J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_4

4. RTOS Application Design

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

The original version of this chapter was revised. A correction to this chapter is available at https://doi.org/10.1007/978-1-4842-8279-3_17

Embedded software has steadily become more and more complex. As businesses focus on joining the IoT, the need for an operating system to manage low-level hardware, memory, and time has steadily increased. Embedded systems implement a real-time operating system in approximately 65% of systems.[1] The remaining systems are simple enough for bare-metal scheduling techniques to achieve the systems requirements.

Real-time systems require the correctness of the computations, the computation's logical correctness, and timely responses.[2] There are many scheduling algorithms that developers can use to get real-time responses, such as

- Run to completion schedulers
- Round-robin schedulers
- Time slicing
- Priority-based scheduling

Definition A real-time operating system (RTOS) is an operating system designed to manage hardware resources of an embedded system with very precise timing and a high degree of reliability.[3]

It is not uncommon for a real-time operating system to allow developers to simultaneously access several of these algorithms to provide flexible scheduling options.

RTOSes are much more compact than general-purpose operating systems like Android or Windows, which can require gigabytes of storage space to hold the operating system. A good RTOS typically requires a few kilobytes of storage space, depending on the specific application needs. (Many RTOSes are configurable, and the exact settings determine how large the build gets.)

An RTOS provides developers with several key capabilities that can be time-consuming and costly to develop and test from scratch. For example, an RTOS will provide

- A multithreading environment
- At least one scheduling algorithm
- Mutexes, semaphores, queues, and event flags
- Middleware components (generally optional)

The RTOS typically fits into the software stack above the board support package but under the application code, as shown in Figure 4-1. Thus, the RTOS is typically considered part of the middleware, even though middleware components may be called directly by the RTOS.

While an RTOS can provide developers with a great starting point and several tools to jump-start development, designing an RTOS-based application can be challenging the first few times they use an RTOS. There are common questions that developers encounter, such as

- How do I figure out how many tasks to have in my application?
- How much should a single task do?
- Can I have too many tasks?
- How do I set my task priorities?

This chapter will explore the answers to these questions by looking at how we design an application that uses an RTOS. However, before we dig into the design, we first need to examine the similarities and differences between tasks, threads, and processes.

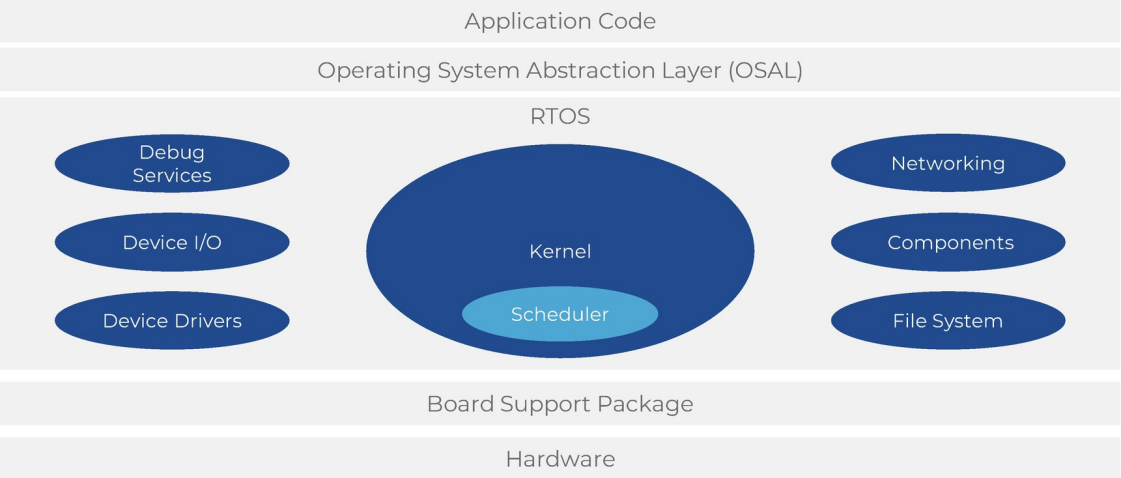

Figure 4-1 An RTOS is a middleware library that, at a minimum, includes a scheduler and a kernel. In addition, an RTOS will typically also provide additional stacks such as networking stacks, device input/output, and debug services

Tasks, Threads, and Processes

An RTOS application is typically broken up into tasks, threads, and processes. These are the primary building blocks available to developers; therefore, we must understand their differences.

A task has several definitions that are worth discussing. First, a task is a concurrent and independent program that competes for execution time on a CPU.[4] This definition tells us that tasks are isolated applications without interactions with other tasks in the system but may compete with them for CPU time. They also need to appear like they are the only program running on the processor. This definition is helpful, but it doesn't represent what a task is on an embedded system.

The second definition, I think, is a bit more accurate. A task is a semi-independent portion of the application that carries out a specific duty.[5] This definition of a task fits well. From it, we can gather that there are several characteristics we can expect from a task:

- It is a separate "program."
- It may interact with other tasks (programs) running on the system.
- It has a dedicated function or purpose.

DefinitionA task is a semi-independent portion of the application that car-

ries out a specific duty.[5]

This definition fits well with what we expect a task to be in a microcontroller-based embedded system. Surveying several different RTOSes available in the wild, you'll find that there are several that provide task APIs, such as FreeRTOS and uC OS II/III.

On the other hand, a thread is a semi-independent program segment that executes within a process.[6] From it, we can gather that there are several characteristics we can expect from a thread:

- First, it is a separate "program."
- It may interact with other tasks (programs) running on the system.
- It has a dedicated function or purpose.

For most developers working with an RTOS, a thread and a task are synonyms! Surveying several different RTOSes available in the wild, you'll find that there are several that provide thread APIs, such as Azure RTOS, Keil RTX, and Zephyr. These operating systems provide similar capabilities that compete with RTOSes that use task terminology.

A **process** is a collection of tasks or threads and associated memory that runs in an independent memory location.[7] A process will often leverage a memory protection unit (MPU) to collect the various elements part of the process. These elements can consist of

- Flash memory locations that contain executable instructions or data
- RAM locations that include executable instructions or data
- Peripheral memory locations
- Shared RAM, where data is stored for interprocess communication

A process groups resources in a system that work together to achieve the application's goal. Processes have the added benefits of improving application robustness and security because it limits what each process can access. A typical multithreaded application has all the application tasks, input/output, interrupts, and other RTOS objects in a single address space, as shown in Figure 4-2.

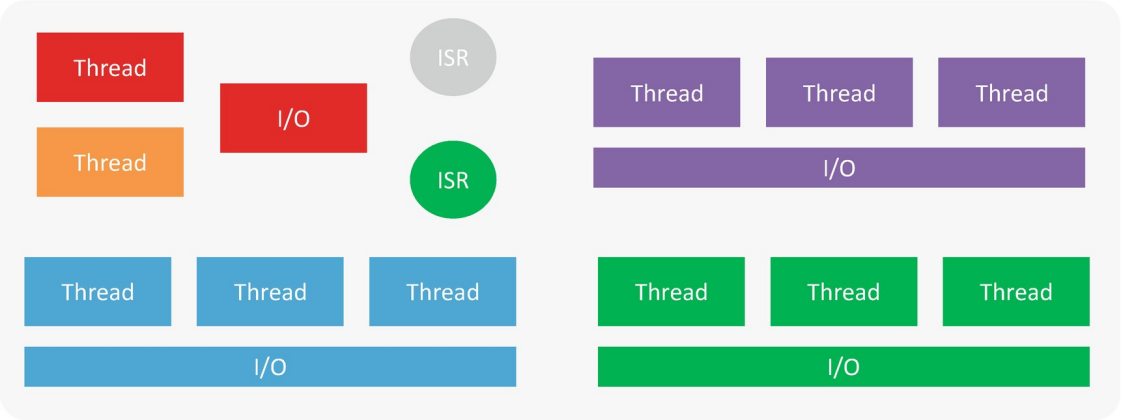

Figure 4-2 A typical multithreaded application has all the application tasks, input/output, interrupts, and other RTOS objects in a single address space[8]

This approach is acceptable and used in many applications, but it does not isolate or protect the various elements. For example, if one of the green threads goes off the rails and starts overwriting memory used by the blue thread, there is nothing in place to detect or protect the blue thread. Furthermore, since everything is in one memory space, everyone can access everyone else's data! Obviously, this may be unwanted behavior in many applications, which is why the MPU could be used to create processes, resulting in multiprocess applications like that shown in Figure 4-3.

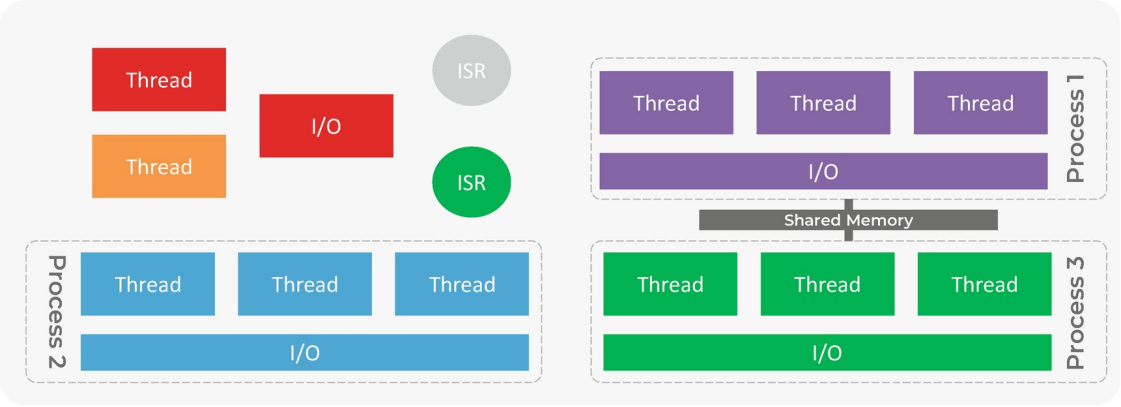

Figure 4-3 A multiprocess application can leverage an MPU to group threads, input/output, interrupts, and other RTOS objects into multiple, independent address spaces[9]

Now that we understand the differences between tasks, threads, and processes, let's examine how we can decompose an RTOS application into tasks and processes.

Task Decomposition Techniques

The question I'm asked the most by developers attending my real-time op-

erating systems courses is, “How do I break my application up into tasks?”. At first glance, one might think breaking up an application into semi-independent programs would be straightforward. The problem is that there are nearly infinite ways that a program can be broken up, but not all of them will be efficient or result in good software architecture. We will talk about two primary task decomposition techniques: feature-based and the outside-in approach.

Feature-Based Decomposition

Feature-based decomposition is the process of breaking an application into tasks based on the application features. A feature is a unique property of the system or application.[10] For example, the display or the touch screen would be a feature of an IoT thermostat. However, these could very quickly be their own task within the application.

Decomposing an application based on features is a straightforward process. A developer can start by simply listing the features in their application. For an IoT thermostat, I might make a list something like the following:

- Display
- Touch screen
- LED backlight
- Cloud connectivity
- Temperature measurement
- Humidity measurement
- HVAC controller

Most teams will create a list of features the system must support when they develop their stakeholder diagrams and identify the system requirements. This effort can also be used to determine the tasks that make up the application software.

Feature-based task decomposition can be very useful, but sometimes it can result in an overly complex system. For example, if we create tasks based on all the system features, it would not be uncommon to identify upward of a hundred tasks in the system quickly! This isn’t necessarily

wrong, but it could result in an overly complex system with more memory and RAM than required.

When using the feature-based approach, it's critical that developers also go through an optimization phase to see where identified tasks can be combined based on common functionality. For example, tasks may be specified for measuring temperature, pressure, humidity, etc. However, having a task for each individual task will overcomplicate the design. Instead, these measurements could all be combined into a sensor task. Figure 4-4 provides an example of a system's appearance when feature-based task decomposition is used.

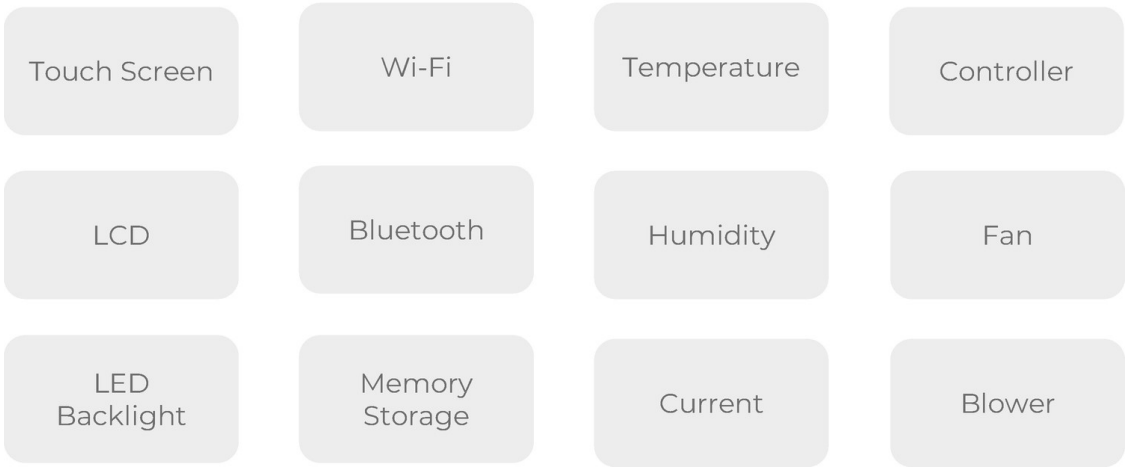

Figure 4-4 An example is feature-based task decomposition for an IoT thermostat that shows all the tasks in the software

Using features is not the only way to decompose tasks. One of my favorite methods to use is the outside-in approach.

The Outside-In Approach to Task Decomposition

The outside-in approach[11] to task decomposition is my preferred technique for decomposing an application. The primary reason is that the approach is data-centric and adheres to the design philosophy principles identified in Chapter 1. Thus, rather than looking at the application features, which will change and morph over time, we focus on the data and how it flows through the application. Let's now look at the steps necessary to apply this approach and a simple example for an IoT-based device.

The Seven-Step Process

Developers can follow a simple process to decompose their applications using the outside-in approach. The outside-in process helps developers think through the application, examine the data elements, and ensure a consistent process for developing applications. The steps are relatively straightforward and include

1. Identify the major components
2. Draw a high-level block diagram
3. Label the inputs
4. Label the outputs
5. Identify first-tier tasks
6. Determine concurrency, dependencies, and data flow
7. Identify second-tier tasks

The easiest way to examine and explain each step is to look at an example. One of my favorite teaching examples is to use an Internet-connected thermostat. So let's discuss how we can decompose an IoT thermostat into tasks.

Decomposing an IoT Thermostat

I like to use an IoT thermostat as an example because they can be sophisticated devices and they have become nearly ubiquitous in the market. An IoT thermostat has

- A wide range of sensors.
- Sensors with various filtering and sample requirements.
- Power management requirements, including battery backup management.
- Connectivity stacks like Wi-Fi and Bluetooth for reporting and configuration management.
- Similarities to many IoT control and connectivity applications make the example scalability to many industries.

There are also usually development boards available that contain many sensors that can be used to create examples.

Like any design, we need to understand the hardware before designing the software architecture. Figure 4-5 shows an example thermostat hardware block

diagram. We can use it to follow then the steps we have defined to decompose our application into tasks.

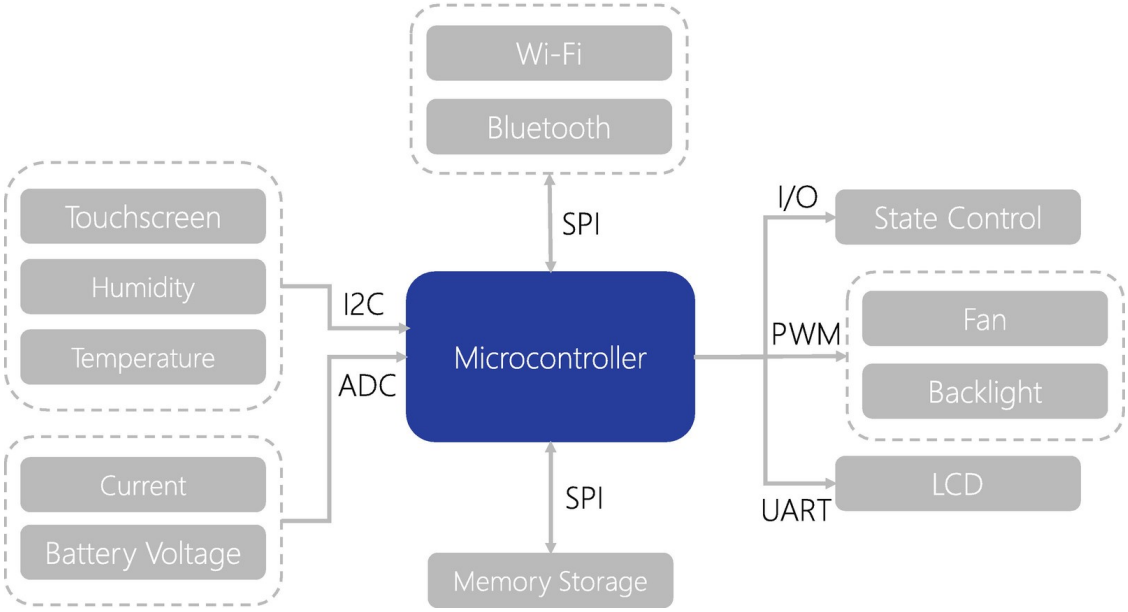

Figure 4-5 The hardware block diagram for an IoT thermostat. Devices are grouped based on the hardware interface used to interface with them

Step #1 – Identify the Major Components

The first step to decomposing an application into tasks is identifying the major components that make up the system. These are going to be components that influence the software system. For example, the IoT thermostat would have major components such as

- Humidity/temperature sensor
- Gesture sensor
- Touch screen
- Analog sensors
- Connectivity devices (Wi-Fi/Bluetooth)
- LCD/display
- Fan/motor control
- Backlight
- Etc.

I find it helpful to review the schematics and hardware block diagrams to identify the major components contributing to the software architecture. This helps me start thinking about how the system might be broken up,

but at a minimum, it lets me understand the bigger picture.

One component that I find people often overlook is the device itself! Make sure you add that to the list as well. For example, add an IoT device to your list if you are building an IoT device. If you are making a propulsion controller, add a propulsion controller. The reason to do this is that this block acts as a placeholder for our second-tier tasks that we will decompose in the last step.

Step #2 – Draw a High-Level Block Diagram

At this point, we now know the major components that we will be trying to integrate into our software architecture. We don't know what tasks we will have yet, though. That will depend on how these different components interact with each other and how they produce data. So, the next logical step is to take our list of components and build a block diagram with them. A block diagram will help us visualize the system, which I find helpful. We could, in theory, create a table for the decomposition steps, but I've always found that a picture is worth a thousand words and much more succinctly gets the point across.

Figure 4-6 demonstrates how one might start to develop a block diagram for the IoT thermostat. Notice that I've grouped the analog and digital sensors into a single sensor block to simplify the diagram and grouped them with the fan/state control components.

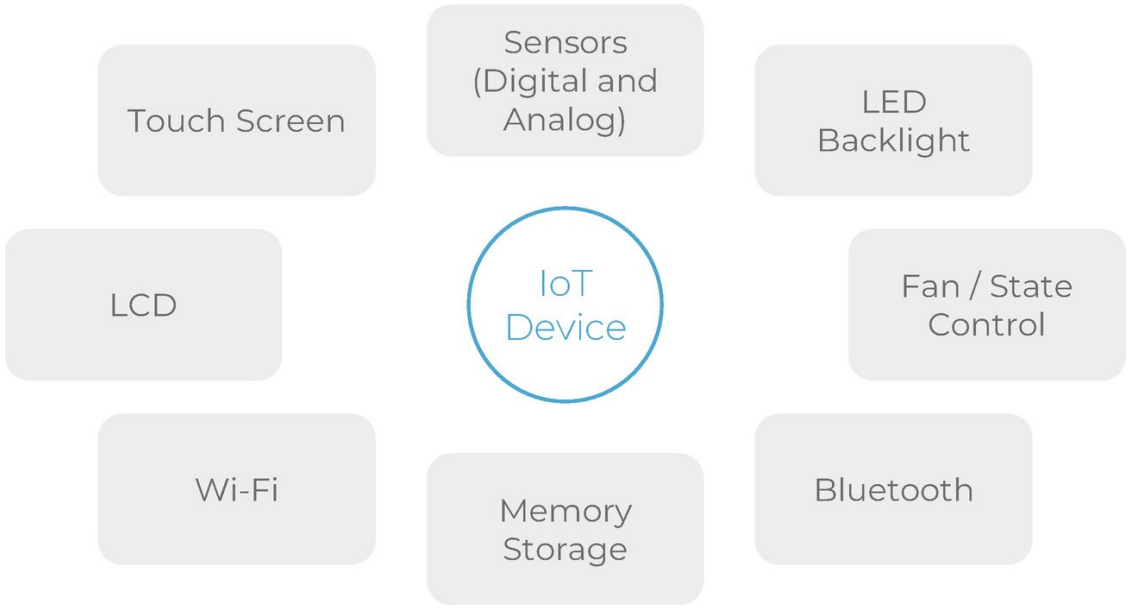

Figure 4-6 The major components are arranged in a way that allows us to visualize the system

Step #3 – Label the Inputs

We don't want to lose sight of our design principles when designing embedded software. We've discussed how important it is to let data dictate the design. When we are working on decomposing our application into tasks, the story is not any different. Therefore, we want to examine each major component and identify which blocks generate input data into the IoT device block, as shown in Figure 4-7.

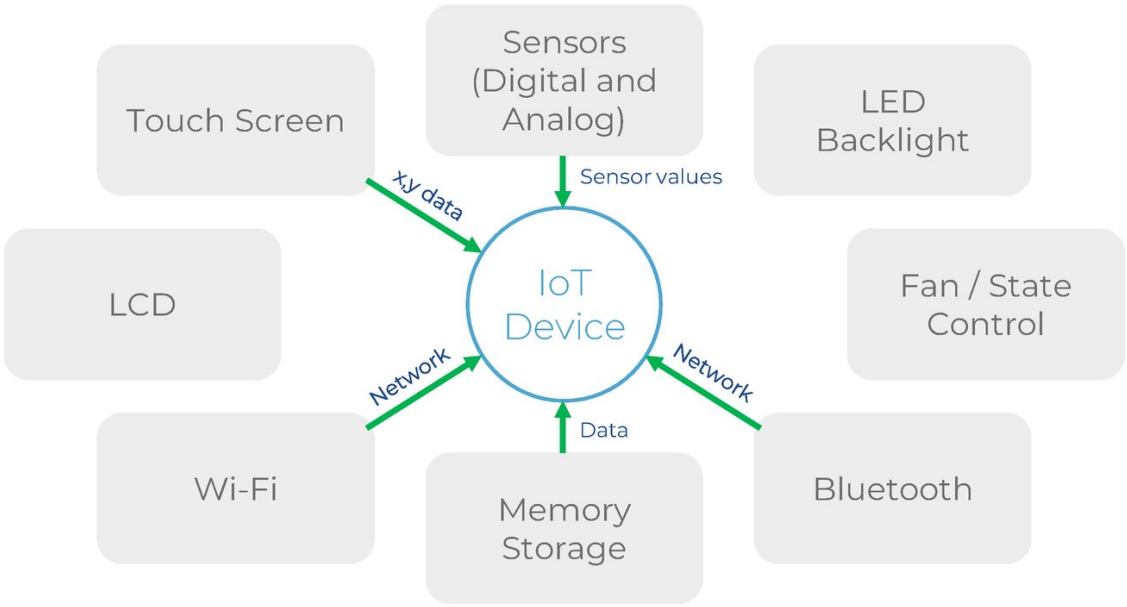

Figure 4-7 In step #3, we identify where input data comes into the system. These are the data sources for the application

In this step, we examine each block and determine what the output from the block is and the input into the application. For example, the touch screen in our application will likely generate an event with x and y coordinate data when someone presses the touch screen. In addition, the Wi-Fi and Bluetooth blocks will generate network data, which we are leaving undefined now. Finally, the sensor block we mark as generating sensor values.

At this stage, I've left the data input into the IoT device block as descriptive. However, it would not hurt to add additional detail if it is known. For example, I might know that the touch screen data will have the following characteristics:

- X and y coordinates.
- Event driven (data only present on touch).

- Data generation is every 50 milliseconds when events occur.
- Coordinate data may be used to detect gestures.

It would be a good idea to either list this information on the diagram or, better yet, add a reference number that can then be looked up in a table with the additional information about the data.

This additional information will help understand the rates at which each task will need to run and even how much work each task must do. In addition, the details will be used later when discussing how to set task priorities and perform a rate monotonic analysis.

Step #4 – Label the Outputs

After identifying all the inputs into the IoT device block, we naturally want to look at all the outputs. Outputs are things that our device is going to control in the "real world."[12] An example of the outputs in the IoT thermostat example can be found in Figure 4-8.

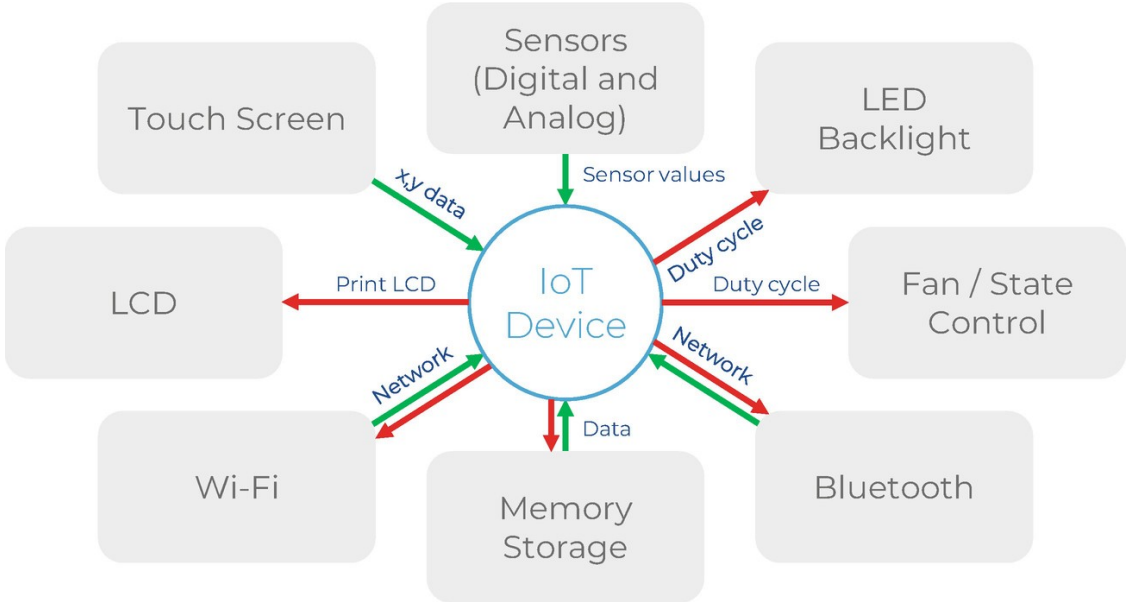

Figure 4-8 In step #4, we identify the system's outputs. These are the data products that are generated by the system

We can see from the diagram that this system has quite a few outputs. First, we have an output to the LCD, which is character data or commands to the LCD. The LED backlight and fan are each fed duty cycle information. There is then data associated with several other blocks as well.

Just like with the input labeling, it would not hurt to add a reference label to the block diagram and generate a table or description that gives more information about each. For example, I might expand on the LED backlight duty cycle output data to include

- Duty cycle 0.0–100.0%.
- The update rate is 100 milliseconds.

For the fan data, I might include something like

- Duty cycle 0.0–100.0%.
- The update rate is 50 milliseconds.
- Controlled by a state machine.

The additional details are used as placeholders for when we identify our tasks. In addition, the details allow us to understand the data we are acting on, the rate at which we are acting on it, and any other helpful information that will help us fully architect the system.

Step #5 – Identify First-Tier Tasks

Finally, we are ready to start identifying the first tasks in our system! As you may have already figured out, these first tasks are specific to interacting with the hardware components in our system! That's really what the outside-in approach is doing. The application needs to look at the outer layer of hardware, identify the tasks necessary to interact with it, and finally identify the inner, purely software-related tasks.

Typically, I like to take the block diagram I was working with and transform it during this step. I want to look at the inputs and outputs to then rearrange the blocks and data into elements that can be grouped. For example, I may take the sensor and touch screen blocks and put them near each other because they can be looked at as sensor inputs to the application. I might look at the Wi-Fi and Bluetooth blocks and say these are connectivity blocks and should also be placed near each other. This helps to organize the block diagram a bit before we identify our first tasks.

I make one critical adjustment to the block diagram; I expand the IoT device

block from a small text block to a much larger one. I do this so that the IoT device block can encompass all the tasks I am about to define! I then go through the system inputs and outputs and generate my first-tier tasks when I do this. The result can be seen in Figure 4-9.

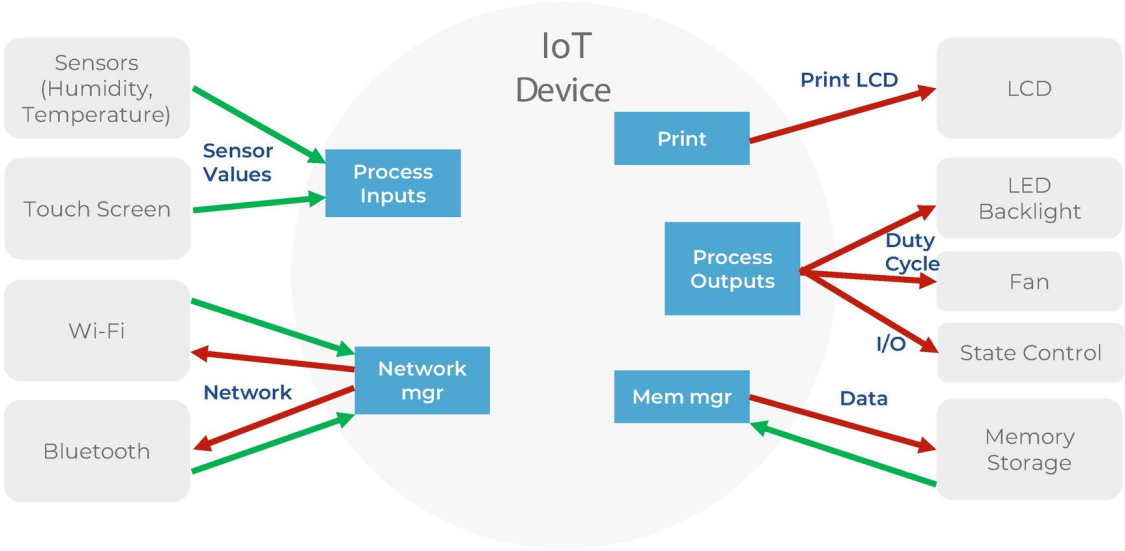

Figure 4-9 In step #5, we group data inputs and outputs and identify the first tasks in the outer layer of the application

Five tasks have been identified in this initial design which include

- Process inputs
- Network manager
- Print
- Process outputs
- Memory manager

What can be seen here is that we are trying to minimize the number of tasks included in this design. We've grouped all the sensor inputs into a single process input task. All connectivity has been put into a network management task. The nonvolatile memory source is accessed through a memory manager gatekeeping task rather than using a mutex to protect access.

This design shows a design philosophy to keep things simple and minimize complexity. I just as quickly could have created separate tasks for all sensors, individual tasks for Wi-Fi and Bluetooth, and so forth. How I break my initial tasks up will depend on my design philosophy and a

preference for how many tasks I want in my system. I try to break my system into layers; each layer does not contain more than a dozen tasks. This is because the human mind generally can only track seven plus or minus two pieces of information at any given time to short-term memory.[13] Limiting the number of tasks in each layer makes it more likely that I can manage that information mentally, reducing the chances for mistakes or bugs to be injected into the design.

Step #6 – Determine Concurrencies, Dependencies, and Data Flow

At this point, we now have our initial, first-tier tasks representing our design's outer layer. This layer is based on the hardware our application needs to interact with and the data inputs and outputs from those devices. The next step is carefully reviewing how that data will flow through the task system and interact with our application task(s). Figure 4-10 shows how one might modify our previous diagram to see this data flow.

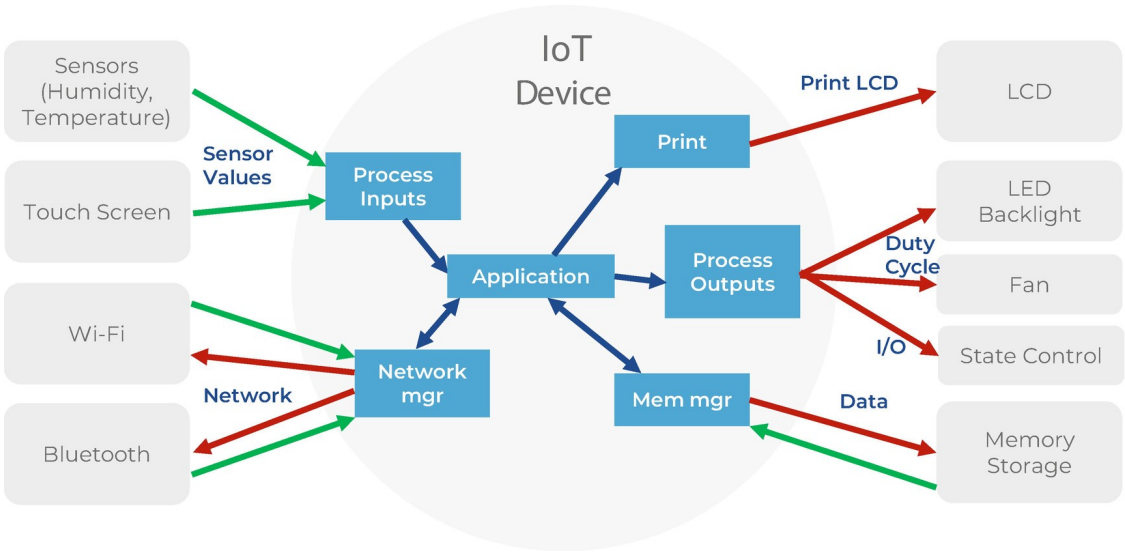

Figure 4-10 Carefully identify the high-level dependencies between the various tasks. This diagram shows arrows between the application task and the input/output tasks

In Figure 4-10, we have added an application task that encompasses all the application's activities. We have not yet decomposed this block but will do that once we understand the surrounding software architecture. The diagram allows us to think through how data will flow from the existing tasks to other tasks in the system. For example, we can decide whether the network manager can directly interact with the memory manager to retrieve and store settings or whether it must go first through

the application. This helps us start understanding how data will flow through our design and helps us understand how tasks might interact with each other.

The diagram also allows us to start considering which tasks are concurrent, that is, which ones may need to appear to run simultaneously and which are feeding other tasks. For example, if we have an output that we want to generate, we may want the input task and the application task to run first to act on the latest data. (We may also want to act on the output data from the last cycle first to minimize jitter in the output!)

Once we have identified our initial tasks, it's the perfect time to develop a data flow diagram. A data flow diagram shows how data is input, transferred through the system, acted on, and output from the system. In addition, the data flow diagram allows the designer to identify event-driven mechanisms and RTOS objects needed to build the application. These will include identifying

- Interrupts and direct memory access (DMA) accesses
- Queues to move data
- Shared memory locations to move data
- Mutexes to protect shared data
- Semaphores to synchronize and coordinate task execution
- Event flags to signal events and coordinate task execution

Figure 4-11 provides an example data flow diagram for our IoT thermostat.

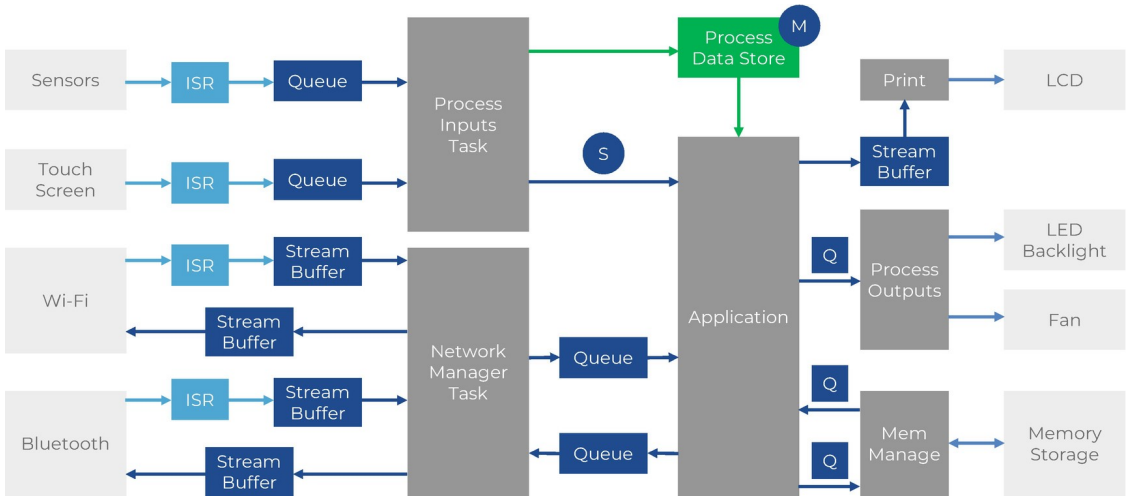

Figure 4-11 Developing a data flow diagram is critical in determining how data moves through the application and identifying shared resources, task synchronization, and resources needed for the application

We now have gone from a hardware diagram to identifying our first-tier tasks and generating a data flow diagram that identifies the design patterns we will use to move data throughout the application. However, the critical piece still missing in our design is how to break up our application task into second-tier tasks focusing specifically on our application. (After all, it's improbable that having a single task for our application will be a robust, scalable design.)

Step #7 – Identify Second-Tier Tasks

The data flow diagram we created in the last step can be beneficial in forming the core for our second-tier task diagram. In this step, we want to look at our application, its features, processes, etc., and break it into more manageable tasks. It wouldn't be uncommon for this generic block we have been looking at to break up into a dozen or more tasks suddenly! We've essentially been slowly designing our system in layers, and the application can be considered the inner layer in the design.

To identify our second-tier or application tasks, I start with my data flow diagram and strip out everything in the data flow outside the tasks that directly interact with the application. This essentially takes Figure 4-11 and strips it down into Figure 4-12. This diagram focuses on the application block's inputs, outputs, and processing.

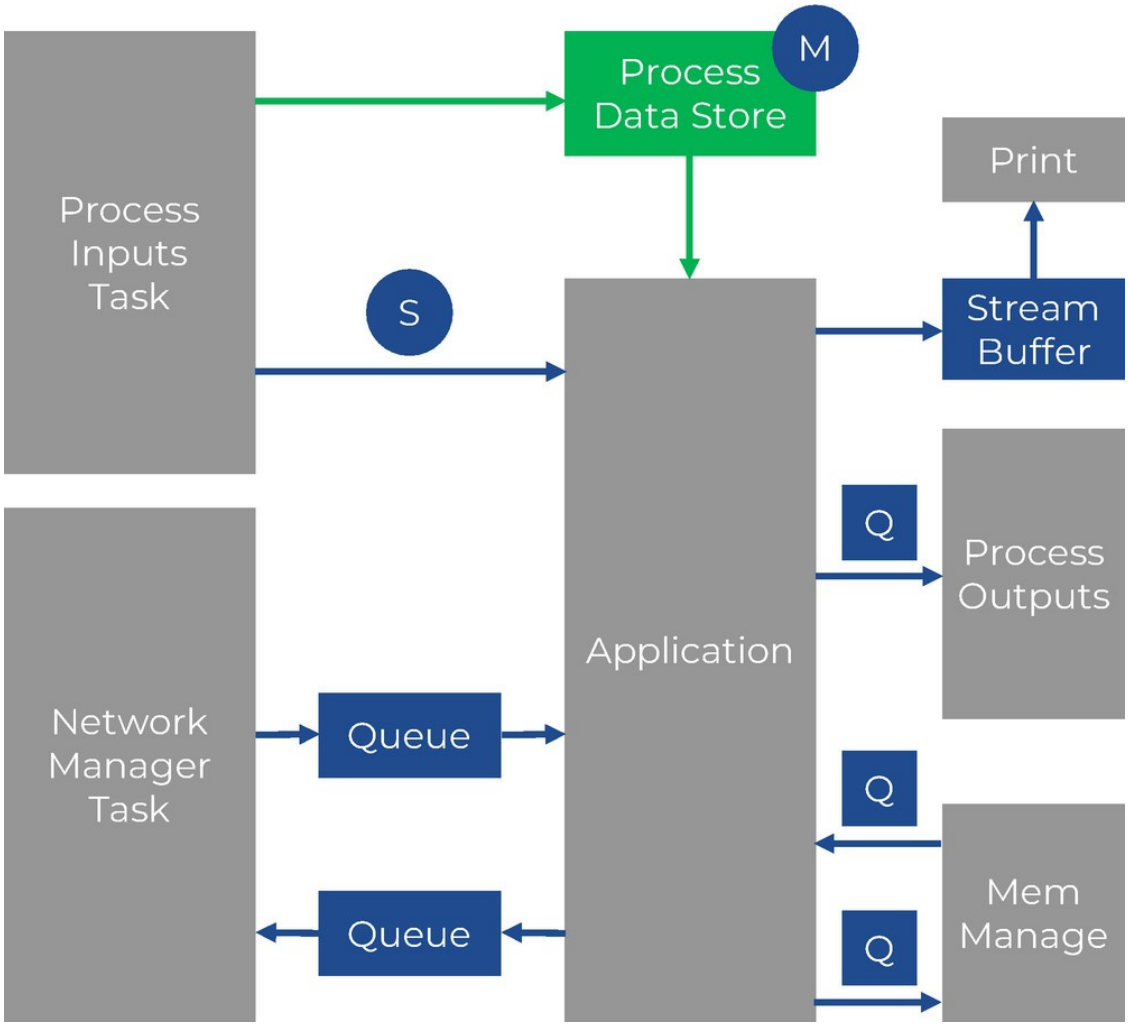

Figure 4-12 Identifying the second-tier application tasks involves focusing on the application task block and all the input and output data from that block

We now remove the application block and try to identify the tasks and activities performed by the application. The approach should look at each input and determine what application task would handle that input. Each output can be looked at to determine what application task would take that output. The designer can be methodical about this.

Let's start by looking at the process input task. Process inputs execute and receive new sensor data. The sensor data is passed to the application through a data store protected by a mutex. The mutex ensures mutually exclusive access to the data so that we don't end up with corrupted data. How process inputs execute would be as follows:

1. Process inputs acquire new sensor data.
2. Process inputs lock the mutex (preventing the application from accessing the data).
3. New sensor data is stored in the data store.

4. Process inputs unlock the mutex (allowing the application to access the data).
5. Process inputs signal the application that new data is available through a semaphore.

The process input task behavior is now clearly defined. So the question becomes, how will the application receive and process the new sensor data?

One design option for the application is to have a new task, sensor DSP, block on a semaphore waiting for new data to be available. For example, the sensor DSP task would behave as follows:

1. First, block execution until a semaphore is received (no new data available).
2. Then, when the semaphore is acquired, begin executing (data available).
3. Lock the sensor data mutex.
4. Access the new sensor data.
5. Unlock the sensor mutex.
6. Perform digital signal processing on the data, such as averaging or filtering.
7. Pass the processed data or results to the application task and pass the data to a gesture detection task.

This process has allowed us to identify those three tasks that may be needed to handle the process input data: a sensor DSP task, a controller task (formerly the application), and a gesture detection task. Figure **4-13** demonstrates what this would look like in addition to filling in the potential second-tier task for other areas in the design.

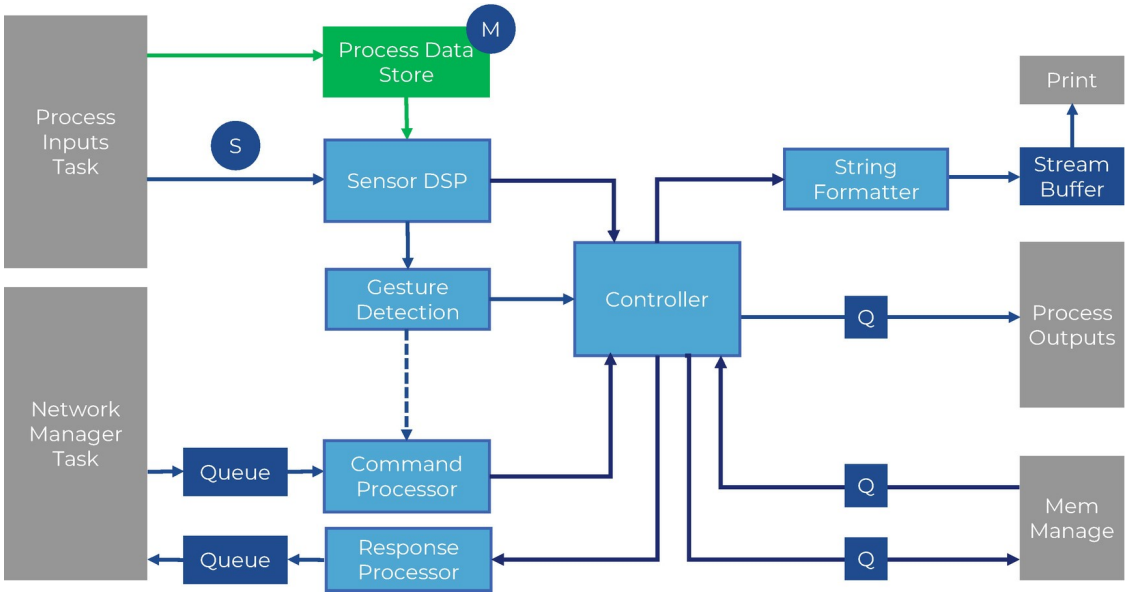

Figure 4-13 Second-tier tasks are created by breaking down the application task and identifying semi-independent program segments that assist in driving the controller and its output

A designer can, at this point, decompose the application further by looking at each block and identifying tasks. Once those tasks are identified, the designer can identify the RTOS objects needed to transfer data and coordinate task behavior. Figure 4-13 doesn't show these additional items to keep the diagram readable and straightforward. For example, the Sensor DSP task has an arrow going into the controller task. How is the result being transferred? Is it through a queue? Is it through a shared data store? At this point, I don't know. Again, it's up to the designer to look at the timing, the data rates, and so forth to determine which design pattern makes the most sense.

Now that we have identified our tasks, several questions should be on every designer's mind:

- How am I going to set my task priorities?
- Can the RTOS even schedule all these tasks?

Let's discuss how to set task priorities and perform a schedulable task analysis to answer these questions.

Setting Task Priorities

Designers often set task priorities based on experience and intuition.

There are several problems with using experience and intuition to set task priorities. First, if you don't have experience, you'll have no clue how to set the preferences! Next, even if you have experience, it doesn't mean you have the experience for the designed application. Finally, if you rely on experience and intuition, the chances are high that the system will not be optimized or may not work at all! The implementers may need to constantly fiddle and play with the priorities to get the software stable.

The trick to setting task priorities relies not on experience or intuition but on engineering principles! When setting task priorities, designers want to examine task scheduling algorithms that dictate how task priorities should be set.

Task Scheduling Algorithms[14]

There are typically three different algorithms that designers can use to set task priorities:

- Shortest job first (SJF)
- Shortest response time (SRT)
- Periodic execution time (RMS)

To understand the differences between these three different scheduling paradigms, let's examine a fictitious system with the characteristics shown in Table 4-1.

Table 4-1 An example system has four tasks: two periodic and two aperiodic

Task Number	Task Type	Response Time (ms)	Execution Time (ms)	Period (ms)
1	Periodic	30	20	100
2	Periodic	15	5	150
3	Aperiodic	100	15	–
4	Aperiodic	20	2	–

In shortest job first scheduling, the task with the shortest execution time is given the highest priority. For example, for the tasks in Table 4-1, task 4 has a two-millisecond execution time, which makes it the highest priority task. The complete task priority for the shortest job first can be seen in Table 4-2.

It is helpful to note that exact measurements for the task execution times will not be available during the design phase. It is up to the designer to estimate, based on experience and intuition, what these values will be. Once the developers begin implementation, they can measure the execution times and feed them back into the design to ensure that the task priorities are correct.

Shortest response time scheduling sets the task with the shortest response time as the highest priority. Each task has a requirement for its real-time response. For example, a task that receives a command packet every 100 milliseconds may need to respond to a completed packet within five milliseconds. For the tasks in Table 4-1, task 2 has the shortest response time, making it the highest priority.

Table 4-2 Priority results for the system are defined in Table 4-1 based on the selected scheduling algorithm

Scheduling Policy	Shortest Response Time	Shortest Execution Time	Rate Monotonic Scheduling
Assigned Task Priority	Task 2 – Highest	Task 4 – Highest	Task 1 – Highest
	Task 4	Task 2	Task 2
	Task 1	Task 3	Undefined
	Task 3	Task 1	Undefined

Finally, the periodic execution time scheduling sets the task priority with the shortest period. Therefore, the task must be a periodic task. For example, in Table 4-1, tasks 3 and 4 are aperiodic (event-driven) tasks that do not occur at regular intervals. Therefore, they have an undefined task pri-

ority based on the periodic execution time policy.

Table 4-2 summarizes the results for setting task priorities for the tasks defined in Table 4-1 based on the selected scheduling policy. The reader can see that the chosen scheduling policy can dramatically impact the priority settings. Conversely, selecting an inappropriate scheduling policy could affect how the system performs and even determine whether deadlines will be met successfully or not.

For many real-time systems, the go-to scheduling policy starts with the periodic execution time, also known as rate monotonic scheduling (RMS). RMS allows developers to set task priorities and verify that all the tasks in a system can be scheduled to meet their deadlines! Calculating CPU utilization for a system based on its tasks is a powerful tool to make sure that the design is on the right track. Let's now look at how verifying our design is possible by using rate monotonic analysis (RMA).

> **Best Practice**While we often default to rate monotonic scheduling, multiple policies can be used simultaneously to handle aperiodic tasks and interrupts.

Verifying CPU Utilization Using Rate Monotonic Analysis (RMA)

Rate monotonic analysis (RMA) is an analysis technique to determine if all tasks can be scheduled to run and meet their deadlines.[15] It relies on calculating the CPU utilization for each task over a defined time frame. RMA comes in several different flavors based on how complex and accurate an analysis the designer wants to perform. However, the basic version has several critical assumptions that developers need to be aware which include

- Tasks are periodic.
- Tasks are independent.
- Preemptive scheduling is used.
- Each task has a constant execution time (can use worst case).
- Aperiodic tasks are limited to start-up and failure recovery.
- All tasks are equally critical.
- Worst-case execution time is constant.

At first glance, quite a few of these assumptions seem unrealistic for the real world! For example, all tasks are periodic except for start-up and failure recovery tasks. This is an excellent idea, but what about a task kicked off by a button press or other event-driven activities? In a case like this, we either must ignore the event-driven task from our analysis or assume a worst-case periodic execution time so that the task is molded into the RMA framework.

The other big assumption in this list is that all tasks are independent. If you have ever worked on an RTOS-based system, it's evident that this is rarely the case. Sometimes, tasks will protect a shared resource using a mutex or synchronize task execution using a semaphore. These interactions make it so that the task execution time may be affected by other tasks and not wholly independent. The basic RMA assumptions don't allow for these cases, although there are modifications to RMA that cover these calculations beyond our discussion's scope.[16]

The primary test to determine if all system tasks can be scheduled successfully is to use the basic RMA equation:

$$\sum_{k=1}^{n} \frac{E_k}{T_k} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

In this equation, the variables are defined as follows:

- E_k is the worst-case execution time for the task.
- T_k is the period that the task will run at.
- n is the number of tasks in the design.

At its core, this equation is calculating CPU utilization. We calculate the CPU utilization for each task and then add up all the utilizations. The total CPU utilization, however, doesn't tell the whole story. Just because we are less than 100% does not mean we can schedule everything!

The right side of the equation provides us with an inequality that we must compare the CPU utilization. Table 4-3 shows the upper bound on the CPU utilization that is allowed to schedule all tasks in a system successfully. Notice that the inequality quickly bounds itself to 69.3% CPU utilization for an infinite number of tasks.

Table 4-3 Scheduling all tasks in a system and ensuring they meet their deadlines

depends on the number of tasks in the system and the upper bound of the CPU utilization

Number of Tasks	CPU Utilization (%)
1	100%
2	82.8%
3	77.9%
4	75.6%
∞	69.3%

RMA allows the designer to use their assumptions about their tasks, and then based on the number of tasks, they can calculate whether the system can successfully schedule all the tasks or not. It is essential to recognize that the basic analysis is a sanity check. In many cases, I view RMA as a model that isn't calculated once but tested using our initial assumptions and then periodically updated based on real-world measurements and refined assumptions as the system is built.

The question that may be on some readers' minds is whether the example system we were looking at in Table 4-1 can be scheduled successfully or not. Using RMA, we can calculate the individual CPU utilization as shown in Table 4-4.

Table 4-4 An example system has four tasks: two periodic and two aperiodic

Task Number	Task Type	Response Time (ms)	Execution Time (ms)	Period (ms)	CPU Utilization (%)
1	Periodic	30	20	100	20.0
2	Periodic	15	5	150	3.33
3	Aperiodic	100	15	100[17]	15.0

Task Number	Task Type	Response Time (ms)	Execution Time (ms)	Period (ms)	CPU Utilization (%)
4	Aperiodic	20	2	50[18]	4.0

The updated table now contains a CPU utilization column. When we sum the CPU utilization column, the result is 42.33%. Can the tasks in this system be scheduled successfully? We have two options to test. First, we can plug n = 4 into the basic RMA equation and find that the result is 75.6%. Since 42.33% < 75.6% is valid, a system with tasks with these characteristics can be scheduled successfully. Second, we could just use the infinite task number value of 69.3% and test the inequality. Either way, the result will show us that we should be able to schedule this system without any issues.

Measuring Execution Time

The discussion so far in this chapter has been focused on designing RTOS applications. Everything that we have discussed so far is important, but the reader might be wondering how you can measure execution time. Our rate monotonic analysis, the priorities we assign our tasks, and so forth, are very dependent upon the execution time of our tasks. For just a few moments, I want to discuss how we can make these measurements in the real world and use them to feed back into our design.

There are several different techniques that can be used to measure task execution time. First, developers can manually make the measurements. Second, developers can rely on a trace tool to perform the measurements for them. Let's look at both approaches in a little more detail.

Measuring Task Execution Time Manually

Measuring task execution time manually can provide developers with fairly accurate timing results. The problem that I often see with manual measurements is that they are performed inconsistently. Developers often

must instrument their code in some manner and then purposely make the measurements. The result is that they are performed at inconsistent times, and it's easy for system timing to get "out of whack."

In many systems, the manual measurements are performed either by using GPIO pins or using an internal timer to track the time. The problem with the manual measurements becomes that it may not be possible to instrument all the tasks simultaneously, making it an even more manual and time-consuming process to make the measurements. For these reasons, I often don't recommend that teams working with an RTOS perform manual measurements on their timing. Instead, tracing is a much better approach.

Measuring Task Execution Time Using Trace Tools

Most RTOS kernels have a trace capability built into them. The trace capability is a setting in the RTOS that allows the kernel to track what is happening in the kernel so that developers can understand the application's performance and even debug it if necessary. The trace capability is an excellent, automated, instrumentation tool for designers to leverage to understand how the implemented application is performing compared to the design.

Within the RTOS, whenever an event occurs such as a task is switched, a semaphore is given or taken, a system tick occurs, a mutex is locked or unlocked, and so forth, the event is recorded in the kernel. As each event occurs, the kernel tracks the time at which the event occurs. If these events are stored and then pushed to a host environment, it's possible to take the event data and reconstruct not just the ordering of the events but also the timing of the events!

Designers need two things to retrieve the event data from the RTOS kernel. First, they need a recording library that can take the events from the kernel and store them in a RAM buffer. The RAM buffer can then either be read out when it is full (snapshot mode) or it can be emptied at periodic intervals and streamed to the host (streaming mode) in real time. Second, there needs to be a host-side application that can retrieve the

event data and then reconstruct it into a visualization that can be reviewed by the developer.

We will discuss tools a bit more in Chapter 15, but I don't want to move on without giving you a few examples of the tools I use to perform RTOS measurements. The first tool that can be used is SEGGER SystemView. An example trace from SystemView can be seen in Figure 4-14. The RTOS generates the events which are then streamed using the SEGGER RTT library through SWD into a SEGGER J-Link which then distributes it through the J-Link server to SystemView. SystemView then records the events and generates

- An event log
- A task context switch diagram in a "chart strip" visualization
- The total CPU utilization over time
- A context summary that lists the system tasks along with statistical data about them such as total runtime, minimum execution time, average execution time, maximum execution time, and so forth.

SystemView is a free tool that can be downloaded by any team that is using a J-Link. SystemView can provide some amazing insights, but as a free tool, it does have limitations as to what information it reports and its capabilities. However, for teams looking for quick and easy measurements without spending any money, this is a good first tool to investigate.

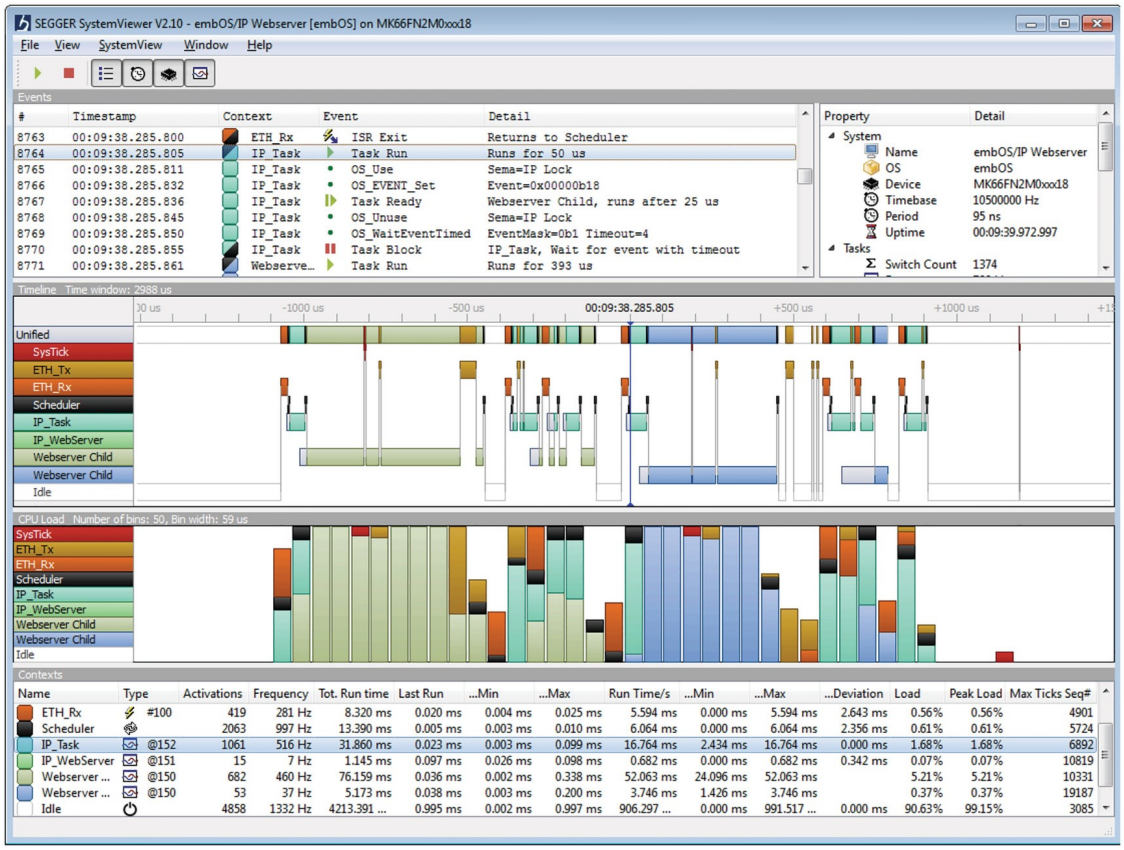

Figure 4-14 An example RTOS trace taken using SEGGER SystemView. Notice the running trace data that shows task context switches and the event log with all the events generated in the system. (Image Source: SEGGER[19])

The second tool that I would recommend and the tool that I use for all my RTOS measurements and analysis is Percepio's Tracealyzer. Tracealyzer is the "hot rod" of low-cost RTOS visualization tools. Tracealyzer provides the same measurements and visualizations as SystemView but takes things much further. The event library can record the standard RTOS events but also allows users to create custom events as well. That means if a developer wanted to track the state of a state machine or the status of a network stack and so forth, they can create custom events that are recorded and reported.

The Tracealyzer library also allows developers to use a J-Link, serial interface, TCP/IP, and other interfaces for retrieving the trace data. I typically pull the data over a J-Link, but the option for the other interfaces can be quite useful. The real power within Tracealyzer is the reporting capabilities and the linking of events within the interface. It's very easy to browse through the events to discover potential problems. For example, a developer can monitor their heap memory usage and monitor stacks for over-

flows. An example of the Tracealyzer user interface can be seen in Figure 4-15.

Okay, so what does this have to do with measuring execution times? As I mentioned, both tools will report for each task in your system, the minimum, average, and maximum execution times. In fact, if you are using Tracealyzer, you can generate a report that will also tell you about the periodicity for each task and provide minimum, average, and maximum times between when the task is ready to execute and when it executes! There's no need to manually instrument your code, you rely on the RTOS kernel and the event recorder which can then provide you with more insights into your system execution and performance than you will probably ever imagine. (You can even trace which tasks are interacting with semaphores, mutexes, and message queues.)

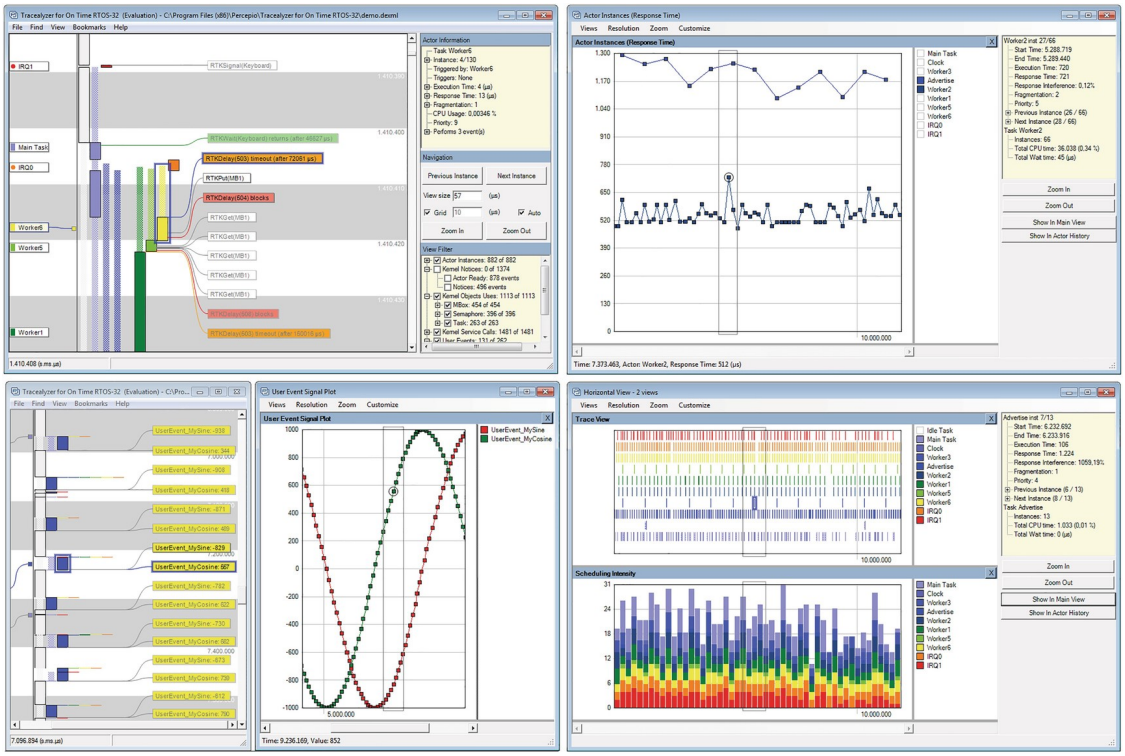

Figure 4-15 An example RTOS trace taken using Percepio Tracealyzer. Notice the multiple views and the time synchronization between them to visualize what is happening in the system at a given point in time. (Image Source: Percepio[20])

I'm probably a little bit biased when it comes to using these tools, so I would recommend that you evaluate the various tools yourself and use the one that best fits your own needs and budget. There are certainly other tools out there that can help you measure your task execution and performance, but these two are my favorites for teams that are on a tight budget. (And the cost for Tracealyzer is probably not even noticeable in

your company's balance sheet; I know it's not in mine!)

Final Thoughts

Real-time operating systems have found their way into most embedded systems, and it's likely they will increase and continue to dominate embedded systems. How you break your system up into tasks and processes will affect how scalable, reusable, and even host robust your system is. In this chapter, we've only scratched the surface of what it takes to decompose a system into tasks, set our task priorities, and arrive at a functional system task architecture. No design or implementation can be complete without leveraging modern-day tracing tools to measure your assumptions in the real world and use them to feed back into your RMA model and help to verify or tune your application.

Action Items
To put this chapter's concepts into action, here are a few activities the reader can perform to start applying RTOS design principles to their application(s):

- Take a few moments to think of a system that would be fun to design. For example, it could be an IoT sensor node, thermostat, weather station, drone, etc. Then, write out a high-level system description and list a few requirements.
- From your system description, decompose the system into tasks using the feature-based decomposition method.
- From your system description, use the outside-in approach to decompose the system into tasks. Then, compare the differences between the two designs.
- Create a data flow diagram for your system.
- Perform an RMA on your system. Use the RMA to identify and set your task priorities. Is your system schedulable?
- Examine your tasks and the supporting objects. Then, break your system up further into processes.
- Review your design and identify areas where it can be optimized.
- Implement a simple RTOS-based application that includes a couple of tasks. The system doesn't need to do anything, just have the tasks execute at different rates and periods.

- Download SystemView and explore the trace capabilities it provides to analyze your simple RTOS application. What are its strengths and weaknesses?
- Download Tracealyzer and explore the trace capabilities it provides to analyze your simple RTOS application. What are its strengths and weaknesses?

These are just a few ideas to go a little bit further. Carve out time in your schedule each week to apply these action items. Even minor adjustments over a year can result in dramatic changes!

Footnotes

1 www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

2 www.embedded.com/program-structure-and-real-time/

3 www.embeddedrelated.com/thread/5762/rtos-vs-bare-metal

4 Unknown reference.

5 https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter1

6 https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter1

7 https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter1

8 This diagram is inspired by and borrowed from Jean Labrosse's "Using a MPU with an RTOS" blog series.

9 This diagram is inspired by and borrowed from Jean Labrosse's "Using a MPU with an RTOS" blog series.

10 www.webopedia.com/definitions/feature/#:~:text=(n.),was%20once%20a%20simple%20application

11 An abbreviated version of this approach is introduced in "Real-Time Concepts for Embedded Systems" by Qing Li with Caroline Yao in Chapter 14, Section 2,

pages 214–216.

12 I mention “real world” here because there will be inputs and outputs in the application layer that don’t physically interact with the world but are just software constructs.

13 Miller, G. (1956), The psychological review, 63, 81–97.

14 The concepts, definitions, and examples in this section are taken from *Real-Time Operating Systems Book 1 – The Theory* by Jim Cooling, Chapter 9, page 6.

15 https://en.wikipedia.org/wiki/Rate-monotonic_scheduling

16 A basic reference for additional techniques is *Real-Time Concepts for Embedded Systems* by Qing Li with Caroline Yao, Chapter 14, and *A Practitioner’s Handbook for Real-Time Analysis*.

17 This is the worst-case periodic rate for this aperiodic, event-driven task.

18 This is the worst-case periodic rate for this aperiodic, event-driven task.

19 https://c.a.segger.com/fileadmin/documents/Press_Releases/PR_151106_SEGGER_SystemView.pdf

20 https://percepio.com/docs/OnTimeRTOS32/manual/