J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_13

13. Configurable Firmware Techniques

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

By the age of 40, I had the opportunity to review and write code for over 150 firmware projects. One thing that I've noticed about firmware projects is that developers rarely write their code in a configurable manner, even if it is warranted. Don't get me wrong; I've seen a lot of progress toward layering architectures, managing coupling, cohesion, and trying to maximize reuse. Unfortunately, however, I've seen a lot to be desired regarding configuration.

I've seen many projects with similar elements and behaviors yet entirely written from scratch for each project. The use of configurations could simplify so many embedded projects, yet many developers are stuck recreating the wheel repeatedly. The projects I see that use configurable firmware techniques often leverage copy and paste and the core mechanism to prepare configuration files. Obviously, this is not very desirable since copy and paste can inject bugs into a code base due to a developer not updating a line of code. (Interruptions in the office, a phone call, or just the mind drifting can cause a slip up.)

This chapter will explore configurable firmware techniques I've been using in my code for over a decade. I've blogged or spoken about some of these techniques, but this chapter will go further behind the curtain and demonstrate how you can write configurable firmware that is also autogenerated. I've used these techniques to write code that is not just configurable but also created at an accelerated rate. Due to the maturity of my scripts and techniques, they don't easily fit into this chapter, so I've streamlined them to give you the core idea, the technique you can use for your firmware.

You are going to learn in this chapter how to write code that is configurable through configuration tables. These tables allow you to write reusable, scalable, and configurable code for many applications. Those tables can also be generated using configuration files and scripts, resulting in your application's C modules. Let's get started!

Leveraging Configuration Tables

The one technique that has the potential to revolutionize the way that you write your embedded software is configuration tables. If you were to look at the code that I write, you'd discover that my code is full of patterns that ease system configuration and management that looks like a table. For example, I might have a table like Listing 13-1 that would be passed to a digital input/output initialization function as a pointer. The initialization function would read this table and then initialize the peripheral registers based on the data stored in the table.

```
static Dio_Config_t const Dio_Config[] =
{
// Name,    Resister, Drive, Filer,   Dir,   State,  Function
```

```
  {PORTA_1, DISABLED, HIGH, DISABLED, OUTPUT, HIGH,
FCN_GPIO},
  {PORTA_2, DISABLED, HIGH, DISABLED, OUTPUT, HIGH,
FCN_GPIO},
  {SWD_DIO, PULLUP,   LOW,  DISABLED, OUTPUT, HIGH,
FCN_MUX7},
  {PORTA_4, DISABLED, HIGH, DISABLED, OUTPUT, HIGH,
FCN_GPIO},
  {PORTA_5, DISABLED, HIGH, DISABLED, INPUT,  HIGH,
FCN_GPIO},
  {PORTA_6, DISABLED, HIGH, DISABLED, OUTPUT, HIGH,
FCN_GPIO},
  {PORTA_7, DISABLED, HIGH, DISABLED, OUTPUT, HIGH,
FCN_GPIO},
  {PORTA_8, DISABLED, HIGH, DISABLED, OUTPUT, HIGH,
FCN_GPIO},
};
```

Listing 13-1 An example digital input/output table

You'll notice that each row in the table represents a GPIO pin, and each column represents a feature. The table is populated with the initialization data for each feature on each pin. For example, the `PORTA_1` pin would have its pull-up/pull-down resistor disabled. The drive strength would be set to high. No filtering would be enabled. The pin would be set as an output with an initial value of high (Vcc) and configured with the GPIO function. Other pins might have functions like MISO, MOSI, CLK, SDA, SCL, etc. The configuration table is nothing more than an array of structures. The structure definition would look something like Listing 13-2.

Configuration tables have a lot of advantages associated with them. First, they are human-readable. I can quickly look at the table and see what is going on. Next, they are easily modified. For example, if you wanted to go in and change `PORTA_7` to initialize in the `LOW` state, you could just change the value in the table, recompile, and make the change! If you had a new microcontroller with more pins, you could just add new rows to the table! If the new microcontroller has fewer pins, then you remove rows. The idea is the same with the features that are represented as columns.

The last advantage I want to mention that is important and that I rarely see exploited is that configuration tables can also be generated automatically using configuration scripts. However, we'll talk a bit more about that later in the chapter. Nevertheless, it's a technique that I think you'll find helpful and may forever change how you write your embedded software.

```
/**
 * Defines the digital input/output configuration table
 * elements that Dio_Init uses to configure the Dio
 * peripheral.
 */
typedef struct
{
   DioChannel_t Channel;  /**< The I/O channel             */
   uint8_t Resistor;      /**< DISABLED,PULLUP,PULLDOWN    */
   uint8_t DriveStrength;       /**< HIGH or LOW           */
   uint8_t PassiveFilter; /**< ENABLED or DISABLED         */
   uint8_t Direction;     /**< OUTPUT or INPUT             */
   uint8_t State;         /**< HIGH or LOW                 */
   uint8_t Function;      /**< Mux Function                */
}DioConfig_t;
```

Listing 13-2 The structure definition for a GPIO peripheral that is populated in an array to initialize a microcontroller's pins

Despite the numerous advantages of using configuration tables, there are some disadvantages that you should also be aware of. First, configuration tables will take up additional flash space. This is because we're creating an array of structures, and every element of our structure and array will be stored in flash. Extra flash space usage isn't a big deal if you are working on a microcontroller with 128 kilobytes or more flash. However, every byte may count if you work on a resource-constrained device with only 32 kilobytes.

Another disadvantage is that the initialization functions are written to loop through the configuration table and then set the specific bits in the peripheral register. Looping through the table can burn unwanted CPU cycles compared to just setting the register to the desired value. Obviously, these extra CPU cycles are only used during start-up, and even on a slower processor running at 48 MHz, it has a minimal impact.

I've found that the benefits of using configuration tables outweigh the disadvantages. Configuration tables can be used for much more than just initializing a peripheral. Let's look at a few familiar places where they can make a big difference.

An Extensible Task Initialization Pattern

Real-time operating systems (RTOS) have steadily found their way into more and more embedded applications. I've noticed that many 32-bit applications start immediately with an RTOS rather than going the bare-metal route. Using an RTOS makes excellent sense in many systems today because it provides a real-time scheduler, synchronization tools, memory management, and much more. However, one exciting thing that I have noticed when I review designs and work with teams in the industry is that they often design their task initialization code in a way that is not very reusable.

When they want to create an RTOS task, many developers will go to their main function and follow the process necessary to complete a task for their specific RTOS. For example, if you are using FreeRTOS, you'll find a handful to several dozen code segments that look like Listing 13-3. There is nothing wrong per se with initializing a task this way, but it's not very scalable. Sometimes, finding what tasks are being created in an application can result in a witch hunt. (I've seen applications where tasks are scattered throughout, which is excellent for reuse since they are all modularized but difficult if you want to analyze the code and see how many tasks are in the system.)

```
xTaskCreate(Task_LedBlink,                  // Task Pointer
            "TaskLedBlink",                 // Task Name
            configMINIMAL_STACK_SIZE,       // Stack Depth
            (void * const) 0,               // Data to Pass
            2,                              // Task Priority
            (TaskHandle_t * const)0 );      // Task Handle
```

Listing 13-3 Applications commonly have code blocks scattered throughout main and the code base to initialize all the application tasks

The problem I have with creating a bunch of these statements to create all the application tasks is multifold. First, if I want to add a new task, I must duplicate the code that is using xTaskCreate. Chances are, I'm going to copy and paste, which means I'll likely inject a bug into the code base because I'll get distracted (squirrel!) and forget to update some parame-

ters like the task priority. Second, if I want to remove a task from the system, I must hunt down the task I'm looking for and remove the xTaskCreate code block by either deleting it or conditionally compiling it out. The final problem that I'll mention is that, from a glance, I have no idea how many tasks might be in an application. Sure, I can use grep or do a search for xTaskCreate and see how many results come up, but that feels clunky to me.

Defining a Task Configuration Structure

A more exciting solution to creating tasks is to create a configuration table that initializes tasks during start-up. The configuration table would contain all the parameters necessary to initialize a task as a single row in the table. If a developer wants to add a new task, they go to the table and add the task. If they want to delete a task, they go to the table and delete the task. If they want to know how many tasks are in the application, they go to the table and count how many rows are in it! Finally, if they want to port the code, they don't have to rewrite a bunch of task creation code. Instead, they update the table with the tasks in the system!

The implementation of the task configuration table is relatively simple. First, a developer creates a structure that contains all the parameters necessary to initialize a task. For FreeRTOS, the structure would look something like Listing 13-4. Looking at each structure member, you'll notice that they directly correspond to the parameters passed into the xTaskCreate function. You'll also notice that we use const quite liberally to ensure that the pointers can't accidentally be incremented or overwritten once they are assigned.

> **Caution** Listing 13-4 shows how the structure would look for a dynamically allocated task in FreeRTOS. Dynamic memory allocation uses malloc, which is not recommended for real-time embedded systems. An alternative approach would be to use the statically allocated API, but I will leave that as an exercise for the reader.

```
/**
 * Task configuration structure used to create a task
configuration
 * table. Note: this is for dynamic memory allocation. We
should
 * create all the tasks up front dynamically and then never
allocate
 * memory again after initialization or switch to static
allocation.
 */
typedef struct
{
    TaskFunction_t const TaskCodePtr;   // Pointer to task
function
    const char * const TaskName;        // String task name
    const configSTACK_DEPTH_TYPE StackDepth;    // Stack
depth
    void * const ParametersPtr;         // Parameter Pointer
    UBaseType_t TaskPriority;           // Task Priority
    TaskHandle_t * const TaskHandle;    // Pointer to task
handle
}TaskInitParams_t;
```

Listing 13-4 A typedef structure whose members represent the parameters necessary to initialize a task using FreeRTOS's xTaskCreate API

Different RTOSes will require different TaskInitParams_t structure definitions. For example, the structure in Listing 13-4 will only work for FreeRTOS. My structure will differ if I use Azure RTOS, formerly known as ThreadX. For example, Listing 13-5 demonstrates what the structure would look like for Azure RTOS. FreeRTOS requires six parameters to initialize a task, whereas Azure RTOS requires ten. Note, this doesn't make FreeRTOS superior! Azure RTOS provides additional features and control to the developer to fine-tune how the created task behaves. For example, you can create a task but not automatically start it based on how the TaskAutoStart parameter is configured.

Tips and Tricks When possible, it's a good idea for Arm processors to limit the number of parameters passed to a function to six. Passing more than six parameters can result in a small performance hit due to the passed parameters not all fitting in the CPU registers. If you have six or more parameters, pass a pointer instead.

```
/**
 * Task configuration structure used to create a task
configuration
 * table.
 */
typedef struct
{
    TX_THREAD * const TCB;          // Pointer to task
control block
    char * TaskName;                // String task name
    void * const TaskEntryPtr;      // Pointer to the task
function
    void * const ParametersPtr;     // Parameter Pointer
    uint8_t ** const StackStart;    // Pointer to task stack
location
    ULONG const StackSize;          // Task stack size in
bytes
    UINT TaskPriority;              // Task Priority
    UINT PremptThreshold;           // The preemption
threadshold
    ULONG TimeSlice;                // Enable or Disable Time
Slicing
    UINT  TaskAutoStart;            // Enable task
automatically?
}TaskInitParams_t;
```

Listing 13-5 A typedef structure whose members represent the parameters necessary to initialize a task using Azure RTOS's tx_thread_create API

Suppose you are working in an environment where you want to have just a single configuration structure for any RTOS you might be interested in using. In that case, you could design your configuration structure around CMSIS-RTOSv2 (or whatever the current API version is). CMSIS-RTOSv2 is designed to be an industry standard for interacting with an RTOS. Clearly, it acts like an operating system abstraction layer (OSAL) or a wrapper. The most common features of an RTOS are exposed, so there may be cool features that developers can't access, but I digress a bit. Designing a configuration structure for CMSIS-RTOSv2 using their osThreadNew API would look like Listing 13-6.

The TaskInitParams_t configuration structure looks much more

straightforward than the other two we have seen. It is simpler because osThreadNew takes a pointer to an attribute structure that contains most of the task's configuration parameters. The structure osThreadAttr_t includes members like

- Name of the thread
- Attribute bits (detachable or joined threads)
- Memory control block
- Size of the memory control block
- Memory for stack
- Size of the stack
- Thread priority
- TrustZone module identifier[1]

Personally, I prefer the method of just passing a pointer to the osThreadAttr_t variable. It's quick, efficient, readable, and configurable. It's also entirely scalable because the API doesn't need to change if the data does (again, I digress ...).

```
/**
 * Task configuration structure used to create a task
configuration
 * table.
 */
typedef struct
{
    osThreadFunc_t TaskEntryPtr;    // Pointer to the task
function
    void * const ParametersPtr;     // Parameter Pointer
    osThreadAttr_t const * const Attributes;   // Pointer to
attributes
}TaskInitParams_t;
```

Listing 13-6 A typedef structure whose members represent the parameters necessary to initialize a task using CMSIS-RTOSv2's osThreadNew API

Once you've decided how you want to define your configuration structure, we are ready to define our task configuration table.

Defining a Task Configuration Table

Now that we have defined TaskInitParams_t, we can create an array of TaskInitParams_t where each element in the array contains the parameters necessary to initialize our task. For example, let's say that I am using FreeRTOS and that my system has two tasks:

1. A blinky LED task that executes every 500 milliseconds
2. A telemetry task that runs every 100 milliseconds

I can define an array named TaskInitParameters as shown in Listing 13-7. You can easily see that each element of the array, represented by a row, contains the parameters necessary to initialize a task in FreeRTOS. Each parameter needed is aligned as a column. Quite literally, our TaskInitParameters array is a configuration table!

Looking at Listing 13-7, you might wonder where I got some of these values. For example, where is Task_Led defined? In my mind, Task_Led is a public function that manages the LED task and is defined in a LED module like task_led.h and task_led.c. The values `TASK_LED_STACK_DEPTH`, `TASK_TELEMETRY_STACK_DEPTH`, `TASK_LED_PRIORITY`, and `TASK_TELEMETRY_PRIORITY` are a different story.

```
/**
```

```
 * Task configuration table that contains all the parameters necessary to initialize
 * the system tasks.
 */
TaskInitParams_t const TaskInitParameters[] =
{
//  Task Pointer,      Task String Name,           Stack Size,          Parameter,  Task
Priority,     Task Handle
//                           Pointer
    {&TaskLed,         "Task_LedBlinky",   TASK_LED_STACK_DEPTH,          NULL,  TASK_LED_
NULL},
    {&TaskTelemetry,   "Task_Telemetry",   TASK_TELEMETRY_STACK_DEPTH,    NULL,  TASK_TELE
NULL}
};
```

Listing 13-7 A typedef structure whose members represent the parameters necessary to initialize a task using CMSI osThreadNew API

I mentioned earlier that I hate having to hunt through a project to find tasks; this includes trying to find where the task priority and memory definitions are. Typically, I will define a module named task_config.h and task_config.c. The task_config.h header file would contain the definitions for all the task configurations in the application! That's right; everything is in one place! Easy to find, easy to change.

There are several different ways that we can define those four configuration values. One way is to use macros, as shown in Listing 13-8. Using macros might appeal to you, or this may be completely horrid depending on your experience and programming philosophy. I look at it as a quick and dirty way to do it that is easy to read, although not really in line with OOP beliefs. However, when using C, it fits into the design paradigm well enough.

```
/*
 * Defines the stack depth for the tasks in the application
 */
#define TASK_LED_STACK_DEPTH          (512U)
#define TASK_TELEMETRY_STACK_DEPTH    (512U)
/*
 * Defines the stack depth for the tasks in the application
 */
#define TASK_LED_PRIORITY             (15U)
#define TASK_TELEMETRY_PRIORITY       (30U)
```

Listing 13-8 Task configuration parameters that may need adjustment can be defined in task_config.h as macros

> **Caution** FreeRTOS defines task priority based on the most significant numeric number. A task with priority 30 has a higher priority than 15. Priority setting is backward compared to most RTOSes, so be careful when setting priorities! (Or use an OSAL that manages the priority behind the scenes.)

Another way to define these configuration parameters, and one you may like a bit better, is to define an enum. First, examine Listing 13-9. You can see that we have created two different enumerations: one for the stack depth and a second for the task priorities. Notice that we must assign the value for each item in the enum, which may overlap! Same values in an enum may or may not bother your software OCD (if you have it) or your desire for perfection.

```
/*
 * Defines the stack depth for the tasks in the application
 */
enum
{
    TASK_LED_STACK_DEPTH = 512U,
    TASK_TELEMETRY_STACK_DEPTH = 512U,
};
```

```
/*
 * Defines the stack depth for the tasks in the application
 */
enum
{
    TASK_LED_PRIORITY        = 15U,
    TASK_TELEMETRY_PRIORITY  = 30U,
};
```

Listing 13-9 Defining task priorities and stack depth as enums

These are just a few C-compatible options for defining the priority and stack depth. The goal here is to keep these items grouped. For example, if I were to perform a rate monotonic analysis (RMA), I may need to adjust the priorities of my tasks. Having all the tasks' priorities, stacks, and even periodicity values in one place makes it easy to change and less error-prone.

Initializing Tasks in a Task_CreateAll Function

With the configuration table for the tasks created in the application completed, the only piece of code currently missing is a task initialization function. You can name this function however you like; however, my preference is to name it Task_CreateAll. Task_CreateAll, obviously from its name, creates all the tasks in the application. So let's look at how we can do this.

First, I usually put the Task_CreateAll function into the task_config module. I do this so that the task configuration table can be locally scoped using static and not exposed to the entire application. You could put Task_CreateAll in a tasks.c module and create an interface in task_config that returns a pointer to the task configuration table. There's a part of me that likes it, but for now, we will keep it simple for brevity's sake.

The interface and prototype for the function are then super simple:

```
void Task_CreateAll(void);
```

You could make the interface more complex using something like

```
TaskErr_t Task_CreateAll(TaskInitParams_t const * const
                                                    TaskParams);
```

I like the second declaration myself, especially the idea of returning a TaskErr_t so that the application can know if all the tasks were created successfully. There's little danger in passing a pointer to the configuration structure, which is stored in flash early in the boot cycle. The chances of memory corruption or bit flips several microseconds after start-up are highly improbable in most environments. However, to simplify the interface, you may want to just use

```
TaskErr_t Task_CreateAll(void);
```

You can decide which version works best for you. For now, we will explore the interface that I listed first, which returns void and has no parameters. (It's the simplest case, so let's start there.)

Once the interface has been defined, we can write the function, shown in detail in Listing **13-10**. The function starts by determining how many tasks are required by the application. The number of tasks in the application can be calculated by taking the size of the TaskInit Parameters array and dividing it by the size of the TaskInitParams_t structure. I've often seen developers create a macro that must be updated with the number of tasks in the system. Manual calculations are error-prone and almost an ensured way to create a bug. I prefer to let the compiler or runtime calcu-

late this for me.

After calculating the number of tasks in the table, Task_CreateAll just needs to loop through each task and call xTaskCreate using the parameters for each task. A for loop can be used to do this. Afterward, the function is complete! You can see the results in Listing **13-10**.

A quick review of Listing **13-10** will reveal a simple bit of code written once to manage the creation of all tasks in our embedded application. In this simple function, we've refactored potentially hundreds of lines of code that would usually be written to initialize a bunch of tasks throughout our application. We've also moved code that was likely scattered throughout the application into one central location. The Task_CreateAll function that we have written could use a few improvements, though.

First, the void return means that the application code cannot know whether we successfully created all the tasks. It is assumed that everything will go perfectly well. Second, our example uses the dynamic task creation method xTaskCreate. If we did not size our heap correctly, it's possible that the task creation function could fail! If xTaskCreate fails, we will never know because we are not checking the return value to see if it is pdPASS (which says the function ran successfully). When I write production code, I can generally get away with a task creation function that doesn't check the return value if we statically allocate tasks. You'll know immediately if you cannot create the task at compile time.

```
void Task_CreateAll(void)
{
    // Calculate how many rows there are in the task table. This
is
    // done by taking the size of the array and dividing it by
the
    // size of the type.
    const uint8_t TasksToCreate = sizeof(TaskInitParameters) /
                                  sizeof(TaskInitParams_t);
    // Loop through the task table and create each task.
    uint8_t TaskCount = 0;
    for(TaskCount = 0; TaskCount < TasksToCreate; TaskCount++)
    {
        xTaskCreate(TaskInitParameters[TaskCount].TaskCodePtr,
                    TaskInitParameters[TaskCount].TaskName,
                    TaskInitParameters[TaskCount].StackDepth,
                    TaskInitParameters[TaskCount].ParametersPtr,
                    TaskInitParameters[TaskCount].TaskPriority,
                    TaskInitParameters[TaskCount].TaskHandle);
    }
}
```

Listing 13-10 The Task_CreateAll function loops through the task configuration table and initializes each task in the table, creating a reusable, scalable, and configurable solution

Autogenerating Task Configuration

The coding technique we've been exploring can easily be copied and pasted into any number of projects. The problem with copy and paste is that there seems to be a sizeable statistical probability that the developer will overlook something and inject a bug into their code. Even if a bug is not injected into the code, I've found that if I copy the task_config module into new projects, I tend to comment out large blocks of code which may live that way for months which is not ideal. The optimal way to manage

the task configuration table and the function to initialize all the tasks is to autogenerate the code.

Code generation has been around in embedded systems for decades, but I've generally only seen it exercised by microcontroller vendors as part of their toolchains. For example, if you look at STM32CubeIDE, it integrates with STM32CubeMx, which allows developers to select their driver and middleware configurations. The tool then generates the source code based on those settings. Of course, developing such tools is time-consuming and expensive. However, you can still take advantage of these techniques on a much more limited basis to improve the configurability of your projects. At a minimum, a code generator will contain four main components: a configuration, templates, the generator, and the output code module. These four components and their interactions can be seen in Figure 13-1.

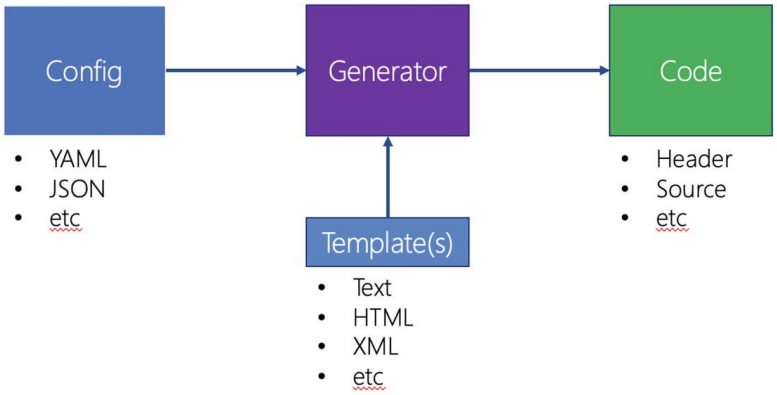

Figure 13-1 The high-level process for creating autogenerated configuration modules

The configuration file stores all the configuration information to generate the code. For example, our task code configuration file would contain information such as the task and all the parameters necessary to configure that task. (We will look more closely at how these files look in the next section.) Several different formats are available to use, such as YAML or JSON. (These formats are even interchangeable using JSON to YAML and YAML to JSON converters.)

Templates are used to create easily updatable files with replaceable text to create the output module. For example, if you have a template to organize how a C header and source file are organized, these could be used as templates. Templates could be text files, an HTML file, XML, etc. Each file type has its advantages and disadvantages, depending on the end goal. We're looking to generate C modules, so plain text files are sufficient.

The generator takes in the configuration file(s) and template(s) and then generates the final code module(s). The generator can be written in nearly any language, such as C++, Java, or Python. However, my personal preference is to write it in Python because almost everyone these days knows Python. If you don't, there are so many online resources that nearly any coding problem can be solved with a quick web search.

At the end of the day, we're looking to generate a C header and source code in the most scalable way possible. In addition, we want to minimize the number of errors generated and generate clean and human-readable code. To understand the entire process, let's build a simple code generator in Python for our task configuration module.

YAML Configuration Files

YAML, Yet Another Mark-up Language, is a file format that lends itself well to configuration files. YAML is minimalistic and follows the Python style of indentation. YAML only allows spaces to be used for formatting and can be used to create lists and maps of data in nearly any data type that a software engineer may want to use. Each YAML file starts with a "---" to denote the start of a YAML document. Each file may have more than a single document included.

For our task configuration module example, we need the YAML file to store the parameter information required to initialize a task. We can do this by creating a dictionary of X tasks where we include a key and a value for the configuration parameter we want to configure. For example, suppose I had a task named "Task Telemetry" with a periodicity of 100 milliseconds, a stack size of 4096, and a priority of 22. In that case, I might save the configuration in a YAML file that looks something like Listing 13-11.

```
---
 Task1:
   TaskName: "Task_Telemetry"
   TaskEntryPtr: "Task_Telemetry"
   PeriodicityInMS: 100
   ParametersPtr: 0
   StackSize: 4096
   TaskPriority: 22
   TaskHandle: "NULL"
```

Listing 13-11 The YAML configuration for a task named Task_Telemetry. All the details necessary to initialize the task are included

The great thing about a YAML configuration file is that if I need to add or remove tasks, I don't have to modify my code! All I need to do is add the new task to the configuration and run my Python generator! For example, if I wanted to add two more tasks to my configuration, my YAML file might become something like Listing 13-12.

```
---
 Task1:
   TaskName: "Task_Telemetry"
   TaskEntryPtr: "Task_Telemetry"
   PeriodicityInMS: 100
   ParametersPtr: 0
   StackSize: 4096
   TaskPriority: 22
   TaskHandle: "NULL"
 Task2:
   TaskName: "Task_RxMessaging"
   TaskEntryPtr: "Task_RxMessaging"
   PeriodicityInMS: 0
   ParametersPtr: 0
   StackSize: 4096
   TaskPriority: 1
   TaskHandle: "NULL"
 Task3:
   TaskName: "Task_LedBlinky"
   TaskEntryPtr: "Task_LedBlinky"
   PeriodicityInMS: 500
   ParametersPtr: 0
   StackSize: 256
   TaskPriority: 5
   TaskHandle: "NULL"
```

Listing 13-12 The YAML configuration file now includes three tasks: Task_Telemetry, Task_RxMessaging, and Task_LEDBlinky

YAML files are a fantastic option for storing configuration data. Configuration files also provide you with a flexible way of configuring source code for different projects and product SKUs without having to maintain "extra" code in your source repository. Instead, it's all one common code base, and the build system chooses which configuration to generate, build, and deploy.

Creating Source File Templates

Templates provide a mechanism from which you can outline how a header and source module will be organized once the final code modules are generated. Templates can be generated in several different ways. For example, a developer could build a template in XML, text, or even an HTML file. The format that we choose depends on the result. An HTML template might make the most sense if you were to autogenerate a report. For C/C++ modules, it's far easier to just use a simple text file.

To generate a module, there are two file templates that we would want to use: a header template file, t_header.h, and a source header file, t_source.c. These files would contain all the typical comments that one would expect in a header and outline the structure of the files. However, the main difference between the template and the final file is that instead of containing all the code, the template files have tags that are replaced by the desired code blocks.

Tags are special text that the Python template library can detect and use for string replacements. For example, you might in your template files have a tag like ${author} which is where the name of the author is supposed to go. Our Python code will specify that the author is Jacob Beningo and then search out all the `${author}` tags and replace it with Jacob Beningo. In the header file, we would include tags such as `${author}`, `${date}`, `${task_periodicity}`, `${task_priorities}`, and `${task_stacks}`. Listing 13-13 shows how an example header file template looks.

The source module template is very similar to the header source template, except for having different tags. The source module has tags for `${author}`, `${date}`, `${includes}`, and `${elements}`. The header files point to the individual task modules that a developer would write, and the elements are our configuration table. The source module template also includes the Task_CreateAll function that allows us to initialize all the tasks in the configuration table. Listing 13-14 shows how an example source file template looks. Note: I would normally also include sections for Configuration Constants, Macros, Typedefs, and Variables for each template, but I've omitted them to try and get the code to fit on as few pages as possible.

```
/*************************************************************
* Title                 :   Task Configuration
* Filename              :   task_config.h
* Author                :   ${author}
* Origin Date           :   ${date}
*************************************************************/
/** @file task_config.h
 *  @brief This module contains task configuration parameters.
 *
 *  This is the header file for adjusting task configuration
 *  such as task priorities, delay times, etc.
```

```
 */
#ifndef TASK_CONFIG_H_
#define TASK_CONFIG_H_
/*****************************************************************
* Includes
*****************************************************************/
/*****************************************************************
* Preprocessor Constants
*****************************************************************/
${task_periodicity}
${task_priorities}
${task_stacks}
/*****************************************************************
* Function Prototypes
*****************************************************************/
#ifdef __cplusplus
extern "C"{
#endif
void Task_CreateAll(void);
#ifdef __cplusplus
} // extern "C"
#endif
#endif /*TASK_CONFIG_H_*/
/*** End of File **************************************************/
```

Listing 13-13 An example template header file for task_config.h with string replacement tags

```
/*****************************************************************
* Title                 :   Task Configuration
* Filename              :   task_config.c
* Author                :   ${author}
* Origin Date           :   ${date}
* See disclaimer in source code
*****************************************************************/
/*****************************************************************
* Doxygen C Template
* Copyright (c) 2013 - Jacob Beningo - All Rights Reserved
*****************************************************************/
/** @file task_config.c
 *  @brief This module contains the task configuration and
 *  initialization code.
 */
/*****************************************************************
* Includes
*****************************************************************/
#include "task_config.h"       // For this modules definitions
#include "FreeRTOS.h"          // For xTaskCreate
${includes}
/*****************************************************************
* Module Preprocessor Macros, Typedefs, etc
*****************************************************************/
/**
 * Task configuration structure used to create a task
 * configuration table. Note: this is for dynamic memory
 * allocation. We create all the tasks up front dynamically
 * and then never allocate memory again after initialization.
 */
typedef struct
{
    TaskFunction_t const TaskCodePtr;
    const char * const TaskName;
    const configSTACK_DEPTH_TYPE StackDepth;
    void * const ParametersPtr;
```

```
    UBaseType_t TaskPriority;
    TaskHandle_t * const TaskHandle;
}TaskInitParams_t;
/*************************************************************
* Module Variable Definitions
*************************************************************/
/**
 * Task configuration table that contains all the parameters
 * necessary to initialize the system tasks.
 */
TaskInitParams_t const TaskInitParameters[] =
{
//    Pointer to Task,   Task String Name,  Stack Size,  Param
Ptr, Priority, Handle
${elements}
};
/*************************************************************
* Function Definitions
*************************************************************/
/*************************************************************
* Function : Task_CreateAll()
*//**
* @section Description Description:
*
*  This function initializes the telemetry mutex and then
*  creates all the application tasks.
*
* @param              None.
* @return             None.
*
* @section Example Example:
* @code
*    Task_CreateAll();
* @endcode
*************************************************************/
void Task_CreateAll(void)
{
    // Calculate how many rows there are in the task table.
    // This is done by taking the size of the array and
    // dividing it by the size of the type.
    const uint8_t TasksToCreate = sizeof(TaskInitParameters) /
                                  sizeof(TaskInitParams_t);
    // Loop through the task table and create each task.
    for(uint8_t TaskCount = 0; TaskCount < TasksToCreate;
TaskCount++)
    {
        xTaskCreate(TaskInitParameters[TaskCount].TaskCodePtr,
                    TaskInitParameters[TaskCount].TaskName,
                    TaskInitParameters[TaskCount].StackDepth,
                    TaskInitParameters[TaskCount].ParametersPtr,
                    TaskInitParameters[TaskCount].TaskPriority,
                    TaskInitParameters[TaskCount].TaskHandle);
    }
}
/*************** END OF FUNCTIONS ***************************/
```

Listing 13-14 An example template source file for task_config.c with string replacement tags

Generating Code Using Python

The Python code we are about to read in our template and configuration

files is written in Python 3.9. Hopefully, it will continue to be compatible with future versions, but we all know that Python changes quite rapidly, and nothing is sacred during those changes. I would also like to point out that the code is literally written in script form and could be refactored and improved dramatically. The tools I've developed to manage a lot of my project configuration would be difficult to include and take up far more pages than any reader probably wants to work through. So, we will look at a simplified version that gets the trick done and demonstrates how this technique can be used.

The Python code aims to read the YAML configuration file and generate strings that will replace the tags in our template files. Those generated strings will complete our C module and should compile and drop into any project using FreeRTOS. A developer should only need to include "task_config.h" and then call Task_CreateAll. (Of course, you must also write the actual task functions we are trying to create, too!)

Figure 13-2 shows the high-level flowchart for the Python script.

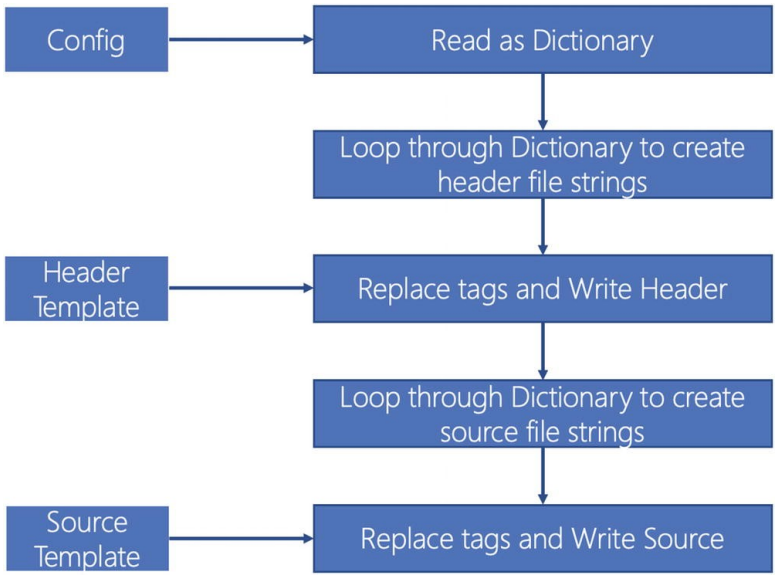

Figure 13-2 The high-level flowchart for the Python code generator

The YAML files are pretty easy to read and manipulate in Python. There is a library called PyYAML[2] that has all the methods necessary to use them. The script reads in the entire YAML file and stores it as a dictionary. The dictionary can then be looped through for each task, and the configuration data read and converted into a string that will replace a tag in the template files.

The script's setup and header portion can be seen in Listing 13-15. The source portion of the script can be seen in Listing 13-16. I'm assuming you understand the basics of Python, but just in case, I'll describe a few high-level techniques that are being used here.

```
import yaml
import string
from datetime import date
today = date.today()
Author = "Jacob Beningo"
# mm/dd/YY
Date = today.strftime("%m/%d/%Y")
if __name__ == '__main__':
```

```
    stream = open("tasksFRTOS.yaml", 'r')
    dictionary = yaml.safe_load(stream)
    #
    # Prepare the header file first
    #
    # Create a string with the task periodicity macro
    # definitions
    periodicityString = ""
    prioritiesString = ""
    stacksString = ""
    LastTaskName = ""
    for task in dictionary:
        # get the data to build the period macros
        taskData = dictionary.get(task)
        periodicity = taskData.get('PeriodicityInMS')
        name = taskData.get('TaskName')
        if periodicity != 0:
            tempString = f"#define {name.upper()}_PERIOD_MS"
            temp1String = f"({str(periodicity)}U)\n"
            periodicityString +=
                                f'{tempString:<50}{temp1String:>10}'
        # get the data to build the priorities string
        priority = taskData.get('TaskPriority')
        tempString = f"#define {name.upper()}_PRIORITY"
        temp1String = f"({str(priority)}U)\n"
        prioritiesString +=
                           f'{tempString:<50}{temp1String:>10}'
        # get the data to build the stack string
        stackSize = taskData.get('StackSize')
        tempString = f"#define {name.upper()}_STACK_SIZE"
        temp1String = f"({str(stackSize)}U)\n"
        stacksString += f'{tempString:<50}{temp1String:>10}'
        LastTaskName = str(name)
    # Read in the task.h template file
    with open("t_task.h") as t_h:
        template = string.Template(t_h.read())
    t_h.close()
    # Substitute the new strings with the template tags
    header_output = template.substitute(author=Author,
                                        date=Date,
                                        task_periodicity=periodicityString,
                                        task_priorities=prioritiesString,
                                        task_stacks=stacksString
                                       )
    with open("task_config.h", "w") as output:
        output.write(header_output)
    output.close()
```

Listing 13-15 Python code to generate the header file

```
    #
    # Prepare the C file
    #
    includeString = ""
    content = ""
    for task in dictionary:
        taskData = dictionary.get(task)
        TaskName = taskData.get('TaskName')
        includeString += '#include "' + TaskName + '.h"\n'
        EntryFunction = '    { &' + \
                                    taskData.get('TaskEntryPtr')
+','
        TaskNameStr = '"'+str(taskData.get('TaskName')) + '",'
        StackSize = str(taskData.get('StackSize')) + ","
```

```
        Parameters = str(taskData.get('ParametersPtr')) + ","
        Priority = str(taskData.get('TaskPriority')) + ","
        Handle = str(taskData.get('TaskHandle')) + "}"
        content += f'{EntryFunction:<25}{TaskNameStr:
<21}{StackSize:^8}
                    {Parameters:^14}{Priority:^8}{Handle:>8}'
        if TaskName != LastTaskName:
            content += '\n'
    # Read in the task.c template file
    with open("t_task.c") as t_c:
        template = string.Template(t_c.read())
    t_c.close()
   # Substitute the new strings with the template tags
    source_output = template.substitute(author=Author,
                                        date=Date,
                                              includes=includeString,
                                        elements=content,
                                        )
    with open("task_config.c", "w") as output:
        output.write(source_output)
    output.close()
```

Listing 13-16 Python code to generate the source file

First, the script uses Python f strings to format the strings. You'll notice statements like:

```
f'{tempString:<50}{temp1String:>10}'
```

This is an f string that contains two variables tempString and temp1String. tempString is left justified with 50 spaces, and temp1String is right justified with ten spaces. (Yes, some assumptions are built into those numbers as to what will allow the configuration values to appear properly in columns.)

Next, you'll notice that several statements use the Python template library, such as

```
template = string.Template(t_c.read())
```

and

```
source_output = template.substitute(…)
```

The substitute method is the one doing all the heavy lifting! Substitute is replacing the tags in our template files with the string we generate from looping through the configuration dictionary.

Finally, after looking through all this code, you might wonder how to use it. I've written this example to not rely on having to pass parameters into the command line or anything fun like that. To run the script, all you would need to do is use the following command from your terminal:

```
python3 generate.py
```

I use a Mac with Python 2.7 and Python 3.9 installed, so I must specify which version I want to use, which is why I use python3 instead of just python. If you run the script, you'll discover that the header file output will look like Listing **13-17**, and the source file output will look like Listing **13-18**. Notice that the tags are no longer in the files and have now been replaced with the generated strings. You'll also notice in the header file that macros for the PERIOD_MS are only generated for two tasks! This is because Task_RxMessaging is event driven and has a period of zero milliseconds.

```
/*************************************************************
* Title                 :   Task Configuration
* Filename              :   task_config.h
* Author                :   Jacob Beningo
* Origin Date           :   04/16/2022
*************************************************************/
/** @file task_config.h
```

```
 *  @brief This module contains task configuration parameters.
 *
 *  This is the header file for adjusting task configuration
 *  such as task priorities, delay times, etc.
 */
#ifndef TASK_CONFIG_H_
#define TASK_CONFIG_H_
/*****************************************************************
* Includes
*****************************************************************/
/*****************************************************************
* Preprocessor Constants
*****************************************************************/
#define TASK_TELEMETRY_PERIOD_MS                    (100U)
#define TASK_LEDBLINKY_PERIOD_MS                    (500U)
#define TASK_TELEMETRY_PRIORITY                      (22U)
#define TASK_RXMESSAGING_PRIORITY                     (1U)
#define TASK_LEDBLINKY_PRIORITY                       (5U)
#define TASK_TELEMETRY_STACK_SIZE                  (4096U)
#define TASK_RXMESSAGING_STACK_SIZE                (4096U)
#define TASK_LEDBLINKY_STACK_SIZE                   (256U)
/*****************************************************************
* Configuration Constants, Macros, Typdefs, and Variables
*****************************************************************/
/*****************************************************************
* Function Prototypes
*****************************************************************/
#ifdef __cplusplus
extern "C"{
#endif
void Task_CreateAll(void);
#ifdef __cplusplus
} // extern "C"
#endif
#endif /*TASK_CONFIG_H_*/
/*** End of File ***************************************************/
```

Listing 13-17 The header file output from the generate.py script

```
/*****************************************************************
* Title                 :   Task Configuration
* Filename              :   task_config.c
* Author                :   Jacob Beningo
* Origin Date           :   04/16/2022
* See disclaimer in source code
*****************************************************************/
/** @file task_config.c
 *  @brief This module contains the task configuration and
initialization code.
 */
/*****************************************************************
* Includes
*****************************************************************/
#include "task_config.h"
#include "FreeRTOS.h"
#include "Task_Telemetry.h"
#include "Task_RxMessaging.h"
#include "Task_LedBlinky.h"
/*****************************************************************
* Module Preprocessor Constants, Macros, etc
*****************************************************************/
/*****************************************************************
* Module Typedefs
*****************************************************************/
```

```
/**
 * Task configuration structure used to create a task
 * configuration table. Note: this is for dynamic memory
 * allocation. We create all the tasks up front dynamically
 * and then never allocate memory again after initialization.
*/
typedef struct
{
    TaskFunction_t const TaskCodePtr;
    const char * const TaskName;
    const configSTACK_DEPTH_TYPE StackDepth;
    void * const ParametersPtr;
    UBaseType_t TaskPriority;
    TaskHandle_t * const TaskHandle;
}TaskInitParams_t;
/*************************************************************
* Module Variable Definitions
*************************************************************/
/**
 * Task configuration table that contains all the parameters
 * necessary to initialize the system tasks.
 */
TaskInitParams_t const TaskInitParameters[] =
{
 {&Task_Telemetry, "Task_Telemetry",   4096, 0,22,      NULL}
 {&Task_RxMessaging,"Task_RxMessaging",4096, 0, 1,      NULL}
 {&Task_LedBlinky, "Task_LedBlinky",    256, 0, 5,      NULL}
};
/*************************************************************
* Function Prototypes
*************************************************************/
/*************************************************************
* Function Definitions
*************************************************************/
/*************************************************************
* Function : Task_CreateAll()
*//**
* @section Description Description:
*
*  This function initializes the telemetry mutex and then
* creates all the application tasks.
*
* @param              None.
*
* @return             None.
*
* @section Example Example:
* @code
*    Task_CreateAll();
* @endcode
*
* @see
*************************************************************/
void Task_CreateAll(void)
{
    // Calculate how many rows there are in the task table.
    // This is done by taking the size of the array and
    // dividing it by the size of the type.
    const uint8_t TasksToCreate = sizeof(TaskInitParameters) /
                                  sizeof(TaskInitParams_t);
    // Loop through the task table and create each task.
    for(uint8_t TaskCount = 0; TaskCount < TasksToCreate;
```

```
TaskCount++)
    {
        xTaskCreate(TaskInitParameters[TaskCount].TaskCodePtr,
                    TaskInitParameters[TaskCount].TaskName,
                    TaskInitParameters[TaskCount].StackDepth,
                    TaskInitParameters[TaskCount].ParametersPtr,
                    TaskInitParameters[TaskCount].TaskPriority,
                    TaskInitParameters[TaskCount].TaskHandle);
    }
}
/*************** END OF FUNCTIONS **************************/
```

Listing 13-18 The source file output from the generate.py script

Final Thoughts

The ability to create configuration tables is a powerful tool for embedded developers. The ability to automatically generate those files from configuration files is even more powerful. The scalability, configurability, and reusability of embedded software can grow exponentially using the techniques we've looked at in this chapter. The question becomes whether you'll continue to hand-code configurable elements of your software or join the modern era and start writing generated, configurable code.

Action Items
To put this chapter's concepts into action, here are a few activities the reader can perform to start using configurable firmware techniques:

- Review your software and identify areas where configuration tables could be used to improve your code's reuse, configurability, and scalability.
- For your RTOS or CMSIS-RTOSv2, create a task_config module. You should include
  - A Task_CreateAll function
  - A TaskInitParams_t structure
  - A TaskInitParamt_t configuration table
- Download the example Python script files from Jacob's GitHub account at **https://github.com/JacobBeningo/Beningo**. Then, run the example to see the default configuration and output.
  - Add a new custom task of your choosing and rerun the script. How did the code change?
  - Change the priorities on several tasks. Rerun the script. How did the code change?
- Modify the Python script to autogenerate the code you created for your RTOS.
- Carefully consider where else you can leverage configurable firmware techniques in your code base.
- If you are serious about building a configurable code based on the techniques in this chapter, consider reaching out to Jacob at jacob@beningo.com for additional assistance.

Footnotes

1 Yes! The interface accounts for Arm's TrustZone thread context management, but we shouldn't be surprised because CMSIS was designed by Arm. Still a useful feature though.

https://pyyaml.org/
