Embedded Systems Design, Analysis and Optimization using the Renesas RL78 Microcontroller

BY ALEXANDER G. DEAN

Micriμm Press
1290 Weston Road, Suite 306
Weston, FL 33326
USA

www.micrium.com

Library of Congress subject headings:

1. Embedded computer systems
2. Real-time data processing
3. Computer software—Development

For bulk orders, please contact Micrium Press at: +1 954 217 2036

ISBN: 978-1-935772-96-5

Please report errors or forward any comments and suggestions to agdean@ncsu.edu

Preface

When Renesas asked me to write another book on embedded system design using the RL78 microcontroller family, I was excited to be able to pick up where our first text left off. Embedded systems draw upon many fields, resulting in many opportunities for creative and synergistic optimizations. This book provides methods to create embedded systems which deliver the required performance (throughput and responsiveness) within the available resources (memory, power and energy). This book can be used on its own for a senior or graduate-level course on designing, analyzing and optimizing embedded systems.

I would like to thank the team that made this book possible: June Hay-Harris, Rob Dautel, and Todd DeBoer of Renesas, and the compositor Linda Foegen. Many thanks go to the reviewers because their comments made this book better, especially John Donovan, Calvin Grier, Mitch Ferguson, and Jean Labrosse.

I would like to thank Bill Trosky and Phil Koopman for opening the doors into so many embedded systems through in-depth design reviews. I also thank the many embedded systems engineers on those projects for helping me understand their goals and constraints as we discussed risks and possible solutions.

I also thank my research students and the students in my NCSU embedded systems courses for bringing their imagination, excitement, and persistence to their projects. I would like to thank Dr. Jim Conrad and his students for their collaboration in developing our previous textbooks; these served as a launch pad for this text.

Finally, I would also like to thank my wife Sonya for sharing her passion of seeking out and seizing opportunities, and our daughters Katherine and Jacqueline for making me smile every day. Finally, I would like to thank my parents for planting the seeds of curiosity in my mind.

Alexander G. Dean
September 2013

Foreword

For more than a decade the microcontroller world has been dominated by the quest for ultra-low power, high performance devices—two goals that are typically mutually exclusive. The Renesas RL78 MCU quickly achieved market leadership by achieving both of these goals with a highly innovative architecture. The RL78 family enables embedded designs that previously would have required some uncomfortable tradeoffs.

However there are no simple solutions to complex problems, and mastering all of the RL78's advanced features is not a task to be undertaken lightly. Fortunately in this book Dr. Dean has crafted a guidebook for embedded developers that moves smoothly from concepts to coding in a manner that is neither too high level to be useful nor too detailed to be clear. It explains advanced software engineering techniques and shows how to implement them in RL78-based applications, moving from a clear explanation of problems to techniques for solving them to line-by-line explanations of example code.

Modern embedded applications increasingly require hardware/software co-design, though few engineers are equally conversant with both of these disciplines. In this book the author takes a holistic approach to design, both explaining and demonstrating just how software needs to interact with RL78 hardware. Striking a balance between breadth and depth it should prove equally useful and illuminating for both hardware and software engineers.

Whether you are a university student honing your design skills, a design engineer looking for leading edge approaches to time-critical processes, or a manager attempting to further your risk management techniques, you will find Alex's approach to embedded systems to be stimulating and compelling.

Peter Carbone
Renesas
September 19, 2013

Contents

Introduction

1.1 INTRODUCTION

The goal of this book is to show how to design, analyze, and optimize embedded computing systems built on Renesas RL78 family microcontrollers (MCUs). In a perfect world, we would just go ahead and design the perfect (optimal) system from the start. Reality prevents this perfection—the complexities of the technology, design process, requirements evolution, and human nature keep us from this goal. Instead, we approximate. We choose a starting point based on our judgment, which comes from experience (which in turn often comes from bad judgment). Based on the predicted, prototyped, or measured performance of that initial design we make changes, measure their impact, and decide whether to keep or discard them. Analysis is a necessary step between design and optimization. By repeating this process we make progress towards a system which meets the design goals.

1.2 DOMINANT CHARACTERISTICS FOR EMBEDDED SYSTEM MARKET SEGMENTS

Economic factors have a major impact on which technology is used (and how) in embedded computing systems. This was explained quite nicely by Nick Tredennick, a veteran processor designer (Tredennick, 2000). Briefly, the embedded systems market can be divided into four segments (cost, power, delay and volume, shown in Figure 1.1) based on the dominant characteristic of that market segment. In the first three segments (low cost, low power and low delay), that characteristic is a constraint or requirement which dominates the design effort. In the last segment (low volume), the characteristic is not a constraint, but instead an indication that the designers have much more flexibility to meet their design goals. Tredennick emphasizes the distinguishing segment characteristic by describing the ideal value—e.g zero-cost vs. low-cost.

- The **zero-cost** segment is by far the largest segment. These are consumer products with modest computational requirements but which are sold in markets with very

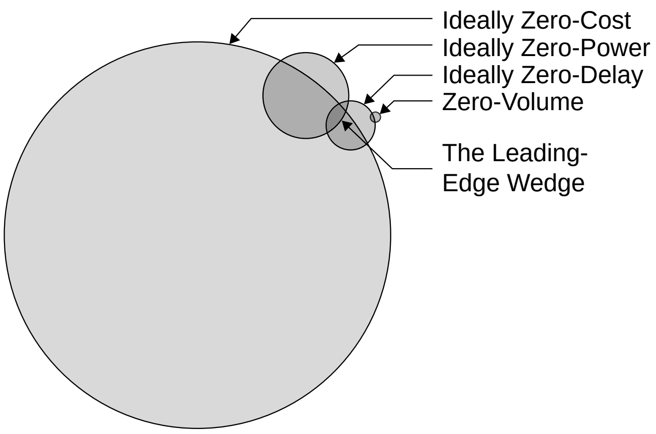


Figure 1.1 Tredennick's Classification of Embedded Systems.

strong price competition. High volumes combined with price pressures lead developers to invest extensively in up-front design to reduce per-unit costs. The high sales volume allows for higher development costs, as these will be divided across all the units sold.

- The **zero-power** segment is much smaller. It includes portable devices which must store or harvest energy to run the system. The devices in this segment still have price pressures, given the nature of markets.
- The **zero-delay** segment is smaller yet. Devices in this category have significant computational requirements in order to provide fast processing throughput. The devices in this segment still have price pressures.
- The **zero-volume** segment is still smaller. In this category, it is quite practical to purchase pre-developed modules in order to simplify development effort because end unit cost is irrelevant. This segment is outside the scope of this text.

Over time the challenges facing embedded system developers are growing. Rising consumer expectations for performance (speed, responsiveness, etc.) are making the **zero-delay/zero-cost** overlap grow. This raises computational requirements; either the existing code must be made faster, or faster hardware must be used. Similarly, growing expectations for longer battery life are making the **zero-power/zero-cost** overlap grow. Tredennick calls the overlap of these first three segments the **leading-edge wedge** and describes its contents as “cheap, highly capable devices that give us instant answers and that work on weak ambient light.”

1.3 LOOKING INSIDE THE BOXES

Creating systems in these overlaps is much easier if we understand how the internal system operates. What determines an embedded system's delay, cost or power? Taking a look at the factors contributing to these top-level metrics gives some insight into how the system design affects each.

For example, a system's **delay** consists of several parts, determined by both the **speed of the code** as well as the **task scheduling** mechanism used. How long does the code take on the critical path from input to output? When does the scheduler start running that code, given that the processor has other ongoing processing responsibilities? Can the critical path code be preempted by other code while executing?

Similarly, **cost** consists of both non-recurring costs (e.g., development) and per-unit production costs. For many embedded systems, the MCU cost is a large factor, while in others it is dwarfed by other costs (e.g., power electronics). MCU costs are very sensitive to internal SRAM size and flash ROM program size. Reducing the requirements for these memories can reduce costs. Choosing an appropriate task scheduling mechanism can reduce costs significantly. Creating a working system quickly reduces development costs. Using existing code libraries and/or a real-time operating system (RTOS) can simplify code development significantly, albeit at the price of less control due to greater abstraction. Similarly, the optimization process followed can affect development costs and progress.

A system's **power** and **energy** use result from a variety of interrelated factors, often leading to a "non-obvious" solution. Relative differences between MCU static and dynamic power characteristics, availability and relative benefits of power saving modes, and peripheral power consumption characteristics all play a role.

1.4 ABSTRACTION VS. OPTIMIZATION

Optimization can be challenging because we develop systems using **abstractions.** When we write a program in source code to perform operation *X* and compile it for a given microprocessor and its instruction set architecture, we use the compiler to convert the source code into object code for our abstracted box *B*.

We can determine that *B* requires a certain amount of power, time, and memory to perform that function. However, without inspection, we do not know how good that solution is. Could we reduce the time used? How difficult would it be? How would that affect the power and memory requirements?

A fundamental constraint of using abstractions is that we don't know how "full" each box is. If it isn't full, we could use a somewhat smaller box, or perhaps build a better solution which still fits in the box. We need to look inside the box to understand it, and for embedded systems this requires expertise in a variety of fields spanning both software and hardware.

1.5 OPTIMIZING A SYSTEM BUILT ON ABSTRACTIONS

A further complicating factor is that in order to build systems of manageable complexity in a reasonable amount of time, we must build systems out of many such boxes without understanding everything in each one. In fact, some boxes may contain other boxes. How do we improve overall performance or resource requirements for a system with so much hidden complexity?

Where do we start? Ideally we would know some critical facts about each box.

- How much of a problem is this particular box? That is, how much does its performance affect the overall system's performance?
- How good is the solution which is currently in the box?
- Is there a relatively easy way to significantly improve what's in the box?

The first question can be answered through preliminary modeling (if the system is simple enough) or profiling an actual system (if the system is complex, or development has proceeded enough). The second and third questions are more challenging and require both breadth and depth of understanding. In order to evaluate a software module, we would need to understand not only the specific solution (e.g., the quality of the algorithm used, the source code implementation, the library code, the resulting machine code) but also what alternatives are available and viable. One of the goals of this text is to provide the reader with the skills needed to address these questions in the domains of code speed, responsiveness, power and energy use, and memory requirements.

1.6 ORGANIZATION

This text is organized as follows. Each group of chapters covers design concepts, analytical methods such as modeling and profiling, and techniques for optimization.

- Chapters 2 and 3 show how to create responsive systems which share a processor's time among multiple software tasks while providing predictable performance.
  - Chapter 2 introduces preemptive and cooperative task scheduling approaches and then shows how to design an application using such schedulers and services. The μC/OS-III real-time kernel from Micrium is used (Labrosse & Kowalski, 2010).
  - Chapter 3 presents real-time scheduling theory for preemptive and cooperative schedulers. These methods allow accurate calculation of the worst-case response time of a system, enabling a designer to determine if any deadlines may ever be missed.

- Chapters 4, 5, and 6 examine how to make code run faster.
  - Chapter 4 covers execution time profiling analysis and understanding the output of the compiler.
  - Chapter 5 shows how to use the compiler effectively to generate efficient code. The IAR Embedded Workbench for RL78 is targeted.
  - Chapter 6 examines optimization methods at the program level, including algorithms and data structures as faster mathematical operations.
- Chapters 7 and 8 show how to create power- and energy-efficient systems.
  - Chapter 7 introduces power and energy models and analytical methods. It then examines the power- and energy-reducing features of the RL78G14 MCU, including low power modes, peripheral snooze modes, and clocking options.
  - Chapter 8 presents methods to apply the features of the RL78G14 to optimize power or energy consumption for embedded applications.
- Chapter 9 examines how to make programs use less memory. It presents methods for evaluating memory requirements (RAM and ROM) and optimization techniques ranging from algorithms and data structures to task scheduling approaches.

1.7 BIBLIOGRAPHY

Labrosse, J., & Kowalski, F. (2010). *MicroC/OS-III: The Real-Time Kernel.* Weston, FL: Micrium Press.

Tredennick, H. L. (2000, August). *The Death of DSP.* Retrieved from http://www.ttivanguard.com/dublin/dspdealth.pdf, accessed 8/16/2013.

Chapter 2

Designing Multithreaded Systems

2.1 LEARNING OBJECTIVES

This chapter examines how to create multithreaded embedded software using a preemptive scheduler. We will explore how to predict worst-case responsiveness, enabling us to create real-time systems—systems which will never miss any task deadlines.

Most embedded systems have multiple independent tasks running at the same time. Which activity should the microprocessor perform first? This decision determines how responsive the system is, which then affects how it determines how fast a processor we must use, how much time we have for running intensive control algorithms, how much energy we can save, and many other factors. In this chapter we will discuss different ways for a microprocessor to schedule its tasks, and the implications for performance, program structure, and related issues.

2.2 MOTIVATION

Consider a trivially simple embedded system which controls a doorbell in a house. When a person at the front door presses the switch, the bell should ring inside the house. The system's **responsiveness** describes how long it takes from pressing the switch to sounding the bell. It is easy to create a very responsive embedded system with only one task. The scheduling approach shown below is an obvious and simple approach.

```
void main (void){
   init_system();
   while(1){
      if(switch == PRESSED){
         Ring_The_Bell();
      }
   }
}
```

Our doorbell is very responsive. In fact, we like it so much that we decide to add in a smoke detector and a very loud beeper so we can be warned about a possible fire. We also add a burglar detector and another alarm bell. This results in the following code shown.

```
void main (void){
   init_system();
   while(1){
      if(switch == PRESSED){
         Ring_The_Doorbell();
      }
      if(Burglar_Detected() == TRUE){
         Sound_The_Burglar_Alarm();
      }
      if(Smoke_Detected() == TRUE){
         Sound_The_Fire_Alarm();
      }
   }
}
```

Going from one task to three tasks has complicated the situation significantly.[1] How should we share the processor's time between these tasks?

- How long of a delay are we willing to accept between smoke detection and the fire alarm sounding? And the delay between the switch being pressed and the doorbell sounding?
- Should the system try to detect smoke or burglars while the doorbell is playing?
- Should the doorbell work while the smoke alarm is being sounded? What about when the burglar alarm is sounding?
- Which subsystem should the processor check first: the doorbell, the smoke detector, or the burglar detector? Or should it just alternate between them?
- Should the doorbell switch be checked as often as the smoke and burglar detectors, or at a different rate?
- What if the person at the door presses the switch again before the doorbell finishes sounding? Should that be detected?

Now that we have to share the processor, we have to worry about how long the bell rings and the alarms sound. If we use a doorbell ringtone which lasts for thirty seconds, then Ring_The_Bell will take at least thirty seconds to run. During this time, we won't know if our house is burning or being robbed. Similarly, what if the firemen come when the alarm is sounding? How quickly should the doorbell respond in that case?

[1] In fact, any number of tasks greater than one complicates the situation!

Our trivial system became much more complicated once we started sharing the processor among different tasks and considering responsiveness and concurrent events. Designers face this same challenge but on a much larger scale when creating embedded systems which must manage multiple activities concurrently while ensuring quick responses (e.g. in microseconds or milliseconds).

2.3 SCHEDULING FUNDAMENTALS

This example reveals the two fundamental issues in scheduling for responsive systems.

- If we have multiple tasks ready to run, which one do we run first? This decision defines the **ordering** of task execution.
- Do we allow one task to interrupt or **preempt** another task?

Both of these decisions will determine the system's **responsiveness,** which is measured by response times for each task.

- How long will it take for the **most important** task to **start running?** To **finish running?** Does this depend on how long any other tasks take to run, and how often they run?
- How long will it take for the **least important** task to **start running?** To **finish running?** We expect it will depend on how long all the other tasks take to run, and how often they run.
- If we allow tasks to preempt each other, then a task may start running very soon but finish much later, after multiple possible preemptions.

These ranges of response times in turn affect many performance-related issues, such as:

- How fast must the processor's clock rate be to ensure that nothing happens "late"?
- How much time do we have available for running compute-intensive algorithms?
- How much energy or power can we save by putting the processor to sleep?
- How quickly can a sleeping processor wake up and start running a task?
- How much power can we save by slowing down the processor?

A software component called a scheduler (or kernel) is responsible for sharing the processor's time among the tasks in the system. One of its main responsibilities is selecting which task to run currently, based on scheduling rules and task states. Figure 2.1 shows a visual representation of some arbitrary scheduling activity. Task *A* is released (becomes ready to run) at the first vertical bar. There is some **latency** between the release and when the task starts running, due to other processing in the system and scheduler overhead. Similarly, there is a **response time** which measures how long it takes task *A* to complete its processing. Some scheduling approaches allow a task to be preempted (delayed) after it has started running, which will increase the response time.

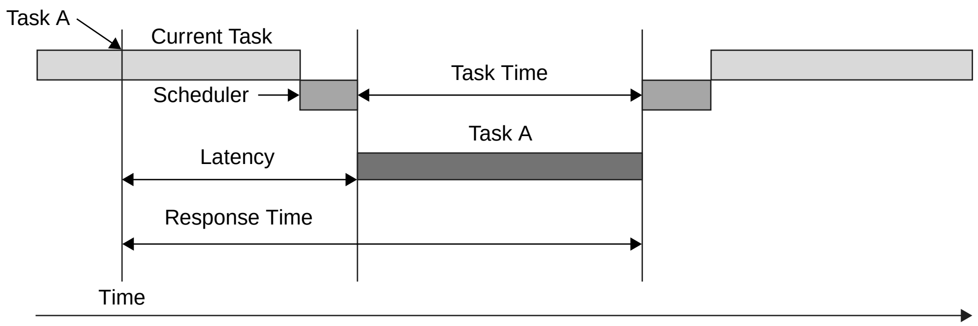


Figure 2.1 Diagram and Definitions of Scheduler Concepts.

2.3.1 Task Ordering

The first factor affecting response time is the **order in which we run tasks.** We could always follow the same order by using a **static** schedule. The code shown for the Doorbell/Fire Alarm/Burglar Alarm uses a static schedule. Figure 2.2a shows an interesting case. If a burglar broke in and a fire broke out just after someone pressed the switch to ring the doorbell, we wouldn't find out about the burglar for almost thirty seconds and the fire for about sixty seconds. We probably do not want these large delays for such critical notifications.

We can change the order based on current conditions (e.g., if the house is on fire) using a **dynamic** schedule. An obvious way to do this is to reschedule after finishing each task. A dynamic schedule lets us improve the responsiveness of some tasks at the price of delaying other tasks. For example, let's prioritize fire detection over burglar detection over the doorbell.

```
void main (void){
   init_system();
   while(1){
      if(Smoke_Detected() == TRUE){
         Sound_The_Fire_Alarm();
      } else if (Burglar_Detected() == TRUE) {
         Sound_The_Burglar_Alarm();
      } else if (switch == PRESSED) {
            Ring_The_Doorbell();
      }
   }
}
```

Notice how this code is different—there are **else** clauses added, which change the schedule to a dynamic one. As long as smoke is detected, Sound_The_Fire_Alarm() will run repeatedly. The burglar alarm and doorbell will be ignored until no more smoke is detected. Similarly, burglar detection will disable the doorbell. This is shown in Figure 2.2b.

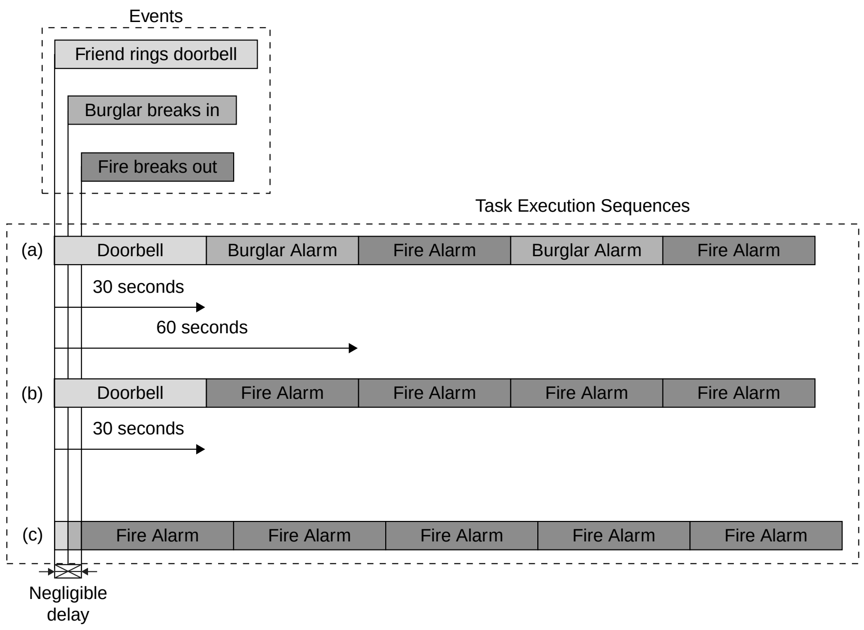


Figure 2.2 Doorbell/fire alarm/burglar alarm system behavior with different scheduling approaches.

This **strict prioritization** may or may not be appropriate for a given system. We may want to ensure some **fairness,** perhaps by limiting how often a task can run. Later in this chapter we present a periodic table-based approach which is much better than this hard-coded design.

2.3.2 Task Preemption

The second aspect to consider is whether one task can **preempt** another task. Consider our thirty-second doorbell ringtone—the task Ring_The_Doorbell will **run to completion** without stopping or yielding the processor.

What if a burglar breaks the window a split second after an accomplice rings the doorbell? In this worst-case scenario, we won't find out about the burglar (or a possible fire) for thirty seconds.[2] Let's say we'd like to find out within one second. We have several options:

- Limit the maximum duration for the doorbell ringtone to one second.
- Add another microprocessor which is dedicated to playing the doorbell ringtone. This will raise system costs.

[2] Imagine what Thomas Crown, James Bond, or Jason Bourne could do in that time!

- Break the Ring_The_Doorbell function into thirty separate pieces (e.g., with a state machine or separate functions), each of which takes only one second to run. This code will be hard to maintain.
- Allow the smoke and burglar detection code to preempt Ring_The_Doorbell. We will need to use a more sophisticated task scheduler which can (1) preempt and resume tasks, and (2) detect events which trigger switching and starting tasks. We will not need to break apart any source code. This will make code maintenance easier. However, we introduce the vulnerability to race conditions for shared data, and we also will need more memory (enough to hold each task's stack simultaneously).

Let's apply this preemption option to our system. We assign the highest priority to fire detection, then burglar detection, and then the doorbell. Now we have the response timeline shown in Figure 2.2c. The system starts sounding the doorbell after the switch is pressed. However, a burglar is detected a split-second after the doorbell is pressed, so the scheduler preempts the Ring_The_Doorbell and starts running Sound_The_Burglar_Alarm. And then a fire is detected after another split-second, so the scheduler preempts Sound_The_Burglar_Alarm and starts running Sound_the_Fire_Alarm. We find out about the fire essentially immediately, without having to wait for the doorbell or buglar alarm to finish sounding. In fact, we may not even hear them.

As with the previous example, we have strict prioritization without control of how often tasks can run. As long as smoke is detected, Sound_The_Fire_Alarm() will run repeatedly. The burglar alarm and doorbell will be ignored until no more smoke is detected. Similarly, burglar detection will disable the doorbell.

2.3.3 Fairness and Prioritization

These examples all show one weakness of our system: prioritizing some tasks over others can lead to starvation of lower priority tasks (they may never get to run). For some systems this is acceptable, but for others it is not. Here are two ways of providing some kind of fairness:

- We can allow multiple tasks to share the same priority level. If both tasks are ready to run, we alternate between executing each of them (whether by allowing each task to run to completion or by preempting each periodically).
- We can limit how often each task can run by defining the task frequency. This is the common approach used for designers of real-time systems. Note that we can still allow only one task per priority level.

2.3.4 Response Time

For the two non-preemptive examples in Figure 2.2, notice how the response time for the fire alarm and the burglar alarm depends on **how long the doorbell sounds.** However, for the preemptive approach those response times are **independent** of how long the doorbell sounds. This is the major benefit of a preemptive scheduling approach: it makes a task's response time essentially **independent** of all processing by lower priority tasks.[3] Instead, only **higher priority** tasks can delay that task.

In Figure 2.3 we present these relationships in a graph. A graph is a mathematical structure used to show how objects (called nodes or vertices) are related to each other (using connections called edges or arcs). Directed edges (with arrows) are used to show relationships in which the node order matters. Tasks and ISRs are nodes, while edges are timing dependences. For example, the edge from B to C indicates that task B's response time depends on task C's duration. We can now compare timing dependences for these three classes of scheduler.

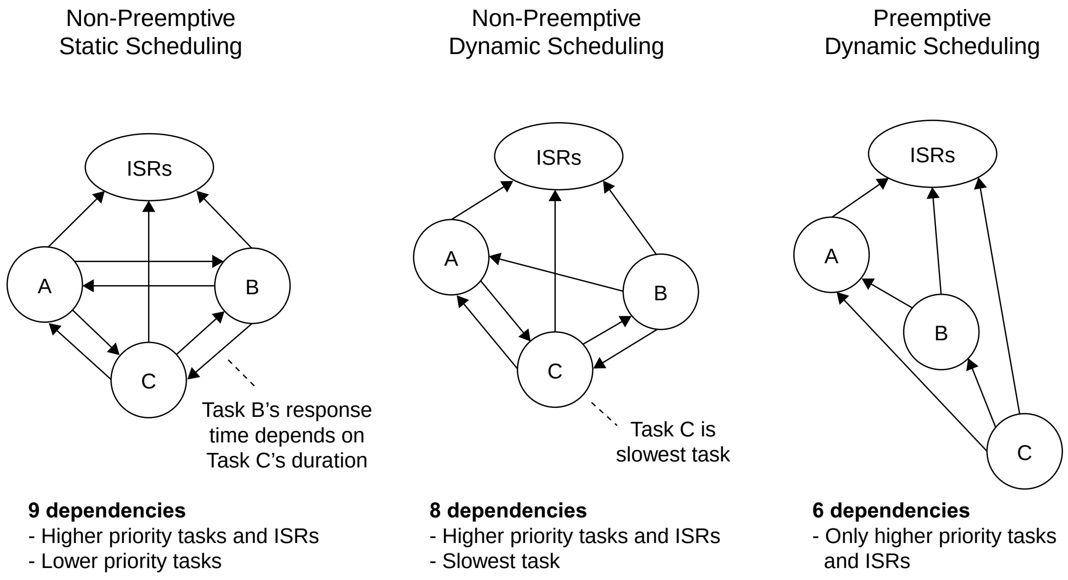

Non-Preemptive Static Scheduling
Non-Preemptive Dynamic Scheduling
Preemptive Dynamic Scheduling
ISRs
A
B
C
Task B's response time depends on Task C's duration
Task C is slowest task
9 dependencies
- Higher priority tasks and ISRs
- Lower priority tasks
8 dependencies
- Higher priority tasks and ISRs
- Slowest task
6 dependencies
- Only higher priority tasks and ISRs

Figure 2.3 Timing dependences of different scheduling approaches.

- With the non-preemptive static scheduler each task's response time depends on the duration of all other tasks and ISRs, so there are nine dependences.[4]
- With the non-preemptive dynamic scheduler, we assign priorities to tasks (A . B . C). In general, a task no longer depends on lower priority tasks, so we have more timing independence and isolation. This accounts for six dependences. The exception is the

[3] There are exceptions when tasks can communicate with each other with semaphores and other such mechanisms, but that is beyond the scope of this introductory text.

[4] Of course, if task code can disable interrupts, then there will be three more edges leading from the ISRs back to the tasks! That would be a total of twelve dependences, which is quite a few to handle.

slowest or longest duration task, which is C in this example. If task C has started running, it will delay any other task, regardless of priority. So the higher priority tasks A and B each have a dependence edge leading to task C in Figure 2.3, which results in a total of eight dependences.

- With the preemptive dynamic scheduler, we also prioritize the tasks (A . B . C). Because a task can preempt any lower priority task, the slowest task no longer matters. Each task can be preempted by an ISR, so there are three dependence edges to begin with. Task A cannot be preempted by B or C, so it adds no new edges. Task B can be preempted by task A, which adds one edge. Finally, task C can be preempted by task A or B, which adds two more edges. As a result we have only six dependences. Most importantly, these dependence edges **all point upwards.**[5] This means that in order to determine the response time for a task, we only need to consider **higher priority tasks.** This makes the analysis much easier.

The real-time system research and development communities have developed extensive precise mathematical methods for calculating worst-case response times, determining if deadlines can ever be missed, and other characteristics of a system. These methods consider semaphores, task interactions, scheduler overhead, and all sorts of other complexities of practical implementations. We provide an introduction to these concepts in the next chapter.

2.3.5 Stack Memory Requirements

The non-preemptive scheduling approaches do not require as much data memory as the preemptive approaches. In particular, the non-preemptive approach requires only **one call stack,** while a preemptive approach typically requires **one call stack per task.**[6]

The function call stack holds a function's state information such as return address and limited lifetime variables (e.g., automatic variables, which only last for the duration of a function). Without task preemption, task execution does not overlap in time, so all tasks can share the same stack. Preemption allows tasks to preempt each other at essentially any point in time. Trying to reuse the same stack space for different tasks would lead to corruption of this information on the stack. For example, task B is running function B3 which is using the stack for storing local data (say, an array of ten floating-point variables). The scheduler then preempts task B to run the higher priority task A, which was running function A2. Function A2 completes and it expects to return to function A1, which called A2. However, the call stack doesn't have the return address to get back to A1. Instead it has floating point variables. When A2 executes its return instruction, the program counter is loaded with data from B3 (a floating-point variable) rather than the return address in A1. And so the processor re-

[5] This is called a DAG or directed acyclic graph.

[6] There are several ways to reduce the number of stacks needed for preemptive scheduling, but they are beyond the scope of this text.

sumes executing code at the wrong address, or accesses an illegal address, and system fails to operate correctly.

As a result of these memory requirements for preemptive scheduling approaches, there are many cost-sensitive embedded systems which use a non-preemptive scheduler to minimize RAM sizes and therefore costs.

2.3.6 Interrupts

Interrupts are a special case of preemption with dedicated hardware and compiler support. They can be added to any of these scheduling approaches in order to provide faster, time-critical processing. In fact, for many systems **only** interrupt service routines are needed for the application's work. The main loop is simply an infinite loop which keeps putting the processor into a low-power idle mode.

When designing a system which splits between ISRs and task code, one must strike a balance. The more work which is placed in an ISR, the slower the response time for other processing (whether tasks or other ISRs[7]). The standard approach is to perform time-critical processing in the ISR (e.g., unloading a character from the UART received data buffer) and deferring remaining work for task code (pushing that character in a FIFO from which the task will eventually read). ISR execution duration affects the response time for other code, so it is included in the response time calculations described in Section 2.3.4 and in Figure 2.3.

2.4 TASK MANAGEMENT

2.4.1 Task States

A task will be in one of several possible states. The scheduler and the task code itself both affect which state is active. With a **non-preemptive dynamic scheduler,** a task can be in any one of the states[8] shown in Figure 2.4a:

- **Waiting** for the scheduler to decide that this task is ready to run. For example, a task which asked the scheduler to delay it for 500 ms will be in this state for that amount of time.
- **Ready to start running** but not running yet. There may be a higher-priority task which is running. As this task has not started running, no automatic variables have been initialized, and there is no activation record.

[7] It is possible to make ISRs interruptable, but this introduces many new ways to build the system wrong. Hence it is discouraged.

[8] We consider preemption by an ISR as a separate state. However, since it operates automatically and saves and restores system context, we consider it as a separate enhancement to the RTC scheduler and leave it out of our diagrams. In fact, the scheduler relies on a tick ISR to track time and move tasks between certain states.

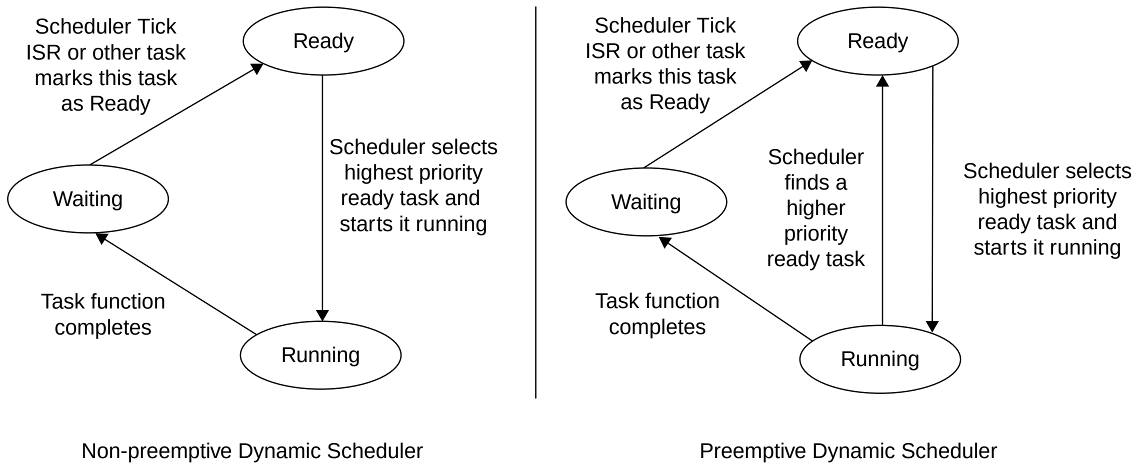

Scheduler Tick ISR or other task marks this task as Ready
Ready
Scheduler selects highest priority ready task and starts it running
Waiting
Task function completes
Running
Non-preemptive Dynamic Scheduler
Scheduler Tick ISR or other task marks this task as Ready
Ready
Scheduler finds a higher priority ready task
Scheduler selects highest priority ready task and starts it running
Waiting
Task function completes
Running
Preemptive Dynamic Scheduler

Figure 2.4 Task States and Transitions for Different Schedulers.

- **Running** on the processor. The task **runs to the completion** of the task function, at which point the scheduler resumes execution and the task is moved to the waiting state. Automatic variables have been initialized, and there is at least one activation record on the stack frame for this task. A single processor system can have only one task in this state.

Consider a task which needs to write a block of data to flash memory. After the software issues a write command to the flash memory controller, it will take a certain amount of time (e.g., 10 ms) for the controller to program the block.[9] We have two options with a non-preemptive kernel:

- Our task can use a busy wait loop until the flash block programming is complete. The task remains in the **running** state while programming. This approach delays other processing and wastes processor cycles.
- We can break the task into a state machine with a state variable indicating which state's code to execute the next time the task is executed. **State one** issues the write command, advances the state variable to two and returns from the task. **State two** checks to see if the programming is done. If it is done, the state variable is advanced to three, otherwise the state remains at two. The state then returns from the task. **State three** continues with the task's processing.

  Consider the behavior of the resulting system. The task remains in state two until it determines that the programming is done. So when the task is in state two,

[9] This write delay is inherent to flash memory hardware because it takes time to charge or discharge the floating gate in each data storage transistor.

it will yield the processor very quickly each time it is called. This allows the scheduler to execute other tasks and share the processor's time better. This approach complicates program design but is practical for smaller systems. However, it grows unwieldy for complex systems.

Allowing tasks to **preempt** each other reduces response time and simplifies application design. With preemption, each task need not be built with a run-to-completion structure. Instead, the task can yield the processor to other tasks, or it can be preempted by a higher-priority task with more urgent processing. For example, our task can tell the scheduler "I don't have anything else to do for the next 10 ms, so you can run a different task." The scheduler then will save the state of this task, and swap in the state of the next highest priority task which is ready to run. This introduces another way to move from running to waiting, as well as a way to move from running to ready. We examine these in detail next.

2.4.2 Transitions between States

We now examine the ways in which a task can move between the various states. These rules govern how the system behaves, and therefore set some ground rules for how we should design our system.

- The transition from **ready to running:**
  - In a non-preemptive system, when the scheduler is ready to run a task, it selects the highest priority ready task and moves it to the running state, typically by calling it as a subroutine.
  - In a preemptive system, when the kernel is ready to run a task, it selects the highest priority ready task and moves it to the running state by restoring its context to the processor. The task's context is a copy of the processor register values when the task was last executing, just before it was preempted by the scheduler.
- The transition from **running to waiting:**
  - In a non-preemptive system, the only way a task can move from running to waiting is if it **completes** (returns from the task function). At this point there is no more execution context for the task (return addresses, automatic variables), so there is no data to save or restore.
  - In a preemptive system, the task can yield the processor.[10] For example, it can request a delay ("Hey, kernel! Wake me up in at least 10 ms!"), or it can wait or **pend** on an event ("Hey, kernel! Wake me up when I get a message!"). This makes application programming much easier, as mentioned before. At this point there still is execution context, so the kernel must save it for later restoration.

[10] What happens if the task function finishes executing depends on the kernel. The task could move to the waiting state, or to a terminated state.

- The transition from **waiting to ready:**
  - In a non-preemptive system using a run-to-completion scheduler, the timer tick ISR sets the run flag to show the task is ready to run. Alternatively, another task can set the run flag to request for this task to run.
  - In a preemptive system, the kernel is notified that some event has occurred. For example, time delay has expired or a task has sent a message to the mailbox called foo. The kernel knows which task is waiting for this event, so it moves that particular task from the waiting state to the ready state.
- The transition from **running to ready:**
  - In a non-preemptive system this transition does not exist, as a task cannot be preempted.
  - In a preemptive system, when the kernel determines a higher priority task is ready to run, it will save the context of the currently running task, and move that task to the ready state.

2.4.3 Context Switching for Preemptive Systems

In preemptive systems, some of these state transitions require the scheduler to save a task's execution context and restore another task's context to ensure programs execute correctly. This is called **context switching** and involves accessing the processor's general-purpose registers.

Figure 2.5 shows an example of the execution context for an RL78 family system as it is executing task A in a system with two tasks (A and B). The CPU uses the program counter PC to fetch the next instruction to execute, and the stack pointer to access the top of the task's stack. The CPU's general purpose registers are used to hold the program's data and intermediate computation results. The PSW holds status bits and control bits.

In order to perform a context switch from task A to task B correctly, we must first copy all of this task-specific processor register information to a kernel data structure called a task control block (TCB) for task A. The kernel uses a TCB to keep track of data for each task which it is managing. This is shown in Figure 2.6.

Second, we must copy all of the data from task control block B into the CPU's registers. This operation is shown in Figure 2.7. Now the CPU will be able to resume execution of task B where it left off.

2.4.4 Implications

At first glance, a preemptive scheduler may seem to be the same as a non-preemptive scheduler, but with a little extra support for saving, switching, and restoring contexts. This apparently small addition in fact has a **major impact** on how programs are structured and built. A task no longer needs to run to completion. Instead, it is allowed to block and wait for an event to occur. While that task blocks (waits), the scheduler is able to work on the

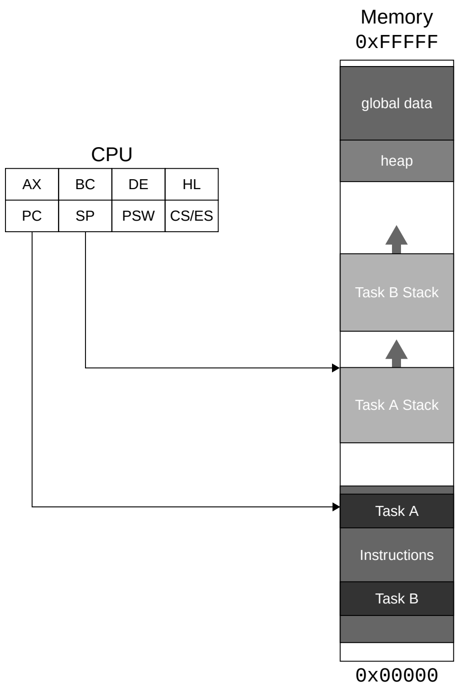


Figure 2.5 Example execution context when executing task A.

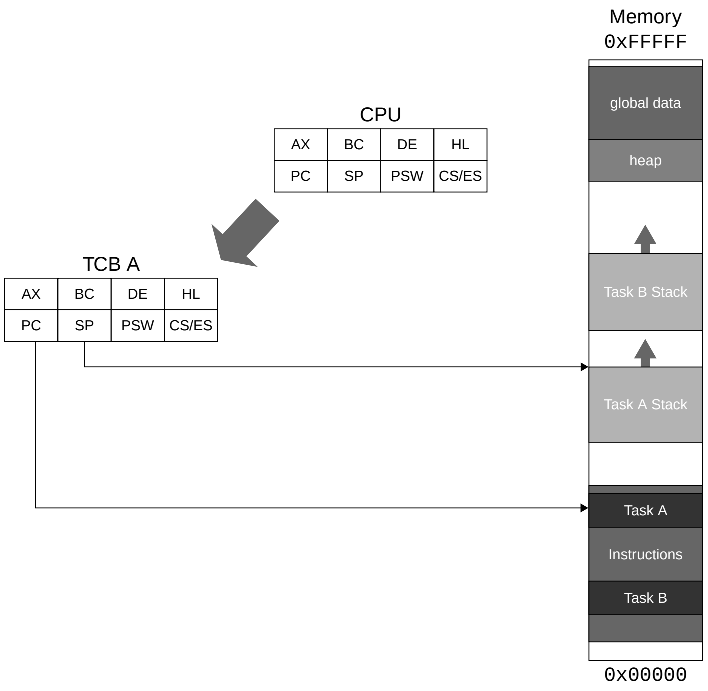


Figure 2.6 Saving task A's context from the CPU registers into task control block for task A.

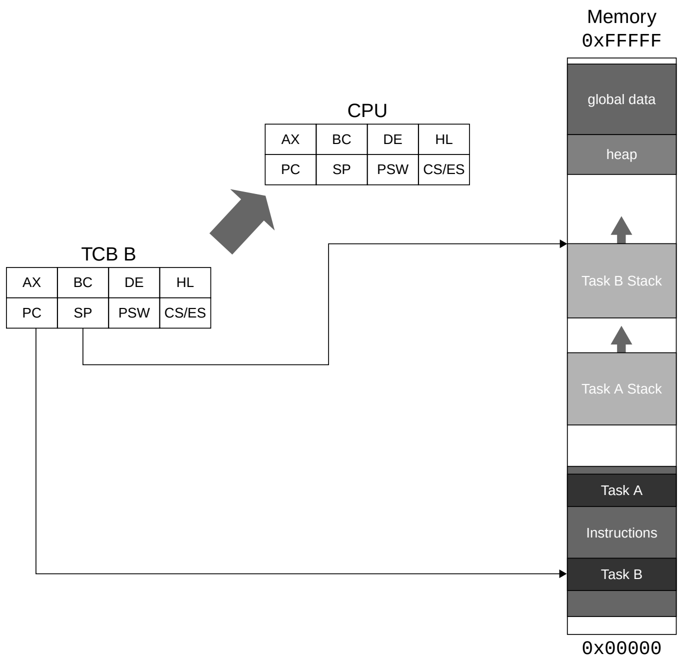


Figure 2.7 Restoring task B's context to the CPU.

next highest priority ready task. When the event occurs, the scheduler will move the task from the blocking state to the ready state, so it will have a chance to run again (when it becomes the highest priority ready task). Because the system is prioritized, it is possible that a low-priority task will never run, instead suffering starvation. All higher priority tasks must block for a given task to be able to run. This opens the door to creating event-triggered multithreaded programs, which are much easier to develop, maintain, and enhance than the equivalent run-to-completion versions.

Since event support is so valuable to (and so tightly integrated with) preemptive schedulers, we refer to **real-time kernels** which include the scheduler, event support, and additional features which build upon both. A real-time operating system (RTOS) may include additional features such as network protocol code, graphical user interfaces, file systems, and standard libraries implemented with predictable timing.

2.5 A NON-PREEMPTIVE DYNAMIC SCHEDULER

We will now examine a flexible non-preemptive scheduler for periodic and aperiodic tasks. We call it the RTC (run-to-completion) scheduler. This simple tick-based scheduler is quite flexible and offers the various benefits:

- We can configure the system to run each task with a given period (e.g., every 40 ms) measured in time **ticks.** This simplifies the creation of multi-rate systems.

- We can define task priorities, allowing us to design the system's response (which tasks are executed earlier) when there are multiple tasks ready to run.
- We can selectively enable and disable tasks.

This scheduler has three fundamental parts.

- **Task Table:** This table holds information on each task, including:
  - The address of the task's root function.
  - The period with which the task should run (e.g., 10 ticks).
  - The time delay until the next time the task should run (measured in ticks).
  - A flag indicating whether the task is ready to run.
- **Tick ISR:** Once per time tick (say each 10 milliseconds) a hardware timer triggers an interrupt. The interrupt service routine decrements the time delay **(timer)** until the next run. If this reaches zero, then the task is ready to release, so the ISR sets its **run** flag.
- **Task Dispatcher:** The other part of the scheduler is what actually runs the tasks. It is simply an infinite loop which examines each task's run flag. If it finds a task with the **run** flag set to 1, the scheduler will clear the **run** flag back to 0, execute the task, and then go back to examining the **run** flags (starting with the highest-priority task in the table).

Figure 2.8 shows a simple example of how this works with three tasks. Task 1 becomes active every twenty time intervals, and takes one time interval to complete. Task 2 is active every ten time intervals, and takes two time intervals to complete. Task 3 becomes active every five time intervals and takes one time interval to complete.

	Priority			Length			Period																			
Task 1		2			1			20																		
Task 2		1			2			10																		
Task 3		3			1			5																		
Elapsed time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Task executed						T3					T2		T3			T3					T2		T1	T3		T3
Time T1	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	20	19	18	17	16	15
Time T2	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5	4	3	2	1	10	9	8	7	6	5
Time T3	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5	4	3	2	1	5
Run T1																					W	W	R			
Run T2											R										R					
Run T3						R					W	W	R			R					W	W	W	R		R

R = Running on processor W = Ready and waiting for processor

Figure 2.8 Scheduler data and processor activity using run-to-completion dynamic scheduling.

If more than one task becomes ready simultaneously (as seen at elapsed time ten), the higher priority task is serviced first. When the higher priority task finishes, the next highest ready task is executed. This repeats until there are no ready tasks.

2.5.1 Task Table

A scheduler uses a table to store information on each task. Each task has been assigned a timer value. A task becomes active at regular intervals based on this value. This timer value is decremented each tick by the timer tick ISR. Once the timer value reaches zero, the task becomes ready to run. To reset this value after it has reached zero, an initial Timer Value variable is used to store the time at which the task has to be active. Two variables, `enabled` and `run`, are used to signal when a task is enabled and when it is ready to run. The function pointer *task indicates to the scheduler which function to perform.

The task's priority is defined by its position within this array. Entry 0 has the highest priority; whenever the scheduler needs to find a task to run, it begins at entry 0 and then works its way through the table.

Note that there is no field specifying how long this task should be allowed to run. Instead, this scheduler allows each task to run to completion—until the task function returns control to the calling function (i.e. the scheduler). The scheduler does not run again until the task function completes.

The scheduler's task table is defined next. Note that we can reduce the amount of RAM required for this table using bitfields to hold single-bit values in the structure.

```
#define MAX_TASKS 10
#define NULL ((void *)0)
typedef struct {
      int initialTimerValue;
      int timer;
      int run;
      int enabled;
      void (* task)(void);
} task_t;
task_t GBL_task_table[MAX_TASKS];
```

Before running the scheduler, the application must initialize the task table as follows:

```
void init_Task_Timers(void){
   int i;
   /* Initialize all tasks */
   for(i = 0; i < MAX_TASKS; i++){
      GBL_task_table[i].initialTimerValue = 0;
```

```
        GBL_task_table[i].run = 0;
        GBL_task_table[i].timer = 0;
        GBL_task_table[i].enabled = 0;
        GBL_task_table[i].task = NULL;
    }
}
```

2.5.2 Managing Tasks

Once the initialization is completed, tasks must be added to the task structure. The new tasks can be added before starting the scheduler or during the scheduler's execution time. When adding a task, the following must be specified: the time interval in which the task has to be active, its priority, and the function on which the task has to operate. The following code shows how adding a task is added:

```
int Add_Task(void (*task)(void), int time, int priority){
    /* Check for valid priority */
    if(priority >= MAX_TASKS || priority < 0)
        return 0;
    /* Check to see if we are overwriting an already scheduled
        task */
    if(GBL_task_table[priority].task != NULL)
        return 0;
    /* Schedule the task */
    GBL_task_table[priority].task = task;
    GBL_task_table[priority].run = 0;
    GBL_task_table[priority].timer = time;
    GBL_task_table[priority].enabled = 1;
    GBL_task_table[priority].initialTimerValue = time;
    return 1;
}
```

We can remove an existing task:

```
void removeTask(int task_number){
    GBL_task_table[task_number].task = NULL;
    GBL_task_table[task_number].timer = 0;
    GBL_task_table[task_number].initialTimerValue = 0;
    GBL_task_table[task_number].run = 0;
    GBL_task_table[task_number].enabled = 0;
}
```

We can also selectively enable or disable a task by changing its **enabled** flag. Note that this does not necessarily start the task running or stop it. Instead, it affects whether the tick ISR manages its timer variable, and whether the scheduler tries to run it.

```
void Enable_Task(int task_number){
   GBL_task_table[task_number].enabled = 1;
}
void Disable_Task(int task_number){
   GBL_task_table[task_number].enabled = 0;
}
```

We can request the scheduler to run the task by incrementing its **run** flag. This does not have any impact until Run_RTC_Scheduler reaches this task in the task table.

```
void Request_Task_Run(int task_number){
   GBL_task_table[task_number].run++;
}
```

Finally, we can change the period with which a task runs:

```
void Reschedule_Task(int task_number, int new_timer_val){
   GBL_task_table[task_number].initialTimerValue = new_timer_val;
   GBL_task_table[task_number].timer = new_timer_val;
}
```

2.5.3 Tick Timer Configuration and ISR

A run-to-completion dynamic scheduler uses a timer to help determine when tasks are ready to run (are released). A timer is set up to generate an interrupt at regular intervals, as explained in Chapter 9. Within the interrupt service routine the timer value for each task is decremented. When the timer value reaches zero, the task becomes ready to run.

```
void RTC_Tick_ISR(void){
   int i;
   for(i = 0; i < MAX_TASKS; i++){
      if(GBL_task_table[i].task != NULL) &&
      (GBL_task_table[i].enabled == 1) &&
      (GBL_task_table[i].timer > 0)){
         if(--GBL_task_table[i].timer == 0){
            GBL_task_table[i].run = 1;
            GBL_task_table[i].timer =
               GBL_task_table[i].initialTimerValue;
         }
      }
   }
}
```

2.5.4 Scheduler

The scheduler looks for ready tasks starting at the top of the table (highest priority task). It runs every ready task it finds, calling it as a function (in line 16).

```
 1. void Run_RTC_Scheduler(void){
 2.    int i;
 3.    /* Loop forever */
 4.    while(1){
 5.       /* Check each task */
 6.       for(i = 0; i < MAX_TASKS; i++){
 7.          /* check if valid task */
 8.          if(GBL_task_table[i].task != NULL){
 9.             /* check if enabled */
10.             if(GBL_task_table[i].enabled == 1){
11.                /* check if ready to run */
12.                if(GBL_task_table[i].run >= 1){
13.                   /* Update the run flag */
14.                   GBL_task_table[i].run--;
15.                   /* Run the task */
16.                   GBL_task_table[i].task();
17.                   /* break out of loop to start at entry 0 */
18.                   break;
19.                }
20.             }
21.          }
22.       }
23.    }
24. }
```

2.6 BUILDING A MULTITHREADED APPLICATION USING A SCHEDULER

In this section we examine how to create a multithreaded application. We create two versions side-by-side, using both preemptive scheduler and non-preemptive schedulers. For the preemptive approach we use the real-time multitasking kernel μC/OS-III (Labrosse &

TABLE 2.1 Common Scheduler Functions

CATEGORY	µC/OS-III	RTC
Task Management	OSTaskCreate	AddTask
	OSTaskSuspend	EnableTask
	OSTaskResume	DisableTask
		RescheduleTask
Time Management	OSTimeDly	
	OSTimeDlyHMSM	
	OSTimeDlyResume	
Resource Management	OSMutexCreate	
	OSMutexPend	
	OSMutexPost	
Synchronization	OSFlagCreate	RequestTaskRun
	OSFlagPend	
	OSFlagPost	
	OSFlagPendGetFlagsRdy	
	OSSemCreate	
	OSSemPend, OSTaskSemPend	
	OSSemPost, OSTaskSemPost	
	OSPendMulti	
Message Passing	OSQCreate	
	OSQPend, OSTaskQPend	
	OSQPost, OSTaskQPost	

Kowalski, 2010), and for the non-preemptive approach we use the run-to-completion (RTC) scheduler.

Table 2.1 shows commonly used functions for the µC/OS-III and RTC schedulers. This is not a complete list, but is provided for reference.

2.6.1 Basic Task Concepts and Creation

We use tasks (threads) and ISRs to build our system. Many tasks may need to run periodically, such as the LED flasher (the embedded "Hello, World" equivalent) shown in Figure 2.9.

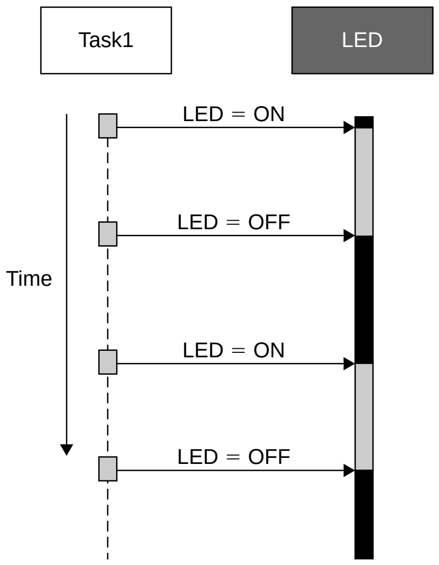


Figure 2.9 Sequence diagram of Task 1 executing periodically and toggling a LED.

2.6.1.1 Non-Preemptive

The RTC scheduler provides periodic task triggering support to make the task function execute at the right times. The Add_Task function call includes a period parameter of 250 (ms in this case) used to set this value. We also need to specify the task priority (1), noting that a lower number indicates higher priority.

```
void main(void){
   ENABLE_LEDS;
   init_Task_Timers();
   Add_Task(Task1,250,1);
   Init_RTC_Scheduler();
   Run_RTC_Scheduler();
}
```

In this non-preemptive scheduling approach, a task function is started once each time it needs to run. Because of this, it uses a run-to-completion structure in each task, so we must ensure that there are **no infinite loops** in the task code.

We also need to declare the state variable as static to make it retain its value from one task execution to the next. Remember that the stack frame holding automatic variables is destroyed upon exiting the function, so the previous value would be lost without this change.

The resulting task is shown below, and is quite simple:

```
void Task1(void)
{
   static char state = 0;
   RED_LED = state;
   state = 1-state;
}
```

2.6.1.2 Preemptive

Next we examine the solution using a preemptive scheduler. An application built on μC/OS-III requires more sophisticated OS configuration due to its flexibility. We step through the standard application skeleton code provided by Micrium with μC/OS-III. The main function performs initialization of the CPU, the BSP (board support package), and the operating system before creating the first task (App_TaskStart) and then starting the scheduler. We will discuss the parameters used in OSTaskCreate shortly.

```
int main (void)
{
   OS_ERR   os_err;
   CPU_Init(); /* Initialize the uC/CPU services */
   BSP_PreInit();
   OSInit(&os_err);   /* Init uC/OS-III. */
   OSTaskCreate((OS_TCB   *)&App_TaskStart_TCB,
         (CPU_CHAR    *)"Start",
         (OS_TASK_PTR  )App_TaskStart,
         (void        *)0,
         (OS_PRIO      )APP_CFG_TASK_START_PRIO,
         (CPU_STK     *)&App_TaskStart_Stk[0],
         (CPU_STK_SIZE )APP_CFG_TASK_START_STK_SIZE_LIMIT,
         (CPU_STK_SIZE )APP_CFG_TASK_START_STK_SIZE,
         (OS_MSG_QTY   )0u,
         (OS_TICK      )0u,
         (void        *)0,
         (OS_OPT       )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
         (OS_ERR      *)&os_err);
   OSStart(&os_err);   /* Start multitasking (i.e. give control to uC/OS-III).*/
   return (0);
}
```

App_TaskStart then performs additional initialization and calls functions to create the application's tasks and kernel objects (e.g., mutexes, semaphores, events, queues, etc.)

```
static void App_TaskStart (void *p_arg)
{
   OS_ERR  err;
   BSP_PostInit();/* Initialize BSP functions          */
   App_TaskCreate();/* Create Application tasks */
   App_ObjCreate(); /* Create Application kernel objects  */
   while (DEF_TRUE) { /* Task body, always written as an infinite
   loop. */
       BSP_LED_Toggle(1);
       OSTimeDlyHMSM(0u, 0u, 0u, 500u,
          OS_OPT_TIME_HMSM_STRICT,
          &err);
   }
}
```

App_TaskCreate then registers the tasks with the scheduler. Each task needs two supporting data structures: a task control block (of type OS_TCB) and a function call stack (of type CPU_STK). Pointers to these data structures are then specified to the scheduler when calling OSTaskCreate (e.g., &App_Task1_TCB, &App_Task1_Stk[0]). Each task is described as a variety of options controlled by parameters. These options include priority, stack size, stack checking, and stack initialization. Each created task is in the ready-to-run state.

Like many other preemptive schedulers, μC/OS-III does not provide support for periodic tasks so we will need to build it into the task (or in a separate dispatcher function, not covered here). As a result, the initialization code does not include any period information.

```
static  OS_TCB App_TaskStart_TCB;
static  OS_TCB App_Task1_TCB;

static CPU_STK App_TaskStart_Stk[APP_CFG_TASK_START_STK_SIZE];
static CPU_STK App_Task1_Stk[APP_CFG_TASK1_STK_SIZE];

static void    App_TaskCreate (void)
{
   OS_ERR os_err;

   OSTaskCreate((OS_TCB *)&App_Task1_TCB,/* Create task 1 */
   (CPU_CHAR            *)"Task1",
   (OS_TASK_PTR          )App_Task1,
   (void                *)0,
   (OS_PRIO              )APP_CFG_TASK1_PRIO,
   (CPU_STK             *)&App_Task1_Stk[0],
   (CPU_STK_SIZE         )APP_CFG_TASK1_STK_SIZE_LIMIT,
   (CPU_STK_SIZE         )APP_CFG_TASK1_STK_SIZE,
   (OS_MSG_QTY           )0u,
   (OS_TICK              )0u,
   (void                *)0,
   (OS_OPT               )(OS_OPT_TASK_STK_CHK |
   OS_OPT_TASK_STK_CLR),
   (OS_ERR              *)&os_err);
}
```

In the preemptive scheduling approach, a task function is started only once. In order to provide repeated executions, we **place the task code in an infinite loop** and then **yield the processor** after completing one iteration. If we do not yield the processor, then no lower-priority tasks will be able to execute, as this task will use all the processor time it can (as allowed by higher-priority tasks). The loop must contain at least one OS call (e.g., yield, pend, delay).

Here we use OSTimeDly to delay further execution of this task for the specified number of time ticks. Note that this is not a precise timing mechanism. There are various ways in which the task can be delayed such that the OSTimeDly call slips one or more time ticks later, reducing the task frequency. Some kernels provide true periodic task scheduling—in µC/OS-III the OS_OPT_TIME_PERIODIC option must be passed to OSTimeDly(). Otherwise we can create a task to trigger periodic tasks using the synchronization methods described later.

Note that the task's state variable is automatic and does not need to be declared as static. This is because the stack frame is never destroyed, since the function Task1 never completes and exits.

```
void Task1(void*data)
{
   OS_ERR error;
   char state = 0;
   for (;;)
   {
      RED_LED = state;
      state = 1-state;
      OSTimeDly(MSEC_TO_TICKS(TASK1_PERIOD_MSEC),
      OS_OPT_TIME_PERIODIC, &error);
   }
}
```

2.6.2 Handling Long Tasks

Some tasks may take so much time to execute that system responsiveness suffers. The execution may be due to a large number of computations or waiting for an operation to complete. For example, consider a task which writes data to flash memory, shown below. This operation could take a long time (e.g., 3 ms to write a page of data to flash memory).

```
1. void Task_Log_Data_To_Flash(void) {
2.    Compute_Data(data);
3.    Program_Flash(data);
4.    while (!Flash_Done()) {
5.    }
6.    if (flash_result==ERROR) {
7.       Handle_Flash_Error();
8.    }
9. }
```

2.6.2.1 Non-Preemptive

With this approach the task must run to completion but we wish to eliminate or reduce busy waiting. To do this we will split the task into two. The first task will do all the work up to busy waiting and then request for the scheduler to run the second task at some future point in time. The second task will perform the completion test. If the flash write is done, then the task is ready to proceed and complete the code (lines 9 and 10). However, if the write is not done, then the task will return almost immediately, yielding the processor but requesting it again in the future. The scheduler will now be able to execute higher-priority tasks, and eventually will come back to this second task.

```
 1. void Task_Log_Data_To_Flash_1(void) {
 2.    Compute_Data(data);
 3.    Program_Flash(data);
 4.    //Ask scheduler to run task 2
 5. }
 6.
 7. void Task_Log_Data_To_Flash_2(void) {
 8.    if (Flash_Done()) {
 9.       if (flash_result==ERROR) {
10.          Handle_Flash_Error();
11.       }
12.    } else {
13.       //Ask scheduler to run task 2
14.    }
15. }
```

2.6.2.2 Preemptive

```
 1. void Task_Log_Data_To_Flash(void) {
 2.    Compute_Data(data);
 3.    Program_Flash(data);
 4.    while (!Flash_Done()) {
 5.       OSTimeDly(3, OS_OPT_TIME_DLY, &error);
 6.       //try again three ticks later
 7.    }
 8.    if (flash_result==ERROR) {
 9.       Handle_Flash_Error();
10.    }
11. }
```

A kernel typically provides a way for a task to explicitly yield control to the scheduler (and hence other tasks). Rather than spin in a busy-wait loop (line 4) until the flash programming is done (indicated by Flash_Done() returning 1), we insert an OSTimeDly(1) call at line 5. This tells the kernel that the task would like to yield control of the processor, and furthermore would like to be placed back into the ready queue after three scheduler ticks have passed. At some point in the future, the scheduler will resume this task's execution, at which point the task will once again check to see if Flash_Done() is true or not. Eventually it will be true and the task will then continue on with the code at line eight following the loop.

2.6.3 Synchronizing with Other Tasks and ISRs

Often tasks need to synchronize with other tasks. For example, we may want Task 1 to be able to signal Task 2 that it should run, as in the non-preemptive flash page write example just discussed. Or perhaps we want an ISR to notify a task that some event has occurred (e.g., a set of analog to digital conversions has completed), and the task should execute (e.g., process the converted data).

2.6.3.1 Non-Preemptive

Task 1 can request for the scheduler to run Task 2 by setting or incrementing Task 2′s Run flag in the scheduler table. Typically this is done with a scheduler API call.

```
void Task1(void)
{
    ...
    Request_Task_Run(TASK2_NUM);
    ...
}
```

2.6.3.2 Preemptive

A kernel typically offers multiple primitives which can be used for synchronization. µC/OS-III features semaphores, message queues, and event flags.

2.6.3.2.1 Synchronization with Semaphores Figure 2.10 shows the desired system behavior. We would like Task1 to run periodically. Each time it runs it should check to see if

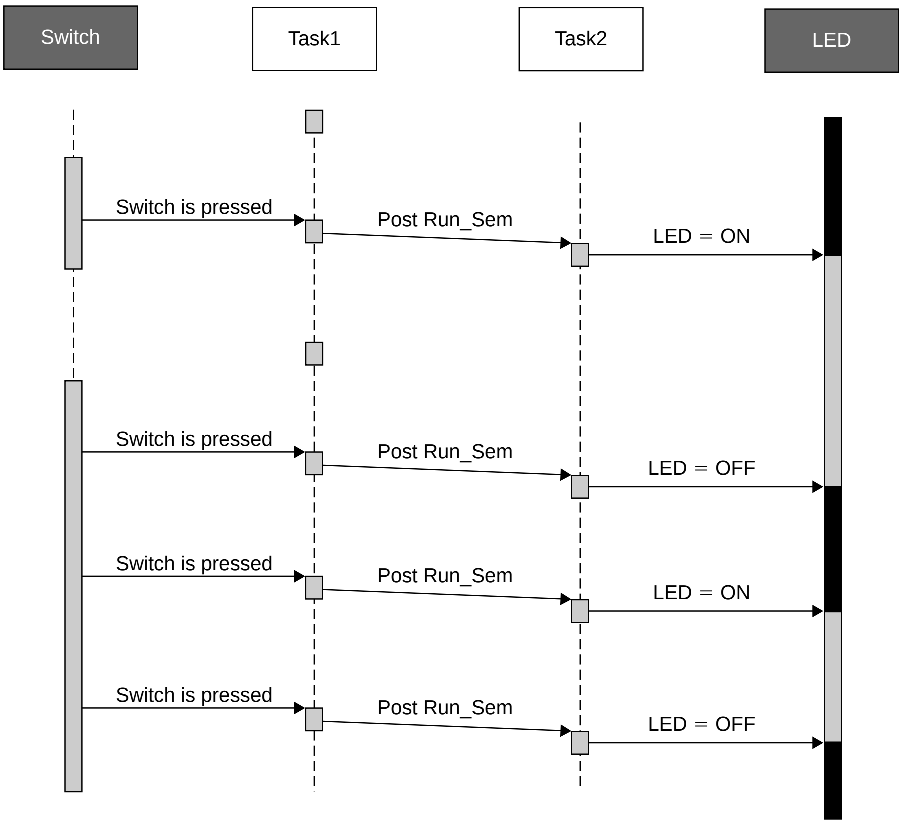


Figure 2.10 Sequence diagram of Task1 triggering Task2 with semaphore.

switch S1 is pressed, perhaps providing debouncing support as well. If the switch is pressed, Task1 should signal Task2 by using the semaphore Run_Sem. Task2 will wait for the semaphore Run_Sem to be signaled. When it is, then Task2 can run, toggling the LED and waiting for the next semaphore signaling.

Details of the code are shown below. Note that the semaphore needs to be created and initialized by the kernel, as shown in line 3. Lines 4 through 6 handle error conditions.

```
 1. OS_SEM * Run_Sem;
 2.
 3. void TaskStartup() {
 4.    ...
 5.    OSSemCreate((OS_SEM *)&Run_Sem, ... &error);
 6.    if (error != OS_ERR_NONE) {
 7.       //error handling
 8.    }
 9.    ...
10. }
11. void Task1(void*data)
12. {
13.    char state = 0;
14.    for (;;)
15.    {
16.       if (!S1) { //if switch is pressed
17.          OSSemPost(Run_Sem); //signal the event has happened
18.       }
19.       OSTimeDly(MSEC_TO_TICKS(TASK1_PERIOD_MSEC));
20.    }
21. }
22. void Task2(void*data)
23. {
24.    char state = 0;
25.    INT8U err=OS_NO_ERR;
26.    for (;;)
27.    {
28.       OSSemPend(Run_Sem, TIMEOUT_NEVER, &err);//await event
29.       if (err == OS_NO_ERR) { //We got the semaphore
30.          YLW_LED = state;
31.          state = 1-state;
32.       }
33.    }
34. }
```

Semaphores can be created by tasks. However, μC/OS-III includes a semaphore in each task because they are a common and useful synchronization mechanism. The names of the functions accessing the task semaphores use TaskSem rather than Sem.

2.6.3.2.2 Synchronization with Event Flags: Event flags enable more flexible synchronization than the one-to-one pattern just described.

First, a task can wait for (pend on) multiple events. For example, a task can be triggered when **any** of the specified events occurs (a logical **or**). Alternatively, a task can be triggered when **all** of the specified events have occurred (a logical **and**). The user creates an event group which is a kernel object with a set of flags stored in a bit field.

Second, multiple tasks can pend on the same event flag. If the event flag is posted, all of the pending tasks will be notified. This allows an event to have multiple results.

A call to OSFlagPost includes these parameters:

- A pointer to the event flag group
- A bitmask indicating which flag to post
- Whether to set or clear the flag
- A pointer to a result code

A call to OSFlagPend includes various parameters:

- A pointer to the event flag group
- A bitmask indicating which flags to monitor
- Whether to wait for all or any events
- Whether to wait for flags to be set or cleared
- A time out value
- A pointer to a result code

2.6.4 Passing Messages among Tasks

A task may need to send data (rather than just an event notification) to another task. Kernels may provide message queues and mailboxes to do this. A mailbox holds one item of data, while a queue can buffer multiple items of data.

2.6.4.1 Non-Preemptive

The RTC scheduler shown does not provide any message-passing support, although it would be possible to add it.

2.6.4.2 Preemptive

μC/OS-III provides support for message passing through its message queues. Message queues can be created by tasks. However, μC/OS-III includes one in each task because they are a common and useful communication mechanism. The names of the functions accessing the task queues use TaskQ rather than Q, as shown in Table 2.1.

A message has three components:

- A pointer to a user defined data object.
- A variable indicating the size of the data object.
- A timestamp of when the message was sent.

There are several message queue operations possible:

- A program can create a queue using the OSQCreate function.
- A task or an ISR can enqueue a message using the OSQPost and OSTaskQPost functions.
- A task can dequeue a message (potentially blocking until it is available) with OSQPend() and OSTaskQPend. The task can specify a time-out value; if no message is received before this time has passed, then the pend function will return with an error result.

Let's consider an example. We would like App_Task1 to run periodically. Each time it runs it should check to see if switch S1 is pressed. If it is, it should signal this by sending a message SWITCH1_PRESSED to App_Task2's internal task queue. There may be other switches present, but App_Task2 is only interested in S1. App_Task2 will block until its task queue is loaded with a message. When it is, then App_Task2 can run, process the received message, and then wait for the next message in the queue.

```
static void App_Task1 (void * p_arg)
{
   OS_ERR os_err;
   CPU_TS ts;

   p_arg = p_arg;
   while (1) {
      if (Switch1Pressed() {
         OSTaskQPost ((OS_TCB *)&App2_TCB,
            (void *) SWITCH1_PRESSED,
            (OS_MSG_SIZE) sizeof(void *),
            (OS_OPT )OS_OPT_POST_FIFO,
            (OS_ERR*) &os_err);
            ... //do other work
            //delay until the next time to run the task
         OSTimeDlyHMSM(0u, 0u, 0u, 150u,
         OS_OPT_TIME_HMSM_STRICT,
         &os_err);
      }
   }

   static void App_Task2 (void * p_arg)
   {
      OS_ERR os_err;
      void * p_msg;
      OS_MSG_SIZE msg_size;

      while (1) {
         p_msg = OSTaskQPend ((OS_TICK) 0,
            (OS_OPT )OS_OPT_PEND_BLOCKING,
            (OS_MSG_SIZE) &msg_size,
            (CPU_TS *)&ts,
            (OS_ERR*) &os_err);
         //process the received message
         ...
      }
   }

```

Note that in this case we typecast our message SWITCH1_PRESSED into a void pointer. We did this because the message to send was small enough to fit into a pointer. In other cases we might need to send longer data, in which case we actually need to use the argument as a pointer to the data. We must then be careful about the lifetime of the data to be sent. Automatic data is located within the declaring function's stack frame and will be destroyed when the function returns. Instead we need to use static data (e.g. a global) or dynamically-allocated data. uC/OS-III provides dynamic memory allocation through its OSMemGet and OSMemPut functions.

2.6.5 Sharing Objects among Tasks

Sometimes tasks may need to share an object such as a variable, a data structure, or a hardware peripheral. Preemption among tasks introduces a vulnerability to data race conditions which does not exist in systems built on run-to-completion schedulers. Now a task can become as bug-prone and difficult to debug as an ISR! The system can fail in new ways when:

- Multiple tasks or ISRs share an object,[11] or
- Multiple instances of a function can execute concurrently.

In order to prevent these failures we need to be careful when designing our system.

2.6.5.1 Shared Objects

If an object is accessed by code which can be interrupted (is not *atomic*), then there is a risk of data corruption. *Atomic* code is the smallest part of a program that executes without interruption. Generally a single machine instruction is atomic,[12] but sequences of instructions are not atomic unless interrupts are disabled.

Consider an example where task A starts modifying object O. Task B preempts it before it finishes. At this point in time object O is corrupted, as it is only partially updated. If task B needs to read or write O, the computation results will be incorrect and the system will likely fail.

```
unsigned time_minutes, time_seconds;
void task1 (void){
   time_seconds++;
   if(time_seconds >= 60){
      time_minutes++;
      time_seconds = 0;
   }
}
void task2 (void){
   unsigned elapsed_sec;
   elapsed_seconds = time_minutes * 60 + time_seconds;
}
```

Here is a more specific example. Our shared object is a pair of variables which measure the current time in minutes and seconds. Task1 runs once per second to increment the seconds,

[11] Hardware registers which change outside of the program's control also introduce problems but we do not discuss them further here.

[12] Some instruction sets (but not the RL78) have long instructions (e.g., string copy, block move) which can be interrupted, in which case those instructions are not atomic.

and possibly the minutes as well. Task2 calculates how many total seconds have elapsed since time zero. There are data races possible:

- If task1 is preempted between lines 4 and 5 or lines 5 and 6, then when task2 runs it will only have a partially updated version of the current time, and elapsed seconds will be incorrect.
- If task2 is preempted during line 11, then it is possible that timeinutes is read **before** task1 updates it and time_seconds is read **after** task 1 updates it. Again, this leads to a corrupted elapsed_seconds value.

2.6.5.2 Function Reentrancy

Another type of shared data problem comes with the use of non-reentrant functions. In this case, the problem arises from multiple instances of the **same function** accessing the same object. Consider the following example:

```
 1. void task1 ( ){
 2.    . . . . . . . . . . . . . . .
 3.    swap(&x, &y);
 4.    . . . . . . . . . . . . . . .
 5. }
 6. void task2 ( ){
 7.    . . . . . . . . . . . . . . .
 8.    swap(&p, &q);
 9.    . . . . . . . . . . . . . . .
10. }
11. int Temp;
12. void swap (*i, *j ){
13.    Temp = *j;
14.    *j = *i;
15.    *i = Temp;
16. }
```

Suppose task1 is running and calls the swap function. After line 13 is executed, task2 becomes ready. If task2 has a higher priority, task1 is suspended and task2 is serviced. Later, task1 resumes to line 14. Since Temp is a shared variable, it is not stored in the TASK subroutine shared data stack. When task1 line 15 is executed, variable *x* (of task1 pointed by variable pointer *i*) gets the wrong value. Such function executions should not be suspended in between or shared by more than one task. Such functions are called **non-reentrant.** The code which can have multiple simultaneous, interleaved, or nested invocations which will not interfere with each other is called reentrant code. These types of code are important for parallel processing, recursive functions, or subroutines, and for interrupt handling. An example of a reentrant code is as follows:

```
void swap (*i, *j ){
   static int Temp;
   Temp = *j;
   *j = *i;
   *i = Temp;
}
```

Since the variable `Temp` is declared within the function, if any other task interrupts the execution of the `swap` function, the variable `Temp` will be stored in the corresponding task's stack and will be retrieved when the task resumes its function. In most cases, especially in a multi-processing environment, the non-reentrant functions should be eliminated. A function can be checked for its reentrancy based on these three rules:

- A reentrant function may not use variables in a non-atomic way unless they are stored on the stack of the calling task or are the private variables of that task.
- A reentrant function may not call other functions which are not reentrant.
- A reentrant function may not use the hardware in a non-atomic way.

When writing software in a system with task preemption or ISRs we need to be careful to never call non-reentrant functions, whether directly or indirectly.

2.6.5.3 High-Level Languages and Atomicity

We can identify **some but not all** non-atomic operations by examining high-level source code. Since the processor executes machine code rather than a high-level language such as C or Java, we can't identify all possible non-atomic operations just by examining the C source code. Something may seem atomic in C but actually be implemented by multiple machine instructions. We need to examine the assembly code to know for sure. Let's examine the following function and determine whether it is atomic or not:

```
static int event_count;
void event_counter (void){
   ++event_count;
}
```

Example 1 in assembly language (not RL78):

```
MOV.L #0000100CH, R4
MOV.L [R4], R5
```

```
3. ADD #1H, R5
4. MOV.L R5, [R4]
5. RTS
```

Consider example 1, and then apply the first rule. Does it use shared variable `event_count` in an atomic way? The `++event_count` operation is not atomic, and that single line of C code is implemented with three lines of assembly code (lines two through four). The processor loads R4 with a pointer to event_count, copies the value of event_count into register R5, adds 1 to R5, and then stores it back into memory. Hence, example 1 **is not atomic and not reentrant.**

However, what if the processor instruction set supports in-place memory operations? In that case, the assembly code could look like this:

Example 1 in assembly language, compiled for a different processor architecture:

```
1. MOV.L #0000100CH, A0
2. ADD #1H, [A0]
3. RTS
```

This code **is atomic,** since there is only one instruction needed to update the value of the event count. Instruction 1 is only loading a pointer to the event count, so interrupting between 1 and 2 does not cause a problem. Hence it **is reentrant.**

The RL78 architecture supports modifications in memory, so the compiler can generate code which takes a single instruction to perform the increment. For example, this instruction is **atomic:**

```
1. INCW      N:int_count
```

Now consider a slightly different example:

```
1. void add_sum (int *j){
2.    ++(*j);
3.    DisplayString(LCDLINE1, Int_to_ascii(*j);
4. }
```

Even though `line 2` in this example is not atomic, the `variable *j` is task's private variable, hence rule 1 is not breached. But consider `line 3.` Is the function `DisplayString` reentrant? That depends on the code of `DisplayString,` which depends on the user. Unless we are sure that the `DisplayString` function is reentrant (and do this recursively for any functions which may be called directly or indirectly by `DisplayString`), example 2 is considered to be non-reentrant. So every time a user designs a function, he or she needs to make sure the function is reentrant to avoid errors.

2.6.5.4 Shared Object Solutions and Protection

In the previous section, we discussed the problems of using shared objects in a preemptive environment. In this section we shall study some methods to protect the shared objects. The solutions provided in this section may not be ideal for all applications. The user must judge which solution may work best for the application. The Resource Management chapter of the uC/OS-III manual (Labrosse & Kowalski, 2010) provides additional explanations, insight, and implementation details.

2.6.5.4.1 Disable Interrupts One of the easiest methods is to disable the interrupts during the critical section of the task. Disabling the interrupts may not take more than one machine cycle to execute, but will increase the worst case response time of all other code, including other interrupt service routines. Once the critical section, or shared variable section, of the code is executed, the interrupt masking must be restored to its previous state (either enabled or disabled). The user must be cautious while disabling or enabling interrupts, because if interrupts are disabled for too long the system may fail to meet the timing requirements. Consult the MCU programming manual to find out how to disable and restore the interrupt masking state. A simple example of disabling interrupts is as follows:

```
#define TRUE 1
#define FALSE 0
static int error;
static int error_count;
void error_counter ( ){
   if(error == TRUE){
      SAVE_INT_STATE;
      DISABLE_INTS;
      error_count++;
      error = FALSE;
      RESTORE_INT_STATE;
   }
}
```

Disabling and restoring the interrupt masking state requires only one or a few machine cycles. Disabling interrupts must take place only at critical sections to avoid increasing response time excessively. Also, while restoring the interrupt masking state the user must keep in mind the need to enable only those interrupts that were active (enabled) before they were disabled. Determining the interrupt masking status can be achieved by referring to the **interrupt mask register.** The interrupt mask register keeps track of which interrupts are enabled and disabled.

2.6.5.4.2 Use a Lock Another solution is to associate a lock variable with each shared object. The lock variable is declared globally. If a function uses the shared object then it sets the lock variable, and once it has finished it resets the lock variable. Every function must test the lock variable before accessing the shared object. If the lock variable is already set, the task should inform the scheduler to be rescheduled once the object becomes available. Since only one variable has to be checked every time before accessing the data, using lock variables simplifies the data structure and I/O devices access. Consider the following example for using a lock:

```
 1. unsigned int var;
 2. char lock_var;
 3. void task_var ( ){
 4.    unsigned int sum;
 5.    if(lock_var == 0){
 6.       lock_var = 1;
 7.       var = var + sum;
 8.       lock_var = 0;
 9.    else {
10.       /* message to scheduler to check var and reschedule */
11.    }
12. }
```

Since it takes more than one clock cycle to check whether a variable is available and use a lock on it, the interrupts must be disabled during lines 5 and 6 of the code. Once again, when the lock is released (in line 8) the interrupts should be disabled since locking and releasing the variable is a critical part of the code. Interrupts should be enabled whenever possible to lower the interrupt service response time. If the variable is not available, the scheduler is informed about the lock and the task goes into a waiting state.

The Renesas RL78 processor family includes a Branch if True and Clear (BTCLR) instruction which can perform a test, clear, and branch in one atomic machine instruction, and therefore does not require an interrupt disable/enable lock around semaphore usage. However, the compiler is not likely to use this instruction, so the programmer must write the assembly code with the BTCLR instruction and other instructions as needed. This limits code portability significantly.

Another challenge with this approach is determining what to do in line 10 if there is no scheduler support. There may be no easy way to tell the scheduler to reschedule this task when the lock variable becomes available again.

2.6.5.4.3 Kernel-Provided Mutex Most operating systems provide locks for shared variables through the use of mutexes.

A mutex is based on a binary semaphore but has additional features. Binary semaphores and therefore mutexes can take two values, 0 or 1. A mutex used for protecting a resource is initialized with the value 1, to indicate the resource is initially available. At this point no task is waiting for the mutex. Once a task requires a data, the task performs a wait operation on the mutex. The OS checks the value of the mutex; for example, if it is available (the value is non-zero), the OS changes the value of the mutex to zero and assigns the mutex to the task. If the value is zero during the wait operation, the task that requested the wait operation is placed on the mutex's waiting list. Once the mutex is released and becomes available, the OS grants it to the highest priority task waiting for it.

A task can ask to wait on a mutex, potentially specifying a time limit for waiting. If the time expires, the kernel returns an error code to the mutex-seeking function for the appropriate response. On the other hand, if the function has obtained the mutex, it can then complete its operation using the shared resource and perform a signal operation, announcing that the mutex is free. The OS checks if any other task is waiting for the mutex. If so, that task is notified that it has obtained the mutex (without changing it's value). On the other hand, if no task is waiting for the mutex, its value is incremented to one. The wait operation is also referred to as Take or Pend, and signal operation is referred to as Release or Post.

μC/OS-III offers both mutexes and semaphores. The difference between the two is that semaphores do not provide priority inheritance, leading to possible priority inversion and much longer (possibly unbounded) response times. Mutexes provide priority inheritance, greatly reducing the worst case response time. Systems with deadlines should use mutexes rather than semaphores when sharing resources.

The following example shows how the mutex LCD_Mutex is used to ensure only one task can access the LCD at a time. Each task must obtain the mutex through an OSMutexPend operation (lines 16 and 34) before using the LCD (lines 21 and 50). When the task is done with the LCD, it must release the mutex with an OSMutexPost operation (lines 32 and 63).

```
 1. static OS_MUTEX LCD_Mutex;
 2.
 3. static void App_ObjCreate (void)
 4. {
 5.    OS_ERR os_err;
 6.    OSMutexCreate((OS_MUTEX *)&LCD_Mutex,
 7.       (CPU_CHAR *)”My LCD Mutex”,
 8.       (OS_ERR *)&os_err);
 9.    ... //create other kernel objects
10. }
```

```

static void App_Task1 (void * p_arg)
{
   OS_ERR os_err;
   CPU_TS ts;

   p_arg = p_arg;
   while (1) {
      ... //do work before using the LCD
      //get mutex for LCD
      OSMutexPend((OS_MUTEX *)&LCD_Mutex,
         (OS_TICK) 0,
         (OS_OPT) OS_OPT_PEND_BLOCKING,
         (CPU_TS *)&ts,
         (OS_ERR*) &os_err);

      //access shared resource
      GlyphSetXY (G_lcd, 30, 24);
      GlyphString(G_lcd, “Task 1”, 6);

      //release mutex
      OSMutexPost((OS_MUTEX *)&LCD_Mutex,
         (OS_OPT) OS_OPT_POST_NONE,
         (OS_ERR*) &os_err);

      OSTimeDlyHMSM(0u, 0u, 0u, 150u,
      OS_OPT_TIME_HMSM_STRICT,
      &os_err);
   }
}
static void App_Task3 (void * p_arg)
{
   OS_ERR os_err;
   uint8_t y;
   CPU_TS ts;

   p_arg = p_arg;
   while (1) {
      //get mutex
      OSMutexPend((OS_MUTEX *)&LCD_Mutex,
         (OS_TICK) 0,
         (OS_OPT) OS_OPT_PEND_BLOCKING,
         (CPU_TS *)&ts,
         (OS_ERR*) &os_err);

      //access shared resource
      for (y = 16; y < 64; y += 8) {
      GlyphSetXY (G_lcd, 0, y);
      GlyphString(G_lcd, “—————————“, 18);
      }

      //release mutex
      OSMutexPost((OS_MUTEX *)&LCD_Mutex,
         (OS_OPT) OS_OPT_POST_NONE,
         (OS_ERR*) &os_err);

      OSTimeDlyHMSM(0u, 0u, 0u, 490u,
      OS_OPT_TIME_HMSM_STRICT,
      &os_err);
   }
}
```

2.6.5.4.4 Kernel-Provided Messages We have seen that a kernel may provide other mechanisms besides semaphores for allowing tasks to communicate, such as message queues. It may be possible to structure your program to use messages to pass information rather than sharing data objects directly. We leave further discussion of this approach to existing books and articles on real-time kernels.

2.6.5.4.5 Disable Task Switching If no other method seems to work, one unattractive option is to disable the scheduler. If the scheduler is disabled, the task switching does not take place and the critical sections or shared data can be protected by other tasks. This method is counter-productive; disabling the scheduler increases response times and makes analysis much more difficult. This is considered bad practice and must be properly justified; hence, consider this method as a last resort.

2.7 RECAP

In this chapter we have seen how the responsiveness of a program with multiple tasks depends on the ordering of the tasks, their prioritization, and whether preemption can occur.

We have seen how the scheduler manages task state based on system behavior, and have examined how to create applications using two different types of schedulers. Finally we have examined how to protect shared data in a preemptive system.

2.8 BIBLIOGRAPHY

Labrosse, J., & Kowalski, F. (2010). *MicroC/OS-III: The Real-Time Kernel*. Weston, FL: Micrium Press. ISBN 978-0-9823375-7-8.

Chapter 3

Real-Time Methods

3.1 LEARNING OBJECTIVES

Most embedded systems have multiple independent tasks running at the same time. Which activity should the microprocessor perform first? This decision determines how responsive the system is, which then affects how fast a processor we must use, how much time we have for running intensive control algorithms, how much energy we can save, and many other factors. In this chapter we will discuss different ways to schedule a system's tasks and the implications for performance and related issues.

3.2 FOUNDATIONS FOR RESPONSE TIME AND SCHEDULABILITY ANALYSIS

In the previous chapter we have seen how allowing (1) dynamic scheduling and (2) preemption of tasks improves a system's responsiveness dramatically. In this section we will introduce the basic analytical methods which enable us to predict the timing behavior of the resulting **real-time** systems accurately. There is an abundance of research papers on real-time scheduling theory; three survey papers stand out for their clarity and context and should be consulted as starting points (Audsley, Burns, Davis, Tindell, & Wellings, 1995; George, Rivierre, & Spuri, 1996; Sha, et al., 2004).

We are mainly concerned with three aspects of a real-time system's behavior:

- How should we **assign priorities to tasks** to get the best performance?
- How long will it take the processor to finish executing all the instructions of a particular task, given that other tasks may disrupt this timing? This is called the **response time.**
- If each task has a deadline, will the system always meet all deadlines, even under the worst case situation? A system which will always meet all deadlines is called **schedulable.** A **feasibility test** will let us calculate if the system is schedulable or not.

3.2.1 Assumptions and Task Model

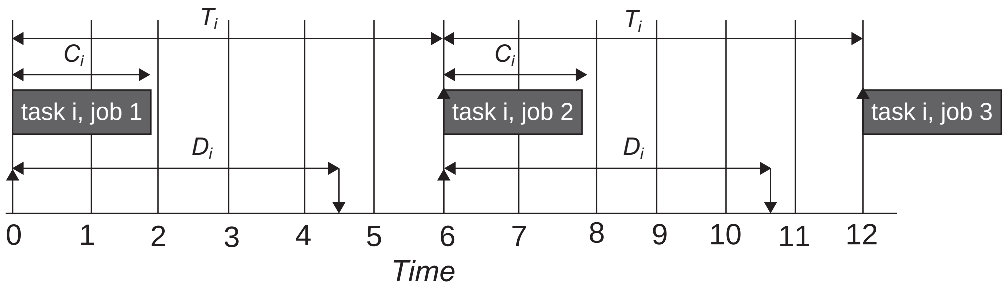


Figure 3.1 Real-time task model.

We model the computational workload according to the following assumptions and restrictions. Basic real-time scheduling analysis begins with this mathematical model:

- We have a single CPU.
- The workload consists of ***n* tasks** τ_i. Each task releases a series of **jobs.**
- Tasks **release** jobs periodically at the beginning of their period T_i.
- When a job is **released,** it is ready to run.
- The deadline D_i for a job may or may not be related to its period T_i. In some cases, if certain relations (e.g., equal) hold for all tasks, analysis may be easier.
- We would like for the job to **complete** before its **deadline** D_i. **Hard real-time** jobs **must** meet their deadlines, while **soft real-time** jobs should meet most of their deadlines.
- No task is allowed to suspend itself.
- For a preemptive scheduler a task can be preempted at any time by any other task. For a non-preemptive scheduler tasks cannot preempt each other.
- The worst-case execution time of each job is C_i. Determining this value is non-trivial because it depends on both software (the control flow may be dependent on the input data) and hardware (pipelining, caches, dynamic instruction execution). Instead, people attempt to estimate a tight bound which is reasonably close to the actual number but not smaller (which would be unsafe).
- Overhead such as scheduler activity and context switches are not represented in the model, so they are assumed to take no time. Because of this unrealistic assumption we need to accept that there will be a slight amount of error in the quantitative analytical results.
- Tasks are independent. They do not communicate with each other in a way which could make one wait for another, and they do not have any precedence relationships.

One important aspect of the workload to consider is the **utilization *U***, which is the fraction of the processor's time which is needed to perform all the processing of the tasks. Utilization is calculated as the sum of the each individual task's utilization. A task's utilization is the ratio of its computation time divided by the period of the task (how frequently the computation is needed):

$$U = \sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

3.2.2 Dividing the Design Space Based on the Workload and Scheduler

TABLE 3.1 Design Space Partitions

	PREEMPTIVE		NON-PREEMPTIVE	
	FIXED PRIORITY	DYNAMIC PRIORITY	FIXED PRIORITY	DYNAMIC PRIORITY
$D_i < T_i$				
$D_i = T_i$				
$D_i > T_i$				
General Case				

As seen in Table 3.1, we divide the design space into partitions based on characteristics of the **workload** and the **scheduler** because for some special cases the analysis is much easier than in the general case.

The relationship between a **task's deadline and period** can be less than, equal to, greater than, or the general case of any relationship. Similarly, the scheduler may or may not allow **preemption by tasks.** Finally, **priority assignment** may be fixed or dynamic (changing at run-time).

3.3 TASK PRIORITY ASSIGNMENT FOR PREEMPTIVE SYSTEMS

We can now examine different scheduling approaches using this foundation as a starting point. One critical question which we haven't answered yet is **how do we assign priorities?** We can assign a fixed priority to each task, or allow a task's priority to vary. The pros and cons for these approaches are discussed in detail elsewhere (Buttazzo, 2005). We first examine fixed-priority assignments.

3.3.1 Fixed Priority

3.3.1.1 Rate Monotonic Priority Assignment—RMPA

Rate monotonic priority assignment gives **higher priorities** to tasks with **higher rates** (execution frequencies). Table 3.2 shows an example of a workload scheduled with RMPA. RMPA is that it is **optimal** for workloads in which **each task's deadline is equal to its period.** There is no other task priority assignment approach which makes a system schedulable if it is not schedulable with RMPA.

TABLE 3.2 Sample Workload

TASK	EXECUTION TIME C	PERIOD *T*	PRIORITY WITH RMPA
τ_1	1	4	High
τ_2	2	6	Medium
τ_3	1	13	Low

3.3.1.2 Rate Monotonic Priority Assignment with Harmonic Periods

A special case of RMPA occurs when the task periods are **harmonic:** a task's period is an exact integer multiple of each shorter period. For example, a task set with periods of 3, 6, 18, and 54 has harmonic periods.

3.3.1.3 Deadline Monotonic Priority Assignment—DMPA

For tasks with the deadline less than the period $D_i < T_i$ RMPA is no longer optimal. Instead, assigning **higher priorities** to tasks with **shorter deadlines** results in optimal behavior. This is another common fixed-priority assignment approach.

3.3.2 Dynamic Priority

Instead of assigning each task a fixed priority, it is possible to have a priority which changes. We still use all of the assumptions in our model defined previously; however, we now need a scheduler which supports dynamic priorities. This means the scheduler must sort tasks, incurring additional computational overhead.

3.3.2.1 Earliest Deadline First

One simple dynamic approach is called ***Earliest Deadline First,*** which unsurprisingly first runs the task with the earliest deadline. This approach is optimal among preemptive scheduling approaches: if a feasible schedule is possible, EDF will find it.

3.4 SCHEDULABILITY TESTS FOR PREEMPTIVE SYSTEMS

Let's look at how to determine whether a given priority assignment makes a workload schedulable.

3.4.1 Fixed Priority

We begin with fixed priority systems.

3.4.1.1 Rate Monotonic Priority Assignment—RMPA

For some workloads it is very easy to determine if the workload is definitely schedulable with RMPA. The **Least Upper Bound (*LUB*)** test compares the utilization of the resulting workload against a function based on the number of tasks.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right) = LUB$$

- If U is less than or equal to the LUB, then the system is definitely schedulable. The LUB starts out at 1. As n grows, the LUB approaches 0.693. This means that **any** workload with RMPA and meeting the above criteria is schedulable.
- If U is greater than the LUB, then this test is inconclusive. The workload **may or may not** be schedulable with RMPA. We will need to use a different test to determine schedulability.[1]

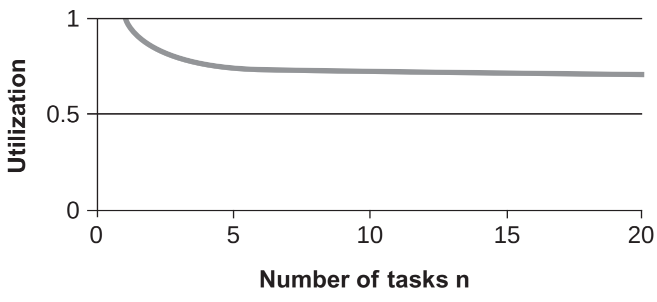


Figure 3.2 Least Upper Bound for RMPA as a function of the number of tasks *n*.

[1] Researchers studying a large number of random task sets found that the average real feasible utilization is about 0.88; however, this is just an average. Some task sets with $0.693 < U < 0.88$ were not schedulable, while some with $0.88 < U < 1$ were schedulable.

Figure 3.2 plots the rate monotonic least upper bound as a function of the number of tasks *n*. The area below the curve represents workloads which are always schedulable with RMPA. For the area above the curve, the test is inconclusive.

EXAMPLE

Let's see how this works for a system with three tasks, as shown in Table 3.2. We first compute the utilization of the workload:

$$U = \sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{1}{4} + \frac{2}{6} + \frac{1}{13} = 0.660$$

We compute the RM *LUB* for $n = 3$ tasks:

$$LUB = 3\left(2\frac{1}{3} - 1\right) = 0.780$$

Since $U \leq LUB$, we know the system is schedulable and will meet all its deadlines. If $U > LUB$, we do not know if the system is schedulable using the *LUB* test. We will use Response Time Analysis to determine schedulability.

3.4.1.2 Rate Monotonic Priority Assignment with Harmonic Periods

Harmonic RMPA has the benefit of guaranteeing schedulability up to 100 percent utilization. Hence, if we are able to adjust task periods, we can make RMPA systems schedulable. The trick is to make task periods **harmonic:** a task's period must be an exact integer multiple of the next shorter period. We can only shorten the period of a task to make it harmonic, as increasing it would violate the original deadline. The challenge is that as we shorten a task's period, we increase the processor utilization for that task. We need to keep utilization at or below 1 to keep the system schedulable.

Our first attempt at period modification lowers the period of task two to four time units, and that of task three to eight time units. The resulting utilization of 1.125 is greater than 1, so the system is not schedulable. Our second attempt lowers the periods of tasks one and three with a resulting utilization of 0.897, so the system is now schedulable.

3.4.1.3 Deadline Monotonic Priority Assignment—DMPA

Deadline Monotonic does not offer a simple utilization-based schedulability test, forcing us to resort to response time analysis instead.

TABLE 3.3 Sample workload with longer task three and harmonic periods.

		ORIGINAL		FIRST HARMONIC PERIOD ATTEMPT		SECOND HARMONIC PERIOD ATTEMPT	
TASK	EXECUTION TIME C	PERIOD T	UTILIZATION	MODIFIED PERIOD T	UTILIZATION	MODIFIED PERIOD T	UTILIZATION
τ_1	1	4	0.250	4	0.250	3	0.333
τ_2	2	6	0.333	4	0.500	6	0.333
τ_3	3	13	0.231	8	0.375	12	0.231
Total			**0.814**		**1.125**		**0.897**
Schedulable			**Maybe**		**No**		**Yes**

3.4.2 Dynamic Priority

EDF will result in a schedulable system if the total utilization is no greater than 1. This simplifies system analysis significantly.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

3.5 RESPONSE TIME ANALYSIS FOR PREEMPTIVE SYSTEMS

In some cases we may need to determine the worst-case response time (the maximum delay between a task's release and completion) for a task set.

3.5.1 Fixed Priority

In order to find the response time for a fixed priority assignment (RM, HRM, DM, etc.) we need to figure out the longest amount of time that a task can be delayed (preempted) by higher priority tasks.

The equation below calculates the worst-case response time R_i for task τ_i as the sum of that task's computation time C_i and the sum of all possible computations from higher-priority tasks, as they will preempt τ_i if they are released before τ_i completes. The tricky part of this equation is that if τ_i is preempted, then it will take longer to complete (R_i will grow), raising the possibility of more preemptions. So, the equation needs to be repeated until R_i stops changing or it exceeds the deadline D_i. Note that the square half brackets $\lceil x \rceil$ signify the **ceiling function,** which returns the smallest integer which is not smaller than the argument x.

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

EXAMPLE

Let's evaluate the response time for the system of Table 3.3 from the previous example (with non-harmonic periods). We will evaluate the response time for the lowest priority task. We begin with a value of 0 for R_i.

$$R_3 = C_3 + \sum_{j=1}^{2} \left\lceil \frac{R_i}{T_j} \right\rceil C_j = 3 + \left\lceil \frac{0}{4} \right\rceil 1 + \left\lceil \frac{0}{6} \right\rceil 2 = 3 + 1 + 2 = 6$$

$$R_3 = C_3 + \sum_{j=1}^{2} \left\lceil \frac{6}{T_j} \right\rceil C_j = 3 + \left\lceil \frac{6}{4} \right\rceil 1 + \left\lceil \frac{6}{6} \right\rceil 2 = 3 + 2 + 2 = 7$$

$$R_3 = C_3 + \sum_{j=1}^{2} \left\lceil \frac{7}{T_j} \right\rceil C_j = 3 + \left\lceil \frac{7}{4} \right\rceil 1 + \left\lceil \frac{7}{6} \right\rceil 2 = 3 + 2 + 4 = 9$$

$$R_3 = C_3 + \sum_{j=1}^{2} \left\lceil \frac{9}{T_j} \right\rceil C_j = 3 + \left\lceil \frac{9}{4} \right\rceil 1 + \left\lceil \frac{9}{6} \right\rceil 2 = 3 + 3 + 4 = 10$$

$$R_3 = C_3 + \sum_{j=1}^{2} \left\lceil \frac{10}{T_j} \right\rceil C_j = 3 + \left\lceil \frac{10}{4} \right\rceil 1 + \left\lceil \frac{10}{6} \right\rceil 2 = 3 + 3 + 4 = 10$$

The estimated response time for task three to complete begins at six time units and grows until it reaches a fixed point at ten time units. Since this is less than the deadline for task three, we know the system is schedulable and will always meet its deadlines.

3.5.2 Dynamic Priority

Finding the worst case response time for a dynamic priority system is much more challenging due to the need to consider every possible combination of task releases, leading to a large amount of analysis. There have been various methods developed to reduce the number of cases which must be examined, but this remains a computationally expensive exercise.

3.6 NON-PREEMPTIVE SCHEDULING APPROACHES

All of the scheduling analysis we just examined depends on the scheduler being able to preempt any task at any time. Let's consider scheduling when preemption is **not** possible. Why? It turns out that we can save large amounts of RAM by using a non-preemptive scheduler. A preemptive system requires enough RAM to store each task's largest possible call stack. A non-preemptive system only requires enough space to store the largest one of all of the task's call stacks. Systems with large numbers of tasks can significantly reduce RAM requirements and correspondingly reduce MCU costs.

Removing preemption means that the processor cannot meet the deadline for a task τ_i with deadline D_i **shorter** than the **duration** of the longest task τ_L plus the actual **computation time** C_i for our task of interest τ_i. This constraint rules out some systems but not all. Systems with tasks whose **deadlines are sufficiently longer than the WCET of the longest task** are promising candidates for using non-preemptive scheduling. How much longer? That depends on the range of deadlines and WCETs and requires quite a bit of analysis.

Another result of removing preemption is that sometimes it is possible to improve a schedule by inserting a small amount of idle time to ensure that a task doesn't start running immediately before something more important is released. Calculating how much idle time to insert, and where, is a computationally hard problem for general task sets, and therefore not feasible. So we must limit ourselves to using a non-idling scheduler and accept that it may not be as good as an idling scheduler.

We will now examine the timing characteristics of these remaining systems. Further analysis and details are available elsewhere (George, Rivierre, & Spuri, 1996). For a given task set and priority class (fixed or dynamic) we have several questions:

- What is the optimal priority assignment?
- Is there an easy schedulability test, and is it exact?
- How do we compute worst-case response time?

3.6.1 Optimal Priority Assignment

For dynamic priority non-preemptive schedulers, EDF turns out to be the optimal for general task sets, in which the deadline does not need to be related to the period in any way.

For fixed priority schedulers, we can consider two cases. In the general case, with deadlines not related to periods, there is a method to calculate the optimal priority assignment (Audsley, 1991). Although this method applies directly to preemptive schedulers, it has been modified to support non-preemptive schedulers as well. This method has a complexity of $O(n^2)$.

There is a case where deadline monotonic is the optimal priority assignment. Two conditions must be met. First, a task's deadline must be no longer than its period. Second, for all pairs of tasks *i* and *j*, task *i* with the shorter deadline must not require more computation than task *j*.

3.6.2 Schedulability Tests

Knowing whether a priority assignment is optimal is of limited value without knowing if it leads to a feasible schedule.

For dynamic priority assignment of general task sets (deadline not related to the period) there is no utilization-based test, but there is an inexact complex analytical test. It provides a sufficient but not necessary condition for showing schedulability. For task sets where a task's deadline equals its period there is an exact analytical test, providing a necessary and sufficient condition for schedulability.

For fixed priority assignment there is no utilization-based test. Instead, one must calculate worst-case response time for each task and verify all deadlines are met.

3.6.3 Determining Worst-Case Response Time

Finding the WCRT for a task in a dynamic priority system is similar to the preemptive case, which is already quite involved. The analysis also needs to consider the possibility of priority inversion due to a later deadline.

Finding the WCRT for a task in a fixed priority system is less daunting. It is similar to the preemptive case but requires considering blocking B_i from the longest lower-priority task. Details are omitted here.

3.7 LOOSENING THE RESTRICTIONS

The assumptions listed in the beginning of the section limit the range of real-time systems which can be analyzed, so researchers have been busy removing them.

3.7.1 Supporting Task Interactions

One assumption is that tasks cannot interact with each other. They cannot share resources which could lead to blocking.

Tasks typically need to interact with each other. They may need to share a resource, using a semaphore to provide mutually-exclusive resource use. This leads to a possible situation called **priority inversion.** If a low priority task τ_L acquires a resource and then is preempted by a higher priority task τ_H which also needs the resource, then τ_H blocks and cannot proceed until τ_L gets to run and releases the resource. In effect, the priorities are inverted so that τ_L has a higher priority than τ_H.

Priority inversion is prevented by changing when a task is allowed to lock a resource. Two examples of such rules are the Priority Ceiling Protocol and the Stack Resource Protocol. The response time analysis equation previously listed can be modified to factor in blocking times.

3.7.2 Supporting Aperiodic Tasks

Another assumption is that each task runs with a fixed period T_i. This is quite restrictive, but it is possible to support aperiodic tasks by finding the minimum time between task releases (inter-task release time) and using this as the period T_i. This approach works but overprovisions the system as the difference between minimum and average inter-task release times grows, limiting its usefulness. There are other approaches (e.g., polling servers) which are beyond the scope of this text.

3.7.3 Supporting Task Interactions

Enabling tasks to share resources with dynamic task priorities is different from static task priorities. With EDF, each job of a task is assigned a priority which indicates how soon its deadline is. Priority inversion can still occur, but now job priorities may change. Researchers have developed approaches such as the Stack Resource Policy (SRP), Dynamic Priority Inheritance, Dynamic Priority Ceiling, and Dynamic Deadline Modification.

Let's look at one example—the Stack Resource Policy. SRP assigns a preemption level to each task in addition to its priority. Each shared resource has a **ceiling,** which is the highest preemption level of any task which can lock this resource. The system is assigned a ceiling which is the highest of all currently locked resource ceilings. These factors are used to determine when a job can start executing. Specifically, a job cannot start executing if it does not both (1) have the highest priority of all active tasks, and (2) have a preemption level greater than the system ceiling.

SRP simplifies analysis of the system because it ensures that a job can only block before it starts running, but never after. In addition, the maximum blocking time is one critical section. These factors lead to a simple feasibility test for periodic and sporadic tasks. For each task i, the sum of the utilizations of all tasks with greater preemption levels and the blocking time fraction for this task must be no greater than one.

$$\forall i, 1 \leq i \leq n \quad \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \leq 1$$

3.7.4 Supporting Aperiodic Tasks

Recall that our task model requires each task to be periodic. If a task's period can vary, we need to choose the minimum period and design the system according to this worst-case, which can lead to an overbuilt system. As with fixed-priority systems, there are ways to relax this limitation. For example, the Total Bandwidth Server (TBS) assigns deadlines for aperiodic jobs so that their demand never exceeds a specified limit U_s on the maximum allowed processor utilization by sporadic tasks. The deadline d_k which depends on the current time r_k and the deadline d_{k-1} assigned for this task's previous job. The deadline is pushed farther out in time as the ratio of the execution time of the request C_k and acceptable sporadic server utilization U_s increases.

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

With this approach, we can guarantee that the entire system is schedulable with EDF if the utilization from periodic tasks (U_p) and the TBS (U_s) is no greater than one.

3.7.5 Supporting Shared Buses

Embedded processors may allow components besides the CPU core to control the address and data buses. Some examples are Direct-Memory Access Controllers (DMAC) and Data Transfer Controllers (DTC). If these devices can seize control of the buses and delay the CPU, then their activities must also be considered as additional tasks with appropriate priority levels. A burst-mode transfer may be modeled easily as a single task, while a cycle-stealing transfer slows whichever task is executing, complicating the analysis.

3.8 WORST-CASE EXECUTION TIME

All of the real-time system timing analysis presented here depends on knowing the *worst-case execution time* (WCET) of each task (including interrupt service routines). For most code, accurately determining the WCET is a non-trivial exercise. Instead, we attempt to estimate a safe *upper bound*—a value which may be greater than the actual WCET, but is definitely not less than the WCET. This makes analysis using that bound *safe* because it may overestimate execution time, but will never underestimate it. The *tightness* of the bound indicates how close it is to the actual (unknown) WCET. As the WCET bound gets tighter the resulting timing analysis grows more accurate and the calculated delays and utilizations decrease, showing the system will respond sooner. A continual goal of researchers in this field is to determine how to tighten WCET estimate bounds, reducing pessimistic overestimation and the resulting overprovisioning of the resulting system.

3.8.1 Sources of Execution Time Variability

Both **software** and **hardware** factors may make the execution time of a given function vary.

First, a function will likely contain different possible control-flow paths, each having a different execution time. Programming constructs such as conditionals complicate the timing analysis, forcing the examination of each case to determine the longest. Loops with execution counts unknown at the time of analysis are not analyzable without making assumptions about the maximum number of iterations. The number of loop iterations and selection of conditional cases may depend on input data.

Second, the MCU's **hardware** may introduce timing variations. The duration of some **instructions** may depend on their input data (for example, multiplying with a zero for either operand can complete early, as the result will be zero). **Pipelined processors** are vulnerable to various hazards. Pipelined instruction processing *overlaps* the execution of different parts of multiple instructions. This increases the instruction throughput (instructions per second), but does not reduce the time taken to execute an individual instruction. Pipelining reduces the amount of time needed to execute code. The deeper the pipeline is, the greater the risk and possible penalty. Most low-end microcontrollers have shallow pipelines, reducing the impact of this risk. When taking a conditional branch, a pipelined processor may stall (due to a control-flow hazard) as it fetches the correct instruction (the target of the taken branch), discarding the previously fetched instruction (from the not-taken path). Some processors may reduce the number of such stalls using a branch target buffer (BTB). However, there may still be stalls where the BTB misses. There may also be data-flow hazards, where one instruction depends upon the result of a prior instruction which has not completed yet.

Cache memories introduce timing variability due to the different access times for hits and misses. There are analytical methods to classify accesses as hits or misses, but these typically do not cover all accesses, leaving some timing variability. Similarly, there are methods to use caches more effectively (e.g., by locking blocks, or locating data carefully) to reduce the number of cache misses. Some memory devices such as **DRAM** include an internal row buffer which behaves as a cache, complicating timing analysis.

3.8.2 RL78 Pipeline

The RL78 CPU pipeline has three stages, as shown in Figure 3.3:

- The IF (Instruction Fetch) stage fetches the instruction from memory and increments the fetch pointer.
- The ID (Instruction Decode) stage decodes the instruction and calculates operand address.

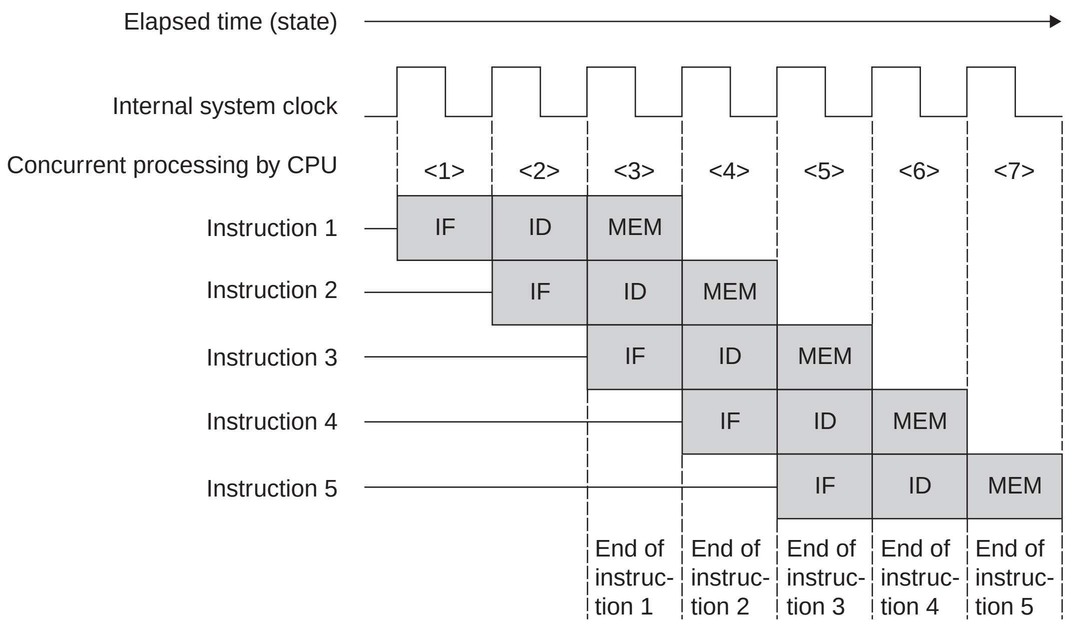


Figure 3.3 RL78 Instruction processing pipeline sequence.

- The MEM (Memory access) stage executes the instruction and accesses the specified memory location.

Many, but not all, RL78 instructions take one cycle to execute once the pipeline is full. The **RL78 Family Users Manual: Software** (Renesas Electronics, 2011) presents execution times for each instruction.

There are various situations in which instructions cannot flow smoothly through the pipeline due to *hazards*. The following situations lead to hazards which stall the pipeline:

- Accessing data from flash or external memory (if present) rather than internal RAM.
- Fetching instructions from RAM rather than from internal flash ROM.
- Current instruction using memory addressed by register written in previous instruction (e.g., indirect access).
- Changing the control flow of the program with instructions such as calls, branches, and returns. Conditional branches are not resolved until the MEM stage, so the target address is not known until two cycles later. Because instruction fetching proceeds sequentially, branches are essentially predicted to be not taken. As a result, taken conditional branches take two more cycles than not-taken conditional branches.

3.8.3 Determining a Worst-Case Execution Time Bound

There are two approaches to determining the WCET. First, we can analyze a task and find the path from entry to exit with the longest possible duration. Second, we can experimentally run the task and measure the execution time.

There are two complications with the first (analytical) approach.

- First, the object code must be analyzed, not the source code. This is because it is the object code that the MCU executes. Manual object code analysis becomes tedious very quickly. There are static timing analysis tools available for a limited number of instruction set architectures and specific processors implementing those ISAs. Developing these analyzers is complicated, and there is a limited market for the tools.
- Second, we must make assumptions about input data. For example, what is the maximum number of times a loop will repeat? The worse our assumptions, the looser the timing bound, and the greater the overestimation.

The main limitation of the second (experimental) approach is that we don't have any guarantee that the *observed* WCET is the actual WCET. Maybe we were very lucky selecting the input data and test conditions, and chose values which led to an uncommonly fast execution. One way to reduce the risk in this area is to make many timing measurements with a wide range of input data, while observing how much of the function has actually been executed (the *code coverage*). As the code coverage resulting from the test cases increases, the risk of unexamined outlier cases decreases. The resulting observed WCET bound is typically scaled up to include a safety factor.

3.9 EVALUATING AND OPTIMIZING RESPONSE LATENCIES

The CPU's interrupt system, the RTOS, and the application program all introduce some delays which limit the responsiveness of the system. In this section we explore the sources of delays and discuss methods to reduce them.

There are three general types of critical path to consider:

- The latency between an interrupt being requested and the corresponding ISR running.
- The latency between an interrupt being requested and a corresponding user task running. The intervening ISR signals through the RTOS or some other mechanism that the task should run.
- The latency between one task signaling an event or other mechanism, causing another task (of higher priority) to run. Again, this uses an RTOS or other mechanism.

For embedded systems, the first two cases are typically the most time-sensitive and so we examine them here.

3.9.1 Methods for Measurement

As with any optimization activity, things go much faster when one can measure the relative contribution of each component to the problem. This indicates where to begin optimization efforts to maximize payoffs.

Some MCUs include mechanisms for logging a trace of executed instructions. With proper triggering and post-processing support, this trace can be used to determine exactly which code executed and how long each component took.

A similar approach is to instrument the software so that it signals externally which code is executing (e.g., with output port bits or a high-speed serial communication link such as SPI). These signals and external interrupt request lines can be monitored to determine what the processor is doing when. This approach requires access to source code to insert the instrumentation instructions.

3.9.2 Interrupt Service Routine

There will be a finite latency between when an interrupt is requested and when the first instruction of the ISR begins to execute. Two components make up this latency.

First, the interrupt may be masked off, forcing the processor to ignore it until it is enabled. Any code which disables this or all interrupts will delay the response of this ISR. Hence if interrupts must be disabled, it should be for as brief a time as possible. Note that most CPUs disable interrupts upon responding to an interrupt request and restore previous masking state upon returning from the interrupt. This means that interrupts are disabled during an ISR's execution, adding to this first latency component. Some time-critical systems may re-enable interrupts within long ISRs in order to reduce response latency.

Second, the CPU will require a certain amount of time to respond to the interrupt. This time may include completing the currently executing instruction, saving some processor context, loading the ISR vector into the program counter, and then fetching the first instruction from the ISR.

3.9.2.1 RL78 Interrupts

The RL78 architecture supports interrupts from many possible sources, both on- and off-chip. When an interrupt is requested, the processor saves some of its execution state (program counter and program status word), executes the ISR corresponding to the interrupt request, and then resumes the execution of the interrupted program.

The address of each ISR is listed in the interrupt vector table in memory. Whenever an interrupt occurs, the processor uses this table to determine the location of the ISR.

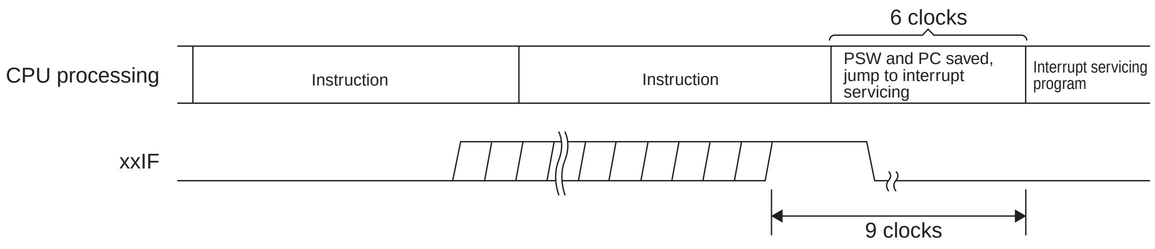

6 clocks
CPU processing
Instruction
Instruction
PSW and PC saved, jump to interrupt servicing
Interrupt servicing program
xxIF
9 clocks

Figure 3.4 Best-case interrupt response time.

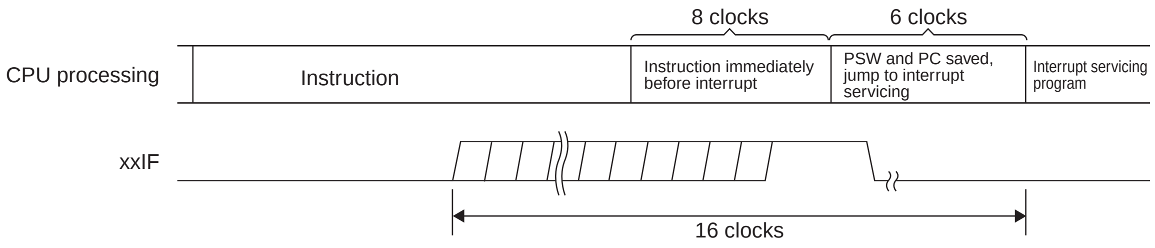

8 clocks
6 clocks
CPU processing
Instruction
Instruction immediately before interrupt
PSW and PC saved, jump to interrupt servicing
Interrupt servicing program
xxIF
16 clocks

Figure 3.5 Worst-case interrupt response time.

1. When an interrupt is requested, the CPU will finish executing the current instruction (and possibly the next instruction[2]) before starting to service the interrupt. Figure 3.4 shows a best-case example, in which the ISR begins executing nine cycles after the interrupt is requested. Figure 3.5 shows the worst case (with a long instruction and an interrupt request hold situation), where this can take up to sixteen cycles.

[2] Certain instructions (called interrupt request hold instructions) delay interrupt processing to ensure proper CPU operation.

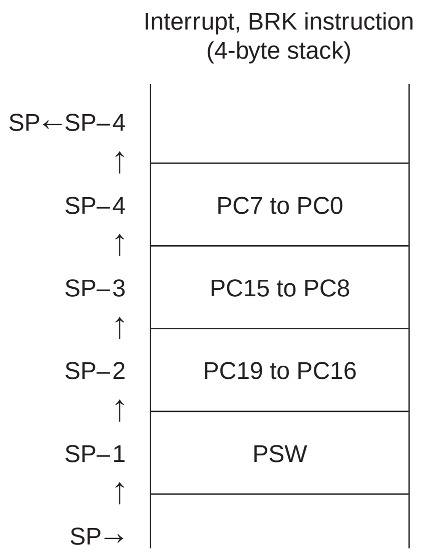


Figure 3.6 Interrupt and BRK instruction push processor status and program counter onto stack.

2. The CPU pushes the current value of the PSW and then the PC onto the stack, as shown in Figure 3.6. Saving this information will allow the CPU to resume processing of the interrupted program later without disrupting it.
3. The CPU next clears the IE bit. This makes the ISR non-interruptible. However, if an EI instruction is executed within the ISR, it will become interruptible at that point.
4. If the interrupt source is a maskable interrupt, the CPU next loads the PSW PR field with the priority of the ISR being serviced (held in the bits ISP0 and ISP1). This prevents the processor from responding to lower-priority interrupts during this ISR.
5. The CPU loads the PC with the interrupt vector for the interrupt source.
6. The ISR begins executing.

3.9.3 Real-Time Kernel

If the system uses a real-time kernel, then the ISR may call a kernel function in order to signal a task to trigger its execution. The kernel code which performs this has critical sections which must be protected. A common way is to disable interrupts, which increases interrupt response latency. Another approach is to lock the scheduler to prevent switching to other tasks. This still requires disabling the interrupts for a brief time, but much less time than the first approach.

Let's examine the critical path from interrupt request to task execution in μC/OS-III using the scheduler lock method (called deferred post in μC/OS-III). This is examined in much greater detail in Chapter 9 of the μC/OS-III manual (Labrosse & Kowalski, 2010).

- CPU interrupt response:
  - The interrupt is requested.
  - After some delay (including interrupt disable time), the interrupt is serviced.
- ISR execution:
  - Disable interrupts, if not already disabled.
  - Execute prologue code to save state.
  - Process the interrupting device (e.g., copying a value from a result or received data register).
  - Potentially re-enable interrupts, if desired.
  - Post the signal to the kernel's interrupt queue.
  - Call OSIntExit, which will switch contexts to the interrupt queue handler task (which is now ready, and is of higher priority than any user task).
- Interrupt queue handler task:
  - Disable interrupts.
  - Remove post command from interrupt queue.
  - Re-enable interrupts.
  - Lock scheduler.
  - Re-issue post command.
  - Yield execution to scheduler.
- Scheduler:
  - Context switch to highest priority task.
- Signaled task (assuming it is the highest priority):
  - Resume execution.

Real-time kernel vendors often provide information on the maximum number of cycles required to perform various operations. These counts will depend on the target processor architecture, compiler, optimization level, and memory system. Similarly, these kernels are designed to disable interrupts for as little time as possible, and will advertise these counts as well.

Some kernels can provide statistics at run-time which allow a designer to monitor various system characteristics. For example, μC/OS-III provides the following information at run-time:

- OSIntDisTimeMax indicates the maximum amount of time that interrupts have been disabled.
- OSSchedLockTimeMax indicates the maximum amount of time which the scheduler was locked.

- OSSchedLockTimeMaxCur indicates the maximum amount of time for which this task locked the scheduler.
- OSStatTaskCPUUsage indicates the percentage of CPU time used by the application.

Kernels are typically configurable in which services they offer, and scalable in how many resources are available. Both of these parameters may affect kernel response time, depending on the implementation details. Hence the kernel should be configured to be lean and efficient, rather than providing extra services which are not needed.

3.9.4 Application

There are various ways in which an application can negatively affect a system's responsiveness.

3.9.4.1 Disabled Interrupts

The application can delay response to interrupts (and dependent processing) through indiscriminate disabling of interrupts. This should be avoided if at all possible, instead disabling only the specific interrupt leading to the ISR causing the race condition.

Similarly, for most CPU architectures the ISRs execute with interrupts disabled. Hence any ISR can be delayed by all higher priority interrupts as well as one lower-priority interrupt. ISRs should be made as short as possible. One way is to leverage the kernel's communication and synchronization mechanisms to hand off as much processing as possible to task-level code where it will not impact responsiveness.

Another approach is to re-enable interrupts immediately upon entering the ISR. This opens the door to all sorts of potential data race problems and should not be done without fully understanding the consequences.

3.9.4.2 Priority Inversion from Shared Resources

If a high priority task shares a resource with a lower priority task then it is possible for priority inversion to occur. Most kernels offer mutexes in order to reduce amount of time a high priority task spends blocking on a shared resource by temporarily raising the priority of the lower priority task currently holding that resource.

3.9.4.3 Deadlines and Priorities

In a fixed-priority system, tasks with higher priorities will have shorter average response latencies than those with lower priorities. Assigning task priority with a Rate Monotonic approach implies that a task's deadline is equal to its period. The Deadline Monotonic approach allows deadlines to be shorter than task periods, allowing for finer control of task responsiveness.

3.10 RECAP

In this chapter we have studied how to calculate the worst-case response time and schedulability for real-time systems. We then examined worst-case execution time concepts. We concluded with an examination of the sequence of activities from interrupt request to response, whether in an ISR or a task.

3.11 BIBLIOGRAPHY

Audsley, N. C. (1991). *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times.* York: University of York, Department of Computer Science.

Audsley, N. C., Burns, A., Davis, R. I., Tindell, K. W., & Wellings, A. J. (1995). Fixed Priority Pre-emptive Scheduling: An Historical Perspective. *Real-Time Systems, 8*(3), 173–198.

Buttazzo, G. C. (2005). Rate Monotonic vs. EDF: Judgment Day. *Real-Time Systems, 29,* 5–26.

George, L., Rivierre, N., & Spuri, M. (1996). *Technical Report RR-2966: Preemptive and Non-preemptive Real-time Uniprocessor Scheduling.* INRIA.

Labrosse, J., & Kowalski, F. (2010). *MicroC/OS-III: The Real-Time Kernel.* Weston, FL: Micrium Press.

Renesas Electronics. (2011). *RL78 Family User's Manual: Software.*

Sha, L., Abdelzhaer, T., Arzen, K.-E., Cervin, A., Baker, T., Burns, A., et al. (2004, November-December). Real Time Scheduling Theory: A Historical Perspective. *Real Time Systems, 28*(2–3), 101–155.

Chapter 4

Profiling and Understanding Object Code

4.1 LEARNING OBJECTIVES

This chapter deals with how to make a program run faster. In particular, it shows how to find the slow parts of a program and address them. There are many guides to optimization which provide a plethora of ways to improve code speed. The challenge is to know **which code** to optimize. This chapter concentrates first on methods to find the slow object code. It then presents methods to help **examine object code** generated by the compiler and understand its relationship to the C source code. This ability is necessary for applying many of the analysis and optimization techniques presented in Chapters 5 and 6.

4.2 BASIC CONCEPTS

There are many reasons why an embedded program many need to run faster: a quicker response, to free up time for using a more sophisticated control algorithm, to move to a slower or less expensive processor, to save energy by letting the processor sleep longer, and so on.

However, an embedded system is built of many parts, any one of which could be limiting performance. The challenge is to find out **which part** of the system is limiting performance. It is similar to a detective story—there are many suspects, but who really did it?

- Was the architecture a bad fit for the work at hand?
- Is your algorithm to blame?
- Did you do a bad job coding up the algorithm?
- Did the person who coded up the free software you are using do a bad job?
- Is the compiler generating sloppy object code from your source code?
- Is the compiler configured appropriately?
- Are inefficient or extra library functions wasting time?
- Is the input data causing problems?
- Are communications with peripheral devices taking too much time?

Clearly there are many **possible** culprits, but we would like to find the most likely ones quickly in order to maximize our benefits.

*The real problem is that programmers have spent far too much time worrying about efficiency in the **wrong places** and at the **wrong times;** premature optimization is the root of all evil (or at least most of it) in programming.*
—Donald Knuth

With this in mind, here is an overview of how to develop fast code quickly:

1. Create a reasonable system design.
2. Implement the system with **reasonable** implementation decisions. Good judgment is critical here. However, don't start optimizing too early.
3. Get the code working.
4. Evaluate the performance. If it is fast enough, then your work is done. If not, then repeat the following as needed:
   a. Profile to find bottlenecks.
   b. Refine design or implementation to remove or lessen them.

4.2.1 Correctness before Performance

Don't try to optimize too early. Make a reasonable attempt to make good design and implementation decisions early on, but understand that it is essentially impossible for puny earthlings like us to create an optimal implementation without iterative development. So start with a **good implementation** based on reasonable assumptions. This implementation needs to be **correct.** If it isn't correct, then fix it. Once it is correct it is time to examine the performance to determine performance bottlenecks.

Certain critical system characteristics do need to be considered to create a good implementation. In particular, one must consider the MCU's native word size, hardware support for floating-point math, and any particularly slow instructions (e.g., divide).

4.2.2 Reminder: Compilation is Not a One-to-One Translation

A compiler translates source code (e.g., in the C language) into object code (e.g., text-based human-readable assembly code or binary CPU-readable machine code).[1] There are many possible correct object code versions of a single C language program. The compiler will generally create a reasonably fast version, but it is by no means the fastest. Part of your role in optimizing software is to understand if the compiler is generating object code which is **good enough.** Some of this can come from reading the compiler manual (IAR Systems).

However, even more understanding comes from examining that code and using your judgment[2] to make this decision. Examining the object code generated by the compiler is a remarkably effective way to learn how to help it generate more efficient code.

4.3 PROFILING—WHAT IS SLOW?

There are many opportunities for optimization in any given program, but where and how should we start? We could waste a lot of time speeding up code which doesn't really impact the system's overall performance.[3] In order to avoid this, we want to identify what parts of the program **take up most of its time.** Optimizing those parts first will give us the biggest payback on our development time. **Profiling** a program shows us where it spends its time, and therefore where we should spend our time for optimization.

4.3.1 Mechanisms

There are four basic approaches to profiling a program.

1. We can **sample the program counter** periodically by interrupting the program to see what it is doing by examining the return address on the call stack, and then looking up what function (or region, to generalize) contains that instruction address. This approach provides the biggest return on development time effort.
2. We can modify each function in the program to **record when it starts and finishes executing.** After the program runs we process the function execution time information to calculate the profile. We don't discuss this further, as it requires extensive modification to the program (for each user function and library function). Some compilers or code post-processing tools provide this support.
3. Some debuggers use a similar approach, **inserting breakpoints** at the start of all functions. Each time the debugger hits a breakpoint or returns from a function it notes the current time and the function name. The debugger uses this information to calculate the execution time profile of the program. This approach incurs debugger overhead with each breakpoint and function return, potentially slowing down the program significantly.

[1] For the purposes of this discussion, the assembly code and machine code are two different representations of the same object code.

[2] Good judgment comes from experience. Experience comes from bad judgment.

[3] You can avoid ten minutes of careful thinking by instead spending the whole day blindly hacking code.

4. We can use hardware circuitry to **extract an instruction address trace** by monitoring the address bus as the program runs. An MCU with an externally-accessible address bus can be connected to a logic analyzer to capture the trace. Alternatively, some MCUs do include dedicated debug hardware to capture the trace and save it internally or send it out through a communication port. The address trace can then be processed to create the profile.

There are two types of profiles: **flat** and **cumulative.** A **flat profile** indicates how much time is spent inside a function *F.* A **cumulative profile** indicates how much time is spent in *F* and all of the functions which it calls, and all the functions called by those functions, and so forth.

4.3.2 An Example PC-Sampling Profiler for the RL78

We will use the PC-sampling approach here for reasons of practicality and performance. There are commercial and open source profiling tools available. For example, the C-Spy debugger in IAR Embedded Workbench for RL78 supports profiling using only breakpoints on the target processor, which limits execution speed.

Instead, let's see how to build a PC-sampling profiler for the RL78 using IAR Embedded Workbench.

4.3.2.1 Sampling the PC

First we need a way to sample the PC occasionally. During system initialization we configure a timer array unit peripheral to generate interrupts at a frequency[4] of 100 Hz. This interrupt is handled at run time by the service routine shown here.

```
#pragma vector = INTTM00_vect
__interrupt void MD_INTTM00(void)
{
   /* Start user code. Do not edit comment generated here */

   volatile unsigned int PC; // at [SP+4]
   unsigned int s, e;
   unsigned int i;

   if (!profiling_enabled)
      return;
```

[4] This is an arbitrary frequency. A higher frequency increases resolution but also timing overhead. A lower frequency reduces resolution and overhead.

```

   profile_ticks++;

   //Extract low 16 bits of return address
   __asm("  MOVW AX, [SP+14]\n"
      "  MOVW [SP+4], AX\n");

   /* look up function in table and increment counter */
   for (i = 0; i < NumProfileRegions; i++) {
      s = RegionTable[i].Start;
      e = RegionTable[i].End;
      if ((PC >= s) && (PC <= e)) {
         RegionCount[i]++;
         return;
      }
   }
   /* End user code. Do not edit comment generated here */
}
```

This ISR needs to retrieve the saved PC value from the stack. Figure 4.1 shows the stack contents upon responding to an interrupt. The address of the next instruction to execute after completing this ISR is stored on the stack in three bytes: PC7–0, PC15–8, and PC19–16. At the beginning of the ISR, they will be at addresses SP+1, SP+2, and SP+3. However, the

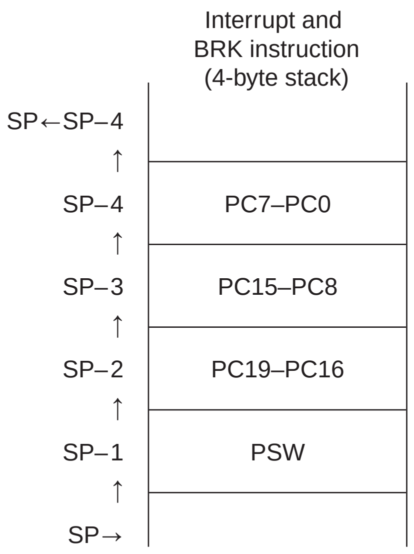


Figure 4.1 Stack contents upon responding to an interrupt.

ISR may push additional data onto the stack so we will need to examine the assembly code generated by the compiler for our ISR before we can definitively identify the offsets from the SP. In our case there is additional data allocated on the stack for local variables, so the high byte of the saved PC (PC19-PC16) is located at SP+16 and the low word (PC0-PC7 and PC8-PC15) is at SP+14. The code at lines 16 and 17 in the listing copies the low word of the saved PC value into register AX and then into the local variable PC on the stack.

4.3.2.2 Finding the Corresponding Code Region

TABLE 4.1 Region Address Information Table

REGION NAME	START ADDRESS	END ADDRESS	COUNT
foo	0×00001234	0×00001267	0
bar	0×00001268	0×00001300	0

So now we have the saved PC, which shows us what instruction the processor will execute after finishing this ISR. What program region (e.g., function) corresponds to that PC? Ideally we would like a table of information such as in Table 4.1.

There are various ways to create such a table. One approach is to process the map file created by the linker. The IAR Embedded Workbench for RL78 generates a **map** file in the output directory (e.g., debug/your_project_name.map) which shows the size and location of each function. Functions are stored in one of three types of code segment:

- CODE holds program code
- RCODE holds start-up and run-time library code
- XCODE holds code from functions declared with the attribute *__far_func*

Here is an example entry from the map file:

```
CODE
   Relative segment, address: 00001238 - 000013C8 (0x191 bytes), align: 0
   Segment part 11.
      ENTRY           ADDRESS     REF BY
      =====           =======     ======
      sim_motion         00001238    main (CG_main)
        calls direct
        CSTACK = 00000000 ( 000000A4 )
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

It shows that the function *sim_motion* starts at address 00001238 and ends at address 000013C8. We can use this information to create a region table entry for the function. We create a type of structure called REGION_T to hold a region's start address, end address, and label (to simplify debugging). The addresses are stored as unsigned ints (16 bits long) because we wish to save space, and our target MCU's program memory is located within the first 64 KB of the address space. This would need to be changed if we needed to store larger addresses.

```
typedef struct {
   unsigned int Start;
   unsigned int End;
   char Name[16];
} REGION_T;
```

We will use an awk script to extract function names and addresses and generate a C file which holds two arrays, as shown in the listing below. Some toolchains offer tools for extracting symbols and their information from the object code. For example, gnu binutils provides nm and objdump.

The first array (RegionTable, lines 2–19) holds the start and end addresses of the functions and their names. The ISR MD_INTTM00 accesses it in lines 21 and 22 provided previously. The array is declared as a const to allow the compiler to place it into ROM, which is usually larger and less valuable than RAM for microcontrollers.

The second array (RegionCount, line 21) is an array of unsigned integers which count the number of times the region was interrupted by the ISR. This array is initialized to all zeros on start-up. The ISR increments the appropriate array entry in line 26. If we do not find a corresponding region, then no region counter is incremented.

```
 1. #include "region.h"
 2. const REGION_T RegionTable[] = {
 3.    {0x00000A46, 0x00000A79, "AD_Init"},   //0
 4.    {0x00000A7A, 0x00000A83, "AD_Start"},   //1
 5.    {0x00000A84, 0x00000A87, "AD_ComparatorOn"},   //2
 6.    {0x00000A88, 0x00000A95, "MD_INTAD"},   //3
 7.    {0x00000A96, 0x00000AAF, "IT_Init"},   //4
 8.    {0x00000AB0, 0x00000ABC, "IT_Start"},   //5
 9.    {0x00000AD4, 0x00000AE8, "MD_INTIT"},   //6
10.    {0x00000AE9, 0x00000B8F, "main"},   //7

    (many lines deleted)

11.    {0x00000A1D, 0x00000A45, "matherr"},   //60
12.    {0x00001606, 0x00001710, "sqrt"},   //61
```

```
    {0x00001711, 0x0000185C, "__iar_Atan"},   //62
    {0x0000185D, 0x00001907, "__iar_Dint"},   //63
    {0x00001912, 0x00001964, "__iar_Dnorm"},   //64
    {0x00001965, 0x00001B1D, "__iar_Dscale"},   //65
    {0x00001B27, 0x00001C66, "__iar_Quad"},   //66
    {0x00001C67, 0x00001DC2, "__iar_Sin"},   //67
};
const unsigned NumProfileRegions=68;
volatile unsigned RegionCount[68];
```

We also have the ISR increment a variable (sample_count) to count how many samples we've taken. We need this value to correctly calculate the profile if any of our samples did not hit any regions in our table.

4.3.2.3 Modifications to the Build Process

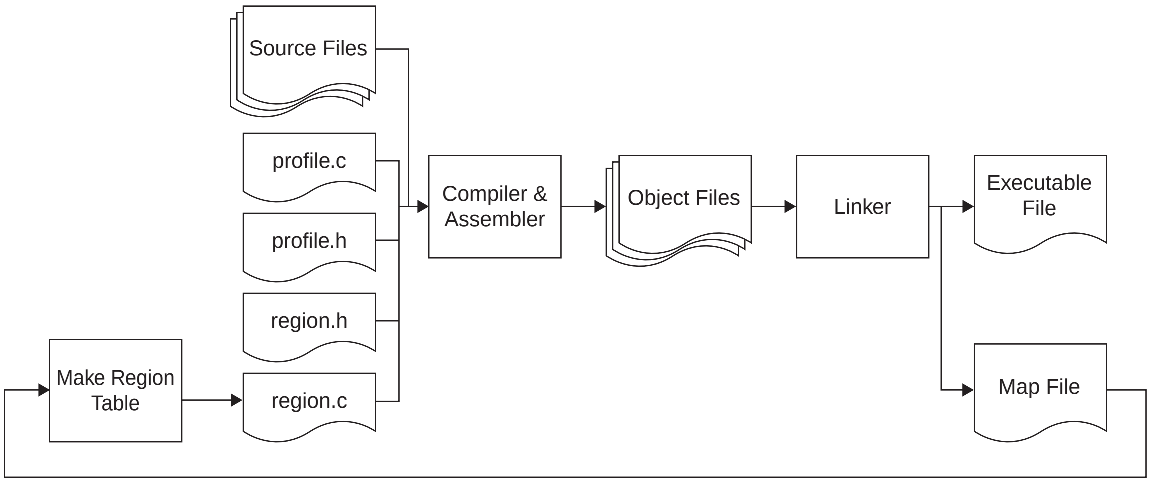


Figure 4.2 Modified build process includes dependency on map file.

Our build process is now more complicated, because the region table depends on the map file, as shown in Figure 4.2. The map file is not created until after the program is fully compiled and linked, so we will need to rebuild the program several times. With suitable tools this build process can be automated.

- We first **build** the program using a dummy region.c file, which contains an empty region table. The resulting map file has the correct number of functions, but with addresses which will probably change, so they are wrong.

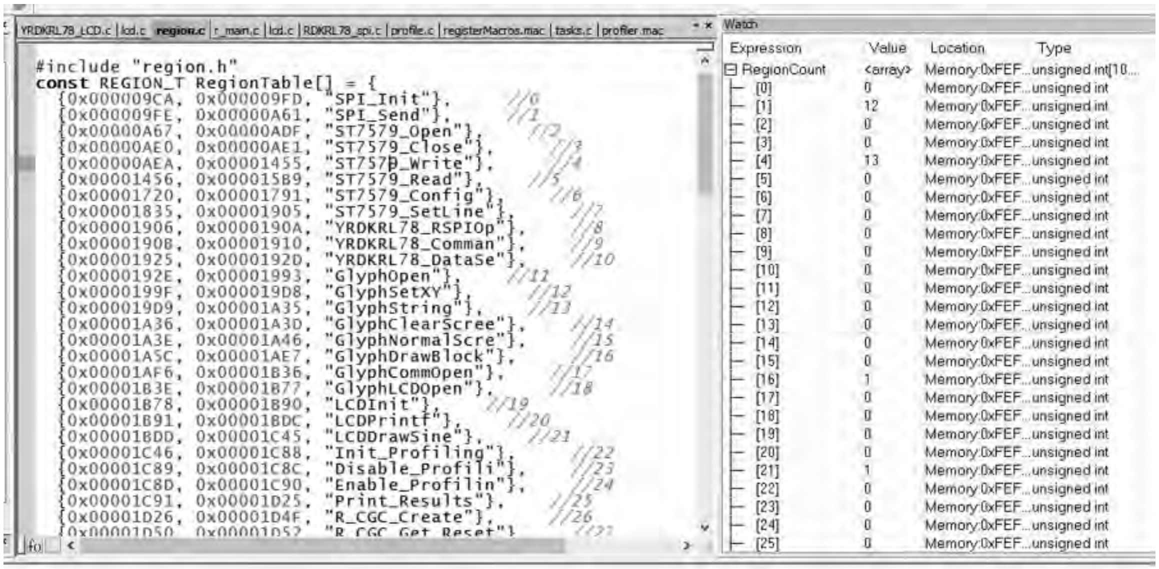
- We run our tool to **create the region table** from the map file. The region table now has the correct number of entries, but the addresses are wrong.
- We **rebuild** the program. The resulting map file has the correct (final) function addresses.
- We run our tool to **create the region table** from the map file.
- We **rebuild** the program for the final time. The resulting executable contains a region table with correct address information for each function.

4.3.2.4 Running the Program

We are now ready to run the program on the target hardware using input data for the test case(s) that we are interested in examining. We let the program run for a sufficiently long time, noting that there may be initialization and other program phases which execute before getting to program steady-state (or the phase we would like to measure). These phases may affect the profiling measurements, in which case we may want to wait to enable profiling until the program reaches a certain operating phase or location. We can then let the program run for a sufficient amount of time. Practically speaking, this means running the program long enough that the relative ratios of the region counts have stabilized. Then we can examine the resulting profile and the corresponding functions.

4.3.2.5 Examining the Resulting Profile

After running the program under appropriate conditions we are ready to examine the profile.

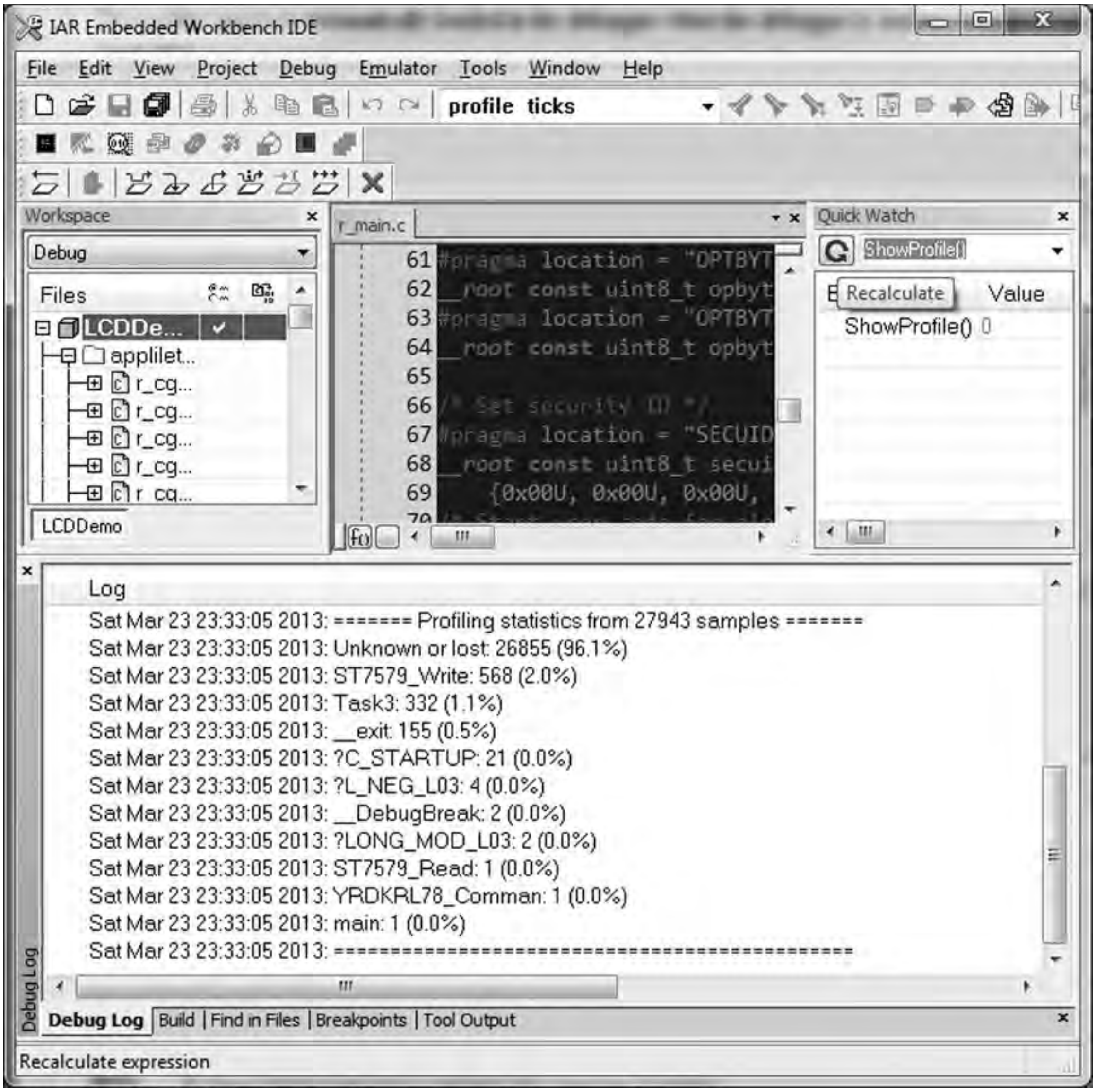


```
#include "region.h"
const REGION_T RegionTable[] = {
  {0x000009CA, 0x000009FD, "SPI_Init"},          //0
  {0x000009FE, 0x00000A61, "SPI_Send"},          //1
  {0x00000A67, 0x00000ADF, "ST7579_Open"},       //2
  {0x00000AE0, 0x00000AE1, "ST7579_Close"},      //3
  {0x00000AEA, 0x00001455, "ST7579_Write"},      //4
  {0x00001456, 0x000015B9, "ST7579_Read"},       //5
  {0x00001720, 0x00001791, "ST7579_Config"},     //6
  {0x00001835, 0x00001905, "ST7579_SetLine"},    //7
  {0x00001906, 0x0000190A, "YRDKRL78_RSPIOp"},   //8
  {0x0000190B, 0x00001910, "YRDKRL78_Comman"},   //9
  {0x00001925, 0x0000192D, "YRDKRL78_DataSe"},   //10
  {0x0000192E, 0x00001993, "GlyphOpen"},         //11
  {0x0000199F, 0x000019D8, "GlyphSetXY"},        //12
  {0x000019D9, 0x00001A35, "GlyphString"},       //13
  {0x00001A36, 0x00001A3D, "GlyphClearScree"},   //14
  {0x00001A3E, 0x00001A46, "GlyphNormalScre"},   //15
  {0x00001A5C, 0x00001AE7, "GlyphDrawBlock"},    //16
  {0x00001AF6, 0x00001B36, "GlyphCommOpen"},     //17
  {0x00001B3E, 0x00001B77, "GlyphLCDOpen"},      //18
  {0x00001B78, 0x00001B90, "LCDInit"},           //19
  {0x00001B91, 0x00001BDC, "LCDPrintf"},         //20
  {0x00001BDD, 0x00001C45, "LCDDrawSine"},       //21
  {0x00001C46, 0x00001C88, "Init_Profiling"},    //22
  {0x00001C89, 0x00001C8C, "Disable_Profili"},   //23
  {0x00001C8D, 0x00001C90, "Enable_Profilin"},   //24
  {0x00001C91, 0x00001D25, "Print_Results"},     //25
  {0x00001D26, 0x00001D4F, "R_CGC_Create"},      //26
  {0x00001D50, 0x00001D52, "R_CGC_Get_Reset"}    //27
```

```
Expression        Value     Location            Type
⊟ RegionCount     <array>   Memory:0xFEF...     unsigned int[10...
  ├─ [0]          0         Memory:0xFEF...     unsigned int
  ├─ [1]          12        Memory:0xFEF...     unsigned int
  ├─ [2]          0         Memory:0xFEF...     unsigned int
  ├─ [3]          0         Memory:0xFEF...     unsigned int
  ├─ [4]          13        Memory:0xFEF...     unsigned int
  ├─ [5]          0         Memory:0xFEF...     unsigned int
  ├─ [6]          0         Memory:0xFEF...     unsigned int
  ├─ [7]          0         Memory:0xFEF...     unsigned int
  ├─ [8]          0         Memory:0xFEF...     unsigned int
  ├─ [9]          0         Memory:0xFEF...     unsigned int
  ├─ [10]         0         Memory:0xFEF...     unsigned int
  ├─ [11]         0         Memory:0xFEF...     unsigned int
  ├─ [12]         0         Memory:0xFEF...     unsigned int
  ├─ [13]         0         Memory:0xFEF...     unsigned int
  ├─ [14]         0         Memory:0xFEF...     unsigned int
  ├─ [15]         0         Memory:0xFEF...     unsigned int
  ├─ [16]         1         Memory:0xFEF...     unsigned int
  ├─ [17]         0         Memory:0xFEF...     unsigned int
  ├─ [18]         0         Memory:0xFEF...     unsigned int
  ├─ [19]         0         Memory:0xFEF...     unsigned int
  ├─ [20]         0         Memory:0xFEF...     unsigned int
  ├─ [21]         1         Memory:0xFEF...     unsigned int
  ├─ [22]         0         Memory:0xFEF...     unsigned int
  ├─ [23]         0         Memory:0xFEF...     unsigned int
  ├─ [24]         0         Memory:0xFEF...     unsigned int
  ├─ [25]         0         Memory:0xFEF...     unsigned int
```

Figure 4.3 Raw profile information RegionCount and RegionTable arrays.

We begin by examining the raw data. As shown in Figure 4.3, we can use a C-Spy Watch window to examine the RegionCount table in the debug window of C-Spy. This provides a raw, unsorted view of the execution counts in the order of region number. It is helpful to bring up the RegionTable in another window to determine which function a given region represents. This is functional but tedious.



```
61 #pragma location = "OPTBYT
62 __root const uint8_t opbyt
63 #pragma location = "OPTBYT
64 __root const uint8_t opbyt
65
66 /* Set security ID */
67 #pragma location = "SECUID
68 __root const uint8_t secui
69     {0x00U, 0x00U, 0x00U,
```

```
Log
Sat Mar 23 23:33:05 2013: ======= Profiling statistics from 27943 samples =======
Sat Mar 23 23:33:05 2013: Unknown or lost: 26855 (96.1%)
Sat Mar 23 23:33:05 2013: ST7579_Write: 568 (2.0%)
Sat Mar 23 23:33:05 2013: Task3: 332 (1.1%)
Sat Mar 23 23:33:05 2013: __exit: 155 (0.5%)
Sat Mar 23 23:33:05 2013: ?C_STARTUP: 21 (0.0%)
Sat Mar 23 23:33:05 2013: ?L_NEG_L03: 4 (0.0%)
Sat Mar 23 23:33:05 2013: __DebugBreak: 2 (0.0%)
Sat Mar 23 23:33:05 2013: ?LONG_MOD_L03: 2 (0.0%)
Sat Mar 23 23:33:05 2013: ST7579_Read: 1 (0.0%)
Sat Mar 23 23:33:05 2013: YRDKRL78_Comman: 1 (0.0%)
Sat Mar 23 23:33:05 2013: main: 1 (0.0%)
Sat Mar 23 23:33:05 2013: ================================================
```

Figure 4.4 Debug log shows profile results in sorted and processed format.

The second method is more sophisticated and leverages C-Spy's macro capabilities. An enterprising student[5] in my class developed a solution which uses C-Spy to display region information (names, sample count, and percentage of total samples) sorted in order of decreasing frequency, as shown in Figure 4.4.[6] This makes analysis much easier. Detailed instructions for configuring and using this solution are included in the profiler documentation.

We can now determine where the program spends most of its time. We simply find the region with the largest number of profile hits and start looking at that region's C and the corresponding assembly code.

4.4 EXAMINING OBJECT CODE WITHOUT GETTING LOST

The abstractions provided by the C programming language and compiler allow us to generate sophisticated, powerful programs without having to pay attention to very many details. The downside of these abstractions is that we lose sight of the actual implementation details which may in fact be slowing down the program significantly.

One of the first steps in optimizing code performance is looking for obvious problems in code which dominates the execution time. Sometimes the compiler generates inefficient object code because of issues in the source code and the semantics of the C programming language. Chapter 5 focuses on how to help the compiler create fast, clean object code. First, however, we need to be able examine that object code so that we can identify suspicious code. Examining object code is a very time-consuming activity unless we have proper guidance and focus.

Let's consider a C function (shown below) which initializes an array with values of the sine function.

```
uint8_t SineTable[NUM_STEPS];

void Init_SineTable(void) {
   unsigned n;

   for (n = 0; n < NUM_STEPS; n++) {
      SineTable[n] = (uint8_t)
((MAX_DAC_CODE/2)*(1 + sin(n * 2 * 3.1415927/NUM_STEPS)));
   }
}
```

[5] Thanks to Daniel Wilson for all this! Professors love it when students come up with great ideas and run with them.

[6] Note that this shows the results of a different profiling run than shown in Figure 4.3.

4.4.1 Support for Mixed-Mode Viewing and Debugging

Most compilers provide the option to generate an **assembly language listing** annotated with comments containing the C source code. For example, in IAR Embedded Workbench this can be controlled by selecting Project -> Options -> C/C++ Compiler -> List -> Output list file. Each time a file is compiled the corresponding listing (.lst) file will be generated in List subdirectory of the output directory (e.g., Debug/List). Each line in this listing may show an address (e.g., 000001), the binary information stored there (e.g., machine code instruction C5), and the assembly language translation of that instruction (e.g., PUSH DE). Similarly, most debuggers provide a similar mixed-mode view for **debugging code.** With IAR Embedded Workbench, the View -> Disassembly window provides an assembly-language view of the program.

If we examine the compiler-generated assembly code listing for Init_SineTable (shown in Section 4.4.2) we can see many specific details but lose the big picture—we can't see the forest because of the trees. And this is just the code for one function! The listing for the complete program is almost overwhelming. Unfortunately, there are many crucial clues contained within these listing files. Some of them but not all are summarized in the linker's map file.

4.4.2 Understanding Function Calling Relationships

We expect a compiler processing Init_SineTable to insert calls to any functions which our code explicitly calls (i.e., sine). Similarly, we expect that using floating point variables on an integer-only MCU will cause the compiler to add calls to library routines which implement that functionality in software (multiply, divide, and add). We also should expect calls to data type conversion code (unsigned to float, float to unsigned).

But is there anything else? How can we determine how much code could actually execute as a result of calling a given function? After all, the more code which executes, the longer the program takes. The abstraction of programming in a high-level language may hide important implementation details (such as extra code) from us. Given our goal of reducing program run time, it is important to see how all the program's functions are related. This will give us insight into how to optimize the program.

4.4.2.1 Examining Object Code

So, which subroutines could a function actually call? We could search the listing file shown below for subroutine CALL instructions but this quickly becomes tedious, especially when we consider that each function may call others (and each of those may call others, and so on).

```
        void Init_SineTable(void) {
          Init_SineTable:
```

```
000000         ; * Stack frame (at entry) *
000000         ; Param size: 0
000000 C3      PUSH   BC   ;; 1 cycle
000001 C5      PUSH   DE   ;; 1 cycle
000002         ; Auto size: 0
            unsigned n;

               for (n = 0; n < NUM_STEPS; n++) {
000002 F6         CLRW   AX   ;; 1 cycle
000003 14         MOVW   DE, AX   ;; 1 cycle
000004 EF41       BR   S:??DAC_Test_0   ;; 3 cycles
000006            ; ---------------------------- Block: 7 cycles
               SineTable[n] = (uint8_t)
((MAX_DAC_CODE/2)*(1+sin(n*2*3.1415927/NUM_STEPS)));
               ??Init_SineTable_0:
000006 30FE42     MOVW   AX, #0x42FE   ;; 1 cycle
000009 C1         PUSH   AX            ;; 1 cycle
00000A F6         CLRW   AX            ;; 1 cycle
00000B C1         PUSH   AX            ;; 1 cycle
00000C 30803F     MOVW   AX, #0x3F80   ;; 1 cycle
00000F C1         PUSH   AX            ;; 1 cycle
000010 F6         CLRW   AX            ;; 1 cycle
000011 C1         PUSH   AX            ;; 1 cycle
000012 30803C     MOVW   AX, #0x3C80   ;; 1 cycle
000015 C1         PUSH   AX            ;; 1 cycle
000016 F6         CLRW   AX            ;; 1 cycle
000017 C1         PUSH   AX            ;; 1 cycle
000018 304940     MOVW   AX, #0x4049   ;; 1 cycle
00001B C1         PUSH   AX            ;; 1 cycle
00001C 30DB0F     MOVW   AX, #0xFDB    ;; 1 cycle
00001F C1         PUSH   AX            ;; 1 cycle
000020 15         MOVW   AX, DE        ;; 1 cycle
000021 01         ADDW   AX, AX        ;; 1 cycle
000022 F7         CLRW   BC            ;; 1 cycle
000023 FD ....    CALL   N:?F_UL2F     ;; 3 cycles
000026 FD ....    CALL   N:?F_MUL      ;; 3 cycles
000029 1004       ADDW   SP, #0x4      ;; 1 cycle
00002B FD ....    CALL   N:?F_MUL      ;; 3 cycles
00002E 1004       ADDW   SP, #0x4      ;; 1 cycle
000030 FD ....    CALL   sin           ;; 3 cycles
000033 FD ....    CALL   N:?F_ADD      ;; 3 cycles
000036 1004       ADDW   SP, #0x4      ;; 1 cycle
000038 FD ....    CALL   N:?F_MUL      ;; 3 cycles
00003B FD ....    CALL   N:?F_F2SL     ;; 3 cycles
00003E 60         MOV    A, X          ;; 1 cycle
00003F C5         PUSH   DE            ;; 1 cycle
000040 C2         POP    BC            ;; 1 cycle
000041 48 ....    MOV    (SineTable & 0xFFFF)[BC], A ;; 1 cycle
   }
000044 A5         INCW   DE            ;; 1 cycle
000045 1004       ADDW   SP, #0x4      ;; 1 cycle
000047            ; ---------------------------- Block: 49 cycles
               ??DAC_Test_0:
000047 15         MOVW   AX, DE        ;; 1 cycle
000048 444000     CMPW   AX, #0x40     ;; 1 cycle
00004B DCB9       BC     ??Init_SineTable_0 ;; 4 cycles
00004D            ; ---------------------------- Block: 6 cycles
   }
00004D C4         POP    DE            ;; 1 cycle
00004E C2         POP    BC            ;; 1 cycle
00004F D7         RET                  ;; 6 cycles
000050            ; ---------------------------- Block: 8 cycles
000050            ; ---------------------------- Total: 70 cycles
```

We see that Init_SineTable can call the functions ?F_UL2F, ?F_MUL (three times), ?F_ADD, and F_F2SL. These are library functions for converting an unsigned long integer to a floating point value, a floating point multiply, a floating point add, and converting a floating point value to a signed long integer.

This is a start. However, can these functions call any other functions? We won't know without examining the assembly code for those functions. Assuming that we have that code (which is not usually the case for library code) we can do this manually but it is quite tedious.

4.4.2.2 Call Graphs

A **call graph** (shown in Figure 4.5) presents all possible calling relationships between functions clearly and concisely (Cooper & Torczon, 2011). Nodes are connected with directed edges pointing toward the called function because calling a function is unidirectional (the return is implicit).

Some code analysis tools can automatically create call graphs. Those which analyze only the **source code** exclude the helper library functions which the compiler will need to link in to create a functioning program. For example, building a call graph using the C source code in Section 4.4 rather than the object code in Section 4.4.2 would have resulted in a call graph in which Init_SineTable calls only the function sine. None of the other functions called by Init_SineTable (whether directly or indirectly) would be present in the graph, misleading us in our analysis.

There are other call graph generator tools which analyze the **object code** and therefore include those additional functions. For our purposes of profiling and optimization we need these in order to get a complete picture. The linker map file typically includes information on the calling relationships between functions. The same enterprising student from my class also developed a tool to form call graphs (Figure 4.5 is one example) from map files generated by IAR Embedded Workbench.

4.4.2.3 Call Graph Analysis

Examining the call graph in Figure 4.5 shows that the program is more complex than we might have expected. The sine function may call __iar_Sin, which in turn may call __iar_Quad, ?FCMP_LT, ?FCMP_EQ, __iar_Errno, ?F_NEG_L04, and other functions. Some of these may call further other functions. In fact, calling sine may lead to executing a total of twenty-nine different functions, and some of them may be called multiple times. This is quite a bit of code, and it may account for a significant amount of program execution time, so it is worth investigating.

4.4.2.4 Forward Reference: Stack Space Requirements

In Chapter 9 we will examine how to measure and reduce memory requirements (both RAM and ROM). The call graph helps us determine the maximum possible stack size and therefore allocate a safe amount of RAM.

The call graph shows the nesting of function calls and this influences the amount of RAM required to hold the procedure call stack. RAM size is strongly correlated with MCU price, so designers of cost-sensitive systems would like to use the minimum possible. However, if the stack overflows its allotted space then the program will malfunction. We would like to allocate just enough stack space (to ensure safety) but not too much (to minimize RAM costs).

We can calculate maximum call stack space required by examining the stack depth at each leaf node.[7] The stack space required at a given node *N* in the call graph is the sum of the size of each activation record on a path beginning at graph's root node (main) and ending at node *N*, including both nodes.

One difference between the preemptive and non-preemptive scheduling approaches described in Chapters 2 and 3 is the amount of RAM required for call stacks. A non-preemptive scheduler requires **only one call stack,** and **shares this stack space** over

[7] In a call graph, a leaf node does not call any subroutines.

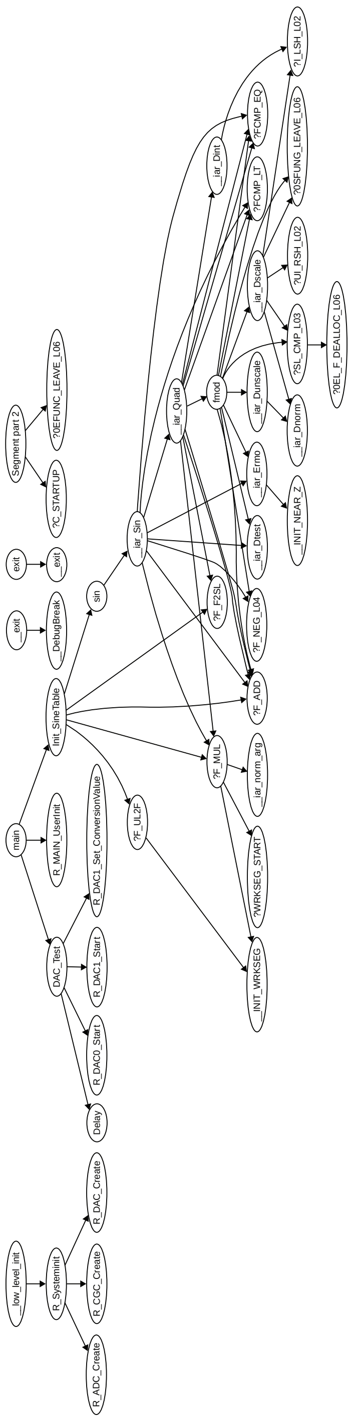


Figure 4.5 Call graph of a portion of the sample program including the Init_SineTable function.

time with the different tasks as they execute sequentially. Only the **largest task stack** needs to fit into RAM. However, a preemptive scheduler requires **one call stack for each task** because preemptions could occur at any point in a task's execution (i.e., any point within that task's call graph). Much of the task's state is stored on the stack, so that must be preserved and not used by other tasks. As a result, a preemptive system needs enough RAM to hold **all task stacks simultaneously,** with each potentially at its largest. Systems with many tasks are quite sensitive to overestimation of stack-depth requirements.

4.4.3 Understanding Function Basics

We can now look into a particular function to examine its basic internal features and methods. We are interested in understanding how the function's object code is related to the source code. For further details, please refer to Chapter 6 of the introductory text (Dean & Conrad, 2013).

Typically each source function is compiled into a separate object language function, but the compiler may optimize the program by eliminating the function or copying its body into the calling function. The code for the function consists of a **prolog,** a **body,** and an **epilog.** Each function also contains instructions to **manage local data storage** in its activation record on the call stack.

- The prolog prepares space for local storage. For example, it may save onto the stack registers which would need to have their values preserved upon returning from the function. It may allocate stack space for automatic variables and temporary results which are only needed within the scope of this function. Finally, it may also move or manipulate parameters which were passed to this function.
- The body of the function performs the bulk of the work specified in the source code.
- The epilog undoes some of the effects of the prolog. It restores registers to their original values as needed, deallocates stack space, possibly places the return value in the appropriate location, and then executes a return instruction.

Functions calls and returns are supported as follows:

- Calling a function involves first moving any parameters into the appropriate locations (registers or stack) according to the compiler's parameter passing conventions. The code then must execute subroutine call instruction.
- Upon returning from a function, argument space may need to be deallocated from the stack. A return value will be in a register or on the stack according to the compiler's value return convention.

How data is accessed depends on its storage class:

- External and static variables have fixed addresses and can be accessed using absolute addressing or a pointer register.
- Automatic variables may be located in registers or on the function's **activation record** (frame) on the call stack. Data on the activation record is accessed using the stack pointer as a base register and an offset which indicates the location within the activation record.
- Some variables may be promoted temporarily to registers by the compiler in order to improve program speed.

Function prologs and epilogs are generally easy enough to understand as they are simple and execution flows straight through from the first instruction to the last without exceptions. However, function bodies tend to be more difficult to understand because their flow of execution changes with loops and conditionals. These **control flow** changes make understanding assembly code much more difficult.

4.4.4 Understanding Control Flow in Assembly Language

Programming languages such as C use braces and indentation to indicate nested control flow behavior. Code which may be repeated or performed optionally is indented, providing a visual cue about the program's behavior.

Assembly code is typically formatted in order to simplify parsing by the assembler. Specific fields begin at specific columns, or after a fixed number of tab characters. All instructions have the same level of indentation, regardless of the amount of control nesting. Similarly, labels indicating branch targets are placed at another level of indentation. Instruction op-codes are at another level, and operands at yet another. This obscures program control flow and makes assembly code examination tiring and error prone.

4.4.4.1 Control Flow Graph

A **control flow graph** (**CFG,** similar to a flowchart) shows the flow of program control (execution) through a function. Jumps, loops, conditionals, breaks, and continues are examples of types of control flow. There are two types of CFG, based on the type of code analyzed. A CFG based on **source code** does not consider assembly code and excludes the impact of the compiler. Some compilers and toolchains generate (or can be extended to generate) such CFGs.

However, we need to understand the object code details in order to perform our code analysis and optimization. We need a CFG which represents the **object code.** Each node in such a CFG represents a **basic block,** a set of consecutive assembly instructions without any control flow changes (Cooper & Torczon, 2011). If an instruction in the basic block executes once, then every other instruction will also execute exactly once.[8] A basic block is **indivisible** from the point of view of program control flow. A conditional branch can only be the last instruction in a basic block. Similarly, a branch target can only be the first instruction in a basic block. CFGs and CGs are essential representations in the static timing analysis tools described in Chapter 3.

A CFG generator program parses the object code in order to create an accurate and complete CFG. This means that the parser must be able to understand which instructions change the program's flow of control and how to determine their targets. Hence a CFG generation tool must be targeted to a specific instruction set architecture.

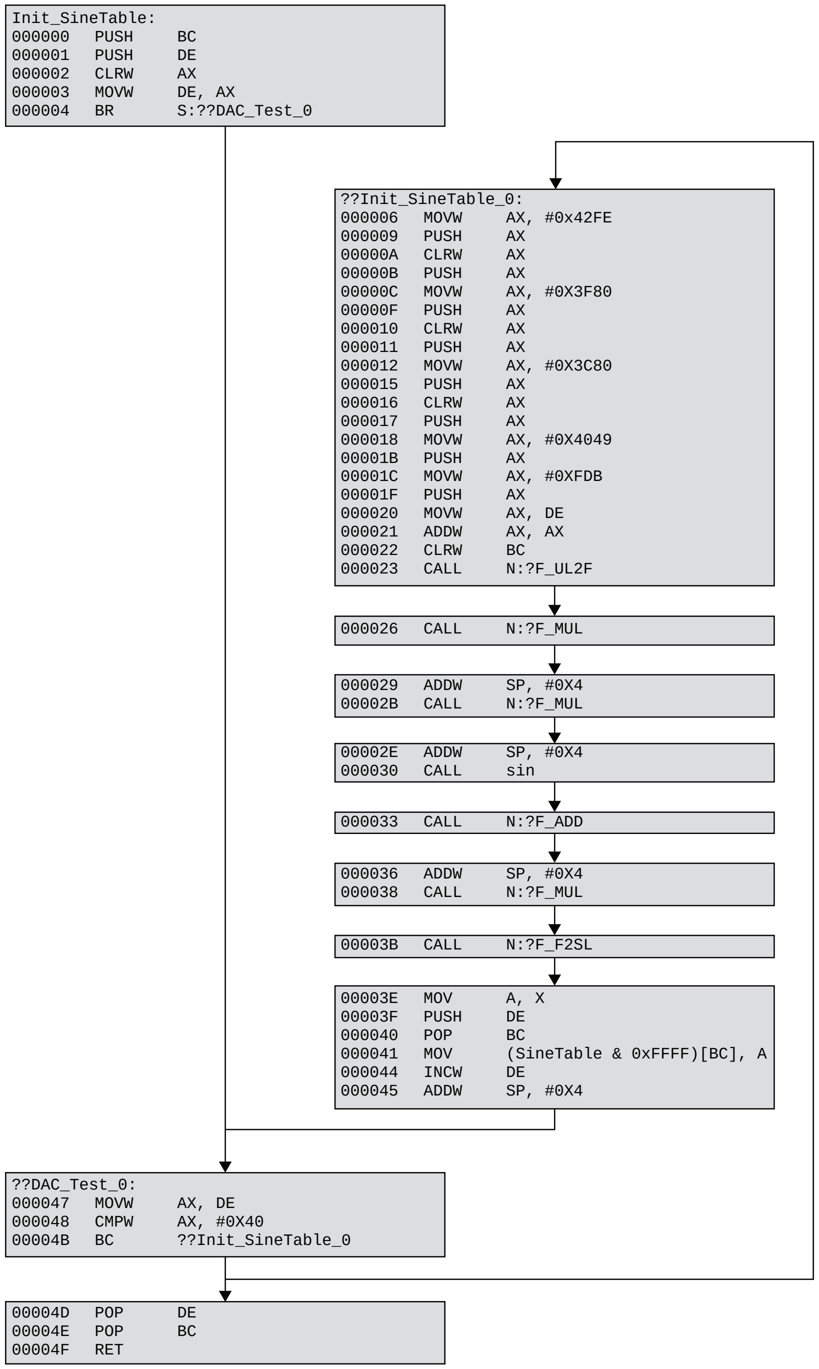
If we have no tool for our instruction set architecture we will need to use the next best solution. We will instead rely on the compiler's or debugger's mixed-mode listings, with the interleaved C code providing guidance on the control flow behavior of the object code.

4.4.4.2 Control Flow Graph Analysis

Figure 4.6 shows the control-flow graph of the object code for the Init_SineTable function. Using this graphical representation we can clearly see the object code's control flow as well the specifics.

- The first basic block (with label **Init_SineTable**) includes the prolog (saving registers BC and DE on the stack) as well as the initialization of the loop control variable (in register DE). The first basic block ends with an unconditional branch to the basic block **??DAC_Test_0.**
- Basic block **??DAC_Test_0** performs the loop test. Recall that the C source code uses a **for** loop, which is a **top-test** loop. This means the code must perform the test before executing the first iteration of the loop body.
    - If the result of the loop test is **true,** then the conditional branch **BC ??Init_SineTable_0** is taken and the processor will branch to the basic block **??Init_SineTable_0.** That basic block begins the loop body.
    - If the result of the loop test is **false,** then the conditional branch will not change the program counter, and execution will instead continue with the next instruction, which is located immediately after the branch instruction. This is the fall-through or not-taken path.

[8] Note that this is under normal program execution and does not consider interrupts, which are normally outside the scope of the compiler.



```
Init_SineTable:
000000   PUSH     BC
000001   PUSH     DE
000002   CLRW     AX
000003   MOVW     DE, AX
000004   BR       S:??DAC_Test_0
```

```
??Init_SineTable_0:
000006   MOVW     AX, #0x42FE
000009   PUSH     AX
00000A   CLRW     AX
00000B   PUSH     AX
00000C   MOVW     AX, #0X3F80
00000F   PUSH     AX
000010   CLRW     AX
000011   PUSH     AX
000012   MOVW     AX, #0X3C80
000015   PUSH     AX
000016   CLRW     AX
000017   PUSH     AX
000018   MOVW     AX, #0X4049
00001B   PUSH     AX
00001C   MOVW     AX, #0XFDB
00001F   PUSH     AX
000020   MOVW     AX, DE
000021   ADDW     AX, AX
000022   CLRW     BC
000023   CALL     N:?F_UL2F
```

```
000026   CALL     N:?F_MUL
```

```
000029   ADDW     SP, #0X4
00002B   CALL     N:?F_MUL
```

```
00002E   ADDW     SP, #0X4
000030   CALL     sin
```

```
000033   CALL     N:?F_ADD
```

```
000036   ADDW     SP, #0X4
000038   CALL     N:?F_MUL
```

```
00003B   CALL     N:?F_F2SL
```

```
00003E   MOV      A, X
00003F   PUSH     DE
000040   POP      BC
000041   MOV      (SineTable & 0xFFFF)[BC], A
000044   INCW     DE
000045   ADDW     SP, #0X4
```

```
??DAC_Test_0:
000047   MOVW     AX, DE
000048   CMPW     AX, #0X40
00004B   BC       ??Init_SineTable_0
```

```
00004D   POP      DE
00004E   POP      BC
00004F   RET
```

Figure 4.6 Control flow graph for object code of Init_SinTable function.

- The loop body consists of a sequence of eight basic blocks beginning with **??Init_SineTable_0** (with relative address 000006). This code loads up various parameters onto the parameter call stacks and then calls a sequence of functions to process them. There are also stack-pointer adjustment instructions to free up stack space used for passing parameters. In the last basic block, the instruction at address 000041 moves the computed value into memory in the correct element of the array SineTable by using the address offset specified in the BC register, which is derived from the index variable stored in DE.
- The last basic block in the program (beginning with relative address 00004D) contains the epilog. The original values of the DE and BC registers are restored and a return from subroutine instruction pops the PC off the stack, allowing the calling function to resume.

4.4.4.3 Oddities

Notice that the compiler has structured and laid out the code somewhat unexpectedly. First, the code for performing the loop test is located **after the loop body,** even though this is a **top-test loop.** Second, the loop body pushes all of the arguments onto the stack before beginning to call those functions. Why didn't the compiler just generate code to push those arguments immediately before each call?

The simple answer is that the compiler had its own reasons and we don't know them. Does this create faster code? Smaller code? Is it easier to debug the object code? Was it easier for the compiler developers to write functions which do the code this way? Does it make it easier for later possible compiler passes to optimize? We don't know.

The lesson to take away is that a compiler has a tremendous amount of flexibility when compiling and optimizing a program. What is actually "under the hood" may be quite different from what we expect to see. When we are analyzing and optimizing software, if we limit ourselves to working at only the source code level then we are ignoring many details, some of which may be critical for us. We can do much better when we examine the actual object code so we can understand why the system does what it does and why. Understanding the object code is much easier when using visualization tools.

4.5 RECAP

In this chapter we have seen that determining **which code to optimize** is an essential step before considering **how to optimize it.** We have learned how to use **execution-time profiling** to find which code dominates the execution time. We have then seen how to **make sense of object code** based both on source code and program structure.

4.6 BIBLIOGRAPHY

Cooper, K., & Torczon, L. (2011). *Engineering a Compiler* (2nd ed.). Morgan Kaufmann.

Dean, A. G., & Conrad, J. M. (2013). *Creating Fast, Responsive and Energy-Efficient Embedded Systems using the Renesas RL78 Microcontroller.* Weston, FL: Micrium Press.

Knuth, D. Turing Award Lecture: *Computer Programming as an Art, Communications of the ACM*, 17(12), Dec. 1974.

Chapter 5

Using the Compiler Effectively

5.1 LEARNING OBJECTIVES

This chapter addresses **how to optimize** a program by using the compiler more effectively. We examine configuration options for the compiler and other parts of the toolchain. Then we explore what optimizations the compiler should be able to perform, and how to help it do them. Finally we evaluate how to reduce computations by precomputing data before run-time or by reusing data calculated at run-time.

5.2 BASIC CONCEPTS

Figure 5.1 shows the various stages in software development, progressing from requirements at the top to object code at the bottom. First, the developer creates a high-level design, which involves selecting the specific architectures and algorithms that define what system components will exist and how they will operate in order to meet the requirements. The developer then implements the algorithms with a detailed design which leads

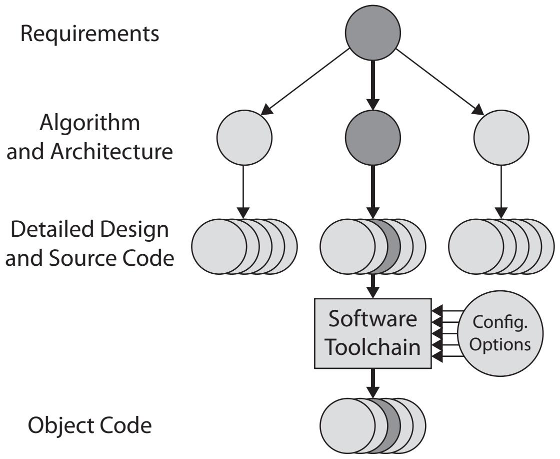


Figure 5.1 Opportunities for optimization in the software development process.

to **source code.** The source code is compiled to object code which can be executed and evaluated.

Note that at each level there are typically multiple approaches possible, but only one is selected, as indicated by the bold arrow. Similarly, there are typically many variants of a specific approach possible, as indicated by the overlapped circles. This means that there are opportunities for optimization at each of these levels. We can improve program performance in various ways: improving the algorithm and architecture, improving the detailed design, improving the source code implementing the detailed design, and improving the quality of the code created by the software toolchain.

In the previous chapter we learned how to use profiling to identify which code is slowing down the program the most, and therefore **what to optimize.** We also learned that which part of the program is slowest is often a surprise.

In this chapter we focus on improving the quality of the code generated by the software toolchain. This involves possibly changing detailed design, source code, and software toolchain configuration options. We examine two areas: how to configure the toolchain properly and how to help the compiler generate good code. We also examine methods for precomputing and reusing data.

An iterative and experimental approach is the best way to evaluate how well the compiler is doing its job, and to determine how to improve it. Examine the object code, modify the source code, recompile the module, and examine the new object code output. Repeat this process as needed.

In the next chapter we will examine how to improve the design at higher levels, touching on algorithms and architectures, as well as detailed design, source code, and mathematical representations and approximations.

5.2.1 Your Mileage Will Vary

We need to keep several points in mind as we consider the optimization process.

- Each program is structured differently and likely has a different bottleneck.
- There may be several different bottlenecks depending on which code is executing. A system with four different operating modes (or four different input events) may have four different bottlenecks, so be sure to profile the code for a variety of operating modes and input conditions.
- A program's bottleneck may move after an optimization is performed. After all, it is just the slowest part of the code. If it is optimized enough, then another piece of code becomes the slowest.
- Different processor architectures have different bottlenecks. Accessing memory in a deeply-pipelined 2 GHz processor may cost 500 cycles. On the RL78, however,

there is generally only a single cycle penalty. Hence optimizations which are effective on one processor architecture may be inconsequential on another.

- Different compilers use different approaches to generate code. Recall that there are many possible assembly language programs which can implement the specification given by a source-level program. One compiler may aggressively unroll loops, while another may not. If you manually try unrolling loops with the first compiler you likely will see no performance improvement. It is valuable to examine the optimization section of the compiler manual for guidance and suggestions.

There are many excellent guides to optimizing code and we do not attempt to duplicate them. In this chapter we examine how to help the compiler generate good code for the RL78 MCU.

5.2.2 An Example Program to Optimize

In order to illustrate the long and winding road of optimizations, let's consider a real program. We will use this program to provide specific optimization examples in this and the next chapter.

We would like to determine the distance and bearing from an arbitrary position on the surface of the earth to the nearest weather and sea state monitoring station. The US government's National Oceanographic and Atmospheric Administration (NOAA) monitors weather and sea conditions near the US using a variety of sensor platforms, such as buoy and fixed platforms. This information is used for weather forecasting and other applications. NOAA's National Data Buoy Center (http://www.ndbc.noaa.gov/ and http://www.ndbc.noaa.gov/cman.php) gathers information from many buoy-mounted (and fixed) platforms and makes it available online. The locations of these platforms are to be stored in the MCU's flash ROM.

Finding the distance and bearing between two locations on the surface of the earth uses spherical geometry. Locations are represented as latitude and longitude coordinates. We use the spherical law of cosines to compute the distance in kilometers:

$$d = \mathrm{acos}(\sin(lat_1) * \sin(lat_2) + \cos(lat_1) * \cos(lat_2) * \cos(lon_2 - lon_1)) * 6371$$

We compute the bearing (angle toward the location) in degrees as follows:

$$a = atan2(\cos(lat_1) * \sin(lat_2) - \sin(lat_1) * \cos(lat_2) * \cos(lon_2 - lon_1),\\ \sin(lon_2 - lon_1) * \cos(lat_2)) * \frac{180}{\pi}$$

Further details are available online at http://www.movable-type.co.uk/scripts/latlong.html. This is a mathematically intensive computation with many trigonometric functions, so we expect many opportunities for optimization.

Let's examine the relevant functions needed to do this work. The function Calc_Distance calculates the distance between two points.

```
float Calc_Distance( PT_T * p1, const PT_T * p2) {
   //calculates distance in kilometers between locations
   (represented in degrees)
   return acos(sin(p1->Lat*PI/180)*sin(p2->Lat*PI/180) +
      cos(p1->Lat*PI/180)*cos(p2->Lat*PI/180)*
      cos(p2->Lon*PI/180 - p1->Lon*PI/180))*6371;
}
```

The function Calc_Bearing calculates the bearing from the first to the second point.

```
float Calc_Bearing (PT_T * p1,  const PT_T * p2){
   //calculates bearing in degrees between locations
   (represented in degrees)
   float angle = atan2(
      sin(p1->Lon*PI/180 - p2->Lon*PI/180)*cos(p2->Lat*PI/180),
      cos(p1->Lat*PI/180)*sin(p2->Lat*PI/180) -
      sin(p1->Lat*PI/180)*cos(p2->Lat*PI/180)*
      cos(p1->Lon*PI/180 - p2->Lon*PI/180)
      ) * 180/PI;
   if (angle < 0.0)
      angle += 360;
   return angle;
}
```

The function Find_Nearest_Point calculates the distance to each point (in line 15) to find the one closest to the current position. It keeps track of the closest point's distance and index in lines 18–20.

```
1. void Find_Nearest_Point(float cur_pos_lat, float cur_pos_lon,
2.    float * distance, float * bearing,  char  * * name) {
3.    //cur_pos_lat and cur_pos_lon are in degrees
4.    //distance is in kilometers
5.    //bearing is in degrees
6.    int i = 0, closest_i;
7.    PT_T ref;
```

```
    float d, b, closest_d=1E10;
    *distance = *bearing = NULL;
    *name = NULL;
    ref.Lat = cur_pos_lat;
    ref.Lon = cur_pos_lon;
    strcpy(ref.Name, "Reference");
    while (strcmp(points[i].Name, "END")) {
       d = Calc_Distance(&ref, &(points[i]) );
       b = Calc_Bearing(&ref, &(points[i]) );
       //if we found a closer point, remember it and display it
       if (d<closest_d) {
          closest_d = d;
          closest_i = i;
       }
       i++;
    }
    d = Calc_Distance(&ref, &(points[closest_i]) );
    b = Calc_Bearing(&ref, &(points[closest_i]) );
    //return information to calling function about closest point
    *distance = d;
    *bearing = b;
    *name = (char * ) (points[closest_i].Name);
}
```

Note that there are various other functions (e.g., for initialization) in the program, but these three do the bulk of the work.

5.3 TOOLCHAIN CONFIGURATION

We begin by examining how to configure the toolchain. Although the default settings should produce correct code, we may be able to improve the code by changing the settings.

5.3.1 Enable Optimizations

Be sure to enable optimization in the project options, as it may not be enabled (or maximized) by default. Most compilers support several levels of optimization, often with selectable emphasis on speed or code size. It is often possible to override the project optimization options as needed for specific source modules (i.e., files). For example, we may want the compiler to optimize for speed in general, except for a module which is rarely executed but contains a large amount of code. Alternatively, size may be so important that we optimize every module for size except for those which dominate the execution time.

5.3.2 Use the Right Memory Model

Compilers for embedded systems typically support multiple memory models, varying in how much memory can be addressed. Using the smallest possible memory model can reduce the amount of code needed, speeding up the program and reducing memory requirements. This is because accessing more possible locations requires longer addresses and pointers, which in turn typically require more instructions and hence memory and execution time.

The RL78 ISA has a one megabyte address space, which requires twenty bits for addressing. Some RL78 addressing modes require the instruction to specify all twenty bits of the address. Others use implicit, fixed values for certain bits so the instruction can specify fewer bits, typically saving space and time.

The RL78 supports addresses and pointers of two sizes: 16 bits and 20 bits. Addressing memory with a 20 bit address involves using an additional 4-bit segment register (ES or CS) to specify the upper portion of the address. This requires the use of additional and slower instructions, reducing code performance and increasing code size.

Compilers and libraries for the RL78 provide memory models which match these address sizes. **Near** memory models can access up to 64 kilobytes of space, using 16-bit addresses. **Far** memory models can access the full one megabyte of space but require 20-bit addresses. Different memory models can be specified for code and data, allowing better optimization.

5.3.3 Floating Point Math Precision

Double-precision floating point math is excessive for most embedded system applications, needlessly slowing and bloating programs. Functions in math libraries often use double-precision floating point math for arguments, internal operations, and return values.

Compilers which target embedded applications may offer an alternative single-precision version of the math library, or single-precision functions within the double-precision library (e.g., sinf vs. sin). Others may only offer the single-precision library (e.g., IAR Embedded Workbench for RL78), or enable all doubles to be treated as single precision floats. Finally, some embedded compilers may allow the user to select between floating point math libraries with increased speed or increased precision (e.g., IAR Embedded Workbench for RL78).

In the next chapter we will examine how to reduce or eliminate the need for floating point math.

5.3.4 Data Issues

5.3.4.1 Data Size

Use the smallest practical data size. Data which doesn't match the machine's native word size will require extra instructions for processing. The native data sizes for the RL78 architecture are the bit, byte, and 16-bit word.

5.3.4.2 Signed vs. Unsigned Data

Some ISAs offer unequal performance for signed and unsigned data, so there may be a benefit to using one type or another. The IAR Compiler manual recommends using unsigned data types rather than signed data types if possible.

5.3.4.3 Data Alignment

Some memory systems offer non-uniform access speeds based on alignment. For example, the RL78 has word-aligned memory, so smaller elements in a structure (e.g., chars) may result in padding bytes, and therefore wasting memory.

5.4 HELP THE COMPILER DO A GOOD JOB

5.4.1 What Should the Compiler be Able to Do on Its Own?

The compiler should be able to perform certain optimizations on its own if you enable optimization in the compiler options. Don't waste your time performing these transformations manually because the compiler should do them automatically.

- Perform compile-time math operations.
- Reuse a register for variables which do not interfere.
- Eliminate unreachable code, or code with no effect.
- Eliminate useless control flow.
- Simplify some algebraic operations (e.g., $x * 1 = x$, $x + 0 = x$).
- Move an operation to where it executes less often (e.g., out of a loop).
- Eliminate redundant computations.
- Reuse intermediate results.
- Unroll loops.
- Manually inline functions (instead use macros).

5.4.2 What Could Stop the Compiler?

Sometimes the compiler can't perform these optimizations, so it helps to examine the object code and determine whether they occurred or not. If they didn't, and the code would benefit significantly, then it makes sense to investigate why the compiler didn't perform them and possibly implement them manually.

Compilers are very careful when it comes to optimizations—the "dark corners" of the semantics of the C language probably allow our program to behave in a certain way which would cause the optimized code to be incorrect. Code should be written to make it clear to the compiler which side effects are impossible, enabling the optimizations. Another reason is that the compiler has a harder time identifying optimization opportunities across more complex program structure boundaries (or when accessing memory).

In this section we examine C language semantics in order to understand their impact on the compiler's possible optimization. This subject is covered in further detail in Jakob Engblom's article "Getting the Least out of your C Compiler"(Engblom, 2002).

5.4.2.1 *Excessive Variable Scope*

Creating variables as globals or statics when they could be local instead (automatics and parameters) limits the compiler in two ways, resulting in slower, larger code and larger memory requirements.

First, because global variables have a program-wide scope, they can be accessed by any function. Consider function f, which performs some operations on global variables. The compiler may be able to optimize the code by loading these variables into registers, operating on them, and then writing the values back to the global memory locations before returning from f. However, if f calls a function g, then the compiler must write the global values back to memory before calling the function. After returning from the function, the compiler will need to reload these global values if they are used again. This adds instructions which slow the program and increase code memory size.

Second, global and static variables are allocated **permanent** storage in data memory. This is because the compiler must assume these variables are alive for the entire duration of the program. The compiler cannot reuse this memory for other variables, reducing available space.

5.4.2.2 *Automatic Type Promotion*

What code will the compiler generate if we try to mix data types in an expression? For example, how is a float variable multiplied by a char variable?

```
float f;
char c;
```

```
3. int r;
4. r = f * c;
```

The compiler does not generate code to perform the mixed multiplication directly. There is no library function to perform this mixed multiplication directly either. Instead, the compiler calls a library routine to **convert (promote) the char variable's data to float type.** Then the compiler can generate code to **multiply the two float values** together, in this case using a call to the floating point math library. Now that we have calculated the result, we need to store it in an int (integer) variable. The compiler will generate a call to **convert the float to an int,** and then store the result in *r*.

The resulting object code must do this:

- call subroutine to convert *c* from char to float
- call subroutine to perform floating point multiply with *f*
- call subroutine to convert result from floating point to integer
- store resulting integer in *r*

So there is quite a bit of work resulting from our simple "$r = f * c$;" expression. Using data types consistently can both reduce the overhead of conversions as well as improve the speed of the actual mathematical operations performed.

This promotion and conversion behavior is defined by automatic type promotion rules in ANSI C. The goal is to preserve the accuracy of data while limiting the number of cases which the compiler and library must support. These promotions may lead to library calls which use additional time and memory. As these promotions aren't immediately obvious in the source code, it is often valuable to examine the generated assembly code when dealing with mixed conversions.

TABLE 5.1 ANSI C Standard for Arithmetic Conversions, Omitting Rules for Converting Between Signed and Unsigned Types

1	If either operand is a long double, promote the other to a long double
2	Else if either is a double, promote the other to a double
3	Else if either is a float, promote the other to a float
4	Else if either is an unsigned long int, promote the other to an unsigned long int
5	Else if either is a long int, promote the other to a long int
6	Else if either is an unsigned int, promote the other to an unsigned int
7	Else both are promoted to int: short, char, bit field

Keep in mind that functions in the math library (prototyped in math.h) are often double-precision floating point. This can cause the compiler to promote all arguments to double-precision floats, and then potentially convert results back to a lower precision format. This wastes time and memory and should be avoided.

5.4.2.3 Operator Precedence and Order of Evaluation

The type promotions described above will depend on the order in which an expression's terms are evaluated. This order is determined by the operator's **precedence** and **associativity.** Precedence determines which type of operator is evaluated first, while associativity specifies the order to evaluate multiple adjacent operators of the same precedence level. The C language semantics are shown in Table 5.2. Use parentheses as needed to change the order of term evaluation in an expression.

TABLE 5.2 C Operator Precedence and Associativity

OPERATOR NAME	OPERATOR	ASSOCIATIVITY
Primary	`() [] . -> ++(post) --(post)`	left to right
Unary	`* & + -! ~ ++(pre) --(pre) (typecast) sizeof()`	right to left
Multiplicative	`* / %`	left to right
Additive	`+ -`	
Bitwise Shift	`>> <<`	
Relational	`<> <= >=`	
Equality	`== !-`	
Bitwise AND	`&`	
Bitwise Exclusive OR	`^`	
Bitwise Inclusive OR	`|`	
Logical AND	`&&`	
Logical OR	`||`	
Conditional	`? :`	right to left
Assignment	`= += -= *= /= %= >>= <<= &= ^= |=`	right to left
Comma	`,`	left to right

```
1. a = b + c * d - e % f / g;
```

For example, the expression above will be evaluated in the order shown in Table 5.3.

TABLE 5.3 Order of Evaluation of Example Code

STEP	A =	B +	C * D	–	E % F	/G
1			c * d			
2					e % f	
3						/g
4		b +				
5				–		
6	a =					

5.5 PRECOMPUTATION OF RUN-TIME INVARIANT DATA

One particularly effective optimization is to pre-compute data which does not change when the program is running (run-time invariant data). How much computation can be done before the program even starts running? The compiler should be able to perform some, while other work may need to be handled with a custom tool such as a spreadsheet which generates data tables. In this section we first examine what the compiler can do, and then what we can do before even running the compiler.

5.5.1 Compile-Time Expression Evaluation

The functions Calc_Distance and Calc_Bearing have many operations on constants—specifically calculating PI/180. The compiler should be able to perform these divisions at compile-time. Let's examine at the assembly code for Calc_Distance and find out. The compiler is set for high optimization, targeting speed without any size constraints.

```
float Calc_Distance( PT_T * p1,  const PT_T * p2) {
   //calculates distance in kilometers between locations
   (represented in degrees)
   return acos(sin(p1->Lat*PI/180)*sin(p2->Lat*PI/180) +
      cos(p1->Lat*PI/180)*cos(p2->Lat*PI/180)*
      cos(p2->Lon*PI/180 - p1->Lon*PI/180)) * 6371;
}
```

TABLE 5.4 Function Calls in Calc_Distance

OPERATION	CALL TARGET	COUNT EXPECTED FROM SOURCE CODE	COUNT MEASURED FROM OBJECT CODE
Arc Cosine	ACOS	1	1
Sine	__iar_Sin	2	5
Cosine	?	3	0
Floating-Point Multiply	F_MUL	10	10
Floating-Point Add	F_ADD	1	1
Floating-Point Subtract	F_SUB	1	1
Floating-Point Divide	F_DIV	0	6

We would expect certain subroutine calls to implement the floating point operations, as shown in the first two columns of Table 5.4. However, the actual object code has some additional calls to F_DIV, the floating point division library subroutine. What is the compiler doing and why?

The compiler should be able to perform the division of PI/180 at compile-time, eliminating the need for run-time division calls. However, the C source code has six floating point divides, so it is obvious that the compiler isn't doing that possible optimization. It seems that the C language's operator precedence rules are getting in the way in the expression p1->Lat*PI/180. Multiplication and division are the same level of precedence and are left-associative, so p1->Lat * PI is performed first, and then the division by 180 is performed. Let's put parentheses around all of the PI/180 terms. With this change the compiler performs the division at compile time and eliminates the calls to F_DIV.

Another unexpected result is the missing three calls to the cosine function and the three extra calls to the sine function __iar_Sin. The compiler is likely using a trigonometric identify to calculate the cosine using the sine function.

5.5.2 Precalculation before Compilation

If we examine Calc_Distance and Calc_Bearing we find that the second parameter to each function is a pointer to a "const" point—one stored in the array points. These points are defined at compile time and then will not change again until we update the list. We could save quite a bit of time with two types of pre-computation:

- Storing a point's latitude and longitude in radians rather than degrees, avoiding the need to convert at run-time.

- Storing the derived trigonometric values. Calc_Distance and Calc_Bearing both use the sine of latitude and cosine of latitude.

So we will modify the spreadsheet we used to create CMAN_coords.c to precompute these values. We will also need to modify the type definition of PT_T to include the sine and cosine of the latitude. This actually simplifies the code quite a bit, as shown below.

```
float Calc_Distance( PT_T * p1,  const PT_T * p2) {
//calculates distance in kilometers between locations (represented in radians)
   return acos(p1->SinLat * p2->SinLat +
   p1->CosLat * p2->CosLat *cos(p2->Lon - p1->Lon)) * 6371;
}
float Calc_Bearing( PT_T * p1,  const PT_T * p2){
//calculates bearing in degrees between locations (represented in degrees)
   float angle = atan2(
      sin(p1->Lon - p2->Lon)* p2->CosLat,
      p1->CosLat * p2->SinLat -
      p1->SinLat * p2->CosLat * cos(p1->Lon - p2->Lon)
      ) * 180/PI;
   if (angle < 0.0)
      angle += 360;
   return angle;
}
```

We will also modify the code in Find_Nearest_Point to convert the current location to radians and save the sine and cosine of latitude.

```
void Find_Nearest_Point(float cur_pos_lat, float cur_pos_lon, float * distance, float * bearing,
   char  * * name) {
   //cur_pos_lat and cur_pos_lon are in degrees
   //distance is in kilometers
   //bearing is in degrees

   int i=0, closest_i;
   PT_T ref;
   float d, b, closest_d=1E10;

   *distance = *bearing = NULL;
   *name = NULL;

   ref.Lat = cur_pos_lat*PI_DIV_180;
   ref.SinLat = sin(ref.Lat);
   ref.CosLat = cos(ref.Lat);
   ref.Lon = cur_pos_lon*PI_DIV_180;
   strcpy(ref.Name, "Reference");
```

5.6 REUSE OF DATA COMPUTED AT RUN-TIME

The compiler may also be able to reuse data which the program has already computed, reducing program size and execution time. One method common to many compilers is called common sub-expression elimination. In this section we examine a function with many opportunities for this type of optimization and then evaluate how well the compiler uses them.

5.6.1 Starting Code

```
float Calc_Bearing( PT_T * p1,  const PT_T * p2){
//calculates bearing in degrees between locations
(represented in degrees)
   float angle = atan2(
      sin(p1->Lon*(PI/180) - p2->Lon*(PI/180))*
      cos(p2->Lat*(PI/180)),
      cos(p1->Lat*(PI/180))*sin(p2->Lat*(PI/180)) -
      sin(p1->Lat*(PI/180))*cos(p2->Lat*(PI/180))*
      cos(p1->Lon*(PI/180) - p2->Lon*(PI/180))
      ) * (180/PI);
   if (angle < 0.0)
      angle += 360;
   return angle;
}
```

Let’s examine the Calc_Bearing function for terms which may be reused. We see that certain terms appear more than once:

- p1->Lon*(PI/180) appears twice
- p2->Lon*(PI/180) appears twice

- p2->Lat*(PI/180) appears three times
- p1->Lat*(PI/180) appears twice

The source code has fourteen floating point multiplies (*). We expect the number of multiplications to be reduced as the compiler optimizes by reusing previous results. After compiling at maximum optimization for speed, we look at the object code. There are fourteen calls to F_MUL, so it appears that these terms computed are once for each appearance in the source code, rather than being reused.

5.6.2 First Source Code Modification

Perhaps the compiler is not reusing the results because dereferencing the pointers p1 and p2 may access global variables which could change? To evaluate this option let's load the terms from memory into local variables p1Lat, p1Lon, p2Lat, and p2Lon.

```
float Calc_Bearing( PT_T * p1,  const PT_T * p2){
   //calculates bearing in degrees between locations (represented in degrees)
   float p1Lon, p1Lat, p2Lon, p2Lat;
   float angle;

   p1Lon = p1->Lon;
   p2Lon = p2->Lon;
   p1Lat = p1->Lat;
   p2Lat = p2->Lat;

   angle = atan2(
      sin(p1Lon*(PI/180) - p2Lon*(PI/180))*
      cos(p2Lat*(PI/180)),
      cos(p1Lat*(PI/180))*sin(p2Lat*(PI/180)) -
      sin(p1Lat*(PI/180))*cos(p2Lat*(PI/180))*
      cos(p1Lon*(PI/180) - p2Lon*(PI/180))
      ) * (180/PI);
   if (angle < 0.0)
      angle += 360;
   return angle;
}
```

The resulting object code still has fourteen calls to F_MUL and it is not clear why the results are not reused.

5.6.3 Second Source Code Modification

Let's explicitly modify the source code to reuse the results. We will use local variables p1LonRad, p1LatRad, p2LonRad, and p2LatRad to hold them.

```
float Calc_Bearing( PT_T * p1,  const PT_T * p2){
   //calculates bearing in degrees between locations
   (represented in degrees)
   float p1LonRad, p1LatRad, p2LonRad, p2LatRad;
   float angle;

   p1LonRad = p1->Lon*(PI/180);
   p2LonRad = p2->Lon*(PI/180);
   p1LatRad = p1->Lat*(PI/180);
   p2LatRad = p2->Lat*(PI/180);

   angle = atan2(
      sin(p1LonRad - p2LonRad)*cos(p2LatRad),
      cos(p1LatRad)*sin(p2LatRad) -
      sin(p1LatRad)*cos(p2LatRad)*cos(p1LonRad - p2LonRad)
      ) * (180/PI);
   if (angle < 0.0)
      angle += 360;
   return angle;
}
```

The resulting object code has nine calls to F_MUL, as expected.

5.6.4 Third Source Code Modification

There are additional reuse opportunities. Examining the source code reveals that cos(p2LatRad) is calculated twice. Perhaps the optimizer could see this more clearly if we pulled these calculations out of the argument list? The resulting code follows.

```
float Calc_Bearing( PT_T * p1,  const PT_T * p2){
//calculates bearing in degrees between locations
(represented in degrees)
   float p1LonRad, p1LatRad, p2LonRad, p2LatRad;
   float term1, term2;
   float angle;

   p1LonRad = p1->Lon*(PI/180);
   p2LonRad = p2->Lon*(PI/180);
   p1LatRad = p1->Lat*(PI/180);
   p2LatRad = p2->Lat*(PI/180);

   term1 = sin(p1LonRad - p2LonRad)*cos(p2LatRad);
   term2 = cos(p1LatRad)*sin(p2LatRad) -
      sin(p1LatRad)*cos(p2LatRad)*cos(p1LonRad - p2LonRad);
   angle = atan2(term1, term2) * (180/PI);
   if (angle < 0.0)
      angle += 360;
   return angle;
}
```

The resulting object code still has seven calls to __iar_Sin.

5.6.5 Fourth Source Code Modification

It looks like we will have to force the compiler to reuse the result. We will create a local variable called cosp2LatRad to hold it.

```
float Calc_Bearing( PT_T * p1,  const PT_T * p2){
//calculates bearing in degrees between locations (represented in degrees)
   float p1LonRad, p1LatRad, p2LonRad, p2LatRad;
   float cosp2LatRad;
   float term1, term2;

   float angle;

   p1LonRad = p1->Lon*(PI/180);
   p2LonRad = p2->Lon*(PI/180);
   p1LatRad = p1->Lat*(PI/180);
   p2LatRad = p2->Lat*(PI/180);
   cosp2LatRad = cos(p2LatRad);

   term1 = sin(p1LonRad - p2LonRad)*cosp2LatRad;
   term2 = cos(p1LatRad)*sin(p2LatRad) -
      sin(p1LatRad)*cosp2LatRad*cos(p1LonRad - p2LonRad);
```

```
    angle = atan2(term1, term2) * (180/PI);
    if (angle < 0.0)
       angle += 360;
    return angle;
}
```

Now there are only six calls to __iar_Sin because we have eliminated one. It is curious that the compiler was not able to reuse these expressions. This reinforces the importance of **examining the object code** to determine **which optimizations were performed.**

5.7 RECAP

In this chapter we focused on improving the quality of the code generated by the software toolchain. This involves possibly changing detailed design, source code, and software toolchain configuration options. We examined two areas: how to configure the toolchain properly and how to help the compiler generate good code. We also examined methods for precomputing and reusing data. We saw that it is important to examine object code to verify that the compiler performed the expected optimizations.

5.8 BIBLIOGRAPHY

Engblom, J. (2002). Getting the Least out of your Compiler. *Embedded Systems Conference.* San Francisco.

High-Level Optimizations

6.1 LEARNING OBJECTIVES

In this chapter we examine high-level approaches to improving program performance. These methods touch on algorithms and data structures as well as mathematical representations and approximations.

6.2 BASIC CONCEPTS

Figure 6.1 shows the various stages in software development, progressing from requirements at the top to object code at the bottom. First, the developer performs creates a high-level design, which involves selecting the specific architectures and algorithms which define what system components will exist and how they will operate in order to meet the

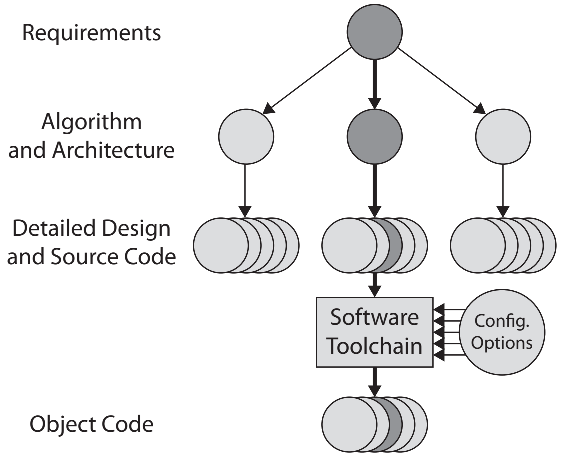


Figure 6.1 Opportunities for optimization in the software development process.

requirements. The developer then implements the algorithms with a detailed design which leads to **source code.**

Embedded systems typically offer many opportunities for creative optimizations. This is because the closed and well-defined nature of embedded systems software gives the developer great flexibility to optimize at multiple levels of the design hierarchy. The methods covered in this chapter typically require much more modification of source code (and therefore effort) than the methods in the previous chapter (which focused on helping the compiler generate good code). Because of these larger effort requirements, we should keep two factors in mind when considering performing these optimizations.

First, it is important to weigh the expected performance gain against the costs of the **development time** spent and the **schedule risk** which is added. Some of the high-level optimizations described in this chapter affect large amounts of a program's source code, **increasing development effort** and **raising schedule risk.** It can be difficult to predict the **quantitative performance benefits** of an optimization accurately before implementing it, introducing additional schedule risk. In comparison, the optimizations in the previous chapter can be implemented much more quickly and therefore with less schedule risk.

Second, **code maintainability** is important yet often suffers when code is optimized. When possible, the wise developer will implement software optimizations in a way which does not reduce its maintainability. The code is likely to be modified in the future for bug fixes, feature changes and upgrades, and as a platform for developing product families and downstream products. Code is usually too expensive to rewrite. Some optimization methods may make it more difficult to maintain the code. Other optimization methods may do the same if implemented badly.

6.3 ALGORITHMS

We begin by examining how to improve the algorithms used to do the work. At times this may require modifying the data structure to enable more efficient algorithms to be applied. A very detailed and thorough examination of algorithms and data structures can be found elsewhere (Knuth, 2011).

6.3.1 Less Computation: Lazy Execution and Early Exits

A common pattern of computation is **performing a calculation** and then **making a decision** based upon the result. In some cases it may be possible to use an intermediate value in the calculation to make the decision early. A related concept (lazy or deferred execution) is to **delay performing the calculations** until it is determined that they are actually needed. It may be that calculated results are never actually used.

Some **algorithms** explicitly leverage these concepts to improve performance. The source code implementation may also offer these opportunities. Will they be used? **Optimization passes in the compiler** try to apply these optimizations but may fail due to limited visibility within the program or caution due to ensuring proper program behavior according to the semantics of C. In these cases it is necessary to modify the source code to either **help the compiler** perform the optimizations, or to **directly implement the optimization** in the source code.

6.3.1.1 Optimization 1

```
1. float Calc_Distance( PT_T * p1,  const PT_T * p2) {
2.    //calculates distance in kilometers between locations
        (represented in radians)
3.    return acos(p1->SinLat * p2->SinLat +
4.       p1->CosLat * p2->CosLat *cos(p2->Lon - p1->Lon)) * 6371;
5. }
```

Consider the example program from Chapter 5. The function Calc_Distance is repeated in the listing above. In line 4 the code multiplies an intermediate result (produced by the arc cosine function) by 6371 to compute the distance in kilometers between two points following a path staying on the surface of the Earth.[1] That intermediate result is an angle measured in radians. It is converted to kilometers by multiplying by the Earth's circumference and dividing by 2π (approximately 6371 km/radian).

Because the angle is proportional to the distance, the two points will also have the smallest angle. This means we can just search for the point which produces the smallest angle. After we have found that minimum angle we multiply it to get kilometers. So we can call the following simplified function to find the closest point, eliminating $N_{\text{Points-1}}$ floating point multiplies.

```
1. float Calc_Distance_in_Unknown_Units( PT_T * p1,  const PT_T * p2) {
2.    //calculates distance in between locations
        (represented in radians)
3.    return acos(p1->SinLat * p2->SinLat +
4.       p1->CosLat * p2->CosLat *cos(p2->Lon - p1->Lon));
5. }
```

[1] This distance is in fact the length of a circular arc between the two points on a sphere. It is the shortest such path on the surface of a sphere. A path going through the sphere would be shorter. We approximate the Earth as a sphere here.

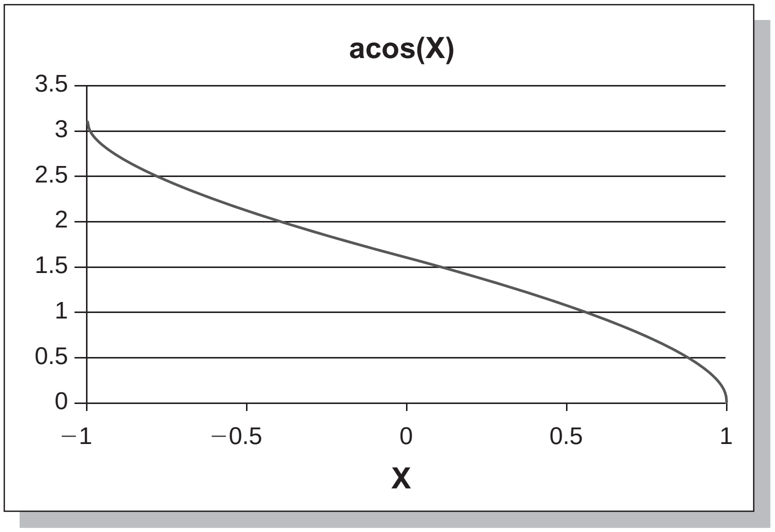


Figure 6.2 Arc Cosine function always decreases as *X* increases.

6.3.1.2 Optimization 2

We can take this a step further. The arc cosine function is a decreasing function, as shown in Figure 6.2, so we don't really need to calculate the arc cosine to find the closest point. Instead, we just need to find the point with the *largest* input (*X* value in Figure 6.2) to the arc cosine, as that will result in the *smallest* output. That will give the smallest distance, as it is just multiplied by a scaling constant to convert to kilometers.

The optimized version of Calc_Distance is renamed and shown below, with changes in lines 3 and 4.

```
1. float Calc_Distance_Partially( PT_T * p1,  const PT_T * p2) {
2.    //calculates cosine of distance between locations
3.    return p1->SinLat * p2->SinLat +
4.       p1->CosLat * p2->CosLat *cos(p2->Lon - p1->Lon);
5. }
```

We need to change the function Find_Nearest_Point (listed below) slightly. First, we need to change various aspects of the code because the arc cosine *increases* as distance *decreases*—we want to find the *largest* intermediate result.

- In line 9 we set the closest_d value to zero.
- In line 23 we look for the *largest* value of d.

Finally, we need to compute the actual distance for the point.

- In line 31 we complete the calculation of the distance for the closest point.

```
 1. void Find_Nearest_Point(float cur_pos_lat, float cur_pos_lon,
 2.    float * distance, float * bearing, char  * * name) {
 3.    //cur_pos_lat and cur_pos_lon are in degrees
 4.    //distance is in kilometers
 5.    //bearing is in degrees
 6.
 7.    int i=0, closest_i;
 8.    PT_T ref;
 9.    float d, b, closest_d=0;
10.
11.    *distance = *bearing = NULL;
12.    *name = NULL;
13.
14.    ref.Lat = cur_pos_lat*PI_DIV_180;
15.    ref.SinLat = MYSIN(ref.Lat);
16.    ref.CosLat = MYCOS(ref.Lat);
17.    ref.Lon = cur_pos_lon*PI_DIV_180;
18.    strcpy(ref.Name, "Reference");
19.
20.    while (strcmp(points[i].Name, "END")) {
21.       d = Calc_Distance_Partially(&ref, &(points[i]) );
22.       //if we found a closer point, remember it and display it
23.       if (d>closest_d) {
24.          closest_d = d;
25.          closest_i = i;
26.       }
27.       i++;
28.    }
29.    b = Calc_Bearing(&ref, &(points[closest_i]) );
30.    //return information to calling function about closest point
31.    *distance = acos(closest_d)*6371;
32.    *bearing = b;
33.    *name = (char * ) (points[closest_i].Name);
34. }
```

6.3.2 Faster Searches

An embedded system may need to search through a large amount of data quickly. Selecting an appropriate **data organization** and an appropriate **algorithm** can simplify the problem significantly. In this section we examine the relationship between data organization and algorithms.

6.3.2.1 *Data Structure Review*

The data structure defines how data elements are connected and organized. Some data structures may store their data in a sorted order. The structure may be fixed at run-time (static), or it may change (dynamic).

Three types of data structure are common in embedded systems software.

- Lists offer sequential access. Each element is contained within a node.[2] Each node is connected with at most two other nodes—a predecessor and a successor. Traversing the list (e.g., to find a specific element) involves following these connections sequentially. Examples of lists include linked lists, queues, circular queues, and double-ended queues.
- Trees offer sequential access as well but offer additional connections between nodes, enabling a hierarchical organization which significantly reduces the average amount of traversal needed. Each node is connected with its parent node[3] and its child nodes.[4] The elements of a tree are stored in a hierarchical structure which reduces access operations. This structure may be explicitly represented by pointers, or implicitly represented by the actual location of the element (e.g., within an array).
- Arrays offer random access—each element can be accessed equally quickly, assuming the program knows which element to access. Arrays are often used to implement static lists, confusing the situation.

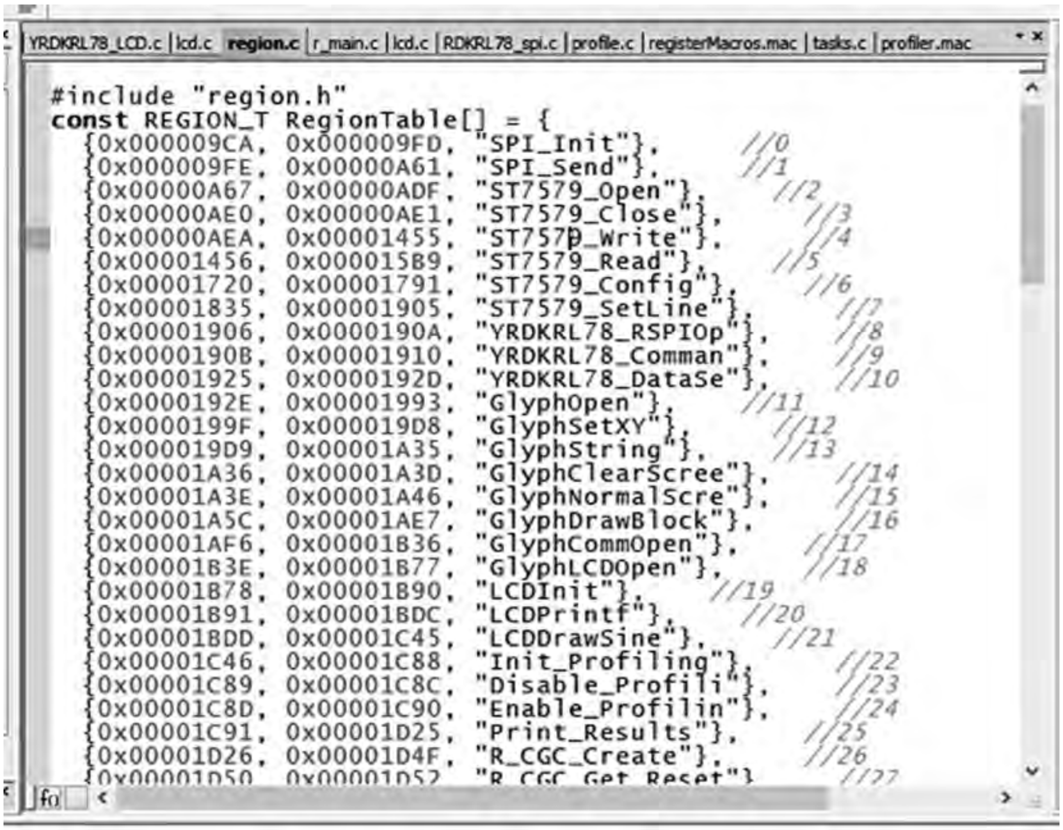
6.3.2.2 *Profiler Address Search*

Consider the execution time profiler in Chapter 4 which samples the program counter periodically to determine which function is currently executing. Figure 6.3 shows the main data structure for this operation—an array which holds the starting and ending addresses for each code region. A search function takes the sampled program counter value which we

[2] Lists and trees typically refer to data elements as *nodes* due to graph theory terminology.

[3] A node with no parent node is called a *root* node.

[4] A node with no child nodes is called a *leaf* node.



```
#include "region.h"
const REGION_T RegionTable[] = {
   {0x000009CA, 0x000009FD, "SPI_Init"},          //0
   {0x000009FE, 0x00000A61, "SPI_Send"},          //1
   {0x00000A67, 0x00000ADF, "ST7579_Open"},        //2
   {0x00000AE0, 0x00000AE1, "ST7579_Close"},        //3
   {0x00000AEA, 0x00001455, "ST7579_Write"},        //4
   {0x00001456, 0x000015B9, "ST7579_Read"},        //5
   {0x00001720, 0x00001791, "ST7579_Config"},        //6
   {0x00001835, 0x00001905, "ST7579_SetLine"},        //7
   {0x00001906, 0x0000190A, "YRDKRL78_RSPIOp"},        //8
   {0x0000190B, 0x00001910, "YRDKRL78_Comman"},        //9
   {0x00001925, 0x0000192D, "YRDKRL78_DataSe"},        //10
   {0x0000192E, 0x00001993, "GlyphOpen"},          //11
   {0x0000199F, 0x000019D8, "GlyphSetXY"},          //12
   {0x000019D9, 0x00001A35, "GlyphString"},          //13
   {0x00001A36, 0x00001A3D, "GlyphClearScree"},        //14
   {0x00001A3E, 0x00001A46, "GlyphNormalScre"},        //15
   {0x00001A5C, 0x00001AE7, "GlyphDrawBlock"},        //16
   {0x00001AF6, 0x00001B36, "GlyphCommOpen"},        //17
   {0x00001B3E, 0x00001B77, "GlyphLCDOpen"},        //18
   {0x00001B78, 0x00001B90, "LCDInit"},          //19
   {0x00001B91, 0x00001BDC, "LCDPrintf"},          //20
   {0x00001BDD, 0x00001C45, "LCDDrawSine"},          //21
   {0x00001C46, 0x00001C88, "Init_Profiling"},        //22
   {0x00001C89, 0x00001C8C, "Disable_Profili"},        //23
   {0x00001C8D, 0x00001C90, "Enable_Profilin"},        //24
   {0x00001C91, 0x00001D25, "Print_Results"},        //25
   {0x00001D26, 0x00001D4F, "R_CGC_Create"},        //26
   {0x00001D50, 0x00001D52, "R_CGC_Get_Reset"}        //27
```

Figure 6.3 Array RegionTable holds information which the profiler searches on each sampling interrupt.

call the search address. This function examines the table to identify which array element holds addresses which **bound** the search address (start address ≤ search value ≤ end address).

In the average case, we would need to search half of the elements ($n/2$) to find the matching region. So the average time complexity of this approach is linear ($O(n)$) with respect to the number of entries n. How can we improve on this?

6.3.2.3 *Sort Data by Frequency of Use*

It may be possible to arrange data so that less run-time work is necessary. Sorting the data elements by expected frequency of use will ensure that the common cases are handled quickly. More frequently used elements are examined before the less common ones, improving average performance.

After the first profiling run we know which regions dominate the execution time of the target program. We could regenerate the region table (shown in Figure 6.3) and **sort it** by placing the most frequently accessed regions at the top where they will be accessed first. This example of "profile-driven profiler optimization" could reduce the execution time of the ISR and hence the profiling overhead.

6.3.2.4 Binary Search

Another approach is to arrange the data elements within this array in order to enable a better search algorithm. For example, a **binary search** has logarithmic time complexity ($O(\log_2 n)$). To enable a binary search the elements need to be sorted so that the addresses are in order (either increasing or decreasing). As the number of data elements to search within grows, the time complexity of the search algorithm becomes more critical. Hence, systems with large data sets can benefit significantly from replacing sequential searches with more efficient ones.

A related search method replaces the array with a specialized abstract data type called a **binary search tree (BST).** This approach requires additional tools to generate the BST and additional code to support the BST.

Examining the profiler table in Figure 6.3 shows the addresses are sorted in increasing order of the first element (start address). This is a side effect of the order in which the linker generates the map file for this particular code and may not always be true. So we need to add a step to the table generation process to sort the table data before generating the source file holding the table.

The search function can then be updated to perform a binary search; for example, with the C standard library function bsearch() which searches a sorted array.

The larger the table, the greater the performance advantage the binary search will provide. For example, a linear search on a table of 160 entries would on average need to compare about eighty entries. A binary search would on average only need to compare a little over seven entries ($\log_2(160) \approx 7.3$), eliminating roughly 90 percent of the effort.

6.4 FASTER MATH REPRESENTATIONS

Unless a microcontroller has a hardware floating point unit, floating point math is emulated in software. It is slow and uses large amounts of memory for the library routines, your code, and the variables. In this section we examine several alternatives to floating point math.

6.4.1 Native Device Integer Math

In some cases it is possible to avoid floating point math by using the device's native integer representations rather than converting to and from formats which require floating point precision. For example, consider a pressure-based water depth alarm. This might be used by a scuba diver to warn of dangerous depths. Or it might be used in the tank of a water heater to determine if the tank is not full and therefore should not turn on the heating elements.

The following program segment measures how far underwater an analog pressure sensor is. It sounds an alarm if it is greater than one thousand feet deep. Line 4 reads the ADC channel connected the pressure sensor. Line 5 converts this reading to a voltage based on the ADC reference voltage and resolution. Line 6 converts this to a pressure in kiloPascals, based on the pressure sensor's specified transfer function. Line 7 converts this pressure to a depth based on the atmospheric pressure, and the fact that pressure increases by 101.3 kPa with every additional thirty-three feet of depth in water. We use floating point math to ensure adequate precision.

```
 1. uint16_t ADC_Code;
 2. float V_sensor, Pressure_kPa, Depth_ft;
 3.
 4. ADC_Code = ad0;
 5. V_sensor = ADC_code*V_ref/1023;
 6. Pressure_kPa = 250 * (V_sensor/V_supply+0.04);
 7. Depth_ft = 33 * (Pressure_kPa - Atmos_Press_kPa)/101.3;
 8. if (Depth_ft > 1000) {
 9.    //sound alarm
10. }
```

We might be able to avoid most or all of the floating point operations if we reverse our use of the ADC-to-depth transfer function. Here we convert from ADC code to voltage, pressure, and depth, and then compare that with the target depth of one thousand feet. We could instead go the other way: determine at compile-time which ADC code represents one thousand feet best, and simply compare ADC_code to that constant value. This eliminates all floating point operations as shown in the following code.

```
uint16_t ADC_Code;

ADC_Code = ad0;
if (ADC_Code> ALARM_ADC_VALUE) {
   //sound alarm
}
```

6.4.2 Fixed Point Math

Fixed point math represents non-integer values with an integer (called a mantissa) which is implicitly scaled by a fixed exponent. Operations on fixed point values consist of integer operations with minor adjustments to handle special cases. This makes the code for fixed point math much smaller than that of floating point, resulting in significantly faster execution.

Floating point math was developed in order to represent a very large range of values dynamically. The mantissa and exponent may need to be adjusted before an operation in order to align the operands correctly. Similarly, the mantissa and exponent may need to be adjusted during or after the operation to maximize precision yet prevent overflow. This processing and support for handling special cases makes software implementations of floating point operations large and slow.

6.4.2.1 Representation

Bit Position	7	6	5	4	3	2	1	0	**. Radix Point**
Bit Weight	$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	
	0	0	0	0	1	0	0	1	

Figure 6.4 Example 1: Bit weighting for byte as integer.

Figure 6.4 shows an example of a byte (0000 1001). If we interpret this byte as representing an integer, then the **least significant bit (LSB)** position has a weight of $2^0 = 1$. The value of the byte with this representation is $8 + 1 = 9$. The **radix point** is located immediately to the right of the bit with a weight of one. For an integer representation that is bit 0 (the LSB).

Bit Position	7	6	5	4	3	**. Radix point**	2	1	0
Bit Weight	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$		$2^{-1} = 1/2$	$2^{-2} = 1/4$	$2^{-3} = 1/8$
	0	0	0	0	1		0	0	1

Figure 6.5 Example 2: Bit weighting for byte as a fixed point value with radix point between bit positions three and two.

It is possible to use scaling to change the range of values which those integers represent by assuming the radix point is in a location other than after the LSB. For example, we could move the radix point left by three bits, as shown in Figure 6.5. This would have the effect of scaling the value of each bit by $2^{-3} = 1/8$. Now the LSB of the byte has a weight of 1/8 rather than 1. With this approach we can represent fractional values with a resolution of 1/8, but cannot represent values greater than 31 1/8.

Bit Position	7	6	5	4	3	2	1	0	−1	−2	.
Bit Weight	$2^9 = 512$	$2^8 = 256$	$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	**Radix point**
	0	0	0	0	1	0	0	1	▨	▨	

Figure 6.6 Example 3: Bit weighting for byte as a fixed point value with radix point two bits right of bit 0.

Alternatively, we could move the radix point right by two bits, as shown in Figure 6.6. This would have the effect of scaling the value of each bit by $2^2 = 4$. Now the LSB of the byte has a weight of four rather than one. With this approach, we cannot represent fractions. However, we can represent values as large as 1020. Note that we cannot represent all values from zero to 1020 uniquely since we no longer can specify the bits with weights 1 and 2. We summarize these three examples in Table 6.1.

TABLE 6.1 Summary of Byte Interpreted with Different Fixed Point Representations

EXAMPLE	MANTISSA	EXPONENT	VALUE REPRESENTED	RESOLUTION
1	9	0	$9 * 2^0 = 9$	1
2	9	−3	$9 * 2^{-3} = 1 + 1/8$	1/8
3	9	2	$9 * 2^2 = 36$	4

It is important to recognize that the exponent is fixed so it is **not stored explicitly** in the variable or anywhere in the code. Instead, the source code is written with an **implicit assumption** of the exponent's value. For **maintainability,** the code should also name the data types, variables, arguments, and functions to indicate the value of the exponent. Sufficient comments should also be provided as well.

6.4.2.2 Unsigned and Signed Values

So far we have only examined unsigned fixed point values. Handling two's complement fixed point values is more complicated.

One common approach which avoids this complexity is to store the absolute value of the mantissa and also store a bit (typically the most-significant bit) to indicate the sign of the mantissa. When operating on two values, one of the first steps is to determine the sign of each operand and determine how to perform the operation, given that the absolute value of the operand's mantissa is stored, rather than the actual mantissa. For example, when multiplying two signed operands, the result will be positive if both operands have the same sign. Otherwise the result will be negative. The absolute values of the two operands are multiplied and then the product's sign is set to be the exclusive-or of the operand sign bits.

6.4.2.3 Notations

In this text we will use the Q$i.f$ notation to clearly show the integer (i) and fraction (f) field sizes. Other notations exist; a format with one sign bit, three integer bits, and twelve fraction bits could be called Q3.12, Q12, or fx3.16.

6.4.2.4 Support Operations

There are several common basic operations used to support fixed point math.

- **Scaling** consists of shifting a value in order to change from one implicit exponent value to another. To increase the number of fraction bits by n, we shift the mantissa left by n. To decrease the number of fraction bits, we shift the mantissa right by n. For example, to convert a value from Q3.12 to Q5.10 we shift the mantissa right by two bit positions ($12 - 10 = 2$).
- Two fixed point values are called **normalized** if they have the same representation: each has i integer bits and f fraction bits. We can **normalize** two values by scaling one to match the other, or by scaling both to a new format.
- **Promotion** converts a value to a longer representation, increasing the total number of bits ($f + i$). This can prevent overflow, described shortly.
- **Rounding** is used to improve the accuracy of the result when truncating one or more least significant bits from a value. The newly-truncated mantissa is typically incremented if the first truncated bit is a one. Rounding is performed after scaling to a value with fewer fraction bits.
- **Overflow** occurs when the result of an operation does not fit into the representation. For example, adding two values in assembly language can result in a carry out of the MSB. Addition can also result in an incorrect change of the sign bit, assuming it is stored in the most significant bit. Multiplying two n bit values in C produces a 2n bit result, but the C language only uses the lower n bits of the result, discarding the upper half. Overflow can be handled in various ways:
  - The operands can be promoted to a type with more total bits before the operation is performed. This slows down the code but maintains accuracy. This is typically required when performing fixed point multiplies in C.
  - The operands can be scaled to a format with fewer fraction bits by shifting them to the right by one or more bits. This introduces error but results in fast code.
  - The overflow can be detected after the operation is performed. The code can then attempt to compensate for the problem or signal an error.

- One way to handle overflow is **saturation.** An operation which provides saturation handles an overflowing result by replacing it with the **closest available value** in that representation. For example, adding two large positive values could result in an overflow into the sign bit, changing it to negative incorrectly. Saturation handles this by returning a result which is the maximum positive value for that representation.

6.4.2.5 Mathematical Operations

We next examine the steps needed to perform the mathematical operations based on a representation with a sign bit and an unsigned mantissa. For clarity we consider formats where both operands have n bits. However, the radix point may be in different locations in the operands (i.e., they can have different numbers of fraction bits). The first operand Op_1 is in format $i_1 . f_1$ and the second operand Op_2 is in $i_2 . f_2$.

Addition and **subtraction** are some of the most basic operations. The number of fraction bits remains the same. The result can be $n + 1$ bits long.

- Normalize the operands.
- Add or subtract the mantissas based on operation type and the exclusive-or of the signs.
- Handle overflow if it occurred.
- Set the sign of the result.

Multiplication does not require the operands to have the same scaling factor. The number of fraction bits increases. The result can be $2n$ bits long.

- If necessary, promote operands to a longer representation to prevent overflow.
- Multiply the mantissas.
- Add the exponents.
- Handle overflow if it occurred.
- The result has $f_1 + f_2$ fraction bits. Scale the resulting mantissa to fit the desired target format.
- Set the sign of the result.

Division does not require the operands to have the same scaling factor. The number of fraction bits decreases. Division is more challenging to implement than the operations above. There are three approaches based on the existence and type of division support available.

First, consider the **C integer division operation** (/). When dividing two **integers,** the result (quotient) is an integer with the fractional bits truncated. Dividing two **fixed point operands** ($\mathrm{Op}_1/\mathrm{Op}_2$) of formats $i_1 . f_1$ and $i_2 . f_2$ creates a result with $f_1 - f_2$ fraction bits. This means that if we **divide two normalized values,** then the result will be an **integer** with no fraction bits.

We can **set the number of fraction bits** in the result by scaling an operand. To make the result have f_r fraction bits we scale the numerator Op_1 by shifting it left by f_r bits, or scale the denominator Op_2 by shifting it right by bits f_r. The first approach could lead to overflow, so we may want to promote the operands to longer formats. The second approach reduces precision of the dividend and therefore increases error in the quotient. The first approach (shifting the numerator Op_1 to the left) is more useful.

Second, consider an **assembly language divide instruction.** This instruction typically produces two results: an integer **quotient** and an integer **remainder.** We can handle the quotient with the same approach as in the C language integer division—shifting the numerator left so the quotient will be in a fixed point format with the desired number of fractional bits. If the remainder is ½ or larger, we round up the LSB of the quotient.

Finally, consider how to perform division **without support,** such as an existing divide operation or function. We can implement fixed point division **indirectly** using **multiplication by a reciprocal.** To **divide Op_1 by Op_2** we instead can **multiply Op_1 by the reciprocal of Op_2.** If Op_2 is a constant then this transformation can be performed at compile time. Otherwise the inverse must be computed at run time, for example using Newton's method of approximation. This involves first estimating the inverse of Op_2 and then improving the accuracy of that estimate with successive iterations of refinement. Each refinement requires **two multiplies,** an **addition** and a **subtraction.** This method is called Newton-Raphson division. We use this division for fixed point values as in the two previous methods.

6.4.2.6 C, Assembly, or Both?

The code for performing these fixed point math operations could be written in a language which is high-level (e.g., C, C++) or low-level (e.g., assembly language). Which is better? Programming in a high-level language might seem to be better due to the ease of code development (when compared with assembly code). However, this approach faces two major problems.

First, because we use integers to represent our fixed point values, the compiler will generate code which treats those values as integers. For most operations with most data values this is not a problem. However, the ANSI C standard defines how to handle, interpret, and modify integers, and its rules **don't always fit well for fixed point math.** For example, multiplying an int (16 bits) with an int (16 bits) produces an int (also 16 bits), even though the hardware produces a 32 bit result. Because of these differences we need to modify our C code in two ways.

- **First** we need to deal with the **C rules** which cause problems. We can try to **prevent** the compiler from handling those cases. We could instead **add code to undo** those undesired effects. Both of these approaches require an in-depth understanding of relevant parts of the C language.
- **Second** we need to **add code** to handle the **fixed point math special cases.**

Both of these coding efforts depend on the processor's word size and available instructions. And the compiler optimization settings are also likely to affect the code. It is quite instructive to compare C code for fixed-point math implementations across different embedded processors (or even compilers for the same processor).

Yet another issue is that the compiler may not be able to generate code which takes full advantage of the processor's instruction set and resources. The C programming language insulates us from many processor implementation details. However, we need to control and monitor those details in order to implement fixed point math operations efficiently. How can we convince the compiler to use the RL78G13's Multiply/Accumulate/Divide peripheral unit? Or to use the RL78G14′s Multiply/Accumulate Halfword instruction?[5]

Considering these factors, the assembly language implementation is usually preferable. The implementation could be an assembly language function, or else inline assembly code in a C function.

6.4.3 RL78 Support for Fixed Point Math

The complexity and speed of fixed point calculations depends on the CPU's support for integer operations. Supporting operations on representations longer than the native integer operations increases code complexity and reduces performance. This is especially acute when performing multiplications and divisions, so native (and fast) hardware support for multiplication and division is quite helpful. Normalization and conversion between formats relies on shifting and rotation, so those are also important.

Let's examine which instructions are available. The RL78 processor family has different CPU cores which implement different versions of the instruction set. Some cores offer multiply and divide instructions for longer data formats. Some RL78 devices include a separate peripheral which can perform multiplication and division.

6.4.3.1 Basic Instructions

All RL78 cores provide a multiply instruction (MULU, shown in Figure 6.7) which multiplies two unsigned bytes in A and X and places the 16-bit result in the AX register. The cores offer fast shifts and rotates of 8- and 16-bit data. The cores use a barrel shifter so that a shift or rotate instruction takes only one clock cycle, regardless of the shift amount or direction.

[5] This is similar to trying to type on a keyboard while wearing mittens.

INSTRUCTION GROUP	MNEMONIC	OPERANDS	BYTES	CLOCK		OPERATION	FLAG		
				NOTE 1	NOTE 2		Z	AC	CY
Multiply, Divide, Multiply & accumulate	MULU	×	1	1	—	AX ← A * X			
	MULHU		3	2	—	BCAX ← AX * BC (unsigned)			
	MULH		3	2	—	BCAX ← AX * BC (signed)			
	DIVHU		3	9	—	AX (quotient), DE (remainder) ← AX ÷ DE (unsigned)			
	DIVWU		3	17	—	BCAX (quotient), HLDE (remainder) ← BCAX ÷ HLDE (unsigned)			
	MACHU		3	3	—	MACR ← MACR + AX * BC (unsigned)		x	x
	MACH		3	3	—	MACR ← MACR + AX * BC (signed)		x	x

Figure 6.7 Multiply, divide, and multiply/accumulate instructions implemented in RL78G14 processors.

6.4.3.2 *Extended Multiply and Divide Instructions*

The RL78G14 processor provides additional instructions. These are shown in Figure 6.7 and include the following operations:

- Signed and unsigned multiplies MULH and MULHU: 16 bit * 16 bit = 32 bit
- Unsigned divide DIVHU: 16 bits/16 bits = 16 bit quotient, 16 bit remainder
- Unsigned divide DIVWU: 32 bits/32 bits = 32 bit quotient, 32 bit remainder
- Signed and unsigned multiply/accumulates MACH, MACHU: 16 bit * 16 bit + 32 bit = 32 bit

6.4.3.3 *Multiply/Divide/Accumulate Unit*

Some RL78 family processors (e.g., G13) include a multiplier/accumulator/divider unit (called MD for brevity) to accelerate certain mathematical operations:

- Signed and unsigned multiplies: 16 bit * 16 bit = 32 bit
- Signed and unsigned multiply/accumulates: 16 bit * 16 bit + 32 bit = 32 bit
- Unsigned divide: 32 bits/32 bits = 32 bit integer quotient, 32 bit integer remainder

Rather than use the general purpose registers such as AX, BC, DE, and HL to hold operands, commands, and results, the MD unit uses its own special function registers. These consist of six 16 bit data registers (MDAH, MDAL, MDBH, MDBL, MDCH, and MDCL) and one 8 bit control register (MDUC). To use the MD unit the program configures MDUC to specify the desired operation, according to Table 6.2.

TABLE 6.2 Multiplier/Accumulator/Divider Operation Selection

DIVMODE	MACMODE	MDSM	OPERATION SELECTED
0	0	0	Multiply, unsigned
0	0	1	Multiply, signed
0	1	0	Multiply/accumulate, unsigned
0	1	1	Multiply/accumulate, signed
1	0	0	Divide, unsigned, generate interrupt when complete
1	1	0	Divide, unsigned, no interrupt generated

TABLE 6.3 Multiplier/Accumulator/Divider Flags

FLAG	DESCRIPTION
MACOF	Multiply/accumulate overflow
MACSF	Multiply/accumulate sign flag
DIVST	Division operation status; 1 = division in progress

TABLE 6.4 MD Operand Locations

OPERATION	MDAH	MDAL	MDBH	MDBL	MDCH	MDCL
Multiply	Multiplier	Multiplicand				
Multiply/ Accumulate	Multiplier	Multiplicand				
Divide	Dividend (high word)	Dividend (low word)	Divisor (high word)	Divisor (low word)		

The program then loads the MD data registers with the input data as shown in Table 6.4. In multiply mode or multiply/accumulate mode, writing to MDAH and MDAL starts the multiplication. In division mode, the DIVST bit must also be set to 1 to start the division. After the operation completes, the results are available in the MD registers as shown in Table 6.5. The status flags shown in Table 6.3 can be examined if needed.

A multiply takes one clock cycle after the last operand is written, while a multiply accumulate takes two clock cycles. A division operation takes 16 clock cycles after the DIVST flag is set. It is possible to configure MDUC so that the MD unit generates an INTMD interrupt when a division completes.

TABLE 6.5 MD Result Locations

OPERATION	MDAH	MDAL	MDBH	MDBL	MDCH	MDCL
Multiply			Product (high word)	Product (low word)		
Multiply/ Accumulate			Product (high word)	Product (low word)	Accumulator (high word)	Accumulator (low word)
Divide	Quotient (high word)	Quotient (low word)			Remainder (high word)	Remainder (low word)

6.4.4 Reduced Precision Floating Point Math

Compilers typically provide floating point data types and operations conforming to the IEEE Standard for Floating Point Arithmetic (IEEE-754). In the previous chapter we discussed the single-precision (32-bit) and double-precision (64-bit) formats and their computational costs. The computational requirements for a software implementation of even the 32-bit format are sizable.

The author once had the opportunity to perform a software design review with a phenomenal team of embedded software developers. During the review the team mentioned that the 32-bit floating point math libraries were too slow for their power-constrained (and therefore very under-clocked) 8-bit microcontroller. So they had modified the math library to support 24-bit floating point math operations. The reduction in the computational requirements enabled the code to meet its timing requirements.

This may seem far-fetched and excessive, but since 2008 the IEEE floating point standard has supported a half-precision (16-bit) format. This reduced precision format was developed for graphics processing units (GPUs) to cut memory requirements and bus traffic, while still supporting the high dynamic range of values needed for graphics. This format uses ten bits for the mantissa, five bits for the exponent, and one bit for the sign. The minimum and maximum positive values which can be represented are $5.96 * 10^{-8}$ and 65504, with similar negative value limits.

The IEEE-754 2008 standard defines the half-precision format as a storage format—the only operations available are conversion to and from other formats. However, some high-throughput microprocessors include hardware support for performing half-precision operations, in addition to single-precision (and possibly double-precision). Similarly, some compilers for these processors support the 16-bit floating point data type.

It will be very interesting to see if any half-precision floating point math libraries for embedded processors exist or are developed. They could simplify the development of embedded systems by providing a solution between full-precision floating point, integer math, and fixed point math.

6.5 FASTER MATH USING APPROXIMATIONS

In this section we examine a valuable approach to approximating mathematical functions which are difficult to compute. There is a wealth of information available online and in print about this and other methods of accelerating mathematical processing for real-time and embedded systems. One example is Crenshaw's extensive book (Crenshaw, 2000).

The trigonometric and other functions in the C math library are very accurate—perhaps more accurate than necessary for your application. These functions are typically implemented with numerical approximations. The library designers ensured high accuracy by using a large number of terms or iterations in the approximation. Perhaps your application doesn't need as much accuracy, in which case you may benefit from implementing your own reduced-precision approximations.

Consider the cosine function. When compiled with proper toolchain settings, the cos library function in the IAR RL78 Embedded Workbench takes about 2420 clock cycles[6] to execute on an RL78G14 family processor. In comparison, a floating point multiply operation takes about 360 cycles. Can we compute a useful approximation of the cosine if we only have time to perform seven floating point multiplies? We will find out at the end of this section.

6.5.1 Polynomial Approximations

How can we approximate the cosine function? If we wish to approximate cosines of small input values (e.g., 0.01), why not just use the constant value of one? After all, $\cos(0) = 1$. This may in fact be adequate. However, the error for this approximation grows as the input value moves farther away from 0 radians.

[6] Note that these execution cycle counts may vary based on different input data due to the various operations necessary to perform floating point math.

We could improve the accuracy of our estimate by including one or more factors based on how far the input value x is from our reference input value of 0 radians. For example, we could include a factor proportional to x, or x^2. This is an example of a **polynomial** approximation. Any arbitrary function $f(x)$ can be represented as a polynomial of x:

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + \cdots$$

This polynomial provides perfect accuracy if we can use an infinite number of terms. However, it cannot be computed practically. To make computation feasible, we will truncate the polynomial to a finite number of terms m by eliminating all terms after term m. This introduces some error, which we will examine shortly. Above, we truncated the polynomial to the first term when we selected the constant value of one for our approximation.

The *degree* of a polynomial is the highest power to which x is raised. Truncating the polynomial reduces the degree of the polynomial from infinity to a finite value. Truncating the polynomial above after the a_4x^4 term will make it have a degree of four.

6.5.2 Optimizing the Evaluation of Polynomials

Polynomials are attractive because they can be computed quickly with some simple optimization.

A polynomial of degree n requires up to n additions and $(n^2 + n)/2$ multiplications. The equation above is in the canonical form and shows all of these operations. Since $x^{n+1} = x * x^n$ we can reuse the result of the previous term if we evaluate the x^n terms in order of increasing degree (left to right in the equation above).

$$f(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + xa_4)))$$

This optimization (called Horner's Rule) reduces the number of multiplications needed from $(n^2 + n)/2$ to n, significantly reducing the amount of computation required.

6.5.3 Determining Coefficients

There are various methods which the function $f(x)$ can use to compute the terms a_n. For example, the Taylor series expansion of the function f can be computed based on successive derivatives of f evaluated at a given reference argument r. A Taylor series evaluated at $r = 0$ is a special case which is also called a Maclaurin series. The *nth* term in the equation uses the *nth* derivative of f, written as f^n.

$$\sum_{n=0}^{\infty} \frac{f^n(r)}{n!}(x-r)^n$$

If we evaluate the cosine function at point $r = 0$, the result is:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Note that there are no terms with an odd degree (exponent). This is because those terms are multiplied by an odd derivative of the cosine function. All of the odd derivatives of the cosine function are in fact the sine function, and $\sin(0) = 0$ so those terms disappear. The sine function expansion is similar but the even derivatives are sines, eliminating the even degree terms:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

One interesting consequence of the alternating sign of terms is that we can estimate the maximum error due to truncation. The error will be no greater than the first term (i.e., the term with the lowest degree) removed by truncation.

6.5.4 Accuracy

Let's evaluate the accuracy of our cosine approximation as we add more terms, as shown in Figure 6.8. The solid line shows the actual value of cosine. We begin with the Degree 0 Taylor plot, which consists of only the first term (1) of the equation. This is in fact the constant value described earlier. The Degree 2 Taylor plot includes the second term (a downward pointing parabola) and improves the accuracy. The Degree 4 Taylor plot includes the third term, and the accuracy improves further. Notice that the error grows quickly as the input argument gets farther from 0. This is because we evaluated the terms for this Taylor series at the input value 0 (so it is also a Maclauren series).

Taylor series expansions are typically not used for approximating functions because there are other approaches (e.g., Chebyshev polynomials, Bessel functions, minimax optimization) which provide better accuracy with the same number of terms (Hart, 1978). Figure 6.9 shows an example comparing an optimized polynomial for cosine with the Taylor series. Furthermore, these other methods can distribute error more evenly across the range of input values rather than let it accumulate at the ends of the range. This reduces the worst-case approximation error.

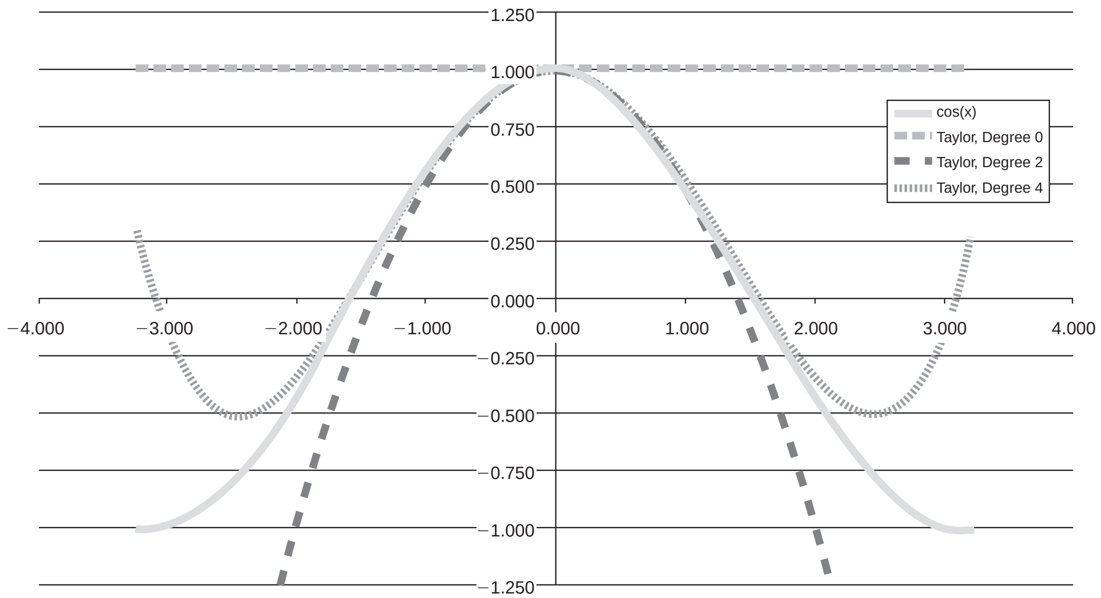


Figure 6.8 Cosine function and Taylor series polynomial approximations.

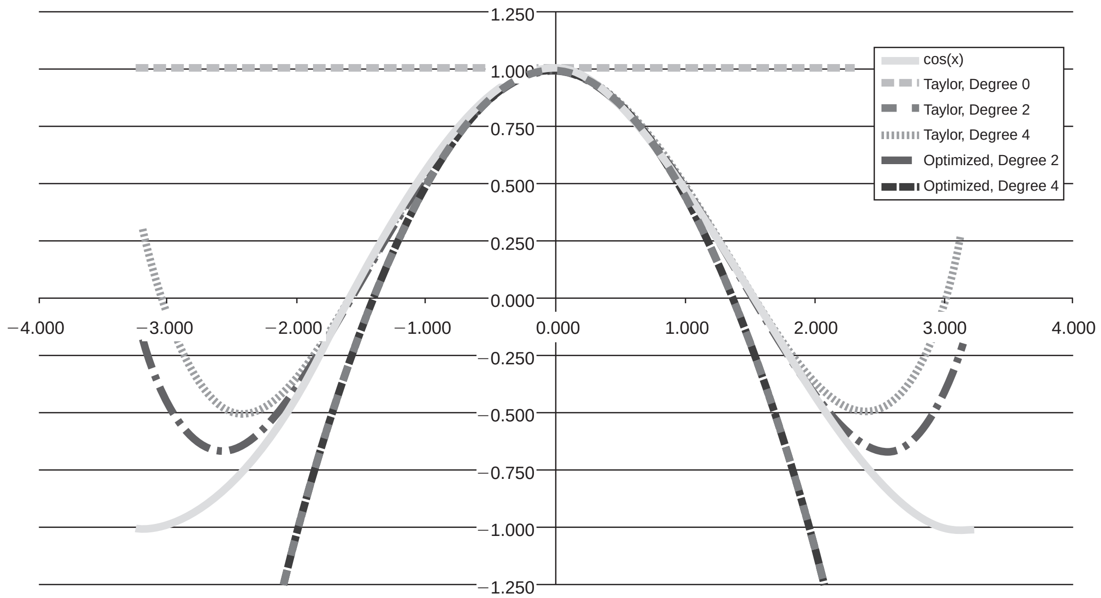


Figure 6.9 Cosine function compared with Taylor series and optimized polynomial approximations.

6.5.5 Approximating Periodic and Symmetric Functions

Functions which are **periodic** or **symmetric** can be approximated over a broader input range with fewer polynomial terms or with greater accuracy by applying certain optimizations.

For example, cosine is **periodic,** repeating every 2π radians. This means $\cos(x) = \cos(x + n2\pi)$. Hence we only need our approximation to be accurate over one period, (e.g., $-\pi$ to π). If the input argument is beyond that range, we can add or subtract $n2\pi$ to bring the argument within the range. Another feature of some functions is **symmetry.** The cosine function is symmetric about 0 radians: $\cos(x) = \cos(-x)$. The cosine function also has the characteristic that $\cos(x) = -\cos(\pi - x)$.

Using these properties we can dramatically improve the accuracy of our cosine approximation for values of x which are far from 0. We perform range reduction to bring the argument within the range of $-\pi/2$ to $\pi/2$ where even a fourth-degree approximation is reasonably accurate.

Further details, coefficients and source code are available elsewhere (Crenshaw, 2000; Ganssle, 1991).

6.5.6 Speed Evaluation

Let's evaluate how long the cosine function and approximations take to execute. For reference, when compiled with default optimization settings, the cosine library function in the IAR RL78 Embedded Workbench takes about 3900 clock cycles to execute on an RL78G14 family processor. After enabling various optimization flags and library configuration flags, the faster version of cos takes about 2421 cycles.

Table 6.6 shows the results of running the code. We see that the degree 6 approximation takes about half as long as the cosine function, which is a significant improvement. Depending on the application, we may even be able to use the degree 4 approximation (see

TABLE 6.6 Execution Cycles Needed for Optimized Floating Point Cosine and Polynomial Approximation Functions

FUNCTION EXECUTED	CLOCK CYCLES USED
Math Library cosine function (iar_cos_small)	2421
Polynomial Approximation , degree 6	1248
Polynomial Approximation , degree 4	883
Polynomial Approximation , degree 2	545

Figure 6.9), in which case the approximation takes only about one third of the cosine function's time.

6.6 RECAP

In this chapter we have seen how to reduce computations by making decisions as early as possible and performing more efficient searches. We have seen how to use the processor's native integer data capabilities to represent and operate on data with fractional values much more quickly than would be possible with floating-point operations implemented in software. Finally, we have seen how to approximate computationally expensive functions with easily-computed polynomials.

6.7 REFERENCES

Crenshaw, J. W. (2000). *Math Toolkit for Real-Time Programming.* Lawrence, KS, USA: CMP Books, CMP Media, Inc.

Ganssle, J. (1991, May). Embedded Trig. *Embedded Systems Programming.* (http://www.ganssle.com/articles/atrig.htm, http://www.ganssle.com/approx.htm).

Hart, J. F. (1978). *Computer Approximations.* Krieger Publishing Co.

Knuth, D. E. (2011). *The Art of Computer Programming* (3rd ed., Vols. 1–4A). Reading, MA, USA: Addison-Wesley.

Chapter 7

Power and Energy Analysis

7.1 LEARNING OBJECTIVES

This chapter deals with how to analyze an embedded system's power and energy use.

The first portion deals with general concepts. We examine the concepts of power and energy and how to model the power use of digital circuits. We discuss power supply approaches and evaluate their efficiency. We then investigate how to measure power and energy.

The second portion examines the RL78G14 processor in detail. We begin with the relationships between voltage, power, and clock frequency. We then examine available features which can reduce power or energy consumption: selecting clock sources, controlling the clock speed, and using low-power standby modes.

7.2 BASIC CONCEPTS

Let's begin by reviewing the concepts of power and energy.

7.2.1 Power and Energy

Power measures the **instantaneous rate** at which **energy is used.** In an electric circuit it is the product of voltage and current. One ampere of current flowing across a one volt drop will dissipate one watt of power.

$$P(t) = V(t)I(t)$$

Energy is the total amount of power used over a specified period time. Energy integrates power (an instantaneous value) over time. Energy (represented with W, as in work) is power integrated over a certain period of time:

$$W(T) = \int_{t=0}^{T} P(t)dt = \int_{t=0}^{T} V(t)I(t)dt$$

One watt being dissipated (and therefore integrated) for a period of one second represents one joule of energy. If we have one Joule of energy, we can use it all in one second if we power a one watt circuit. If we use a ¼ watt circuit, then it will take four seconds to use that one joule.

Do we want to save power, energy, or both?

- In some cases, there is a limited amount of **energy** available. For example, consider a rechargeable NiMH AA cell with a nominal capacity of 2500 mAH, or 2.5 AH. This means the cell can supply 2.5 A for one hour. We will assume the average voltage is 1.2 V.[1] Multiplying 1.2 V by 2.5 AH gives the cell energy capacity as 3 Watt-Hours, or 3 * 60 * 60 J = 10800 J.
- In some cases, the power budget is limited. There may be **limited power** available. For example, a photovoltaic panel may produce at most 100 mW, and less when the sky is cloudy. In other applications there is **limited cooling** available. The power dissipated by the circuit heats it above the ambient temperature, based upon thermal resistance and available cooling. In order to keep the circuit from overheating we need to limit the power dissipated.

7.2.2 Digital Circuit Power Consumption

Let's take a look at how digital circuits use power and energy. The inverter in Figure 7.1 dissipates power in two ways.

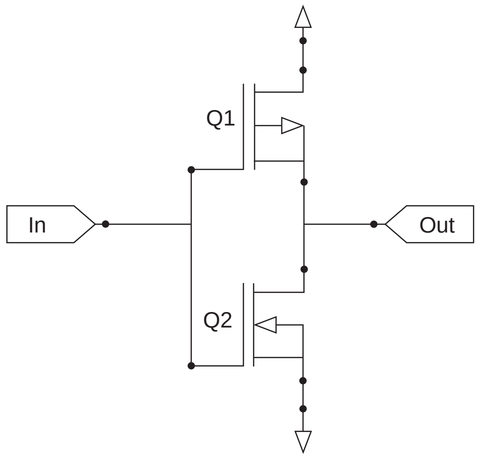


Figure 7.1 Example digital inverter circuit.

[1] A NiMH cell's voltage is not constant. It depends on the state of charge, load current, and temperature. The voltage is roughly 1.35 V when fully charged, and falls to roughly 1.0 V when mostly discharged.

- When the **input signal is not changing,** one transistor is on (saturated or active) and the other is off. For example, a low input level will turn on Q1 and turn off Q2. A high input level will turn off Q1 and turn on Q2. In either case, since the transistors are in series, the total resistance is quite large, and only a small amount of current flows from V_{DD} to ground. This current leads to a **static power component** which is proportional to the square of the supply voltage. The power is still dissipated even though there is no switching, so it is independent of the clock frequency.
- When the **input signal is changing,** then as the input voltage changes from one logic level to another, both transistors will be on simultaneously, leading to shoot-through current flowing for a brief time from V_{DD} to ground. In addition, some components in the circuit have capacitance (e.g., gates, wires) which must be charged or discharged in order to change the voltage level of a signal. This current leads to a **dynamic power component** which is proportional to the square of the supply voltage. It also depends on the frequency of the switching (f_{sw}).

The resulting total power dissipated can be modeled as the sum of the static and the dynamic power components:

$$P = S_P V_{DD}^2 + C_P V_{DD}^2 f_{sw}$$

S_p and C_p are proportionality constants representing conductance and capacitance and can be experimentally derived.

7.2.3 Basic Optimization Methods

The power equation gives us some insight into how to reduce the **power** consumption for a digital circuit.

- Lowering the supply voltage will reduce power quadratically for both terms. For example, cutting V_{DD} to 80 percent of its original value will reduce power to $(80\%)^2 = 64\%$ of its original value.
- Shutting off the supply voltage for unused circuits will eliminate all of their power.
- Disabling the clock ("clock gating") for unused circuits will eliminate their dynamic power.
- Reducing the switching frequency for circuits which are used will reduce their dynamic power proportionately.

Reducing energy is a bit more involved. As we reduce the supply voltage, transistors take longer to switch because when they turn on they are operating closer to the threshold voltage V_{Th}, so they do not turn on as strongly (since a saturated MOSFET's current depends on $(V_{GS} - V_{Th})^2$).

$$f_{max} = \frac{K_P(V_{DD} - V_{Th})^2}{V_{DD}}$$

Looking at this from the point of view of a CPU core lowering the clock frequency means that the processor has to be active longer to complete the same amount of processing work. Optimizing energy is more complex than optimizing power since it requires us to balance multiple factors. Slowing the clock f_{max} lets us lower V_{DD} and therefore both static and dynamic power. Slowing the clock also raises the computation time, so that power is integrated over a longer time, potentially raising total energy used. If there is a low-power standby mode available, in many cases it turns out that the best approach is to run the processor as fast as possible (at the minimum V_{DD} possible for that frequency) when there is work to do, and put the processor into standby mode otherwise.

7.3 MEASURING POWER

We can calculate the power used by a circuit by multiplying the current used by the supply voltage. We can measure the current directly using a multimeter, or indirectly by measuring the voltage drop across a resistor.

7.3.1 MCU Power

The RDK includes provisions measuring the MCU's current. As shown in Figure 7.2, power from the 3V3 power rail (in the center) flows to two power connections on the MCU. MCU Vdd powers the MCU internals and some of the I/O buffers, while MCU EVdd powers the remaining I/O buffers. EVdd must be less than or equal to Vdd. The RDK connects both of those power connections to 3V3 using zero ohm resistors R108 and R109. Figure 7.3 shows location of these various components on the RDK itself.

In order to measure MCU current we need to do the following:

- Remove resistors R108 and R109
- Short out jumper JP9 to connect MCU Vdd and MCU EVdd
- Measure current across JP7

A **multimeter** will give us an **average** current reading. This is adequate for many situations. However, there are times when we would like to see how much current the MCU uses **over time** as the program executes, entering different modes and using different peripherals. One way to determine the current and power over time is to convert the current into a voltage and then display it on an oscilloscope. We can display power if the oscillo-

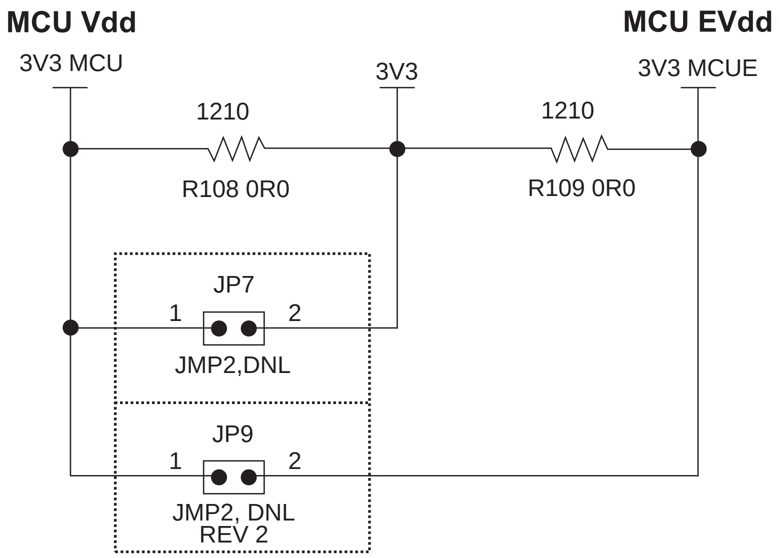


Figure 7.2 RDK circuit for measuring MCU current.

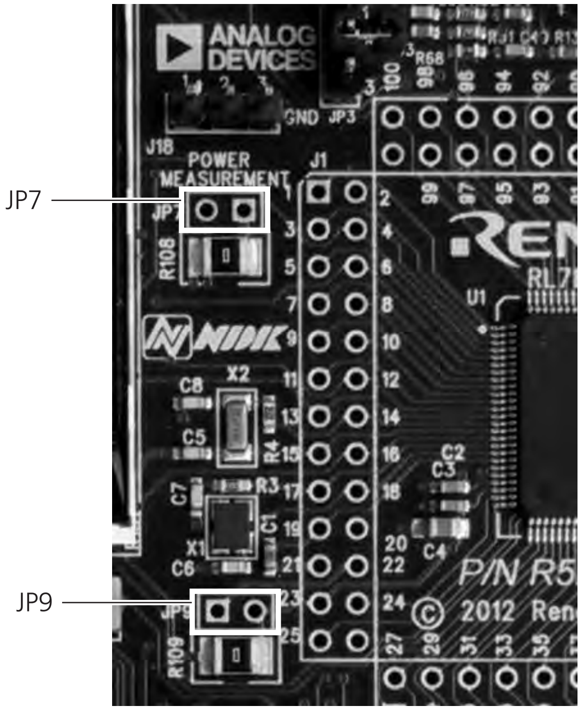


Figure 7.3 Location of power measurement components on RDK.

scope supports multiplying two analog inputs: one input will be the current, and the second will be the supply voltage for the circuit. Analyzing the data is easier if we modify the program to generate pulses on digital output pins as the processor executes the activities whose power we are trying to measure.

One way to convert the current into a voltage is to insert a small resistor (of resistance R) in series with either the power rail or ground connection. A current I through the circuit also flows through the resistor, creating a voltage drop of $V_r = I * R$. We solve for the current: $I = V_r/R$. We now multiply this by the voltage across the circuit V_c to calculate the circuit's power use: $P_c = V_c * I = V_c * V_r/R$. There are also integrated circuits designed specifically for performing this current sensing operation, and they include an amplifier to improve the sensitivity to small currents.

7.3.2 RDK Power

To measure the RDK's total power we need to include the current consumed by all devices on the RDK, not just the MCU. We can measure the current coming in to the 5VIN supply rail at auxiliary power connectors J16 or J17, or at the USB connector J19. One way to measure the current is to modify a USB cable as shown in Figure 7.4:

- Removing the plastic outer jacket.
- Cut either the red wire (5 V) or the black wire (ground).
- Strip a small amount of insulation from the two ends of the cut wire.
- Connect an ammeter or current sense resistor across the ends of the cut wire.

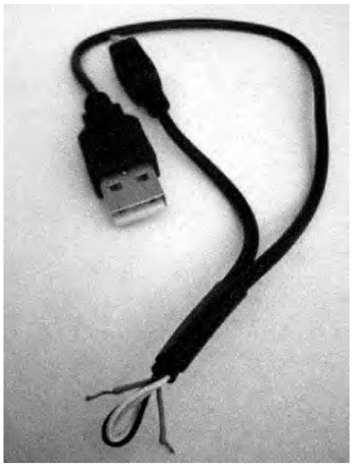


Figure 7.4 USB cable modified to allow current measurement.

If you plan to use a current sense resistor, it is better to cut the black wire (ground). That will eliminate about five volts of offset from the measured voltage V_r, improving the accuracy and removing some ground offset issues.

It is also important to know that the red wire provides a nominal 5 V, but this may vary due to various factors (powered vs. unpowered hub, etc.). So be sure to check that voltage before performing power calculations.

7.4 MEASURING ENERGY

Remember that energy (W) is power integrated over time:

$$W(T) = \int_{t=0}^{T} P(t)dt = \int_{t=0}^{T} V(t)I(t)dt$$

How can we measure (in a practical way) the amount of energy a circuit uses? We can take advantage of the fact that the energy W in a capacitor is related to the capacitance C and the voltage V:

$$W = \frac{CV^2}{2}$$

If we power the circuit with a capacitor then the capacitor voltage V will fall as the circuit uses energy. We can measure capacitor voltage before and after the circuit operation and then calculate the capacitor energy before and after. The difference is the amount of energy used.

In Figure 7.5 we examine how to measure how much energy the MCU uses in order to perform a certain operation. The operation begins at time t_1 and the voltage is V_1. The

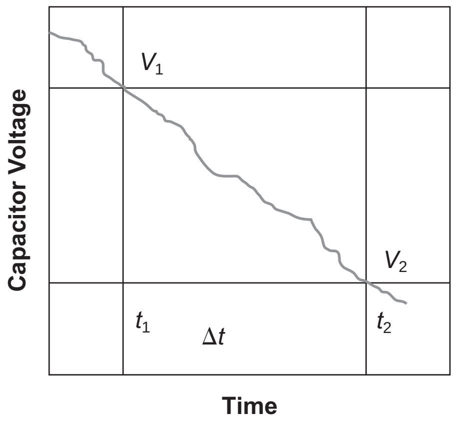


Figure 7.5 Capacitor voltage falls over time based on circuit's power and energy use.

voltage falls at varying rates as the circuit uses different amounts of power. The operation finishes at time t_2 and the voltage is V_2. We can calculate the energy used as:

$$W = \frac{CV_1^2}{2} - \frac{CV_2^2}{2} = C\frac{V_1^2 - V_2^2}{2}$$

We can also calculate the average power as:

$$P = C\frac{V_1^2 - V_2^2}{2(t_2 - t_1)}$$

One helpful relationship is that a load drawing a constant current I will take t seconds to discharge the capacitor from V_1 to V_2:

$$t(I) = C\frac{V_1 - V_2}{I}$$

Another relationship is that a load with a constant resistance R will take t seconds to discharge the capacitor from V_1 to V_2:

$$t(R) = -RCln\frac{V_2}{V_1}$$

7.4.1 Using Ultracapacitors

Ultracapacitors offer very high capacitances yet low leakage so they are very useful for this type of circuit. For example, 1 F capacitor charged to 5.0 V could power a 10 mA circuit for 340 seconds until the voltage fell to 1.6 V (the minimum operating voltage for the MCU).

Note that ultracapacitors are not ideal capacitors. As shown in Figure 7.6, they consist of the equivalent of many small capacitors connected with resistors. This means that if the

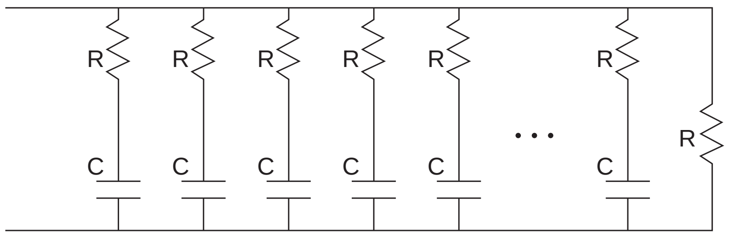


Figure 7.6 Equivalent circuit for ultracapacitor consists of many small capacitors connected with resistors.

capacitor is charged (or discharged) to a new voltage and then disconnected, the voltage will change for a period of time as the charge equalizes across the internal capacitors. In addition, the capacitance value may vary significantly from the rated value (e.g., −20% to +85%) due to manufacturing, temperature, age, and other factors. It is important to keep these limitations in mind when using ultracapacitors to measure energy use.

7.4.2 MCU Energy

Let's see how to use an ultracapacitor to measure the amount of power used by the MCU on the RDK. Figure 7.7 shows the power connections for the MCU. We can connect an ultracapacitor to the MCU supply rails (3V3_MCU and 3V3_MCUE) to power just the MCU and not the other hardware on the RDK.

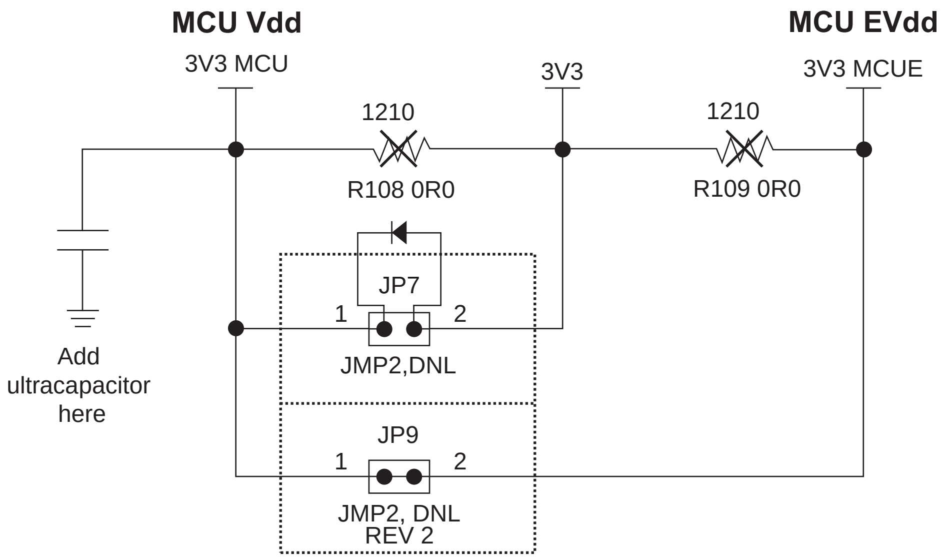


Figure 7.7 Measuring MCU energy on the RDK by adding an ultracapacitor.

If we disconnect R108 and R109, and short JP9, then all MCU power will need to come across JP7 from the 3V3 rail. We need to insert a diode across JP7 in order to ensure that the ultracapacitor powers only the 3V3 MCU and 3V3 MCUE rails, but not the 3V3 rail.

7.4.2.1 Input Voltage Protection

We need to be careful to ensure that all input signals to the MCU are no greater than its supply voltage (on 3V3_MCU). First, a sufficiently high voltage could damage the MCU. Second, each MCU input pin is protected with a diode connected to the MCU's internal

Vdd rail. If that input pin's voltage is high enough above the internal Vdd rail, then the diode will begin to conduct and the MCU will be powered in part by that input signal. This will reduce the amount of current being drawn from the 3V3_MCU rail and introduce error into our measurements. In some cases this can provide all of the current the MCU needs, enabling operation with no voltage present on Vdd.

7.4.3 RDK Energy

Measuring the RDK energy presents different challenges from measuring the MCU energy. The RDK can draw a large amount of current (e.g., 300 mA) when the WiFi module is transmitting. We need an ultracapacitor which is large enough to power the system for the time of measurement. An ideal 1 F capacitor starting at 5.0 V would be able to provide 300 mA for 6.7 seconds until the voltage falls to 3.0 V, the minimum operating voltage for the WiFi module. One approach is to use a capacitor (or several) with **high capacitance.** Another is to **disable unnecessary components** in order to eliminate their power consumption. We will examine how to do this in the next chapter.

7.5 POWER SUPPLY CONSIDERATIONS

The power supply can play a large role in a system's energy and power efficiency. Both voltage converters as well as switches (e.g. diodes, transistors) must be considered.

7.5.1 Voltage Converters

These devices convert power from an input voltage to an output voltage. This conversion may be needed for proper operation, reduced noise, improved energy efficiency, or otherwise better performance. Some converters, called **voltage regulators,** use **feedback** to ensure that the output voltage is fixed regardless of changes in the input voltage (within a given range). There are two common types of voltage converter. Each of these can be used as a regulator with the addition of feedback control.

7.5.1.1 Linear

A linear converter produces an output voltage which is lower than the input voltage. It uses a transistor which behaves as a variable resistor to drop the voltage to the output voltage level. The power dissipated depends in part on output current multiplied by the difference between the input and output voltages. The larger the voltage drop or output current, the greater the power loss is. There is also a second power loss term resulting from the **quies-**

cent current I_q (or ground current) flowing from the input pin to the ground pin. The resulting power loss is the sum of these terms:

$$P_{loss} = I_{out} * (V_{out} - V_{in}) + I_q * V_{in}$$

Power loss due to quiescent current can be significant when drawing only a small output current. For example, Figure 7.8 shows the quiescent current for a linear voltage regulator. Even with no load, this regulator draws 1.8 mA. There are other linear regulators available with lower quiescent currents which would serve a low power application better.

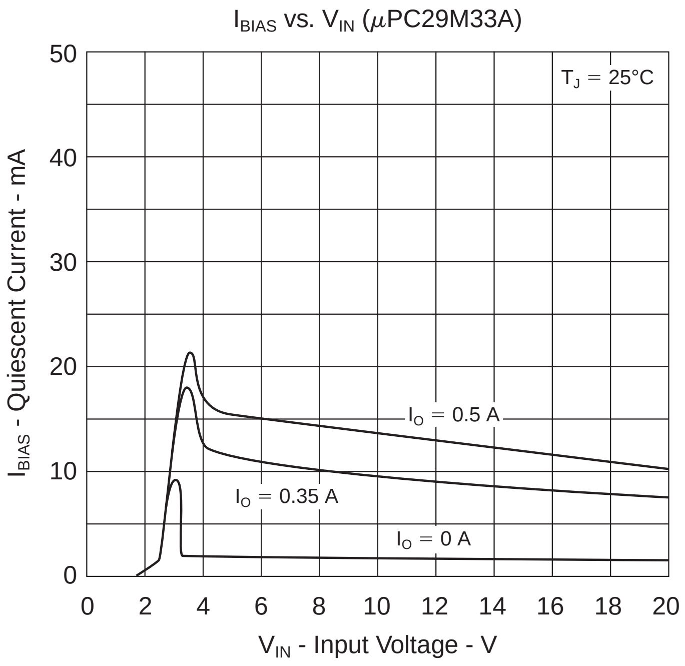


Figure 7.8 Quiescent current for linear voltage regulator as function of input voltage.

7.5.1.2 Switch-Mode

There are various types of switch-mode power converters. A **buck** converter produces an output voltage which is **lower** than the input voltage. A **boost** converter produces an output voltage which is **higher** than the input voltage. There are other types as well.

A switch-mode converter is typically much more efficient than a linear converter because it stores energy (using an inductor and a capacitor) and switches it (using transistors and possibly a diode) to perform the voltage conversion. The voltage conversion ratio is determined by the **duty cycle** of the periodic switching activity of the transistors (and possibly diodes).

The losses in the converter come from the non-ideal (parasitic) characteristics of these components. Conduction losses result from a component's non-zero resistance when current is flowing through it. Switching losses result from having to charge and discharge parasitic capacitances (e.g., transistor gate, diode junction), and also from the transistor's time in a lossy, partially conductive state between being fully off and fully on.

7.5.1.3 Trade-Offs

Linear regulators for low-power applications (e.g., < 1 W) are small and inexpensive, yet are likely to be less energy-efficient. Switching regulators for low-power applications are larger and more expensive than linear regulators due to the need for additional components, some of which are relatively large (e.g., inductors). However, they tend to be more efficient.

7.5.2 Power Gating Devices

Power supplies often need to control which sources provide power and which loads are powered. Diodes provide automatic gating while transistors can be switched on or off based on a control signal.

7.5.2.1 Diodes

Diodes can be used to provide protection by ensuring that current only flows in one direction. This allows multiple possible circuits to drive a given supply rail safely. Power loss in a diode is equal to the product of its forward voltage drop V_f and the current I. Note that the forward voltage drop depends on the current I.

$$P_{loss}(I) = I * V_f$$

7.5.2.2 Transistors

Transistors can be used to control whether power is provided to a domain or not. When the transistor is on, the power loss is equal to the product of the drain-to-source resistance r_{DS} and the square of the current I.

$$P_{loss}(I) = I^2 * r_{DS}$$

7.6 RDK POWER SYSTEM ARCHITECTURE

Let's examine the architecture of the power system for the RDK, shown in Figure 7.9.

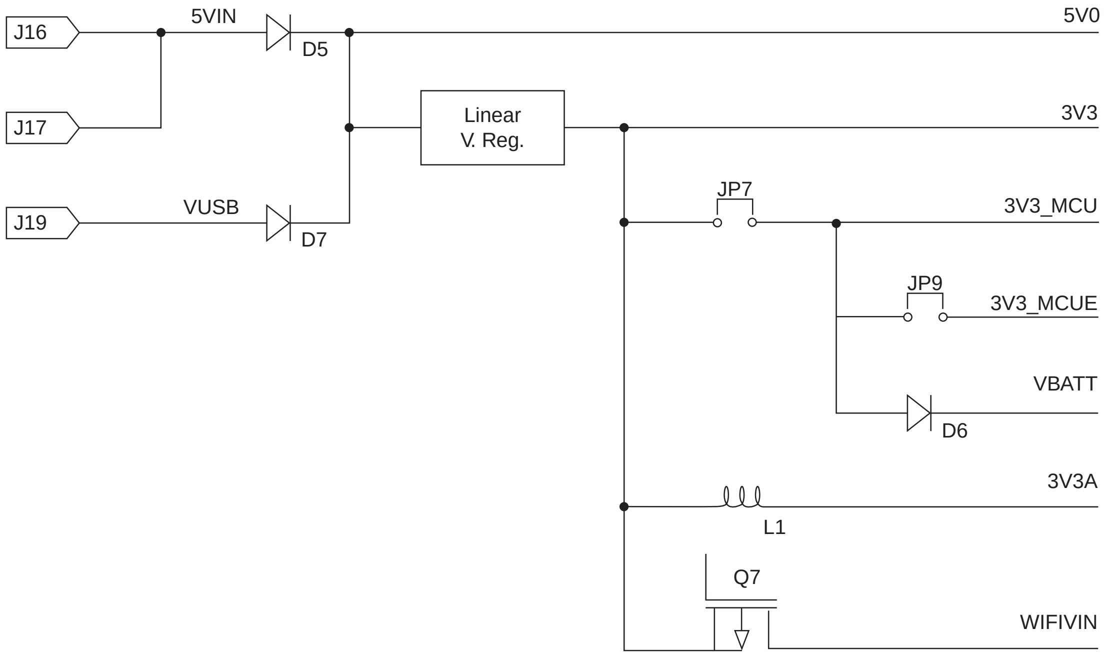


Figure 7.9 RDK power supply system drops nominal 5 V input to 3.3 V regulated output and offers switching and protection.

The RDK has the following nine power domains:

- Input power (nominally at 5 V) can be supplied through the USB jack (J19) to drive the **VUSB** domain.
- Input power (again nominally 5 V) can also be provided with a barrel connector (J16) or a two pin 0.1″ header (J17) to drive the **5VIN** domain.
- The **5V0** domain is driven by either 5VIN or VUSB (whichever has a higher voltage). Diodes D5 and D7 are used do this safely by protecting the 5VIN rail from the VUSB rail. Without the diodes, if power were applied at different voltages to J19 and J16 or J17, then the power supplies would likely be damaged due to excessive current.[2]
- A low-dropout (LDO) linear voltage regulator (U22, type UPC29M33A) draws its power from the 5V0 rail and drops it to a fixed 3.3 V to drive the **3V3** rail.
- Power for the MCU is provided using **3V3_MCU** and **3V3_MCUE.** The current on these rails can be monitored at jumpers JP7 and JP9, after removing shorting resistors R108 and R109 discussed previously.

[2] Note that J16 and J17 are not protected with diodes, so do not try to power the board from both connectors simultaneously.

- **VBATT** charges an ultracapacitor which provides standby power for the WiFi module when the RDK is not powered.
- Power for noise-sensitive analog portions of the RDK is provided on the **3V3A** rail, which is connected to the 3V3 rail using an inductor to reduce noise.
- The WiFi module's power comes from the **WIFIVIN** rail, which can be switched on or off using the P-channel MOSFET transistor Q7.

The major devices in each power domain are listed here:

- 5V0: LED1, LED17, optocoupler Q3 for triac, LED2, backlight LED for LCD, Eink display and driver U21, TK debugger U25
- 3V3: serial EEPROM U2, zero crossing detector U3, ambient light sensor U4, LCD, micro SD card interface, speaker amplifier U8, accelerometer U13, temperature sensor U12, RS232 transceiver U14, LIN transceiver U15, LED3-LED16, IR emitter D4, IR detector U19, pushbuttons SW2–4
- 3V3_MCU: MCU U1
- 3V3_MCUE: MCU U1
- 3V3A: headphone amplifier U7, microphone U10 and microphone amplifier U9, potentiometer VR1
- WIFIVIN: Gainspan WiFi module U16, WiFi serial EEPROM U18
- VBATT: ultracapacitor C72 for WiFi module

7.7 RL78 VOLTAGE AND FREQUENCY REQUIREMENTS/POWER AND ENERGY CHARACTERISTICS

Let's look at the RL78/G14 data sheet's electrical characteristics section. As shown in Table 7.1, the minimum voltage required **increases** as the desired clock frequency **increases.** The transistors need higher voltages to switch faster and support higher clock frequencies.

TABLE 7.1 Minimum Supply Voltages Required for Different Operating Frequencies

OPERATING FREQUENCY	MINIMUM V_{DD}
4 MHz	1.6 V
8 MHz	1.8 V
16 MHz	2.4 V
32 MHz	2.7 V

TABLE 7.2 RL78G14: Energy per Clock Cycle for Various Speeds and Voltages

FREQUENCY (MHz)	VOLTAGE (V)	CURRENT (mA)	POWER (mW)	ENERGY PER CYCLE (pJ)
0.032768	2.0	0.005	0.010	305.2
4	2.0	1.40	2.80	700.0
8	2.0	1.30	2.60	325.0
32	3.0	5.4	16.2	506.3

Next let's examine how much energy is used per clock cycle at four different operating points, as shown in Table 7.2. This data was gleaned from the hardware manual and is a good starting point to see how the MCU behaves.

- The **lowest power** operating point runs the clock **as slow as possible** (using the 32.768 kHz oscillator), and therefore can use a low voltage (2 V) to reduce power. The resulting power is 10 μW. Each clock cycle uses 305.2 pJ of energy.
- The **lowest energy** operating point also occurs when using the low-speed oscillator. Running at higher speeds consumes more energy per clock cycle, although the 8 MHz operating point is almost as low (325 pJ).

It might seem that the lowest power and lowest energy operating points will always be the same. However, reality is a bit more complicated. This comparison is biased by the remarkably low power consumption of the 32 kHz oscillator. If we examine the operating points using just the high speed oscillator, we find the lowest energy point is at 8 MHz. Remember that static power is wasted from the point of view of computation. As we increase the clock frequency, we divide the overhead of static power over more clock cycles. This reduces the overhead per instruction. In general, the most energy-efficient way to use an MCU is to run at the most energy-efficient speed when there is any work to do, and shift into an extremely low-power standby state otherwise.

It is interesting to note that the MCU can operate with lower power or energy than these levels by using a lower operating voltage. Notice that $V_{DD} = 1.6$ V is sufficient for running at 32 kHz (see Table 7.1). This is an optimal operating point: it provides the highest clock frequency which will run at a given voltage. We would expect the power to fall to about $(1.6\text{ V}/2.0\text{ V})^2 = 64\%$ of the original value. If we scale all of the power and energy calculations according to the minimum voltages we can predict the minimum power and energy required, as shown in Table 7.3.

The results of the calculations show that the power used for the 32 kHz case falls by nearly one half, and the energy falls by about one third. The other cases see a smaller increase because the relative voltage drops are smaller (2.0 V to 1.8 V, and 3.0 V to 2.7 V).

TABLE 7.3 Estimated Power and Energy for RL78G14 Running at Optimal Operating Points

FREQUENCY (MHz)	VOLTAGE (V)	ESTIMATED POWER (mW)	ESTIMATED ENERGY PER CYCLE (pJ)
0.032768	1.6	0.006	195.3
8	1.8	2.106	263.3
32	2.7	13.122	410.1

7.8 RL78 CLOCK CONTROL

Let's start by looking at how we control the clock frequency and perform clock gating for the RL78 family of MCUs (Renesas Electronics Corporation, 2011a).

7.8.1 Clock Sources

There are multiple oscillators in an RL78 family MCU, as shown in Figure 7.10. The first three can be used to clock the CPU and most of the peripherals (serial and timer array units, analog to digital converter, I2C interface, etc.).

- The high-speed on-chip oscillator is configurable and generates clock signal f_{IH} at certain frequencies from 1 to 32 MHz. It also can generate a clock signal of up to 64 MHz for timer operation. These frequencies are approximate, and the accuracy can be increased by adjusting the value in the trimming register HIOTRM.
- The high-speed system clock oscillator uses an external crystal, resonator, or clock signal to generate the signal f_{MX}, which can range from 1 to 20 MHz.
- The subsystem clock oscillator uses an external 32.768 kHz resonator, crystal, or clock signal to generate the f_{XT} signal, which is called f_{sub} when clocking the MCU or peripherals.

There is a fourth oscillator available as well:

- The low-speed on-chip oscillator generates the f_{SUB} signal at approximately 15 kHz (+/−2.25 kHz). This signal can only be used by the watchdog timer, real-time clock, or interval timer.

7.8.2 Clock Source Configuration

There are several registers used to configure the various clocks and how they are used.

- The Clock Operation Mode Control Register (CMC) determines critical parameters such as oscillation amplitude and frequency range, and whether external pins

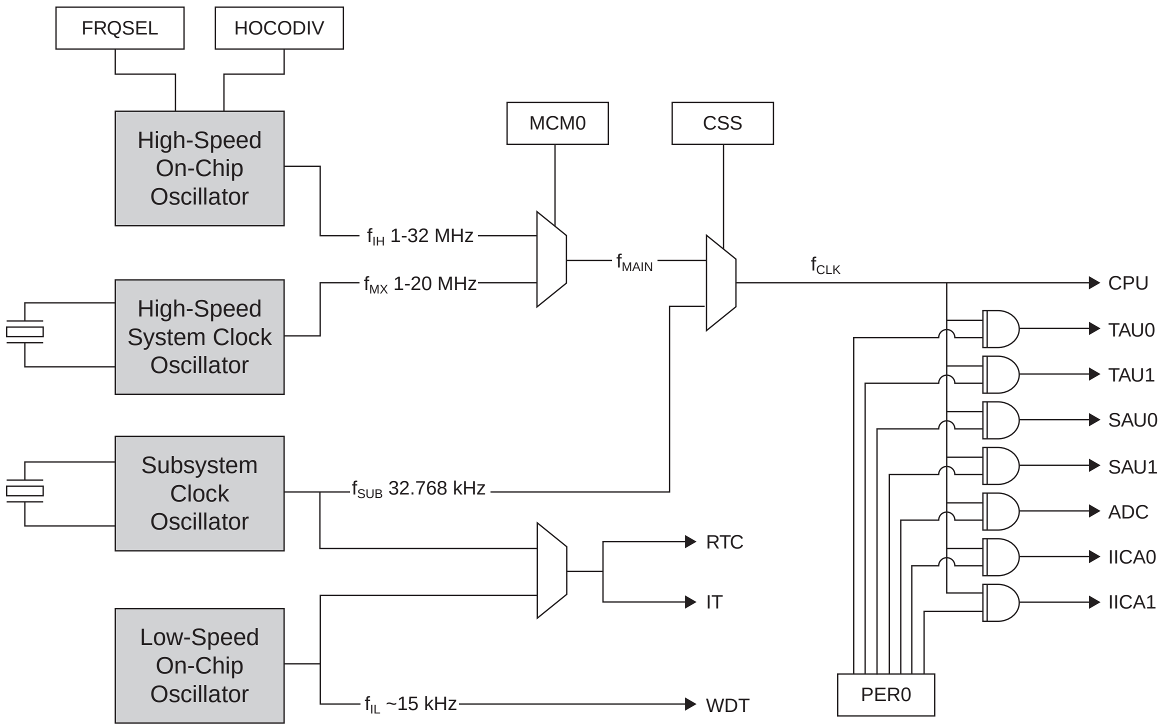


Figure 7.10 RL78 clock system overview.

are used as crystal connections or input ports. It can only be written once after reset in order to protect it from corruption by abnormal program behavior.

- The System Clock Control Register (CKC) selects the CPU/peripheral hardware clock (f_{CLK}) using the CSS bit, and the main system clock (f_{MAIN}) using the MCM0 bit.
- The Clock Operation Status Control Register (CSC) controls whether an oscillator is stopped or running. The MSTOP bit controls the high-speed system clock oscillator, the XTSTOP bit controls the subsystem clock oscillator, and the HIOSTOP bit controls the high-speed on-chip oscillator.
- The Peripheral Enable Register 0 (PER0) allows the program to disable the clock signals for unused peripherals in order to save power. The analog to digital converter, and each timer array unit, serial array unit, and IIC unit, can be controlled independently.

7.8.3 Oscillation Stabilization

There is some time delay between when the high-speed system clock oscillator is started and when it runs at the correct frequency and amplitude, as shown in Figure 7.11. There are two registers associated with controlling and monitoring this time delay.

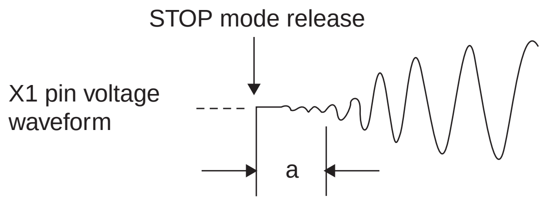


Figure 7.11 High-speed system clock oscillator start-up time.

- The Oscillation Stabilization Time Select Register (OSTS) specifies how long the MCU waits for the X1 clock to stabilize when coming out of stop mode. Delays from 2^8 to 2^{18} X1 counts are possible (i.e., from tens of microseconds to tens of milliseconds).
- The Oscillation Stabilization Time Counter Status Register (OSTC) indicates how much time has elapsed since coming out of stop mode. Each bit is set to one as the time threshold passes and remains at one.

7.8.4 High-Speed On-Chip Oscillator Frequency Selection

The high-speed on-chip oscillator's output frequency f_{IH} can be selected in two ways.

First, the FRQSEL bits of option byte 000C2H can be used to specify a speed.

					FREQUENCY OF THE HIGH-SPEED ON-CHIP OSCILLATOR CLOCK	
FRQSEL4	FRQSEL3	FRQSEL2	FRQSEL1	FRQSEL0	f_{HOCO}	f_{IH}
1	1	0	0	0	64 MHz	32 MHz
1	0	0	0	0	48 MHz	24 MHz
0	1	0	0	0	32 MHz	32 MHz
0	0	0	0	0	24 MHz	24 MHz
0	1	0	0	1	16 MHz	16 MHz
0	0	0	0	1	12 MHz	12 MHz
0	1	0	1	0	8 MHz	8 MHz
0	1	0	1	1	4 MHz	4 MHz
0	1	1	0	1	1 MHz	1 MHz
Other than above					Setting prohibited	

Figure 7.12 Oscillator speed selection with option byte 000C2H.

Second, the frequency select register HOCODIV can be used, as shown in Figure 7.13. There are two possible sets of frequencies based on whether the FRQSEL3 bit of option byte 000C2H is set to 1 or 0.

			SELECTION OF HIGH-SPEED ON-CHIP OSCILLATOR LOCK FREQUENCY			
			FRQSEL4 = 0		FRQSEL4 = 1	
HOCODIV2	HOCODIV1	HOCODIV0	FRQSEL3 = 0	FRQSEL3 = 1	FRQSEL3 = 0	FRQSEL3 = 1
0	0	0	f_{IH} = 24 MHz	f_{IH} = 32 MHz	f_{IH} = 24 MHz f_{HOCO} = 48 MHz	f_{IH} = 32 MHz f_{HOCO} = 64 MHz
0	0	1	f_{IH} = 12 MHz	f_{IH} = 16 MHz	f_{IH} = 12 MHz f_{HOCO} = 24 MHz	f_{IH} = 16 MHz f_{HOCO} = 32 MHz
0	1	0	f_{IH} = 6 MHz	f_{IH} = 8 MHz	f_{IH} = 6 MHz f_{HOCO} = 12 MHz	f_{IH} = 8 MHz f_{HOCO} = 16 MHz
0	1	1	f_{IH} = 3 MHz	f_{IH} = 4 MHz	f_{IH} = 3 MHz f_{HOCO} = 6 MHz	f_{IH} = 4 MHz f_{HOCO} = 8 MHz
1	0	0	Setting prohibited	f_{IH} = 2 MHz	Setting prohibited	f_{IH} = 2 MHz f_{HOCO} = 4 MHz
1	0	1	Setting prohibited	f_{IH} = 1 MHz	Setting prohibited	f_{IH} = 1 MHz f_{HOCO} = 2 MHz
Other than above			Setting prohibited			

Figure 7.13 Oscillator speed selection with HOCODIV register.

7.9 RL78 STANDBY MODES

In addition to the normal operating mode of executing instructions, the RL78 offers several standby modes in which the processor cannot execute instructions but other portions of the MCU continue to operate. Figure 7.14 presents a state diagram showing the halt, stop, and snooze states and possible transitions among them.[3] Note that the peripherals are functioning in the Halt and Operating states, while most are off in the Stop and Snooze modes. Turning off the peripherals dramatically reduces power consumption.

[3] Note that this is a simplification of the complete state diagram presented in the RL78G14 hardware manual.

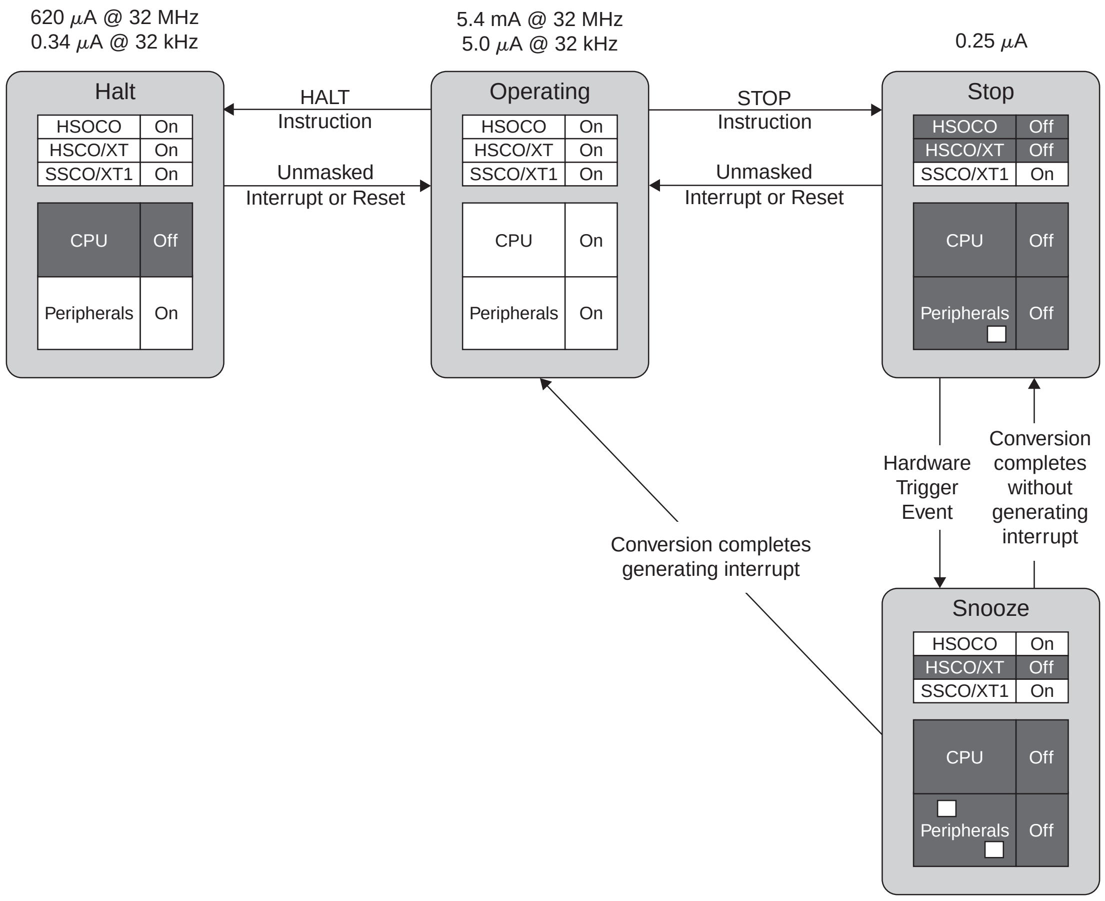


Figure 7.14 MCU operating and standby state transitions with snooze mode.

A circuit's current consumption can be cut significantly by using the standby states: starting at 5.4 mA when executing instructions at 32 MHz, current falls to 620 µA when halted but with clocks running at 32 MHz, and falls further to 0.25 µA when stopped with only the 32.768 kHz subsystem clock running.

Table 7.4 shows which portions operate in the different standby modes. Note that which clock source is used affects which subsystems can operate in HALT mode.

Let's examine each of the available standby states next.

7.9.1 Halt

A program executes the HALT instruction to enter the halt mode. The CPU stops executing instructions, but some or all peripherals continue to operate. The CPU clock continues to

TABLE 7.4 MCU Subsystem Operation in Standby Modes

SUBSYSTEM	HALT		STOP	SNOOZE
	MAIN SYSTEM CLOCK	SUBSYSTEM CLOCK		
Port	y	y	y	y
Power On Reset	y	y	y	y
Voltage Detection Circuit	y	y	y	y
External Interrupt	y	y	y	y
Key Interrupt	y	y	y	y
Real-Time Clock	y	y	y	y
Interval Timer	y	y	y	y
Clock/Buzzer	y	partial	partial	partial
Watchdog Timer	config.	config.	config.	config.
Serial Array Unit	y	config.	wake to snooze	partial
Analog to Digital Converter	y	n	wake to snooze	y
Digital to Analog Converter	y	config.	y	y
Comparator	y	config.	partial	partial
I2C Array	y	n	adx. match wake to operating	n
Data Transfer Controller	y	config.	n	y
Event Link Controller	y	y	y	y
Timer Array Units	y	config.	n	n
Timers RJ, RD, RG	y	config.	partial	n
General Purpose CRC	n	n	n	n
High-Speed CRC	y	y	n	n
Illegal Memory Access Detection	DTC only	DTC only	n	DTC only

run. If the main system clock is used then all peripherals will be able to operate, while if the subsystem clock is used some peripherals will be unavailable.

The MCU exits the halt mode if it receives an unmasked interrupt request or a reset signal.

7.9.2 Stop

A program executes the STOP instruction to enter the stop mode, which shuts down most peripherals in order to save additional power. The stop mode shuts down the main oscillator (X1 pin).

The MCU exits the stop mode if it receives an unmasked interrupt request or a reset signal. When exiting stop mode, the oscillator must start up again so it incurs the stabilization delay described earlier.

7.9.3 Snooze

In order to use the Snooze mode the program must configure the peripheral accordingly, and then the MCU enters the Snooze mode with a STOP instruction. The following peripherals can be configured to move the processor from STOP to SNOOZE mode when an appropriate trigger condition occurs:

- A/D converter: upon receiving a conversion request from INTRTC, INTIT, or ELC
- Serial array unit when configured for CSI or UART mode: upon receiving data
- Data transfer controller: upon activation event occurring

For example, the ADC can be triggered by the real-time clock or the interval timer, as shown in Figure 7.15. When triggered, the ADC will assert a signal starting up the high-speed on-chip oscillator clock. The clock starts up (and stabilizes for the specified delay) and is fed to the ADC, which uses it to perform the specified conversion(s). After the conversion(s) complete there are two possible actions. If the conversion result is within a specified range, then no interrupt is generated. Otherwise an interrupt is generated.

- If no interrupt is generated, the clock stops running and the system goes back into snooze mode.

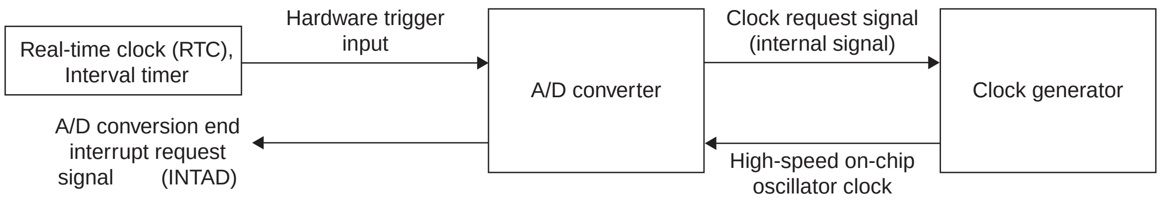

Real-time clock (RTC), Interval timer
Hardware trigger input
A/D converter
Clock request signal (internal signal)
Clock generator
A/D conversion end interrupt request signal (INTAD)
High-speed on-chip oscillator clock

Figure 7.15 Example of snooze mode operation with ADC.

- If an interrupt is generated, the clock continues running. The interrupt signal wakes up the MCU from STOP mode so it can execute instructions.

Further details are available in a white paper (Renesas Electronics America Inc., 2011b).

7.10 RECAP

In this chapter we have seen how voltage and switching frequency determine the amount of power used by a digital circuit. We have also examined the relationship between power and energy. We have investigated the RL78 MCU family's design features which can reduce power and energy use.

7.11 REFERENCES

Renesas Electronics Corporation. (2011a). *RL78/G13 16-Bit Single-Chip Microcontrollers User's Manual: Hardware.*

Renesas Electronics America Inc. (2011b). *White Paper—RL78 Microcontroller Family: Using the Snooze Mode Feature to Dramatically Reduce Power Consumption.*

Chapter 8

Power and Energy Optimization

8.1 LEARNING OBJECTIVES

This chapter begins with a discussion of how to create power and energy models of an embedded system. We present models for both loads (such as the MCU and external devices) and power supply subsystems. We continue by examining optimization methods. We first survey techniques for peripherals and then examine MCU techniques in more detail in order to leverage voltage scaling, frequency scaling and standby modes.

8.2 MODELING SYSTEM POWER

It is extremely helpful to build mathematical models of a system's power and energy use because these models help us identify which components or subsystems dominate the overall behavior. By examining those first we can improve the performance quickly and effectively.

8.2.1 Basic Power Models

A component may have more than one operating mode. For example, an LED may be on or off. We need to create a power model for each mode. We can then select the appropriate model based on the component's operating mode.

8.2.1.1 Modeling Passive Components

The power dissipation of a **resistor** with resistance R can be expressed as a function of the voltage across it (V_R) or the current through it (I_R):

$$P_R = \frac{V_R^2}{R} = I_R^2 R$$

Ideal **capacitors** and **inductors** store energy perfectly and therefore do not dissipate power. However, real devices have parasitic resistances which lead to power loss. A capacitor has a parasitic resistance in series (equivalent series resistance) and in parallel. Inductors have parasitic resistances in series (equivalent series resistance).

8.2.1.2 Modeling Semiconductors

8.2.1.2.1 Diodes: The power which diodes (including light-emitting diodes (LEDs)) dissipate when they are forward biased is nonlinear because the current depends upon the voltage across it. We can calculate power (using a simplification of Shockley's ideal diode law) as:

$$P_D = V_F I_D = V_F I_S e^{(V_F/(nV_T))}$$

Here I_S is the reverse bias saturation current, V_T is the thermal voltage (about 25.85 mV at room temperature, 23° C), and n is the ideality factor (often approximated as 1).

For simplicity, we can determine the forward voltage for a given current by inspecting the component datasheet. Figure 8.1 shows the characteristics of a diode on the RDK used for power supply protection (D5 is Schottky diode, part number SBR2U30P1). This plot shows we should expect a forward voltage of about 120 mV when 10 mA of current is flowing through the diode when operating with an ambient temperature $T_A = 25°$ C.

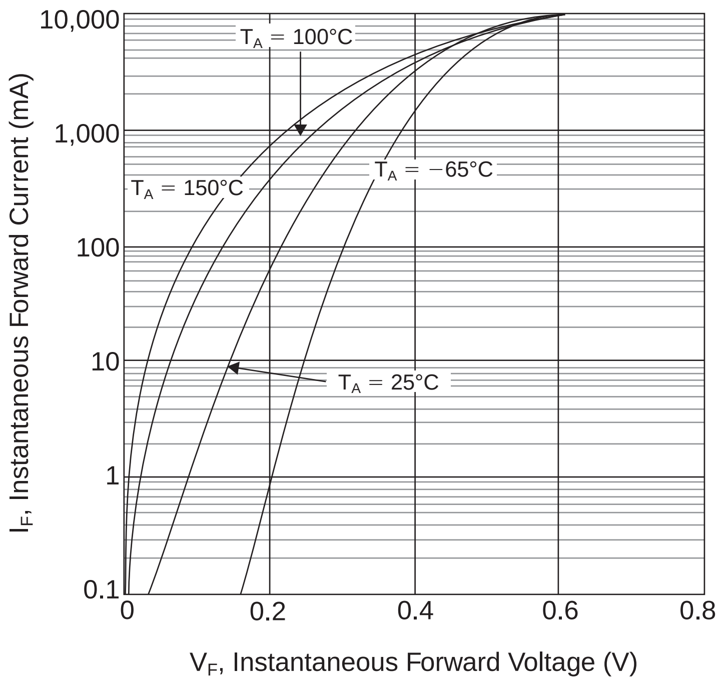


Figure 8.1 Current as a function of voltage for a Schottky diode.

We can also measure an actual component to find these values. Note that the forward voltage for a given current can vary significantly with different types of diodes, ranging from low (e.g., 0.1 V for a Schottky diode), to moderate (e.g., 0.7 V for a PN junction diode) to high (e.g., 1.6 V to 2 V for a red LED, 2.5 V to 3.7 V for a blue LED).

8.2.1.2.2 Transistors Used as Switches: Often transistors are used as switches to enable or disable power to other components. We consider bipolar and field-effect transistors. We assume that the transistors are either fully on or fully off, and spend negligible time in any intermediate states.

A bipolar transistor operating as a switch will either be on (in the saturation mode) or off (cutoff). When the transistor is saturated we can calculate the power loss as the product of voltage V_{CE} and collector current I_C.

$$P_Q = V_{CE} I_C$$

For simplicity we can derive these values from the component datasheet. Figure 8.2 shows the characteristics of an NPN transistor on the RDK (Q1 is an NPN transistor, part number MBT2222A). For collector currents of less than 20 mA we can approximate the V_{CE} as about 50 mV, simplifying the power model significantly.

A field-effect transistor (e.g., a MOSFET) will either be on (in the active mode) or off (cutoff). When the transistor is on, the drain-to-source current I_{DS} depends mainly on the

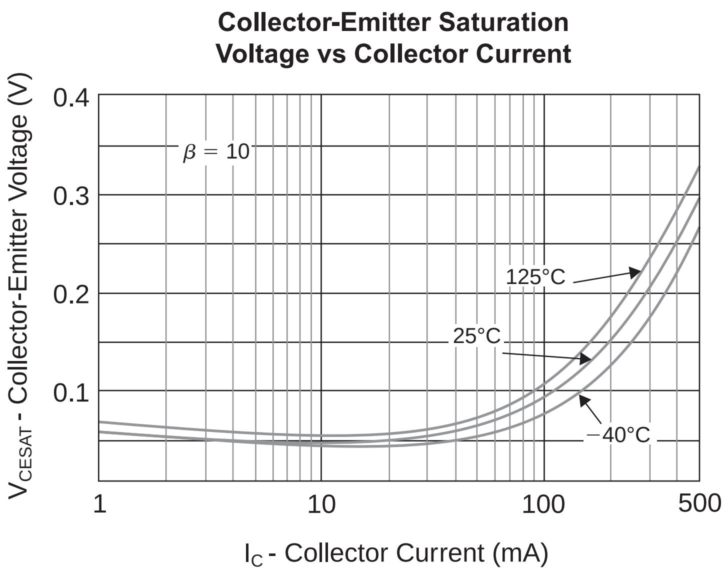


Figure 8.2 Voltage drop of saturated BJT as a function of collector current.

square of the overdrive voltage V_{OV} (equal to gate-to-source voltage V_{GS} minus the threshold voltage V_{th}). This means that the drain-to-source channel can be modeled as a resistor of value R_{DS} given a fixed overdrive voltage. The power loss of a FET in the active mode can be modeled using that resistance:

$$P_Q = I_{DS}^2 R_{DS}$$

The value of R_{DS} can be found in the datasheet for the device. For example, Figure 8.3 shows R_{DS} for a MOSFET on the RDK (Q2 is an N-channel device, part number RQK0609CQDQS). With $V_{GS} = 2.5$ V, the channel resistance R_{DS} is 90 mΩ.

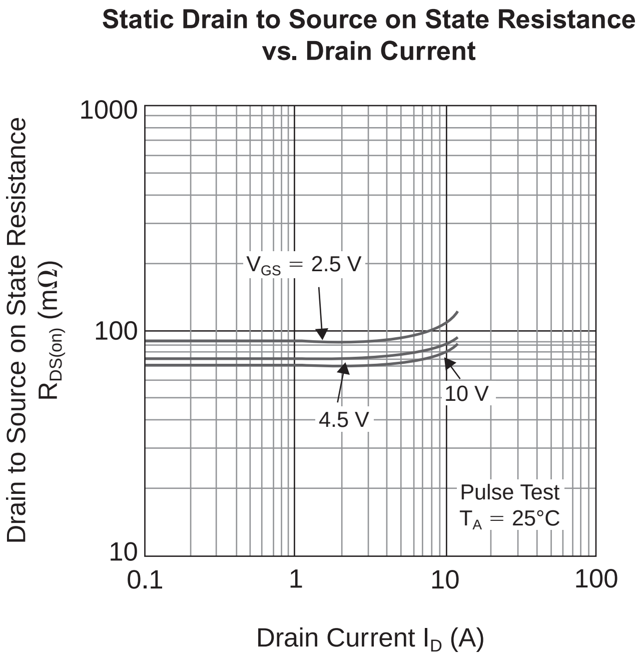


Figure 8.3 Drain to source resistance of MOSFET in active mode is approximately linear for low to moderate drain currents.

8.2.1.3 Modeling Digital Circuits

As shown in the previous chapter, we can model the power of a digital circuit as the sum of two components—the **static** and **dynamic** power.

$$P = S_P V_{DD}^2 + C_P V_{DD}^2 f_{CLK}$$

S_p and C_p are proportionality constants representing conductance (the inverse of resistance) and capacitance. These constants can be calculated by fitting curves to data derived experimentally or from the device datasheet.

We model the MCU and peripherals with the digital circuit model. Typical MCUs offer a variety of operating modes possible with different clock sources, so there may be a slightly different model for each mode.

8.2.2 Modeling the Power System

The power system may include **protection diodes, transistor switches,** and **voltage regulators.** Diodes and transistors can be modeled as described above. Voltage regulators require more discussion. There are two types of voltage regulators to consider: **linear** and **switching.**

A linear regulator uses a transistor as a variable resistor to drop the voltage to the output voltage level, dissipating some power. Power is also lost because a quiescent current I_q (or ground current) flows from the input pin to the ground pin. The resulting power loss is the sum of these terms:

$$P_{loss} = I_{out}*(V_{out} - V_{in}) + I_q*V_{in}$$

The power loss of a switching regulator depends greatly on the internal components and design parameters and operating modes. Extensive information is available in power electronics texts (Erickson & Maksimovic, 2001). Vendors of switching regulator modules typically provide plots showing efficiency as a function of load current.

8.2.3 Example: RDK Power System

Let's examine the RDK power system, shown in Figure 8.4. Further details were presented in the previous chapter.

Consider a 10 mA load on the 3V3 rail. It will dissipate its own power $P_{load} = 10$ mA * 3.3 V = 33 mW. It will also lead to additional power dissipation in the linear voltage regulator and either D5 or D7. The linear voltage regulator's power loss will be the sum of two terms: power loss due to the voltage drop ((5.0 V - 3.3 V)*10 mA = 17 mW) and the quiescent current (1.8 mA * 5.0 V = 8 mW). The total loss in the regulator is 25 mW. The loss in the diode (per Figure 8.1) is about 70 mV * 10 mA = 0.7 mW. In order to provide 33 mW of power to the load at 3.3 V, the power system uses an additional 25.7 mW, so the total power use is 58.7 mW. For an application with limited power or energy, the voltage regulator would be an excellent starting point for optimization.

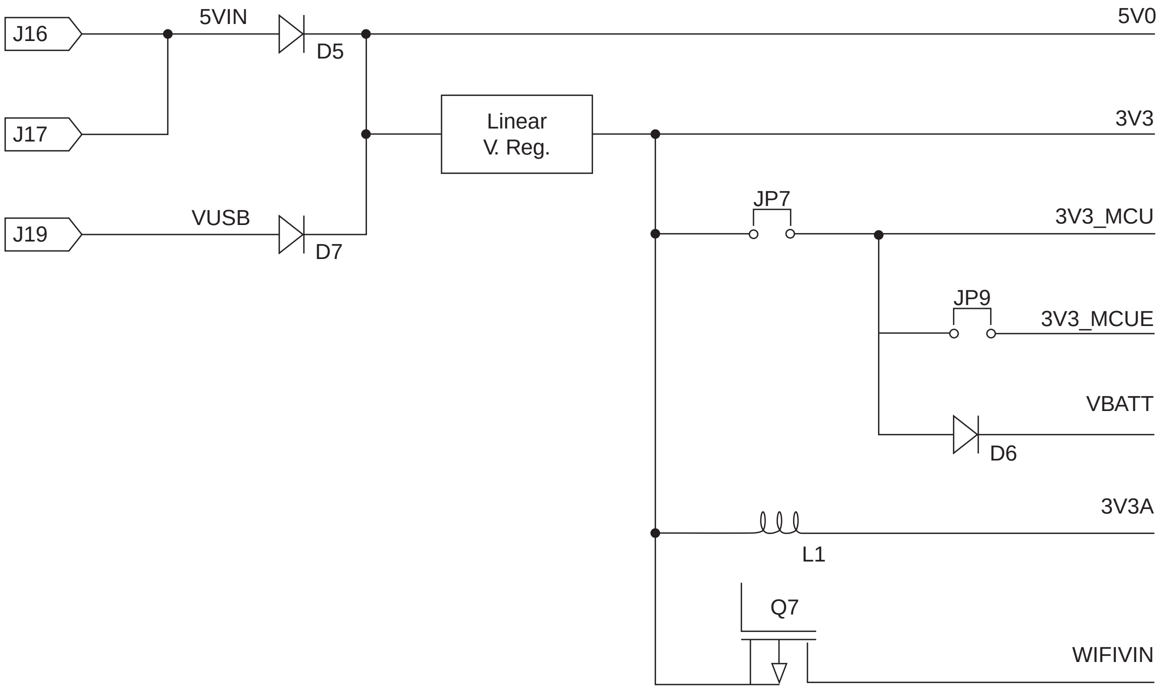


Figure 8.4 Overview of the RDK power system.

Now consider a larger 20 mA load being driven by the 5V0 rail. It will dissipate its own power (P_{load} = 20 mA * 5 V = 100 mW). It will also lead to additional power loss due to a voltage drop across the diode D5 or D7, depending on which power source is used. These diodes will have a forward voltage drop of about 80 mV (per Figure 8.1), leading to an additional power loss of 80 mV * 20 mA = 1.6 mW. This is a smaller loss due to the lack of use of a voltage regulator. Circuits which can operate without voltage regulators can save power.

8.2.4 Example: RDK Power Model

In some projects we may be given a clean slate and will be able to design the system hardware from scratch, selecting from a wide range of components and selecting the best. However, it is quite common to work with existing **(legacy)** hardware designs, where only minor hardware changes are feasible. In either case it is quite helpful to have a power model so we can focus our optimization efforts.

Let’s consider how power is used in the RDK. We can apply the modeling methods listed above to determine how much power is used by each component when operating (active) and then order them by greatest power first. Figure 8.5 shows the RDK components which use more than 20 mW each. The power system which supplies 3.3 V uses the most power (603 mW) primarily because of the linear regulator's low efficiency. The WiFi module is next, using nearly 500 mW when transmitting or receiving. Next comes the infrared-emitting LED and the white LED backlight for the LCD, and then the 5 V power system (protection diodes D5 and D7), the debugger and the green LEDs. This type of analysis shows where to start when seeking to reduce power or energy consumption.

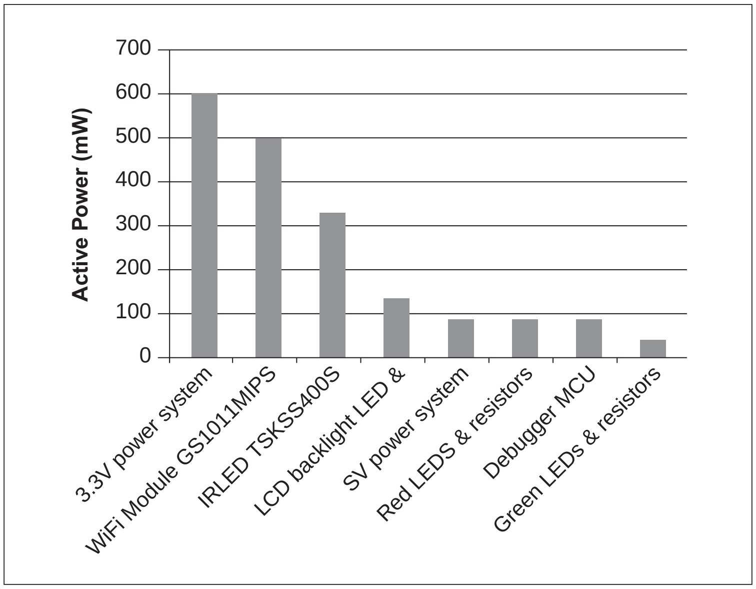


Figure 8.5 Estimated power consumption (mW) for RDK components using more than 20 mW when active.

Figure 8.6 shows the RDK components which use between 2 and 20 mW when active. Notice that the RL78/G14 MCU is running at full speed (32 MHz) at 3.3 V, but it only uses slightly more power than the 5 V power indicator LED (with current limiting resistor). These power models are for active components: LEDs which are lit, a WiFi module which is transmitting or receiving, or an an EEPROM which is being written. This models the RDK's maximum power use and assumes all peripherals are used simultaneously. We would like to enable only the peripherals which are needed for an application. The RDK

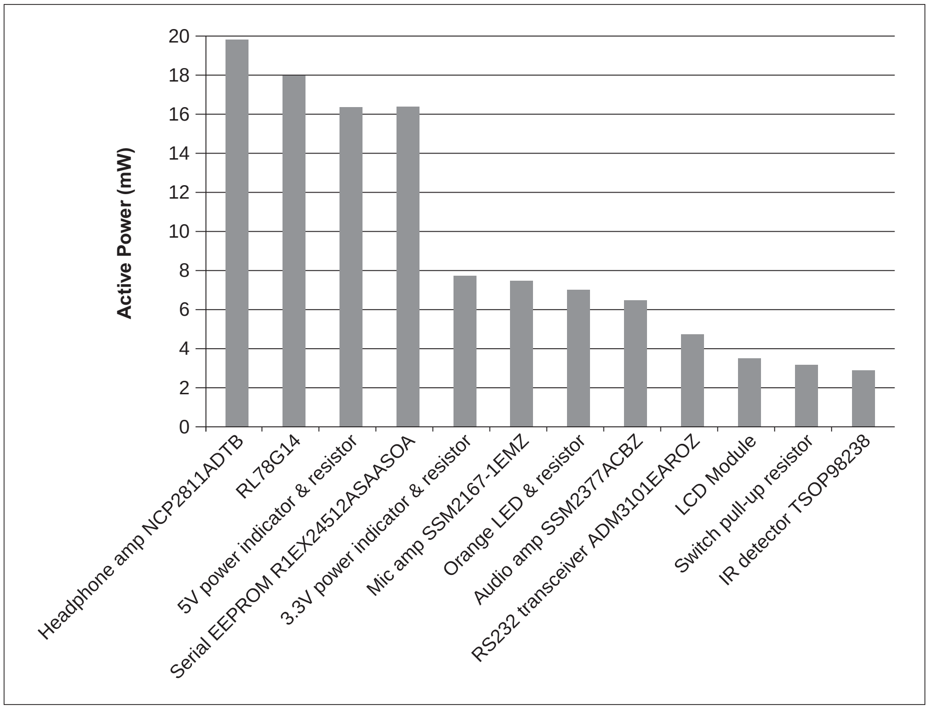


Figure 8.6 Estimated power consumption (mW) for RDK components using less than 20 mW when active.

allows the WiFi module to be disabled through hardware control using switch SW4. The RDK supports disabling some of its peripherals under software control using the GPIO control signal outputs listed in Table 8.1. Peripherals on the SPI or I2C bus can typically be disabled by sending a specific command on the bus. When disabled, these peripherals enter a very low-power standby mode. Similarly, it is possible to hold the debugger MCU in reset mode, reducing its power significantly.

TABLE 8.1 Control Signals for Disabling Peripherals on RDK

CONTROL SIGNAL	GPIO PORT AND BIT	MCU PIN
LCD Backlight Enable	P00	97
LCD Reset	P130	91
LCD SPI Chip Select	P145	98
Headphone Amplifier Enable	P04	93
Speaker Amplifier Enable	P06	41
Microphone Amplifier Enable	P05	42
MicroSD Card SPI Chip Select	P142	1

8.2.5 Modeling System Energy

We compute energy by integrating power over time. To create a precise and time-accurate energy model, we need to know **when** and for **how much time** each component in the power model is active. Often it is sufficient to estimate the duty cycle for each component—what fraction of time it is active. This allows for the use of weighted averages, and is the approach we will use.

8.3 REDUCING POWER AND ENERGY FOR PERIPHERALS

Let's see how we can reduce an embedded system's peripheral device power and energy consumption. The power equations previously presented give us some insight into how to reduce the **power** consumption for the system.

- Selecting more efficient components can reduce power. For example, using high-brightness LEDs will reduce the current required, reducing total power use. Alternatively, using a display such as the Eink display will consume no power until the image needs to change.
- Lowering the supply voltage will reduce power quadratically for both terms. For example, cutting V_{DD} to 80% of its original value will reduce power to $(80\%)^2 = 64\%$ of its original value.
- Some devices offer a standby mode with very low power requirements. This may be controlled through a logic level input (for example, as in Table 8.1), or through a command on a communication port such as SPI or I2C.
- Devices without standby modes can have their supply voltage shut off, eliminating all of their power. Transistors or dedicated switch ICs can be used to perform this switching. LEDs can be pulse-width modulated to control brightness.
- Disabling the clock ("clock gating") for unused circuits will eliminate their dynamic power. This is the approach used for the RL78′s internal peripherals.
- Reducing the switching frequency for circuits which are used will reduce their dynamic power proportionately.

Energy for peripherals can be reduced with the methods above, as well as by limiting the time for which the circuits are active.

- There may be built-in low-energy modes available for use (e.g., in the accelerometer).
- Faster communications can reduce the amount of time that a peripheral or the MCU needs to be active. A serial protocol such as SPI can be run at very high speeds (tens of MHz) when communicating with some devices (e.g., microSD memory cards). Data may be compressed before transmission, reducing active time.

- Peripheral devices may allow configuration of parameters which affect the active time, such as conversion rate, conversion resolution, settling times, noise filtering, time-outs, message size, etc.

8.4 REDUCING POWER AND ENERGY FOR THE MCU

Minimizing the power or energy used by an MCU is an interesting challenge which requires balancing various factors to reach the design goal.

Consider reducing the supply voltage to a digital circuit. This will clearly reduce power consumption. However, the transistors will take longer to switch because they are operating closer to the threshold voltage V_{th}, so they will not turn on as strongly. Looking at this from the point of view of a CPU core, lowering the clock frequency means that the processor has to be active longer to complete the same amount of processing work.

Optimizing energy thus requires us to balance multiple factors. Slowing the clock f_{CLK} lets us lower V_{DD} and therefore both static and dynamic power. Slowing the clock also raises the computation time, so that power is integrated over a longer time, potentially raising total energy used.

We will see later that if there is a low-power standby mode available, the best approach might be to run the processor as fast as possible (at the minimum V_{DD} possible for that frequency) when there is work to do, and put the processor into the standby mode otherwise.

One of the most important properties of the program the MCU will run is how many CPU instruction execution cycles it requires to do its work. For **non-real-time** systems we model the program as requiring C execution cycles every second to complete its work. The resulting utilization U of the processor is C/f_{CLK} given a clock frequency of f_{CLK}. For **real-time systems** the situation is more complicated, and the subject of extensive ongoing research. In some cases, the model defined above is applicable. We leave further discussion of these concepts to future work.

8.4.1 Optimization Approaches

In this section we measure the power and other characteristics of the RL78/G14 MCU on the RDK using an adjustable power supply and other test equipment. This allows us to evaluate many more operating conditions than are covered in the datasheet. The previous chapter used values from the datasheet, so there will be slight differences in values between these two chapters.

Given that our program requires C CPU execution cycles per second, how can we improve power or energy? We have several methods available: **scaling** (adjusting) the operating voltage, scaling the CPU clock frequency, and using standby modes such as stop and halt.

Scaling decisions can be made at design time and then be **static** (fixed) during program run-time. Alternatively, they may be made **dynamically** at run-time, assuming proper hardware and software support. The scaling may be applied to the entire system or to parts of it. There may be some domains with no scaling, and other individual domains within the system which can be scaled independently. For example, a multi-tasking system may scale MCU voltage and frequency differently on a per-task basis. Or the MCU voltage may be scaled while the LCD voltage is not scaled.

8.4.2 Voltage Scaling

One straightforward approach to reducing power and energy use is to reduce the supply voltage. The amount of reduction is constrained by the minimum voltage constraints of the MCU and any other circuitry on that supply rail. The minimum supply voltage for digital logic is related to the target operating frequency. Table 8.2 shows these specifications for the RL78G14 family MCUs.

TABLE 8.2 Minimum Supply Voltages Required for Different MCU Clock Frequencies

OPERATING FREQUENCY (f_{CLK})	MINIMUM SUPPLY VOLTAGE V_{DD}
4 MHz	1.6 V
8 MHz	1.8 V
16 MHz	2.4 V
32 MHz	2.7 V

Let's assume we've developed a functioning prototype of embedded system on the RDK with the MCU running at 8 MHz. The RDK uses a 3.3 V supply rail to power the MCU. We can run the MCU at 1.8 V instead, since that is the minimum supply voltage needed for 8 MHz. Reducing the supply voltage from 3.3 V to 1.8 V would reduce MCU power and energy to $(1.8\text{ V}/3.3\text{ V})^2 = 0.298$ of their original RDK-based values. This is a major improvement—eliminating 70.2% of the power and energy required by the MCU.

Peripheral logic will also benefit from such voltage scaling, assuming that it can operate at the lower voltage. If not, the peripherals must be powered at an adequate voltage (based on their requirements) and level-shifting circuitry may be needed to convert signals safely between the voltage domains.

Remember that we will need to generate the new supply voltage rail. As discussed previously, voltage converters and regulators are not 100% efficient, so some power will be lost in the conversion. This loss needs to be balanced against the gain from reducing MCU (and peripheral) power in order to determine whether this optimization is worthwhile.

8.4.3 MCU Clock Frequency Scaling

One approach to improving power or energy consumption is to scale the clock frequency at which the MCU processor core runs. The RL78G14 family of MCUs supports both internal and external clock sources. The internal high speed oscillator (HOCO) can generate an MCU clock signal f_{CLK} at speeds of 1, 4, 8, 12, 16, 24, and 32 MHz. An external oscillator running at a different frequency is also possible. There is also a subsystem oscillator available which runs at 32.768 kHz, offering exceptionally low power consumption. In this discussion we will only examine using the HOCO due to time and space constraints.

8.4.3.1 Power Analysis

Figure 8.7 shows the power consumption of the MCU at 3.3 V running at these speeds (using the HOCO). It also shows a linear approximation of power based on frequency. The total power (for $V_{DD} = 3.3$ V) can be modeled using this linear approximation as:

$$P_{MCU} = 2.893\ mW + \left(f_{CLK} {*} 0.4187 \frac{mW}{MHz} \right)$$

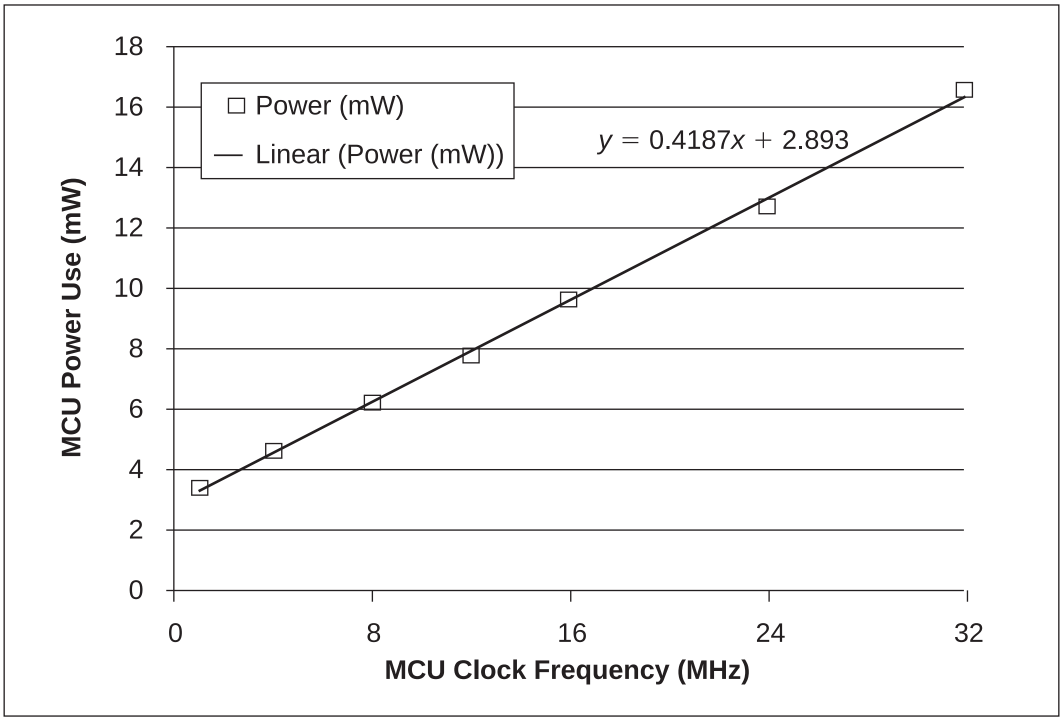


Figure 8.7 RL78G14 MCU power consumption at 3.3V (R5F014PJAFB).

This equation shows the static power term (2.893 mW), which is not affected by the clock rate. It also shows the dynamic power term which depends linearly on the clock rate. We see that the **lowest power** is used by running the MCU at the **lowest frequency.**

8.4.3.2 Energy Analysis

We can use this power model to determine the amount of energy needed for the MCU to perform one clock cycle of instruction processing by dividing the power required by the clock frequency, as shown in Figure 8.8.

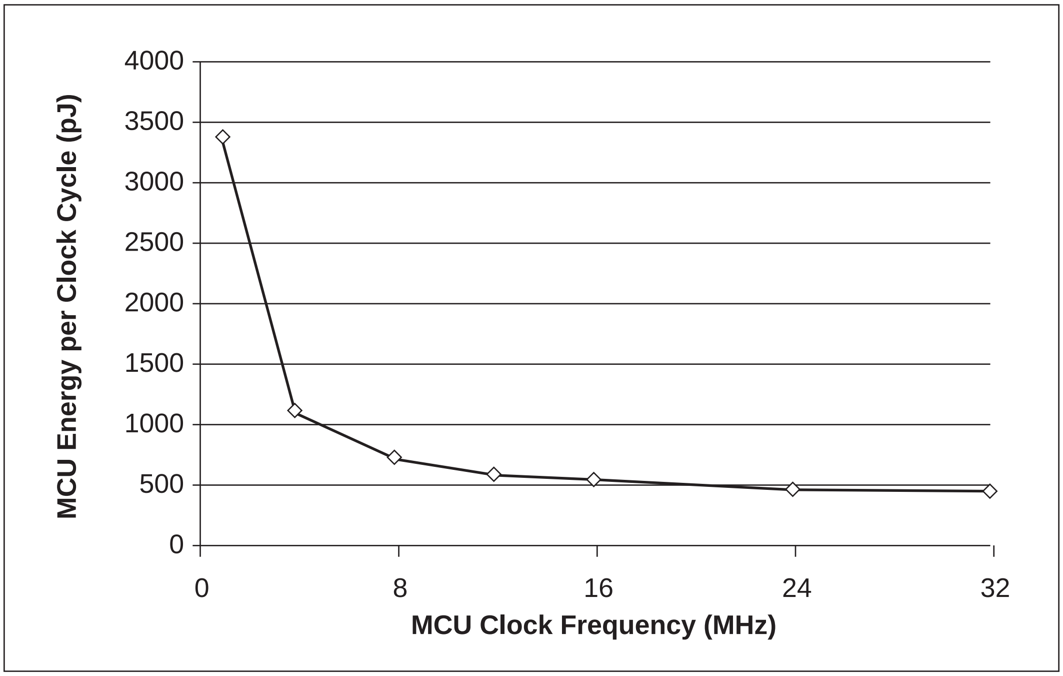


Figure 8.8 RL78G14 MCU energy consumption per clock cycle at 3.3V (R5F014PJAFB).

We can model the energy per clock cycle based on the MCU power model and the clock frequency:

$$E_{MCU\_per\_cycle} = \frac{2.893\ mW + \left(f_{CLK} * 0.4187\ \frac{mW}{MHz}\right)}{f_{CLK}} = \frac{2893\ pJ * MHz}{f_{CLK}} + 418.7\ pJ$$

This equation clearly shows the impact of the static power component (2.893 mW) being reduced as it is spread over more and more clock cycles (f_{CLK}). It also shows the dynamic power component of 0.4187 mW/MHz (equal to nJ) per clock cycle. We see that the **lowest energy** results from running the MCU at the **highest frequency.**

8.4.3.3 Selecting the Operating Frequency

In order to reduce power we reduce the frequency f_{CLK} of the CPU as much as possible, but no lower than C. Recall that C is the maximum number of execution cycles the program requires each second to complete its work. This ensures the resulting utilization does not exceed 1, and all work is completed.

This approach reduces power and energy. Given the power model of the RL78G14, we can see that dynamic power will be reduced based on the number of compute cycles C required per second:

$$P_{MCU} = 2.893\ mW + \left(C * 0.4187 \frac{mW}{Hz} \right)$$

When using the HOCO we are limited to selecting the smallest HOCO frequency f_{HOCO} which is not less than C. If we use an external oscillator instead, then we can select a device which produces the desired frequency exactly.

8.4.4 MCU Voltage and Clock Frequency Scaling

The previous section evaluates power and energy at a fixed supply voltage of $V_{DD} = 3.3$ V. If we can run the MCU at a lower voltage, we can reduce **both static and dynamic power** since they depend quadratically on the voltage. We can therefore also reduce energy.

8.4.4.1 Power Analysis

Table 8.3 shows MCU characteristics for four different **operating frequencies.** The second column shows the **minimum supply voltage** V_{DDmin} for each of these frequencies as spec-

TABLE 8.3 RL78G14 Power and Energy Measured at Various Operating Points.

OPERATING FREQUENCY (f_{CLK})	MINIMUM SUPPLY VOLTAGE V_{DDMIN} (V)	POWER (mW)	ENERGY PER CYCLE (pJ)	REDUCTION FROM V_{DD} = 3.3V
4 MHz	1.6 V	1.09	272.6	76.48%
8 MHz	1.8 V	2.167	270.9	65.23%
16 MHz	2.4 V	6.308	394.3	34.29%
32 MHz	2.7 V	12.812	400.4	22.26%

ified by the MCU documentation. Each frequency/voltage pair **(operating point)** results in the **lowest MCU power and energy** dissipation for that frequency.

The third column shows the **power** used by the MCU at each power point, while the fourth shows the **energy per clock cycle.** Because the minimum voltages are all less than the 3.3 V used above, we reduce power and energy use. Notice that the lowest energy operating point is no longer the highest frequency. This is because we have reduced the supply voltage to the minimum possible for the clock frequency. This results in much greater energy savings for the lower clock frequencies, as they can run at lower voltages.

Figure 8.9 shows the improvement in power consumption achieved with the use of voltage scaling. We can model the MCU power for these frequencies as:

$$P_{MCU} = -0.8279\ mW + \left(f_{CLK}{*}0.4282\ \frac{mW}{MHz}\right)$$

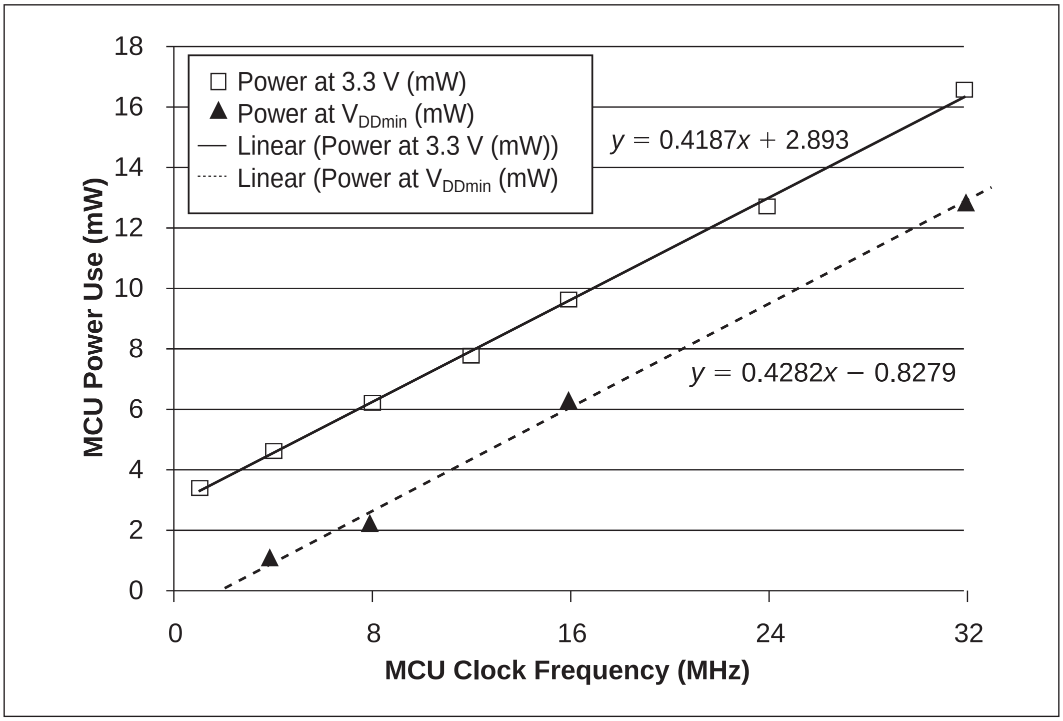


Figure 8.9 RL78G14 MCU power consumption with supply voltage scaled down to V_{DDmin} (R5F014PJAFB).

Note that the clock frequencies of 1, 4, 12, and 24 MHz are not included because the minimum supply voltages for those frequencies are not specified. We would need to run the processor at the minimum voltage for the next higher specified frequency. For example, to run at 12 MHz we would need to use the supply voltage requirement for 16 MHz (2.4 V). The resulting power use would fall somewhere between the 3.3 V value and the trend line for power using V_{DDmin}.

8.4.4.2 Energy Analysis

Figure 8.10 shows the improvement in energy required per compute cycle when voltage scaling is used. Note that the energy per cycle at 4 and 8 MHz is almost equal, and similarly the energy per cycle at 16 and 32 MHz is almost equal. The 4 and 8 MHz operating points are much more energy efficient than the 16 and 32 MHz points, using about 32% less energy. For some applications this may be very useful.

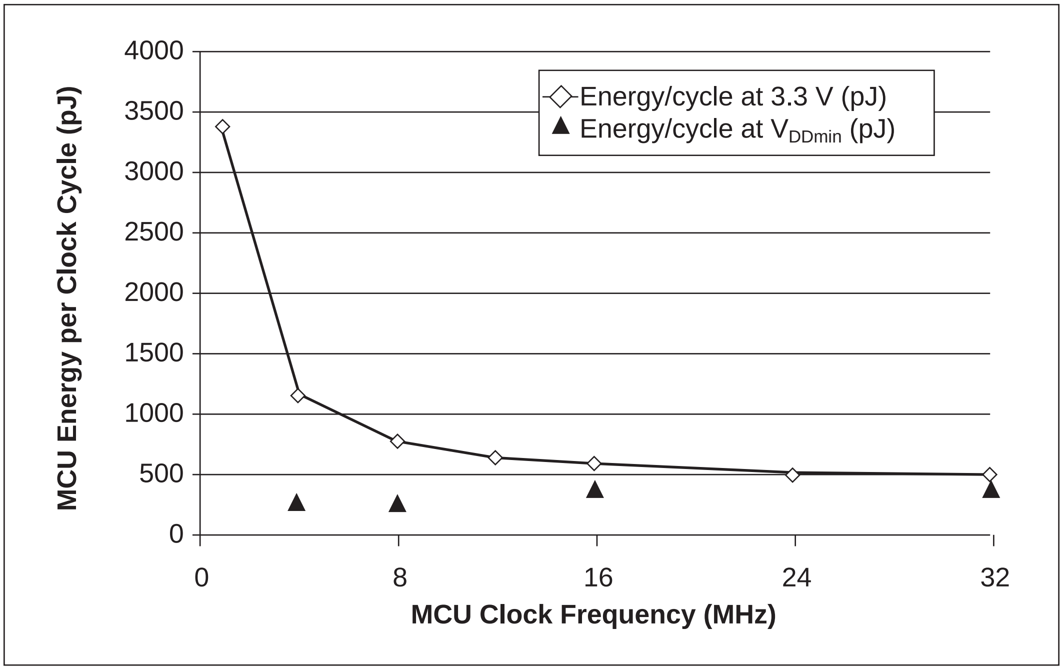


Figure 8.10 RL78G14 MCU energy consumption per clock cycle with supply voltage scaled down to V_{DDmin} (R5F014PJAFB).

We can update our energy model based on the power model:

$$E_{MCUpercycle} = \frac{-0.8279\ mW + \left(f_{CLK} * 0.4282\ \frac{mW}{MHz}\right)}{f_{CLK}} = \frac{-827.9\ pJ * MHz}{f_{CLK}} + 428.2\ pJ$$

This equation is only valid at the four operating points in Table 8.3. Using other frequencies will require operating at the minimum voltage of the next highest frequency, increasing power and energy.

By combining both voltage and frequency scaling we see significant improvements in the power and energy required for computation.

8.4.4.3 Selecting the Operating Point

We select the operating point with the following steps. First we determine the minimum clock frequency based on C, as in Section 8.4.3.3.When using the HOCO, if f is not a valid HOCO frequency, then we will use the smallest HOCO frequency f_{HOCO} which is not less than C. Next, we find the operating point in Table 8.3 with the minimum power or energy that supports operating at f_{HOCO}, depending on which parameter we are trying to optimize. We then select that operating voltage.

8.4.5 MCU Standby Mode

As described in the previous chapter, the RL78 family of MCUs offers several standby modes (halt, stop, and snooze) in which the processor cannot execute instructions but other portions of the MCU continue to operate. All of the peripherals can function in the Halt mode, while most are off in the Stop and Snooze modes. Turning off the peripherals and oscillators dramatically reduces power consumption but reduces device functionality and increases wake-up times.

Switching between modes takes a certain amount of time. Figure 8.11 shows the times when the MCU uses the HOCO as its clock in operating, halt, and snooze modes. Some delays result from powering up an oscillator and allowing it to stabilize while others are from reset processing. These delays may be ignored if the total transition time is small compared with the computational time required (C/f_{CLK}).

We place the MCU into a low-power standby mode when it is idle. When active, the MCU runs at a fixed frequency (e.g., 16 MHz). We can calculate the average power used by the MCU as a weighted average.

$$P_{MCU} = \frac{C}{f_{CLK}} P_{Active} + \left(1 - \frac{C}{f_{CLK}}\right) P_{Standby}$$

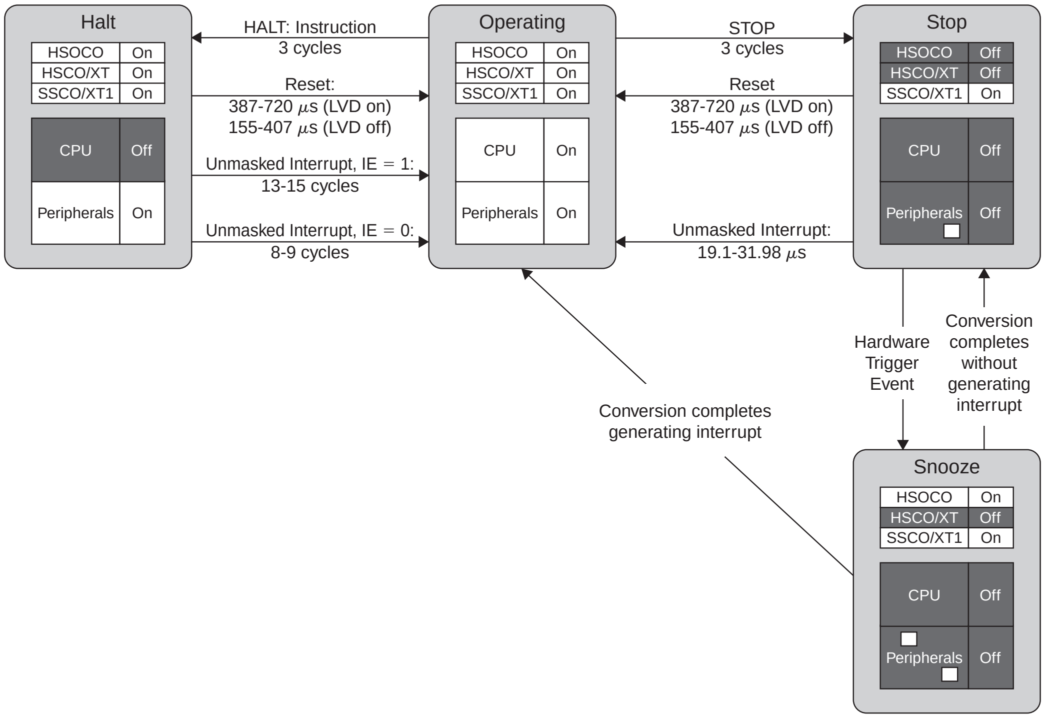

Halt
HSOCO On
HSCO/XT On
SSCO/XT1 On
CPU Off
Peripherals On
HALT: Instruction 3 cycles
Reset: 387-720 µs (LVD on) 155-407 µs (LVD off)
Unmasked Interrupt, IE = 1: 13-15 cycles
Unmasked Interrupt, IE = 0: 8-9 cycles
Operating
HSOCO On
HSCO/XT On
SSCO/XT1 On
CPU On
Peripherals On
STOP 3 cycles
Reset 387-720 µs (LVD on) 155-407 µs (LVD off)
Unmasked Interrupt: 19.1-31.98 µs
Stop
HSOCO Off
HSCO/XT Off
SSCO/XT1 On
CPU Off
Peripherals Off
Hardware Trigger Event
Conversion completes without generating interrupt
Conversion completes generating interrupt
Snooze
HSOCO On
HSCO/XT Off
SSCO/XT1 On
CPU Off
Peripherals Off

Figure 8.11 Transition delays when using standby modes and high-speed on-chip oscillator (HOCO) as CPU clock.

The standby modes can be used with the voltage and frequency scaling methods described, but we leave this discussion as future work.

8.5 RECAP

We have examined how to create power and energy models for an embedded system. We have used them to evaluate the impact of various possible changes for peripherals and the MCU. We then examined how voltage and frequency can be scaled down to reduce an MCU's active power dramatically. Finally we investigated the power savings possible using the MCU's standby mode.

8.6 REFERENCES

Erickson, R. W., & Maksimovic, D. (2001). *Fundamentals of Power Electronics* (2nd ed.). Norwell, Massachusetts, USA: Kluwer Academic Publishers.

Chapter 9

Memory Size Optimization

9.1 LEARNING OBJECTIVES

This chapter focuses on analyzing a program's memory use (for code and data) and then presents methods for reducing it. We begin by examining which types of memory are required by different components of the program. We then examine tool support for measuring these requirements in order to determine where to start optimizing. We then examine how to improve data memory use followed by code memory use, using language features, toolchain support, and better coding styles. Finally we examine how to reduce memory requirements for multitasking systems.

9.2 DETERMINING MEMORY REQUIREMENTS

9.2.1 Why? Cost

A microcontroller includes both RAM and ROM (typically flash ROM). RAM size often is the main factor in determining the relative cost of an MCU. For most MCUs in cost-sensitive markets it is impossible to add fast memory externally due to pin count constraints. Supporting single-cycle access would require bringing out the address bus (e.g., 20 bits), the data bus (e.g., 16 bits) and the control signals (e.g., 3 bits). The only way to add memory is to replace the MCU with one with more memory. This constraint makes it important to ensure that the program fits within the available memory.

9.2.2 A Program's Memory Use

Table 9.1 shows where the different portions of a program are stored in an MCU's memory. The compiler and linker use memory **segments** to hold program information. These are identified with bold borders in the table. There are three basic types of memory segment:

- CODE: used for executable code
- CONST: used for data which is placed in ROM
- DATA: used for data which is placed in RAM

TABLE 9.1 Diagram Showing Where Different Parts of Program are Stored in Memory.

	RAM			ROM	
DATA	*DATA segment*	Globals and Statics		*CONST segment*	Initialization Data
		Function Call Stack	PC, etc.		
			Arguments		
			Callee-Saved Registers		Const Data
			Local Variables		
			Temporary Storage		
		Heap			
CODE				*CODE segment*	Instructions

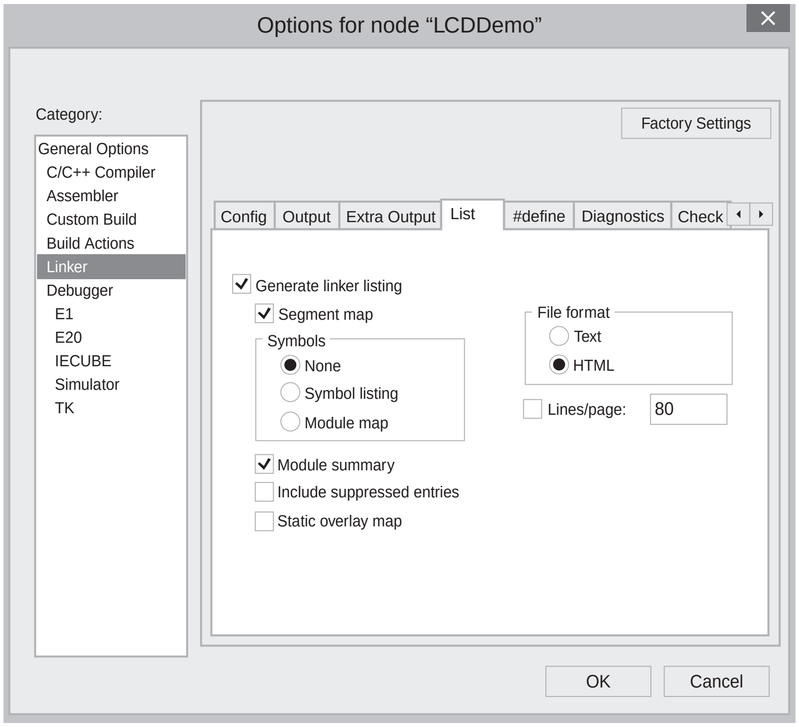
One complication with determining a program's memory requirements is that some are difficult or impossible to compute before running the program. The light gray entries in the table indicate memory sections with fixed sizes that are easily computed statically (i.e., at compile and link time, without running the program). The dark gray entries show memory sections which are difficult or impossible to compute statically. For example, the total amount of call stack space needed depends on the subroutine called nesting behavior, which may be data-dependent (e.g., a recursive function to compute the Fibonacci series, or a call to sprintf). For these sections, we must **estimate** the **worst-case values** in order to allocate enough space. A program with an overflowing stack is challenging to debug and should be avoided.

There is some overlap between the optimizations for program **speed** and those for **memory size.** A program which executes faster because the work is done with fewer instructions uses less code memory. A program which operates on less data (whether smaller items, or fewer items, or both) will be faster and uses data memory. It may also use less code memory. As a result, many of the optimizations used for speed can also benefit program size. However, there are also other optimizations which primarily benefit the size and have less effect on speed.

9.2.3 Linker Map File

The linker can generate a **map file** which provides information on how much memory of each type is used. With IAR Embedded Workbench, set the project options to generate a linker listing which includes a module summary and a segment map, as shown in

Figure 9.1. Each time the project is linked the map file will be generated. We can now examine the different portions of the map file.



[✓] Generate linker listing
[✓] Segment map
Symbols
● None
○ Symbol listing
○ Module map
[✓] Module summary
[ ] Include suppressed entries
[ ] Static overlay map
File format
○ Text
● HTML
[ ] Lines/page: 80

Figure 9.1 Linker Options for generating the map file.

9.2.3.1 Memory Summary

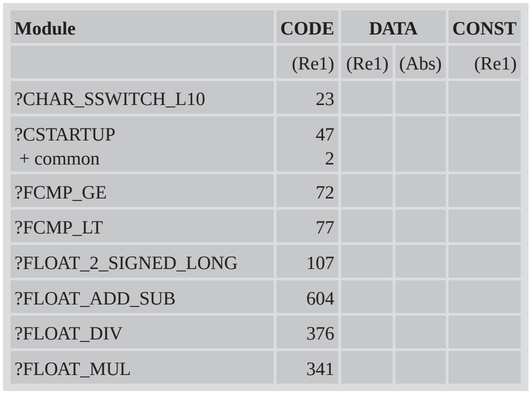
```
11 634 bytes of CODE  memory
   909 bytes of DATA  memory (+1 089 absolute)
 3 400 bytes of CONST memory
Errors: none Warnings: none
```

Figure 9.2 Linker memory summary in the map file.

Figure 9.2 shows the high-level information memory summary. This includes the size of each segment type. We can calculate the total RAM segment size: 909 bytes are needed for data. The MCU’s special-function registers are also listed (1089 bytes) but can be ignored. The total amount of ROM needed is the sum of the CODE and CONST segment sizes, or 15,034 bytes.

9.2.3.2 *Module Summary*

The module summary shows how much memory each module (C or assembly language source file, or library function) requires in each of the three types of segment. To generate this summary, be sure the "module summary" box is checked in project options.



Module	CODE	DATA		CONST
	(Re1)	(Re1)	(Abs)	(Re1)
?CHAR_SSWITCH_L10	23			
?CSTARTUP + common	47 2			
?FCMP_GE	72			
?FCMP_LT	77			
?FLOAT_2_SIGNED_LONG	107			
?FLOAT_ADD_SUB	604			
?FLOAT_DIV	376			
?FLOAT_MUL	341			

Figure 9.3 Start of module summary.

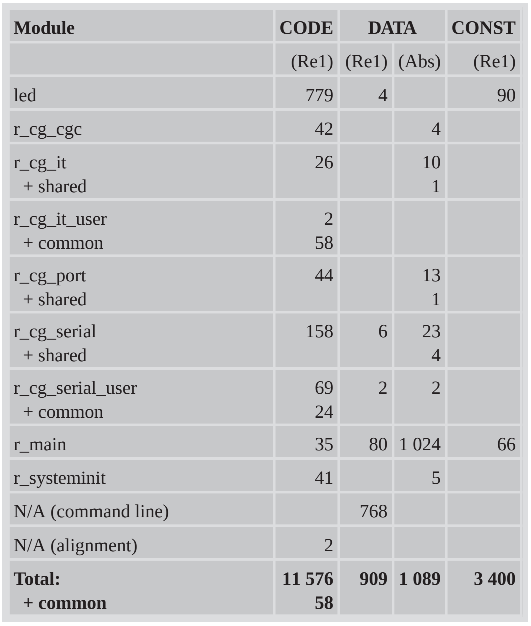
Figure 9.3 shows an example of the beginning of the module summary for a program. All of these modules are library functions which use only code memory. Note that **Rel** indicates that a segment is relocatable (can be moved to a different address), while **Abs** indicates the segment must be located at a fixed, absolute address.

If we look farther down in the module list (see Figure 9.4) we find modules which also use the DATA and CONST segments. For example, r_main uses 80 bytes in the relocatable DATA segment, 1024 bytes in the absolute DATA segment, and 66 bytes in the CONST segment.

9.2.3.3 *Analyzing the Map File*

We now see that the map file gives us the raw information needed to identify the largest modules. We can optimize most effectively if we start with the largest module and then work our way down to smaller modules until we meet our goals. To target RAM use, we sort based on the relocatable DATA segment size. To target ROM use we sort based on the sum of the CODE and CONST segment sizes, as the ROM holds both of those segments.

It is helpful to use automation to sort the modules, starting with the one using the most of the segment of interest. The linker can generate the map file in either text or HTML formats. The text format is helpful when using text-based processing tools to process the map file. The HTML is useful for human interpretation as well as copying into a spreadsheet for program processing.



Module	**CODE**	**DATA**		**CONST**
	(Rel)	(Rel)	(Abs)	(Rel)
led	779	4		90
r_cg_cgc	42		4	
r_cg_it	26		10	
+ shared			1	
r_cg_it_user	2			
+ common	58			
r_cg_port	44		13	
+ shared			1	
r_cg_serial	158	6	23	
+ shared			4	
r_cg_serial_user	69	2	2	
+ common	24			
r_main	35	80	1 024	66
r_systeminit	41		5	
N/A (command line)		768		
N/A (alignment)	2			
Total:	**11 576**	**909**	**1 089**	**3 400**
+ common	**58**			

Figure 9.4 End of module summary.

9.3 OPTIMIZING DATA MEMORY

How can we reduce or otherwise improve a task's use of data memory? The obvious coding practice of **using the smallest adequate data type** improves data size, code speed, and code size, so it is worth following. In this section we explore additional methods.

9.3.1 Using Toolchain Support

Be sure to select the **smallest memory models** into which the program will fit. Typically code and data memory models can be specified separately. Using a larger memory model will force the compiler to generate longer and slower code in order to handle longer addresses (e.g., when accessing static variables, using pointers, and calling subroutines).

Compilers typically allow the user to specify the **optimization effort level** (e.g., level 3) as well as the **goal** of the optimization (e.g., speed, size, or possibly a balanced approach to both). These options should be used as appropriate.

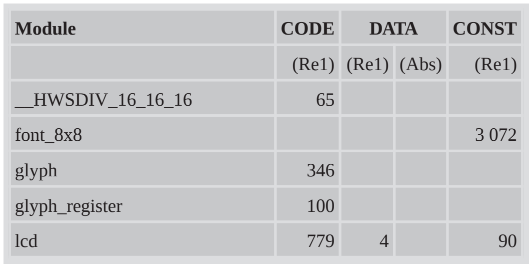
Note that it is often possible to use **different** optimization settings for a specific module. With EW, we can override the default project settings for a module. This allows us to use a finer-grain approach to optimization. For example, we could optimize the large modules for size, and optimize the rest for speed.

The compiler aligns the elements of a structure based on the target architecture's word size and/or the size of the largest element. Space for padding is added to align shorter elements, wasting RAM. This padding can be eliminated by packing the data structure (e.g. with the directive #pragma packed). The resulting code uses less RAM but requires additional instructions to perform packing and unpacking. This concept can be applied to integer elements of non-standard sizes by using C's bitfield width specifier to indicate the number of bits required to store an element.

9.3.2 Leveraging Read-Only Data

Some data in a program is only read and never written. This data can be stored in the ROM in order to free up the RAM. The **const** type modifier for a variable directs the compiler to place the variable in read-only memory rather than RAM.

For example, consider the Glyph graphics code, which contains font bitmaps for rendering text on a graphical LCD. The bitmaps are declared as initialized arrays, and therefore are allocated space in two places in memory. The DATA segment in RAM holds the values which the program accesses. The CONST segment in ROM holds the initial array data values; the C start-up code copies these values into the DATA segment in RAM to initialize them. These bitmaps are read-only, and do not need to be stored in RAM. Instead only need to be stored in ROM. Figure 9.5 shows that moving a bitmap (font_8x8) into ROM from RAM freed up 3072 bytes of valuable RAM.



Module	**CODE**	**DATA**		**CONST**
	(Re1)	(Re1)	(Abs)	(Re1)
__HWSDIV_16_16_16	65			
font_8x8				3 072
glyph	346			
glyph_register	100			
lcd	779	4		90

Figure 9.5 Memory requirements from map file.

9.3.3 Improving Stack Memory Size Estimation

We may be able to **reduce the amount of space allocated** for the stack if we have a **more accurate estimate** of the worst-case size. The call stack is a dynamic structure which grows and shrinks as the program executes. We need to allocate enough space to handle its worst-case (largest) size. We added a margin of error to our initial stack space allocation because we were not confident of its accuracy. We can reduce that margin by improving the accuracy of the estimate.

The amount of space allocated for the stack is defined in Project Options -> General Options -> Stack. The default size (e.g., 128 bytes) is likely to be too small for many programs and will need to be increased.

9.3.3.1 Analytical Stack Size Bounding

Because the call graph shows the nesting of function calls it is a good starting point to evaluate the maximum stack size. We can calculate space required by examining the stack depth at each leaf node.[1] The stack space required at a given node *N* in the call graph is the sum of the size of each activation record on a path beginning at graph's root node (main) and ending at node *N*, including both nodes. Note that the activation record size within a function can vary. For example, a function may push arguments onto the stack before calling a subroutine, which will increase the activation record size. Because of this reason we need to consider the activation record size for a function at each point with a subroutine call.

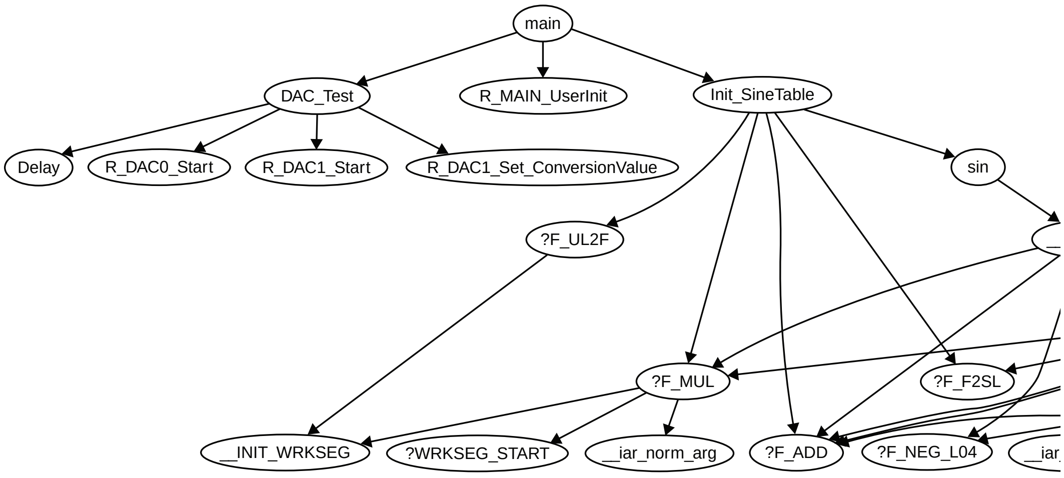


Figure 9.6 Detail of call graph with function Init_SineTable.

IAR Embedded Workbench provides some help. Each module's assembly listing shows the maximum stack use per function. Note that these figures **do not include the return address** (four bytes) which is pushed onto the stack by the call instruction. Those four bytes will need to be added. Figure 9.7 shows various examples:

- The function Delay uses two bytes and does not call any other functions (it is a leaf function).

[1] In a call graph, a leaf node does not call any subroutines.

- The function Main uses zero bytes of stack space and calls three functions as subroutines (DAC_Test, Init_SineTable and R_MAIN_UserInit).
- Init_SineTable uses a maximum of 20 bytes, but when it calls sin it is only using 12 bytes.
- The activation record size listed when calling a function will include the space used for any parameters which have been pushed onto the stack.

```
Maximum stack usage in bytes:

CSTACK Function
------ --------
  12   DAC_Test
    10   -> Delay
    10   -> R_DAC0_Start
    10   -> R_DAC1_Set_ConversionValue
    10   -> R_DAC1_Start
   2   Delay
  20   Init_SineTable
    12   -> sin
   0   R_MAIN_UserInit
   0   main
     0   -> DAC_Test
     0   -> Init_SineTable
     0   -> R_MAIN_UserInit
```

Figure 9.7 Example of stack usage information included in each assembly language listing file generated by compiler.

We can start to evaluate the total stack use at each node (representing a function) on the call graph shown in Figure 9.6. The total stack use at function f is the sum of the stack use of each function on the path starting with the root (main) and ending with the node representing function f.

- Main uses zero bytes. It was called as a subroutine by the reset ISR, so stack depth is four bytes here (due to the return address).
- DAC _Test uses a maximum of 12 bytes. The maximum stack depth at this point in the call graph is the sum of the stack depth at the calling function (four bytes at main), DAC_Test's activation record (12 bytes), and the return address (four bytes), for a total of 20 bytes.
- Delay uses a maximum of two bytes. The maximum stack depth at this point is 20 bytes + two bytes + four bytes = 26 bytes.

We now continue this process for the remainder of the call graph. However, soon we reach a problem: how much stack space is needed by the library functions, such as ?F_UL2F, _INIT_WRKSEG, ?F_MUL, _WRKSEG_START, and ?F_ADD? We need to analyze the code for these libraries to determine the stack space required. Source code would be easiest to analyze, but it is possible (though tedious) to use the disassembler to examine the object code. Some vendors may include information on stack space requirements for library functions in order to simplify stack depth analysis.

9.3.3.2 Experimental Measurement

One approach to estimating the maximum stack size is to execute the code and measure how much stack space has been used. Before program execution the stack space should be initialized with a known pattern. The program is then run for some time with realistic test input data and conditions. The stack space can then be examined to determine how much of the initial pattern has been overwritten. This is sometimes called examining the "high-water mark." Some debuggers (including C-Spy) provide a graphical indication of current stack use.

This approach tells us only how much stack space was used for **one specific run** of the program. It does not indicate the maximum possible stack use. As a result, this is an empirical estimate with limited confidence. To improve the confidence in the estimate we should repeat the test with a wide range of input data and conditions, monitoring the variation and limits of observed stack use. We may also want to evaluate the code coverage of our test cases to ensure that most or all of the code is being executed.

After running enough experiments the maximum observed stack space should settle down to a fixed value. We add a margin of safety (e.g., 20%) to this observed value in order to determine the amount of stack space to allocate.

It is also possible to build this type of measurement mechanism into software which executes as the program runs, allowing live monitoring of stack use and potentially detecting a stack overflow condition. A lighter-weight approach is to sample the stack pointer value periodically to record its maximum value, or to confirm that the current stack pointer value is within a valid range of stack addresses.

9.3.4 Reducing the Size of Activation Records

Functions require stack space for several purposes, but there are two in particular to look out for when trying to reduce stack space use.

First, **automatic variables** are allocated memory space in the stack frame of the declaring function. Automatic variables are more space-efficient than static variables because their memory space can be reused after the function exits. Static variables occupy memory space for the entire duration of the program.

Sometimes the compiler can reuse stack space for automatic variables within a function if their live ranges do not overlap. The compiler may even promote automatic variables to registers and eliminate their use of the stack. However, some variables are not promoted to registers. The most important from the point of view of stack space optimization is variables which are simply too large to be promoted to registers.

We see in Figure 9.8 that the function LCDPrintf (from the Glyph graphics library included with the RL78G14) uses 108 bytes on the stack. If we examine the source code (shown below) we see that an automatic variable called buffer is declared as an array of 100 characters, so it is placed on the stack. Does the buffer really need to be 100 characters long? Can we even fit 100 characters onto the LCD, given its dimensions of 96 by 64 pixels? It helps to determine how large an array really needs to be, and then add a small safety margin. In addition, add bounds-checking code to detect errors.

```
Maximum stack usage in bytes:

CSTACK Function
------ --------
   2   LCDCenter
   4   LCDChar
     4   -> GlyphChar
     4   -> GlyphSetXY
   4   LCDCharPos
     4   -> GlyphChar
     4   -> GlyphSetXY
   0   LCDClear
     0   -> GlyphClearScreen
   8   LCDClearLine
     8   -> GlyphEraseBlock
   2   LCDFont
     2   -> GlyphSetFont
   2   LCDInit
     2   -> GlyphClearScreen
     2   -> GlyphNormalScreen
     2   -> GlyphOpen
   0   LCDInvert
     0   -> GlyphInvertScreen
 108   LCDPrintf
   108   -> LCDStringPos
   108   -> vsprintf
```

Figure 9.8 Stack usage for functions in object module LCD.

```
void  LCDPrintf(uint8_t aLine, uint8_t aPos, char *aFormat, ...)
{
   uint8_t  y;
   char buffer[100];
   va_list marker;

   ... (deleted)
}
```

Second, **arguments and return values** are passed on the stack if they cannot be passed in registers (which would be faster and use less memory).

TABLE 9.2 Locations of Function Arguments.

ARGUMENT SIZE	REGISTERS USED FOR PASSING
8 bits	A, B, C, X, D, E
16 bits	AX, BC, DE
24 bits	stack
32 bits	BC:AX
Larger	stack

The compiler allocates arguments to registers (shown in Table 9.2) by traversing the argument list in the source code from left to right. When it runs out of registers it starts passing arguments on the stack. The stack pointer points to the first stack argument and the remaining arguments are at higher addresses. All objects on the stack are word-aligned (the address is a multiple of two), so single byte objects will take up two bytes. Some arguments are always passed on the stack:

- Objects larger than 32 bits
- Structures, unions and classes (except for those 1, 2 or 4 bytes long)
- Functions with unnamed arguments

Return values up to 32 bits long are returned in registers (A, AX, A:HL, or BC:AX), while longer values are returned on the stack.

Consider the following code which generates a formatted string using sprintf:

```
sprintf(buffer, "$APRMC,%02d%02d%02d,A,%02d%06.3f,N,%03d%06.3f,"
   "W,%05.1f,%04.1f,%06ld,%05.1f,W*", hr, min, sec, lat_deg, lat_min,
   lon_deg, lon_min, speed, track, date, var);
```

Let's examine the object code listing generated by the compiler. We select maximum optimization (level 3) for code size.

```
CMP0      N:DifferentTalker
BZ        ??sim_motion_14
PUSH      BC
PUSH      AX
MOVW      AX, [SP + 0x2E]
MOVW      BC, AX
MOVW      AX, [SP + 0x2C]
PUSH      BC
PUSH      AX
```

```
10. MOVW      AX, [SP + 0x2E]
11. MOVW      BC, AX
12. MOVW      AX, [SP + 0x2C]
13. PUSH      BC
14. PUSH      AX
15. MOVW      AX, [SP + 0x2E]
16. MOVW      BC, AX
17. MOVW      AX, [SP + 0x2C]
18. PUSH      BC
19. PUSH      AX
20. MOVW      AX, [SP + 0x28]
21. MOVW      BC, AX
22. MOVW      AX, [SP + 0x26]
23. PUSH      BC
24. PUSH      AX
25. MOVW      AX, [SP + 0x1C]
26. PUSH      AX
27. MOVW      AX, [SP + 0x2A]
28. MOVW      BC, AX
29. MOVW      AX, [SP + 0x28]
30. PUSH      BC
31. PUSH      AX
32. MOVW      AX, [SP + 0x24]
33. PUSH      AX
34. PUSH      DE
35. MOVW      AX, [SP + 0x3A]
36. PUSH      AX
37. MOVW      AX, [SP + 0x3A]
38. PUSH      AX
39. MOVW      BC, #`?<Constant "$APRMC,%02d%02d%02d,A ...">`
40. MOVW      AX, [SP + 0x2E]
41. CALL      sprintf
42. ADDW      SP,#0x22
```

The object code listed above uses 58 bytes of code space. The reason so much code is generated is that numerous parameters (13) are being passed to sprintf. At first the compiler uses registers for parameter passing: the buffer pointer is in AX, and the format string pointer is in BC. However, after that point it runs out of registers, so each remaining parameter must be passed on the stack. The parameter's value must first be loaded from the variable in the stack frame (e.g., instructions 5 through 7) and then it can be pushed onto the stack (e.g., instructions 8 and 9).

9.3.5 Use Stack-Friendly Functions

Some functions such as printf and scanf use large amounts of stack space in order to support a rich range of formatting options. It is often possible to use functions which are less powerful but less stack-hungry, such as ftoa, itoa, atof, and atoi.

9.4 REDUCING CODE MEMORY

9.4.1 Language Support

We can help the compiler delete code for functions which will never be called. Using the **static** keyword to modify a function declaration (e.g., in file.c) will indicate to the compiler and linker that the function will not be called by any function outside of file. c. As a result, if no function inside file.c calls the function either, then the linker can delete that function's object code from the module.

9.4.2 Compiler and Toolchain Configuration

The compiler should be configured to generate code for the particular type of MCU rather than a more generic target which might lack some instructions or hardware accelerators. For example, the RL78G14 family of MCUs uses a core with support for various multiply, divide, and multiply/accumulate instructions operating on several data lengths. The RL78G13 family of MCUs uses an older core which supports only a multiply instruction (8 bit * 8 bit). Programs compiled for the older core will call run-time library functions to perform the operations which are not supported directly as instructions.

Some toolchains (including IAR EW) offer multiple versions of library functions which reduce code size by eliminating features which are not needed. There may be various versions of libraries based on desired features or accuracy.

- Different versions of the studio functions **printf** and **scanf** may offer subsets of all possible formatting options (excluding formatting floating-point values, eliminating field width specifiers, etc.).
- Different versions of the **floating-point math library** may allow the user to reduce the size of the code at the expense of less precision and a smaller input range.
- The **C run-time library** may allow omission of features such as multibyte support, locale, and file descriptors.
- Some libraries may support the use of hardware accelerators (e.g., multiply/divide unit).

9.4.3 Removing Similar or Identical Code

If large amounts of source code are duplicated, or are very similar, then there may be an opportunity for size optimization. There are various compilers optimizations which try to remove common code from a program. One approach is to create a subroutine out of the common code and then replace the duplicated code with calls to the subroutine. This is called function "out-lining" (as opposed to "in-lining"), or procedural abstraction. A related approach is to move the common code within the function and remove the duplicates.

Sometimes the compiler is not able to apply this type of optimization, but the programmer may be able to do so and improve code size.

9.4.3.1 Cut-and-Paste Source Code

Consider the code in the listing below.[2] It is used to generate test messages in the NMEA-0183 format but with various errors. It is an excerpt from a larger section of code with 17 test cases.

```
if(DifferentTalker)    // Error in talker ID - not GPS
   sprintf(buffer, "$APRMC,%02d%02d%02d,A,%02d%06.3f,N,%03d%06.3f,"
   "W,%05.1f,%04.1f,%06ld,%05.1f,W*", hr, min, sec, lat_deg, lat_min,
   lon_deg, lon_min, speed, track, date, var);
else if(DifferentSenType)    // Error in sentence type - not GLL
   sprintf(buffer, "$GPGLC,%02d%02d%02d,A,%02d%06.3f,N,%03d%06.3f,"
   "W,%05.1f,%04.1f,%06ld,%05.1f,W*", hr, min, sec, lat_deg, lat_min,
   lon_deg, lon_min, speed, track, date, var);
else if(IllegalInField)    // letter in field
   sprintf(buffer, "$GPRMC,%02d%02d%02d,A,%02d%06.3fa,N, %03d%06.3f,"
   "W,%05.1f,%04.1f,%06ld,%05.1f,W*", hr, min, sec, lat_deg, lat_min,
   lon_deg, lon_min, speed, track, date, var);
else if(IllegalAsField)    // Illegal separator
   sprintf(buffer, "$GPRMC;%02d%02d%02d;A;%02d%06.3f;N;%03d%06.3f;"
   "W;%05.1f;%04.1f;%06ld;%05.1f;W*", hr, min, sec, lat_deg, lat_min,
   lon_deg, lon_min, speed, track, date, var);
```

[2] Note that the C compiler concatenates sequential string literals: "abc" "def" is processed as "abcdef" allowing long string literals to be broken across multiple lines of source code.

Yes, it looks like lazy coding, but is that so bad? Writing the code was fast—cut and paste, and then modify the format string for each test case.

There are two major drawbacks to this approach. First, **code maintenance** will be more **difficult** if we need to modify each case to fix a common issue. Second, the **code size** will be much larger than necessary. Each test case uses about 58 bytes of object code, as discussed previously. There are a total of 17 test cases consuming about 986 bytes. This copy-and-paste coding style does indeed have a negative impact on memory requirements.

9.4.3.2 Improving the Source Code with an Array

It is often possible to identify common code and remove duplicates. The compiler was able to identify that certain operations (e.g., CALL sprintf) appear in each test case, and moved that code out into a common basic block, saving a little space (about seven bytes per case).

We can take this idea further by identifying the common cases and then using code-space efficient mechanisms such as arrays and loops built for the parameterized solution shown below:

```
char Formats[17][] = {
   "$APRMC,%02d%02d%02d,A,%02d%06.3f,N,%03d%06.3f,W,%05.1f,%04.1f,"
   "%06ld,%05.1f,W*",
   "$GPGLC,%02d%02d%02d,A,%02d%06.3f,N,%03d%06.3f,W,%05.1f,%04.1f,"
   "%06ld,%05.1f,W*",
   "$GPRMC,%02d%02d%02d,A,%02d%06.3fa,N,%03d%06.3f,W,%05.1f,%04.1f,"
   "%06ld,%05.1f,W*",
   "$GPRMC;%02d%02d%02d;A;%02d%06.3f;N;%03d%06.3f;W;%05.1f;%04.1f;"
   "%06ld;%05.1f;W*", ... (deleted) ...
};

if(DifferentTalker)
   format_num = 0;   //error in talker id - not gps
else if(DifferentSenType)
   format_num = 1;   //error in sentence type - not gll
else if(IllegalInField)
   format_num = 2;   //letter in field
else if(IllegalAsField)
   format_num = 3;   //illegal separator
   ...
sprintf(buffer, Formats[format_num], hr, min, sec, lat_deg,
   lat_min, lon_deg, lon_min, speed, track, date, var);
...
```

We could even improve this code further by deleting the if/else chain. One approach would be to compute the index into the array based on the conditions which are tested. Another would be to merge these error codes into one integer variable in order to allow direct indexing.

9.4.3.3 Tables of Function Pointers

Arrays can hold more than just data. An **array of function pointers** is effective for quickly selecting which code to execute in response to an input value. Embedded software may need to process received messages based upon their type. Rather than perform a sequence of comparisons, or use a switch statement, it may be more practical to use a function pointer table to execute the correct processing code.

9.5 OPTIMIZATION FOR MULTITASKING SYSTEMS

As discussed previously, supporting task-level preemption usually requires **one call stack per task.**[3] The function call stack holds a function's state information such as return address and limited lifetime variables (e.g., automatic variables, which only last for the duration of a function). Without task preemption, task execution does not overlap in time, so all tasks can share the same stack. Preemption allows tasks to preempt each other at essentially any point in time. Trying to reuse the same stack space for different tasks would lead to corruption of this information on the stack and system failure.

9.5.1 Use a Non-Preemptive Scheduler

For some systems, it may be possible to use a non-preemptive scheduler rather than one which is preemptive. A non-preemptive scheduler requires **only one call stack,** and **shares this stack space** over time with the different tasks as they execute sequentially. Only the **largest task stack** needs to fit into RAM. However, a preemptive scheduler requires **one call stack for each task** because preemptions could occur at any point in a task's execution (i.e., any point within that task's call graph).[4] Much of the task's state is stored on the stack, so that must be preserved and not used by other tasks. As a result, a preemptive system needs enough RAM to hold **all task stacks simultaneously,** with each potentially at its largest.

[3] There are several ways to reduce the number of stacks needed for preemptive scheduling. For example, see the description of the Stack Resource Policy in Chapter 3.

[4] If the tasks are all written to run to completion and never block, it may be possible to use a single stack with a preemptive scheduler.

9.5.2 Improve the Accuracy of Stack Depth Estimates

A system with preemptive scheduling is more sensitive to the effects of stack size overestimation because of the larger number of call stacks. For these systems it may be worth investing the time in developing support for bounding maximum stack depth more accurately.

9.5.3 Combining Tasks to Reduce Stack Count

In some systems it may be possible to reduce the number of task stacks required by merging tasks. If there are multiple independent tasks which run to completion and do not have tight timing requirements, they may be combined into a single task consisting of the original tasks as subtasks. The task acts as a state machine controller or non-preemptive scheduler and runs one of the subtasks each time it executes. Note that if any of the subtasks blocks, all other subtasks will block as well, impacting system responsiveness and perhaps even introducing deadlock. This approach can reduce stack space use significantly in some systems.

9.6 RECAP

This chapter focused on analyzing a program’s memory use (for code and data) and then presented methods for reducing it. We examined methods to determine a program's memory requirements with the goal of targeting the largest modules first. We then examined how to improve data memory use and code memory use. For both we discussed language features, toolchain support, and better coding styles. Finally we discussed methods to reduce stack memory requirements for multitasking systems.

Index