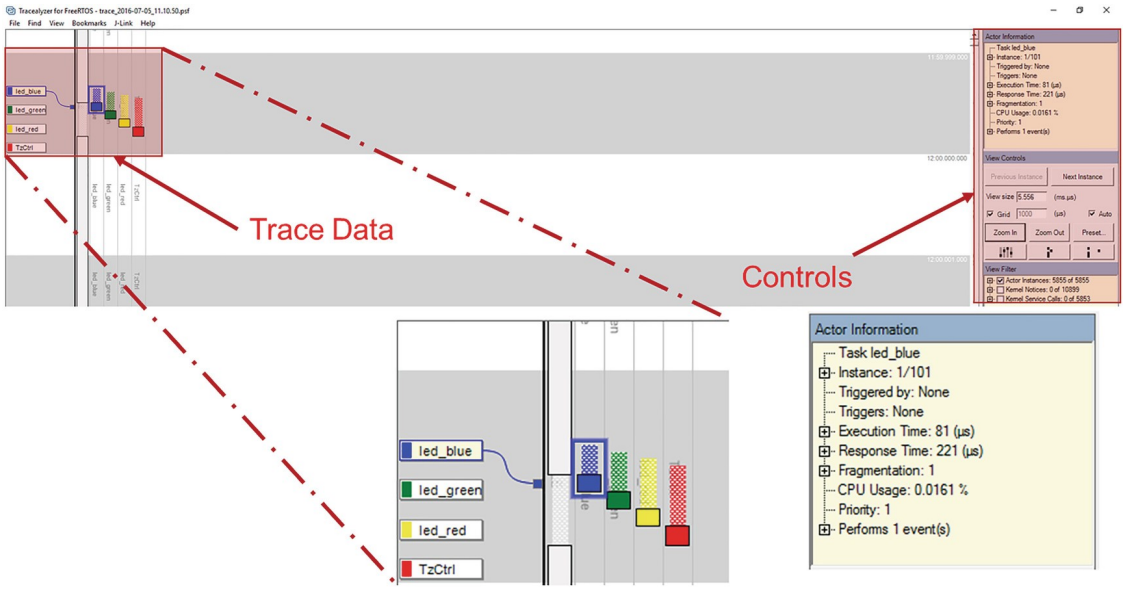


Figure 10-5 Percepio Tracealyzer allows developers to trace their RTOS application to gain insights into memory usage, performance, and other critical runtime data

There are quite a few insights developers can gain from these tools. First, early dynamic code analysis can help developers monitor their applications as they build them. Examining the data can then help to establish trends. For example, if I've seen that my CPU usage typically increases by 1–2% per feature but suddenly see a 15% jump, I know that perhaps I have a problem and should investigate the changes I just made.

Phase 7 – Debug Messages and Trace

The last phase for jump-starting software development to minimize bugs is to set up debug messages and trace capabilities. Finally, phase 7 is about getting data out of the embedded system and onto a host machine where the developer can see and understand how the application behaves. The capabilities used can range from simple printf messages to real-time data graphing.

An excellent first example of a capability set up in this phase is printf through the Instruction Trace Macrocell (ITM), if you are using an Arm part that supports it. The ITM provides 32 hardware-based stimulus channels that can be used to send debug information back to an IDE while minimizing the number of CPU clock cycles involved. The ITM is a hardware module internal to some Arm Cortex-M processors and typically sends the data back to a host computer through the Serial Wire Output (SWO) pin of the microcontroller. The debug probe then monitors the SWO pin and reports it back to the host where it can be decoded and displayed in an IDE.

Listing **10-1** shows an example of how a developer would utilize the ITM_SendChar CMSIS API to map printf to the ITM through _write.

```
int _write(int32_t file, uint8_t *ptr, int32_t len)
{
    for(int i = 0; i < len; i++)
    {
        ITM_SendChar((*ptr++));
    }
}
```

Listing 10-1 An example of how to map printf to the ITM in an Eclipse-based environment

In addition to setting up printf, developers would also ensure that they have their assertions set up and enabled at this stage. As we saw earlier, assertions can catch defects the moment they occur and provide us with useful debug information. Other capabilities can be enabled at this point as well. For example, some microcontrollers will have data watchpoints or can leverage the serial wire viewing to sample the program counter (PC) and perform other readings on the hardware. These can all help a developer understand their application and the state of the hardware, making it more likely that it is found and fixed immediately as a defect occurs.

When the Jump-Start Process Fails

There are going to be times when you are not able to prevent a defect in your code. A requirement is going to be missing or changed. You'll have a foggy mind day and inject a defect into your code. Embedded software to-

day is just too complex, and the human mind can't keep track of everything. When defects occur, developers must have troubleshooting skills to help them root out the defect as quickly as possible.

When it is time to roll up your sleeves and troubleshoot your system, there are eight categories of troubleshooting techniques that you have at your disposal that you can see in Figure 10-6. Starting at the top of the figure, you have the most straightforward troubleshooting technique, which involves using breakpoints. Then, moving clockwise, techniques become more advanced and provide more insights into the system.

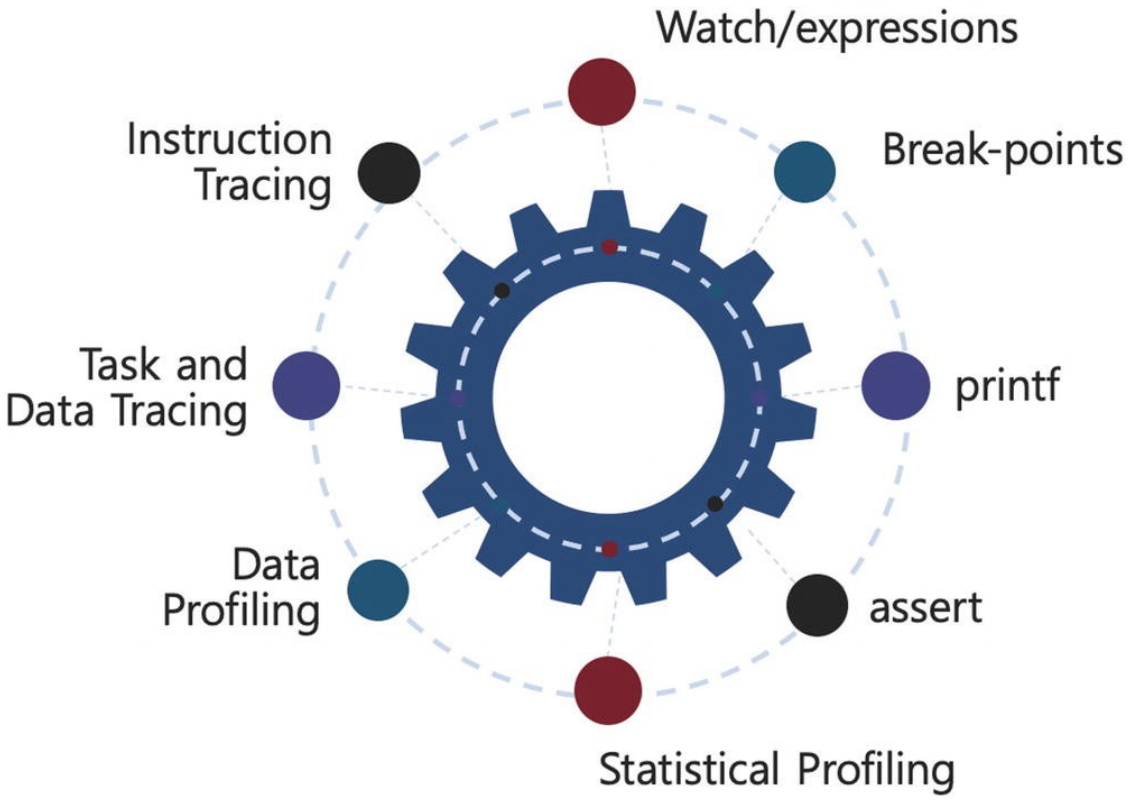


Figure 10-6 The technique categories used to find and remove defects from an embedded system

Let's briefly define what each of these troubleshooting techniques is:

Watch/expressions – An essential technique used to watch memory and the result of calculations. This technique is often used with breakpoints.

Breakpoints – One of the least efficient troubleshooting techniques, but one of the most used. Breakpoints allow the application execution to be halted and the developer to step through their code to evaluate how it is executing. This technique often breaks the real-time performance of an application while in use.

Printf – Using the printf facility to print "breadcrumbs" about the application behavior to understand its behavior. Using printf will often break real-time performance, but it depends on the implementation.

Assertions – A technique for verifying assumptions about the state of a program at various points throughout. Great for catching defects the moment they occur.

Statistical profiling – A method for understanding code coverage and application behavior. Statistical profiling periodically samples the PC to determine what code areas are being executed and how often. It can provide general ideas about performance, code coverage, and so forth without sophisticated instruction tracing tools.

Data profiling – A technique that leverages the capabilities of the microcontroller architecture and debugger technology to read memory during runtime simultaneously. The memory reads can be used to sample variables in near real time and then plot them or visualize them in the most effective way to troubleshoot the system.

Task and data tracing – A technique that uses an event recorder to record task change and other events in an RTOS. Task and data tracing can visualize CPU utilization, task switching, identify deadlocks, and understand other performance metrics.

Instruction tracing – A technique for monitoring every instruction executed by a microprocessor. The method helps debug suspected compiler errors, trace application execution, and monitor test coverage.

Now that we understand a little bit about what is involved in each of the technique categories, it's a good idea to evaluate yourself on where your skills currently are. Look through each category in Figure **10-6**. For each category, rank yourself on a scale from zero to ten, with zero being you know very little about the technique and ten being that you are a master of the method. (Note: This is not a time to be humble by giving yourself a nine because you can continually improve! If you've mastered the technique, give yourself a 10!)

Next, add the sum for each category to have a total number. Now let's analyze our results. If your score ranges between 0 and 40, you are currently stumbling around in the dark ages. You've probably not mastered many techniques and are less likely to understand the advanced techniques. You are crawling out of the abyss if your score is between 40 and 60. You've mastered a few techniques but probably are not fully utilizing the advanced techniques. Finally, if your score is between 60 and 80, you are a fast, efficient defect squasher! You've mastered the techniques available to you and can quickly discover defects in your code.

If your score is a bit wanting, you do not need to get worried or upset. You can quickly increase your score with a little bit of work. First, start by reviewing how you ranked yourself in each category. Identify the categories where you ranked yourself five or less. Next, put together a small plan to increase your understanding and use of those techniques over the coming months. For example, if you don't use tracing, spend a few lunch periods reading about it and then experimenting with tracing in a controlled environment. Once you feel comfortable with it, start to employ it in your day-to-day troubleshooting.

Final Thoughts

As developers, we must move beyond the thinking that bugs are crawling into our code and giving us grief. The reality is that there are defects in our code, and we're most likely the ones putting them there! In this chapter, we've explored a few ideas and a process you can follow to help you prepare for the inevitable defects that will occur in your code but will help you discover them as quickly as possible and remove them.

Changing how you think about bugs will 100% remove all bugs from your code! Instead, there will be only defects. Using the processes we've discussed, though, will allow you to catch defects quickly. As a result, you'll be able to decrease the time you spend troubleshooting your system, which should reduce stress and help developers become more effective. The benefits of having a software development process to minimize defects include

- Working fewer hours to meet deadlines

- More time to focus on product features
- Improved robustness and reliability
- Meeting project deadlines
- Detailed understanding of how the system is behaving and performing

Action Items

To put this chapter's concepts into action, here are a few activities the reader can perform to start decreasing how much time they spend debugging their code:

- Calculate what it costs you and the company you work for each year to debug your code:
  - What percentage of your time do you spend debugging?
  - Convert the percentage to hours spent.
  - Calculate your fully loaded salary and calculate your average hourly rate.
  - What are the total time and the total dollar amount?
- Based on how much time you spend each year debugging, what can you do to decrease the time by 10%? 20%? 50%? Put an action plan in place.
- What do you think about bugs? Do you need to change how you think about bugs and take more responsibility for them?
- Review the seven phases of the Jump-Starting Software Development to Minimize Defects process. Which phases do you need to implement before writing any additional code?

Footnotes

1 https://en.wikipedia.org/wiki/Defect

2 https://yourdiectionary.com/error

3 www.computerhope.com/jargon/e/error.htm

4 I'm making this number up from experience; I have no real evidence of this at this time.