J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_9

9. Application Modeling, Simulation, and Deployment

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

Embedded software developers traditionally do most of their work on development boards and product hardware. Unfortunately, there are many problems with always being coupled to hardware. For example, product hardware is often unavailable early in the development cycle, forcing developers to create “Franken boards” that cobble together enough hardware to move the project forward slowly. There are also often inefficiencies in working on the hardware, such as longer debug times. Modern developers can gain an incredible advantage by leveraging application modeling and simulation.

This chapter will explore the fundamentals of modeling and simulating embedded software. We will look at the advantages and the disadvantages. By the end of this chapter, you should have everything you need to explore further how you can leverage modeling and simulation in your development cycle and processes.

The Role of Modeling and Simulation

Chapter 1 shows that modern embedded software development comprises many parts. In fact, throughout this book, we have looked at the makeup of modern embedded software to be represented by the diagram shown in Figure 9-1. When we write embedded software, we use configurators, models, and handwritten code to create our application. Then, that application can be run in three scenarios: in our test harnesses, a simulator or emulator, and finally on our target hardware.

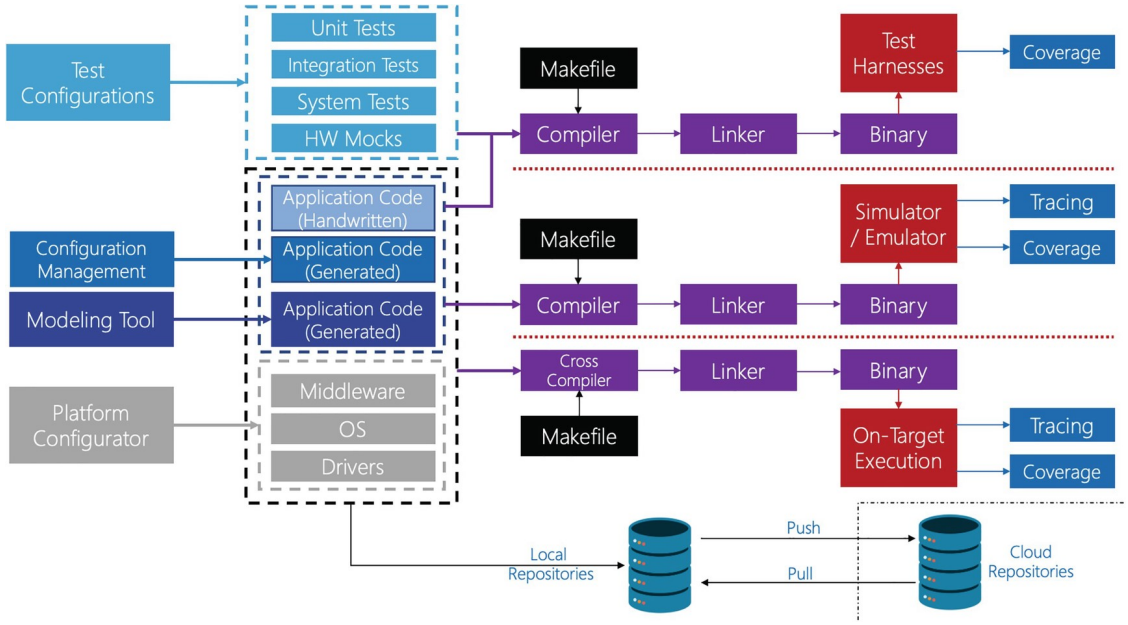

Figure 9-1 Modern embedded software applies a flexible approach that mixes modeling and configuration tools with hand-coded development. As a result, the code is easily tested and can be compiled for on-target or simulated execution

Traditionally, embedded software developers skip the modeling and simulation and go directly to the hardware. The tradition dates to the early days of embedded software when the software was tightly coupled to the hardware, and the use of objects, abstractions, encapsulation, and other modern software techniques wasn't used. As a result, developers had no choice but to get the hardware running first and then work on the application. Unfortunately, going this route often leads to lower-quality application code.

Each piece in the modern embedded software development diagram helps lead teams to create firmware faster with a higher level of quality. However, two pieces are overlooked far too often today: the use of modeling to prove out software and running the software in a simulator or an emulator. Both techniques provide several benefits that are overlooked by teams.

First, teams can become more efficient. Modeling and simulating the application code allows the team and customers to run the application code at the highest logic levels. Getting sign-off early on the high-level behavior can help to minimize scope creep and better focus developers on the features that need to be developed.

Next, modeling and simulation often lead to early defect discovery. The cheapest point in a development cycle to fix bugs is as soon as they happen or, better yet, before they ever happen! For example, suppose a requirement specification defect can be found early. In that case, it can save considerable money and help optimize the delivery schedule by removing painful debug time.

Finally, at least for our discussions, the last benefit is that modeled code can be autogenerated. Hand coding by humans is an error-prone endeavor. I like to believe that I'm a pretty good software developer, yet, if it were not for all the processes that I put in place, there would be a lot of defects and a lot of time spent debugging. Using a tool to model and generate the software makes writing and maintaining that code less error-prone.

There are too many benefits to leveraging modeling and simulation for teams to ignore and skip over them. So let's start our exploration of modeling, simulating, and deploying by looking at software modeling.

Embedded Software Modeling

The greatest return on investment in modeling embedded software is in the application's high-level, hardware abstracted pieces. The application code is the area of the code base that contains the business logic for the device. The application shouldn't have low-level dependencies or be tightly coupled to anything that requires the hardware. When we design software this way, we can model and even simulate the application and test our customers' requirements before we get too far down the development path.

Modeling is a method of expressing the software design that uses an abstract, picture-based language. The most used language to model software systems is the Unified Markup Language (UML). UML has been standardized for decades, providing standard models that can express nearly any software design.[1] For example, expressing a software system using state machines or state diagrams is very common. UML provides the standard-

ized visual components that one would expect to see represented in these types of diagrams.

Embedded software applications can be modeled in several different ways. First, designers can use a stand-alone UML modeling tool. A stand-alone tool will allow the designer to create their models. The more advanced tools will even generate code based on that model. The second method designers can use is UML modeling tools, including the capability to simulate the model. Finally, advanced tools often allow a designer to model, simulate, and generate code. Let's look at how some models can look with different toolchains.

Software Modeling with Stand-Alone UML Tools

The first modeling technique, and the one that I see used the most, is to use a stand-alone tool to model the software system in UML. UML provides a visual framework for developers to design various aspects of the software system. For example, if we were designing a system with two states, a SYSTEM_DISABLED and a SYSTEM_ENABLED state, we could use a state machine diagram to represent our system, as shown in Figure 9-2.

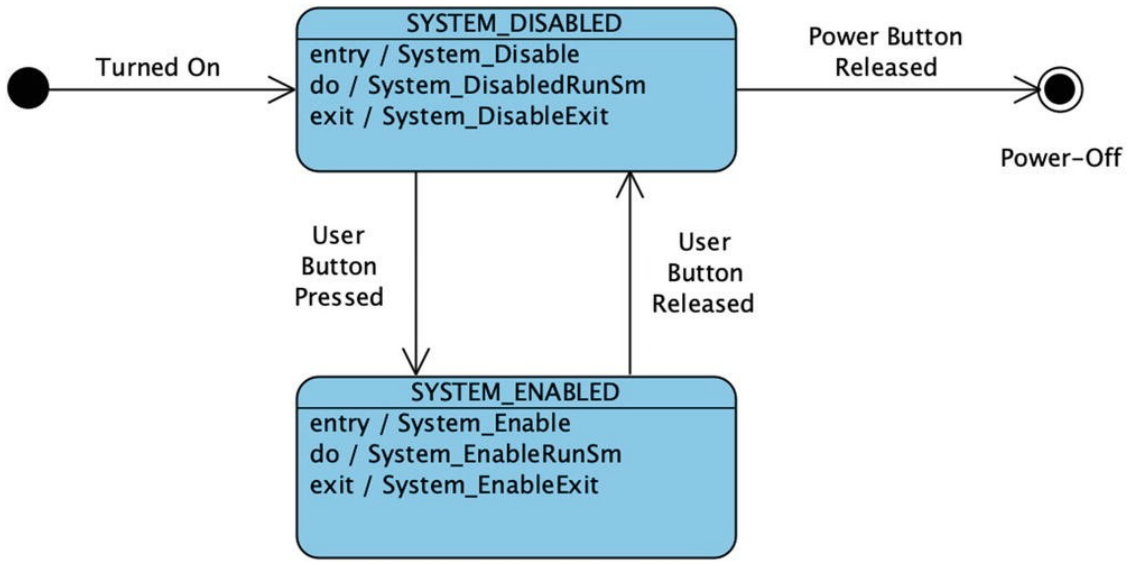

Figure 9-2 A simple state machine model demonstrates how the system transitions from the SYSTEM_ENABLED state to the SYSTEM_ENABLED state

The state diagram in Figure 9-2 is drawn using **Visual Paradigm**,[2] a low-cost modeling tool. The model shows how we transition initially to the SYSTEM_DISABLED state through the system being powered on (starting

from the solid black circle). When the system is first powered on, the SYSTEM_DISABLED state's entry function, System_Disable, is run once to disable the system. After the entry function is executed, the state machine will run System_DisabledRunSm during each state machine loop. It will continue to do so until the user button is clicked.

When the user button is clicked, the SYSTEM_DISABLED exit function will run, SystemDisableExit, to run any state cleanup code before transitioning to the SYSTEM_ENABLED state. Once there, the System_Enable entry function runs once, and then the System_EnableRunSm function runs while the system remains in SYSTEM_ENABLED. The transition from the SYSTEM_ENABLED state to the SYSTEM_DISABLED state follows a similar process.

As you can see, the state machine diagram creates a simple visualization that is very powerful in conveying what the system should be doing. In fact, what we have done here is modeled a state machine architecture diagram. Therefore, with some guidance, every engineer and even customers or management should be able to read and verify that the system model represents the requirements for the system.

Software Modeling with Code Generation

With a software system model, we can leverage the model to generate the system software if the correct tools are used. For example, in our previous state machine example, we could generate the C/C++ code for the state machine. That state machine code could then be integrated into our code base, compiled, and run on our target. If changes were required, we could then update our model, generate the state machine code again, and compile our code to get the latest update. Using a tool to generate our code is often lower risk and has a lower chance for a defect to be in the code. Unfortunately, I've not had much success with the code generation features of Visual Paradigm. However, I have had more success with IAR Visual State software.

Visual State isn't a complete UML modeling tool. It focuses explicitly on state machines. If we wanted to create a class diagram of our software,

Visual State would not be the tool to use. However, if we want to create, model, simulate, verify, and generate code for a state machine, Visual State will work just fine. Let's use an example to see how we can perform some of this functionality.

When I work on space systems, nearly every system has a heater to manage the spacecraft's temperature. A heater state machine can be simple with just three states: HEATER_OFF, HEATER_ON, and HEATER_IDLE. Modeling the heater controller using a UML diagram would look like Figure 9-3. The figure shows the three states and the events that cause the state transitions. For example, if the heater controller is in the HEATER_OFF state, the On_Command event will cause a transition of the controller to the HEATER_ON state. The next transition depends on the event and the controller's state.

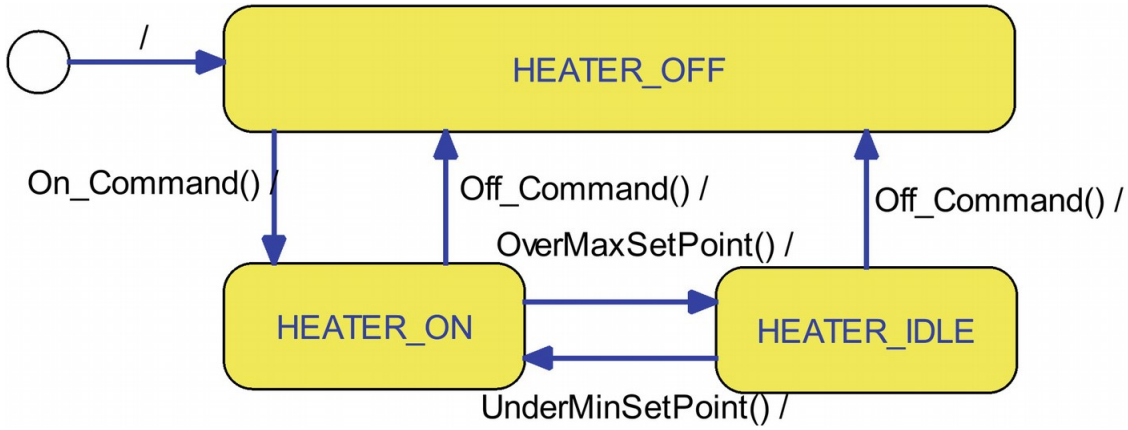

Figure 9-3 A heater controller state machine modeled in the IAR Visual State tool

I think the reader can see how powerful a model and code generation can be. However, using a model, though it is not all rainbows and sunshine, there are some issues that need to be thought through up front. One of the issues designers will often encounter with software modeling tools that generate code is that they need to find a way to connect their existing code outside the modeled code with the generated model. For example, if a state machine is designed that uses interrupts or waits for a message in an RTOS queue, some "glue logic" needs to be provided.

Figure 9-4 shows a simplified example of how the developer code converts hardware- or software-generated inputs into events that the state machine code can use. First, the developer inputs events, makes calls to the state machine API, and passes any data to the state machine. The state machine then acts on the events and generates action. Next, the developer needs to add code to take action

and convert it to the desired output that the embedded system can use.

Figure 9-4 Glue logic in user code converts inputs and outputs to/from the state machine

BewareGenerated code can be convenient, but defects can exist in the generator tool. So proceed with caution and test everything!

Software Modeling with Matlab

A second modeling technique designers can take advantage of is leveraging a tool like Matlab. Matlab is a powerful, multidisciplinary tool that provides designers and developers with many toolkits to model state machines, algorithms, and systems and even train machine learning models. Matlab is extremely powerful and can be used by multidisciplinary teams to model far more than just software; however, their Stateflow toolbox is an excellent tool for modeling state machines.

Figure 9-5 shows a simple state machine diagram created in state flow to represent the various heating states of an oven. A designer can quickly visually show three possible states: HEATER_OFF, HEATER_ON, and HEATER_IDLE. The transition into and out of each state is specified using the [] notation. For example, transitioning from the heating state to the idling state can occur if the temperature is [too hot] or from idling to heating if the temperature is [too cold]. Those definitions are not specific for an engineering application and would undoubtedly have some logic associated with them, but I think the reader can see the point.

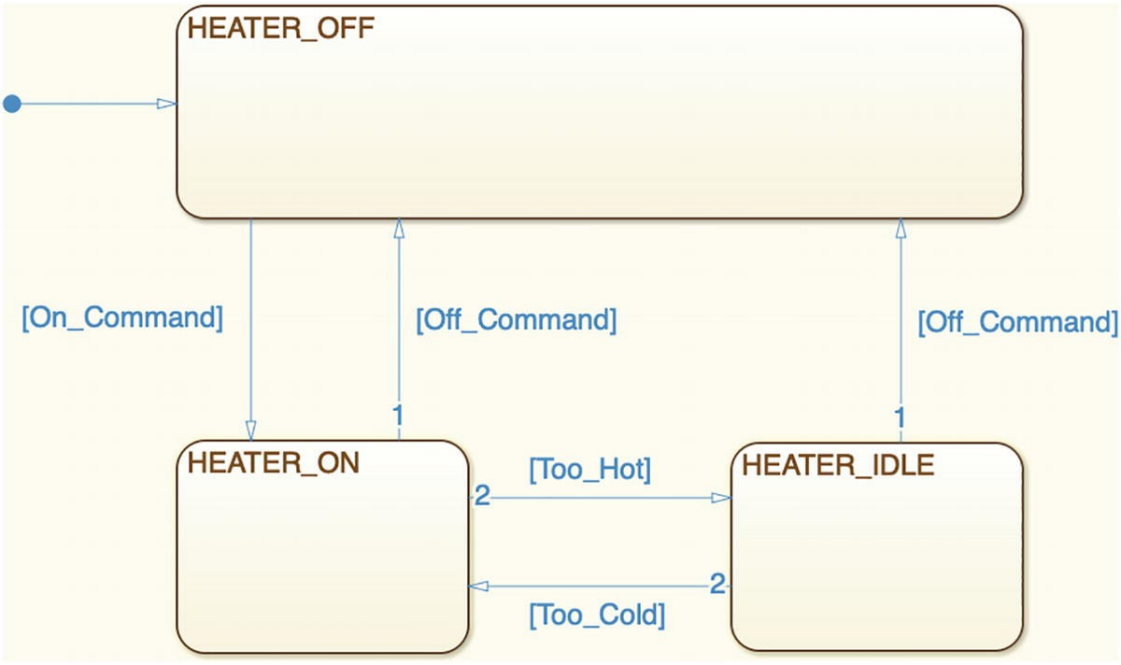

Figure 9-5 An example Matlab state machine visualization that shows the various states for a heater and causes for the state transitions

Visualizing what the system and the software are supposed to do is critical to ensuring that the right software is developed. For example, the diagram in Figure 9-5 can be presented to management and the customer and easily verified that it is the required state behavior for the system. If it is wrong, the designer can make a few quick adjustments to the diagram and get the sign-off. Then, the developers can either hand-code based on the diagram, or if they have the Embedded Coder toolbox, they can generate the state machine code themselves.

Embedded Software Simulation

Creating a software system model is a great way to understand the system that will be built. The ability to run that model in a simulation can provide insights into the design that otherwise may not have been possible without running production software on the device. In addition, simulation can provide insights to fix problems quickly and efficiently before they are allowed to balloon out of control.

There are several ways embedded developers can simulate their embedded software applications. The number of options is nearly dizzying! Let's explore a few options to get you started. Just keep in mind that there are

many options out there.

Simulation Using Matlab

One tool that designers can use to simulate embedded software is Matlab. Matlab allows designers not just to visualize but also to run the model, make measurements, and even generate C/C++ code from the model. As a result, Matlab is quite powerful as a development tool, and a price tag comes with that power. As we discuss throughout this book, though, the price isn't so much an issue as long as there is a solid return on investment for the purchase.

The best way to see how we can use Matlab to simulate embedded software is to look at an example. Let's say that we are building a new widget that has a heating element associated with it. The temperature must be maintained between some configurable minimum value and a configurable maximum value. The current temperature will determine whether the heater should be enabled or disabled. If the heater is enabled, then the temperature will rise. If it is disabled, then the temperature will fall.

We can view our system as having several parts. First, a state machine named Heater Controller manages whether the heater is on or off. The Heater Controller will input several temperature values: minimum, maximum, and current. The Heater Controller also can be enabled or disabled. Second, the system also has a Temperature state machine that determines whether the temperature rises or falls based on the Heater Controller's HeaterState. Figure 9-6 demonstrates what the Simulink model might be like.

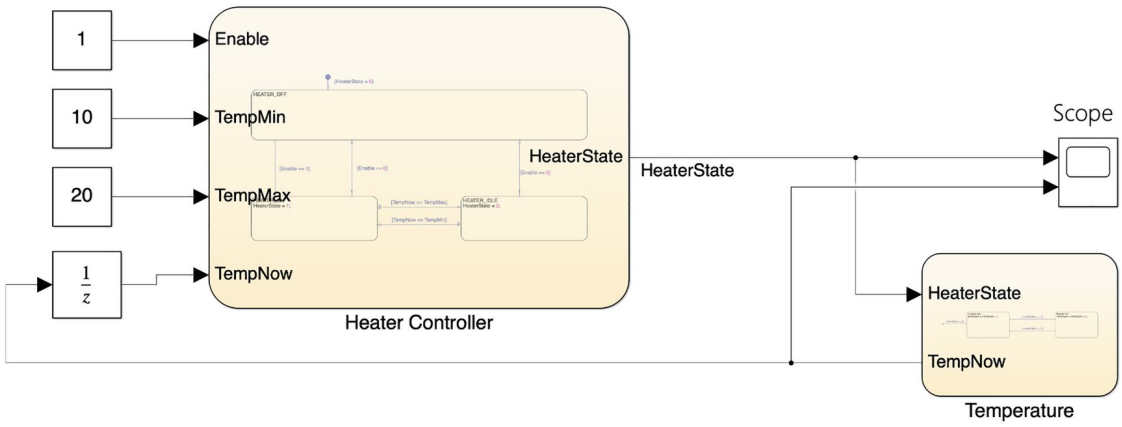

Figure 9-6 Heater Controller state machine with supporting Simulink functionality to simulate the state machines' behavior fully

There are two things I'd like to point out in Figure 9-6. First, notice the input to the Heater Controller TempNow port. I've placed a unit delay, represented by the 1/z block, to allow the simulation to run. Since Heater Controller depends on TempNow and Temperature relies on HeaterState, we have a logic loop that creates a "chicken and egg" issue. Placing the unit delay removes this issue and allows the model to simulate successfully. Second, on the upper right, I've added a scope to the model so that we can see the output of the temperature and the HeaterState throughout a 25-second simulation.

At this point, we've just defined the inputs and outputs of the two state machines needed for our simulation. We also need to define these state machines. Figure 9-7 shows the implementation for the Heater Controller state machine. Notice that it comprises just three states: HEATER_OFF, HEATER_ON, and HEATER_IDLE. We enter the state machine in the HEATER_OFF state and initialize HeaterState to 0. When the Enable input port signal is 1, which we've hard-coded using a constant, the state machine will transition from HEATER_OFF to HEATER_ON. In HEATER_ON, the HeaterState is changed to 1, representing that the heating element should be enabled.

Our controller is designed to maintain the temperature within a specific range. If TempNow exceeds or is equal to TempMax, then we transition to the HEATER_IDLE state and set HeaterState to 0. There is a transition back to HEATER_ON if the temperature falls below or equal to TempMin. That is it for the Heater Controller design!

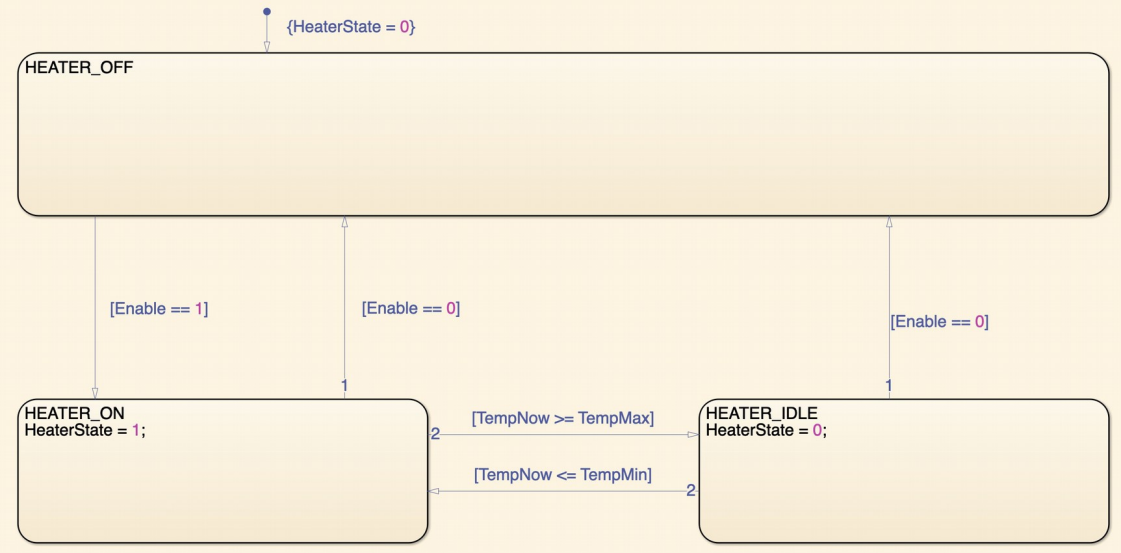

Figure 9-7 The Heater Controller state machine design

Next, we must define how the temperature will behave in the Temperature state machine. The temperature will have two possible states; it falls when the heater is off or rises when the heater is on. We will set the initial temperature to 2 degrees when the system starts. After that, the temperature increase or decrease will be in 2-degree increments. Figure 9-8 shows how the temperature state machine is implemented.

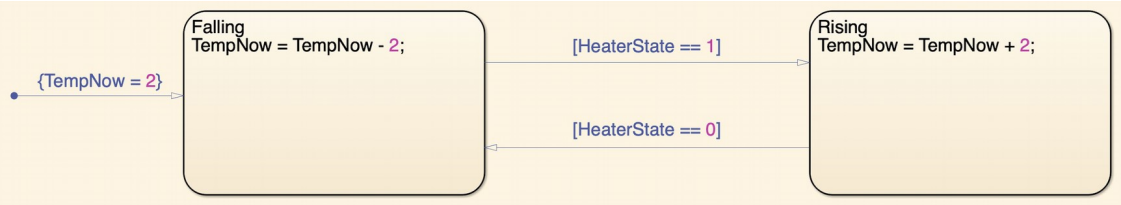

Figure 9-8 The Temperature feedback state machine design

We have all the pieces implemented in the model to represent how the system behaves and simulate it! Pressing the green run button in Simulink compiles the model and runs it. Let's check in and see if the system behaves as we want it to. Let's first check out the Sequence Viewer, a tool within Matlab. The Sequence Viewer allows us to see that state transitions in our state machines during the simulation. Figure 9-9 provides an example of the system we have just been designing.

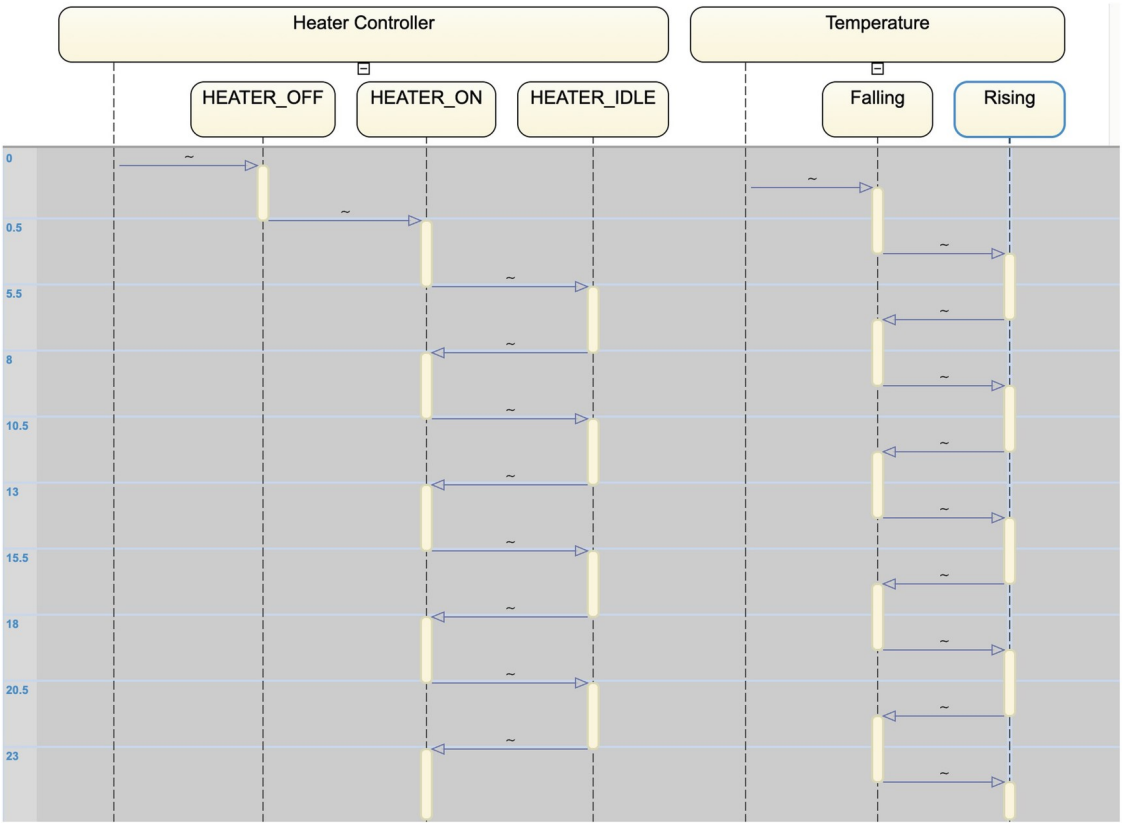

Figure 9-9 The sequence diagram representing the state transitions for the Heater Controller and Temperature simulation

Notice that in the simulation, we enter the HEATER_OFF state and then transition to the HEATER_ON state. For the rest of the simulation, the Heater Controller bounces between the HEATER_ON and the HEATER_OFF states. The Temperature state machine in lockstep bounces between the temperature falling and the temperature rising. The state transitions behave as we expect, but does the temperature? Figure 9-10 shows the output on our scope for the HeaterState and TempNow.

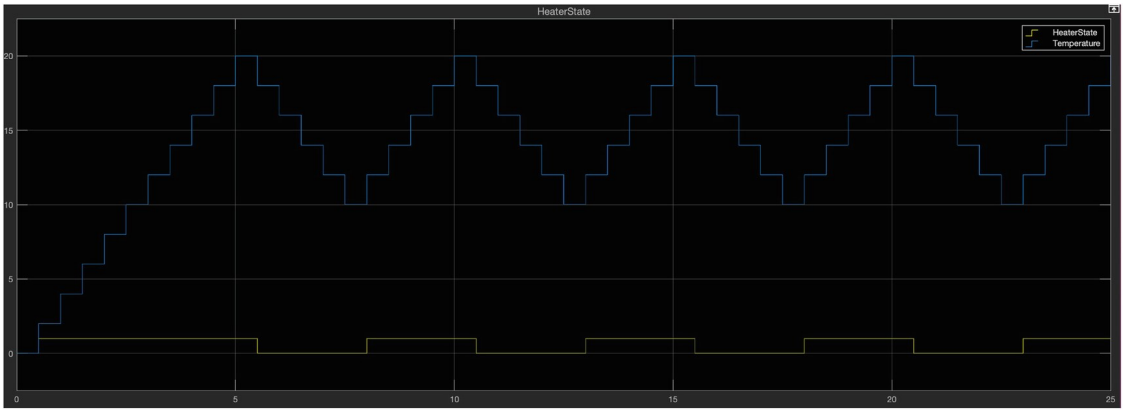

Figure 9-10 The scope measurements for the HeaterState and Temperature

The temperature starts to rise from zero when the HeaterState is enabled. The temperature rises to the 20-degree TempMax value, where we can see the HeaterState turnoff. The temperature then falls until it reaches 10 degrees, TempMin. The HeaterState then returns to an enabled state, and the temperature rises. We can see for the simulation duration that the temperature bounces between our desired values of TempMin and TempMax!

Before we move on, I want to point out that there is more than one way to do things and that models can be as sophisticated as one would like. For example, the model we just looked at is completely state machine, Stateflow, based with hard-coded parameters. I asked a colleague of mine, James McClearen, to design the same model within any input from me. His Matlab model for the Heater Controller can be seen in Figure 9-11.

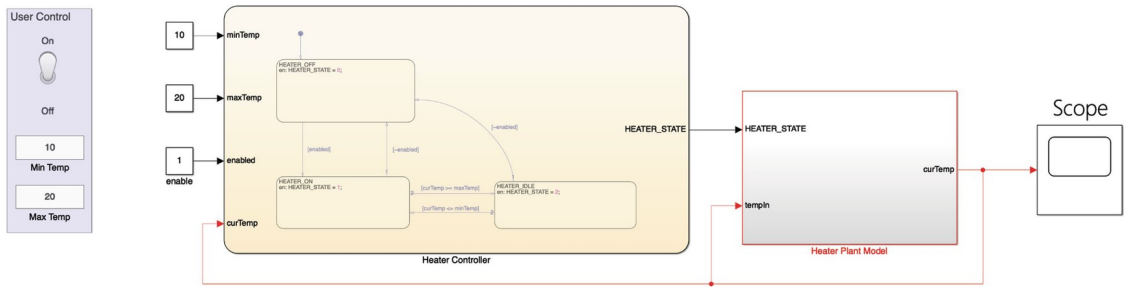

Figure 9-11 Another version of the heater model

There are a few subtle differences between his model and mine. First, instead of modeling the temperature response as a state machine, he just created it as a standard Simulink block. Next, he added a user control box to manage the enable/disable state and set the minimum and maximum temperatures. Finally, he didn't need the unit delay block because his Heater Plant Model included a different transfer function on the output as shown in Figure 9-12.

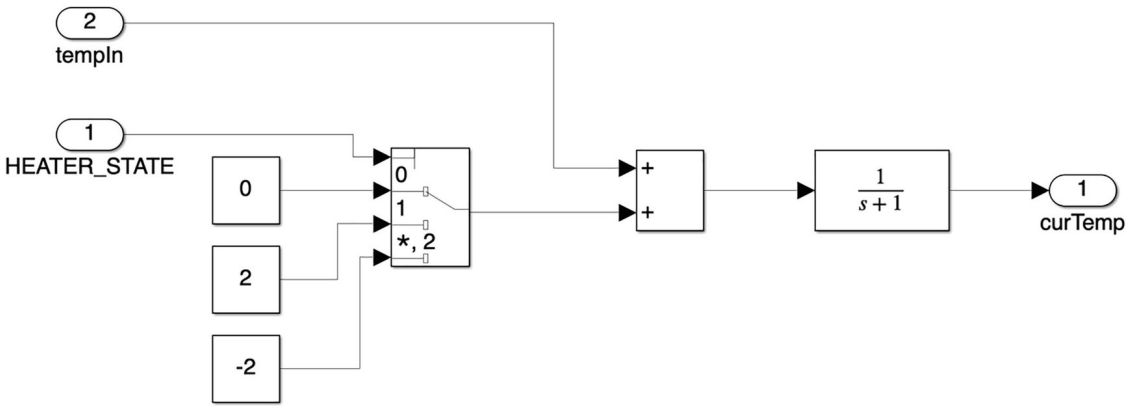

Figure 9-12 The Heater Plant Model in Simulink which uses "discrete" logic blocks rather than a state machine to simulate the temperature response

Which model is right? Both! Each model produces the same results. The first model was developed by an embedded software consultant with basic to intermediate Matlab skills focusing on demonstrating state behavior. The second model was developed by a mechanical engineering consultant, focused on modeling it the way a Matlab expert would.

We've just successfully simulated how we want our Heater Controller embedded software application to behave, and we've done it without any hardware in the loop! At this point, we have several options available on how we proceed. First, we can continue adjusting the simulation to test different conditions and use cases to ensure it behaves as needed under all conditions. Next, we can use our visualization to hand-code the state

machine in C/C++. Alternatively, we can use the Embedded Coder toolbox, another tool within Matlab, to generate the C/C++ code for the Heater Controller and then deploy it to an embedded target. How you proceed is entirely up to your own needs and comfort level.

> **Note** We don't need to generate code for the Temperature state machine. TempNow would be generated by a thermocouple fed into the Heater Controller API.

Software Modeling in Python

Using a visualization and simulation tool like Matlab is not the only way to model embedded software. Designers can also model the behavior of their systems by using a software language that is not an embedded language and runs on a host computer. For example, a designer might decide that they can rapidly model the system by writing the application code in Python!

I know that writing code in Python leaves a bad taste in many embedded software developers' mouths, but Python can be a game-changing tool for developers if they are willing to embrace it. For example, I had a customer who we were designing an electronic controller with. The controller had several states, telemetry outputs, and commands that it would respond to under various conditions. To ensure that we understood the controller's inputs, outputs, and general behaviors, we started by spending several weeks creating a model of how the controller should behave in Python.

The Python controller was written to simulate the electronic controller's behavior. We provided a host computer running the Python model that used a USB to serial adapter to communicate with the other subsystems like it was the actual controller. The customer verified that the model behaved as needed from an input, output, and command standpoint. It didn't have the full real-time performance or energy constraints the end controller would have; however, we were able to get the customer to sign off. Once we had the sign-off, we started to design and build the real-time

embedded controller that would provide those same modeled inputs, outputs, and behaviors.

At first glance, modeling the system in another language in code that we essentially just threw away may seem like a waste. However, it was far more effective to build a rapid prototype of the system and shake out all the interfaces to the controller so that the customer could sign off on what we would build. If that had not been done, we undoubtedly would have started to design and build something that would not have met the customers' requirements. We then would have had to incrementally adjust and probably would have ended with a rat's nest of code as the customer changed their minds and evolved the product. Instead, we got early sign-off with fixed requirements we could work from. The customer received a representative model they could then use to develop and test their systems against, which then takes the pressure off the need to deliver the actual product as quickly as possible. With the pressure off, the chances of rushing and creating defects dramatically decrease.

> **Note** There isn't any such static project where requirements don't change. Modeling the system up front can help minimize changes and fully define the core product features.

Using Python is just one example. There are certainly plenty of additional methods that can be used to simulate an embedded system.

Additional Thoughts on Simulation

There are several other ways that developers simulate their embedded software applications without the need for the underlying embedded hardware. I don't want to go into too much detail on all these methods. I could easily put together another full book on the topic, but I at least would like the reader to know that they exist.

One tool that I have found to be helpful in creating simulations is to use **wxWidgets**.[3] wxWidgets is a cross-platform GUI library that provides language bindings for C++, Python, and several other languages. wxWidgets is interesting because with the C++ compiler, it's possible to compile an

RTOS. For example, FreeRTOS has a Windows port that can be compiled into wxWidgets. Once this is done, a developer can create various visualizations to simulate and debug embedded software. Dave Nadler gave a great talk about this topic at the 2021 **Embedded Online Conference**[4] entitled "**How to Get the Bugs Out of your Embedded Product**."[5]

Another tool to consider is using **QEMU**.[6] QEMU is a popular generic and open source machine emulator and virtualizer. Over the last several years, I've seen several ports and developers pushing to emulate Arm Cortex-M processors using QEMU. For example, if you visit `www.qemu.org/docs/master/system/arm/stm32.html`, you'll find an Arm system emulator for the STM32. The version I'm referring to supports the netduino, netduinoplus2, and stm32vldiscovery boards.

Leveraging QEMU can provide developers with emulations of low-level hardware. For embedded developers who like to have that low-level support, QEMU can be a great way to get started with application simulation without removing the low-level hardware from the mix. The STM32 version I mentioned earlier has support for ADC, EXTI interrupt, USART, SPI, SYSCFG, and the timer controller. However, there are many devices missing such as CAN, CRC, DMA, Ethernet, I2C, and several others. Developers can leverage the support that does exist to get further down the project path than they would have otherwise.

> **Note** There are many QEMU emulator targets. Do a search to see if you can find one for your device (and a better STM32 version than I found).

There are certainly other tools out there for simulation, but I think the core idea is that the tool doesn't matter. The idea, the concept, and the need within teams are that we don't need to be as hardware dependent as we often make the software out to be. Even without a development board, we can use a host environment to model, simulate, debug, and prove our systems long before the hardware ever arrives. Today, there aren't enough teams doing this, and it's a technique that could dramatically transform many development teams for the better.

Deploying Software

The final output option for our software that isn't a test harness or a simulator is to deploy the embedded software to the hardware. I suspect that deploying software to the hardware is the technique that most readers are familiar with. Typically, all that is needed is to click the debug button in an IDE, magic happens, and the software is now up and running on the target hardware. For our purposes in this book, we want to think a little bit beyond the standard compile and debug cycle we are all too familiar with.

In a modern deployment process, there are several mechanisms that we would use to deploy the software such as

- The IDE run/debug feature
- A stand-alone flash tool used for manufacturing
- A CI/CD pipeline job to run automatic hardware-in-loop (HIL) tests
- A CI/CD pipeline job to deploy the software to devices in the field through firmware-over-the-air (FOTA) updates or similar mechanism

Since you are undoubtedly familiar with the IDE run/debug feature, let's examine a few of the other options for deploying software.

Stand-Alone Flash Tools for Manufacturing

Once developers get away from their IDE to deploy software, the software deployment process begins to get far more complex. When manufacturing an embedded product, a big concern that comes up is how to program units on the assembly line at the manufacturing facility. Having a developer with their source and a flashing tool doesn't make sense and is in fact dangerous! When we begin to look at how to deploy code at the manufacturing facility, we are looking to deploy compiled binaries of our application.

The biggest concern with deploying software at the manufacturing facility is the theft of intellectual property. The theft can come in several different ways such as

- Access to a company's source code
- Reverse-engineering compiled binaries

- Shadow manufacturing[7]

It's interesting to note that with today's tools and technology, someone can pull your compiled code off a system and have readable source code in less than ten minutes! A big push recently has been for companies to follow secure device manufacturing processes.

What is necessary for manufacturing is for companies to purchase secure flash programmers. A secure flash programmer can hold a secure copy of the compiled binary and be configured to only program a prespecified number of devices. A secure programmer solves all the concerns listed earlier in a single shot.

For example, the secure flash programmers will not allow a contract manufacturer to have access to the binaries. The binaries are stored on the flash programmer, but not accessible. In addition, the fact that the secure flash programmer will only allow a specified number of devices to be programmed means that the devices programmed can be easily audited. A company doesn't have to worry about 1000 units being produced that they don't know about. The secure flash programmer tracks each time it programs a device, creating a nice audit trail.

The secure manufacturing process is quite complex, and beyond the book's scope, but I at least wanted to mention it since many of you will encounter the need to deploy your firmware and embedded software to products on the assembly line.

CI/CD Pipeline Jobs for HIL Testing and FOTA

A very common need is to deploy embedded software within a CI/CD pipeline as part of Embedded DevOps processes. The CI/CD pipeline will typically have two job types that need to be performed. First, software will need to be deployed to the target that is part of a HIL setup. The second is that the software will need to be deployed to the target that is in the field. Let's explore what is required in both situations.

Figure **9-13** shows an example architecture that one might use to deploy

software to a target device. As you can see, we have the target device connected to a flash programming tool. That programming tool requires a programming script to tell it what to program on the target device. The program script can be as simple as a recipe that is put together as part of a makefile that knows what commands to execute to program the target.

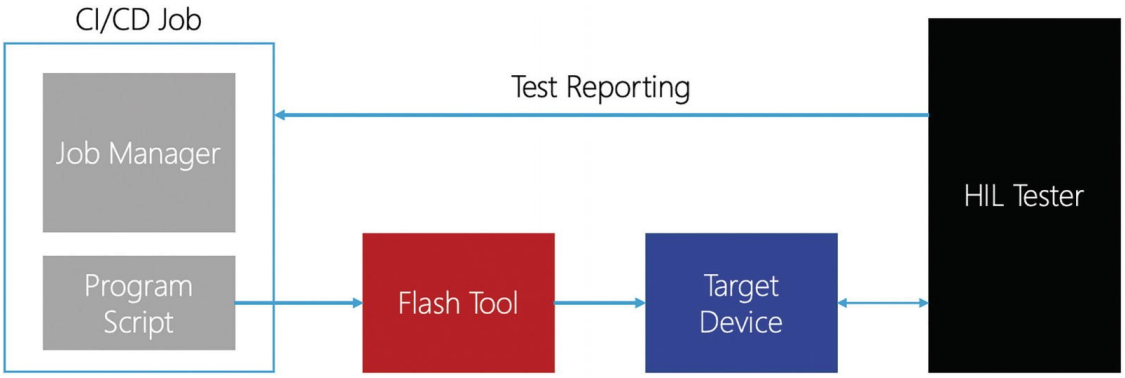

Figure 9-13 Example architecture for deploying to a target device in a CI/CD HIL test job

For example, using a typical setup using a SEGGER J-Link Ultra+, I would install the J-Link tools on a Docker image that is part of my CI/CD framework. My Git repository would contain a folder that has the J-Link command script included. The command script is just the commands necessary to program a device such as commands to reset, erase, program, and verify the device. The script is necessary because there is no high-level IDE driving the programming process.

With the command file in place, a simple recipe can then be used within the main makefile to call the J-Link utilities and run the command script with the desired application to program. If you've lived in a Windows IDE environment, the process can seem complicated at first. Rest assured, a little bit of digging and it's relatively simple. In fact, you'll find an example for how to add a deployment job to a CI/CD pipeline using a SEGGER J-Link in Appendix C. Once it's set up for the first time, there is usually very few changes that need to be made to maintain it.

Adding production deployment to a CI/CD pipeline can be complicated. It often involves careful collaboration with the operations team. There are several different approaches I've used in the past to perform production deployments. First, we've used our CI/CD pipeline in conjunction with the tools in Amazon Web Services. As part of our deploy job, we can create

AWS jobs for pushing firmware to our deployed devices. I've found this approach has worked well. There is a bit of a learning curve, so I recommend starting with small batches of deployments and then eventually moving to larger ones once you have the operational experience.

The second method that I've seen used but have not yet personally used is to leverage a tool like **Pelion**.[8] Pelion provides companies with the mechanisms to manage fleets of IoT devices in addition to managing their device life cycles. Teams can integrate their deployment pipelines into the Pelion IoT management platform and then go from there. Like I mention, I've seen the solution but I have not used it myself, so make sure if you do use a platform like this that you perform your due diligence.

Final Thoughts

The need to model, simulate, and deploy software systems are not techniques reserved for big businesses with multimillion-dollar budgets. Instead, modeling and simulation are tools that every development team can use to become more efficient and deliver software that meets requirements at a higher-quality level. At first, modeling and simulation can appear intimidating, but steady progress toward implementing them in your development cycles can completely transform how you design and build your systems.

Using a CI/CD pipeline to deploy software to hardware devices locally and in the field can dramatically change how companies push firmware to their customers. I can't tell you how many times per day I get a notification that Docker has a new software update. Docker is continuously deploying new features and capabilities as they are ready. Could you imagine doing that with your customers or team? Providing a base core functionality and then continuously deploying new features as they become available? It might just help take the heat off, delivering the entire product if it can be updated and enhanced over time.

There is a lot more that can be done with simulation and the CI/CD pipeline. There are improvements from an organizational standpoint, along

with additional capabilities that can be added. For example, the deployToTarget we created only works on a local embedded device. A fun next step would be integrating this into an over-the-air update system that can be used to push production code to thousands of devices. Granted, you might want to start by deploying to just a half dozen boards scattered throughout the office.

I think that you will find that modeling, simulation, and automated deployment are great additions to the current processes that you are using. As I've mentioned, you'll find that these enable you to resolve countless problems well before the hardware is required. When the hardware is needed, you now have a simple process to deploy your application to the hardware. With some modification, hardware-in-loop testing can even be performed, adding a new level of feedback to the application development.

Action Items

To put this chapter's concepts into action, here are a few activities the reader can perform to get more familiar with modeling, simulation, and deployment:

- Examine Figure 9-1 again. What areas of this diagram are you missing? What will you do over the coming quarters to add in the missing pieces?
- Research and identify several tools that will allow you to model your software. Pick a few to download trials for. Model a simple system like the Heater Controller. Share the model with several colleagues.
  - How quickly do they understand the design intent?
  - How quickly and accurately can they hand-code the model?
  - If the model changes, what issues are encountered trying to update the code by hand?
  - What tool best fits your design needs?
- Use a tool like Matlab to design and simulate a simple system.
  - How did the tool affect your understanding of the system?
  - How did the tool affect your efficiency, bug quality, and evolution of the software product?
- Create a simple model of a product using Python on a host computer. Can you see the benefits of having a functionally equivalent system in a host environment? What are the benefits? Any cons?

- Schedule time to work through the deployment example in Appendix C. You'll find that you can easily create a CI/CD pipeline that can use a J-Link to program a board automatically.

Footnotes

1 The discussion of UML is beyond the scope of this book, but you can learn more about it at www.uml.org.

2 www.visual-paradigm.com/

3 www.wxwidgets.org/

4 https://embeddedonlineconference.com/

5 https://embeddedonlineconference.com/session/How_to_Get_the_Bugs_Out_of_your_Embedded_Product

6 www.qemu.org/

7 Shadow manufacturing is when the contract manufacturer builds your product during the day and by night is building your product for themselves.

8 https://pelion.com/