AN3007

Getting Started with FreeRTOS on megaAVR® 0-series

Features

- Configuration and Basic Features of FreeRTOS™
- Debugging and Typical Errors
- Atmel | START Example Code

Introduction

Author: Eira Mørch-Thoresen, Microchip Technology Inc.

FreeRTOS is a real-time operating system kernel for embedded devices. It is designed to be small and simple, and thus, does only consist of a few files written mostly in C.

Microcontrollers are often used for real-time embedded applications, meaning that the embedded system must be able to respond to certain events within a strictly defined amount of time. To ensure that the system meets these deadlines, the RTOS has a scheduler that decides which task to run at any instance of time.

The FreeRTOS provides features for tasks, task communication, and scheduling, and has become the de facto standard real-time operating system (RTOS) for microcontrollers. The primary design goals of FreeRTOS are robustness, ease of use and a small footprint.

This document starts by describing how FreeRTOS can be configured and then goes on to explain blocking functions, inter-task communication schemes, and scheduling. A section about debugging is included as well, before the section about the demo code. The application note also provides UML diagrams for each of the tasks in the demo.

© 2019 Microchip Technology Inc.

Table of Contents

1. Relevant Devices

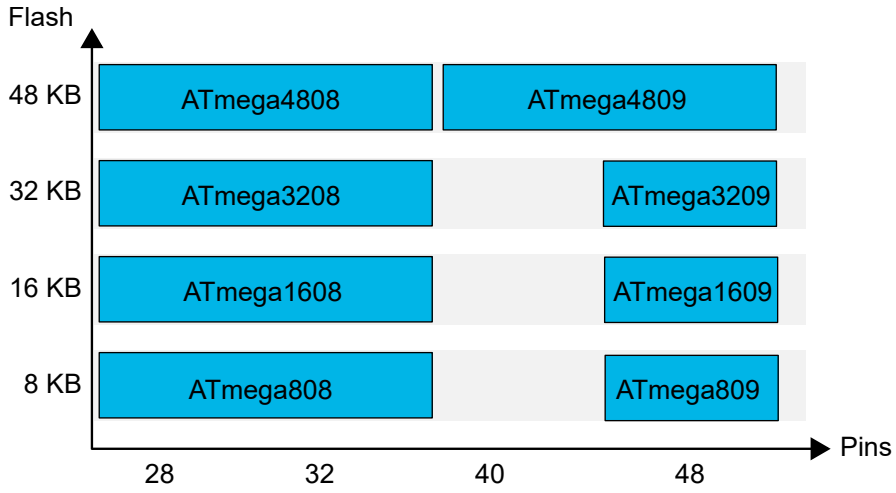
This chapter lists the relevant devices for this document.

1.1 megaAVR® 0-series

The figure below shows the megaAVR® 0-series devices, laying out pin count variants and memory sizes:

- Vertical migration is possible without code modification, as these devices are fully pin and feature compatible.
- Horizontal migration to the left reduces the pin count and, therefore, the available features.

Figure 1-1. megaAVR® 0-series Overview



Devices with different Flash memory sizes typically also have different SRAM and EEPROM.

2. Starting from Atmel | START

The easiest way to create a FreeRTOS application is to download an example from Atmel | START and work from there. This ensures that all FreeRTOS files will be included and working correctly. The demo code can then be removed, and the application development can begin.

In this application note, the following Atmel | START example project is used: ATmega4809 FreeRTOS Example.

3. Configuring FreeRTOS

FreeRTOS uses a configuration file called “FreeRTOSConfig.h”. By altering this, the FreeRTOS application can be customized to behave in the desired manner. The configuration file must be included in the pre-processor path.

Typical configurations to alter are the memory size, stack options, as well as clock and tick rates. Additionally, priorities and preemption of tasks can be configured. The configuration file contains a lot of options, and this application note does only cover a subset of the available options. For a full overview, refer to https://www.freertos.org/a00110.html.

3.1 Configure Clock and Tick Rate

A real-time operating system (RTOS) uses a system tick which is the time unit the timers are based on. FreeRTOS uses the microcontroller's TCB0 timer to generate its own tick interrupt. The FreeRTOS kernel measures the time using the tick, and every time a tick occurs, the scheduler checks if a task should be woken up or unblocked.

The `configCPU_CLOCK_HZ` define must be configured for the FreeRTOS timings to be correct. The frequency of the TCB0 clock should be entered here. The TCB0 will by default operate at the same clock frequency as the CPU for the devices listed in the “Relevant Devices” section.

The `configTICK_RATE_HZ` is used to determine the frequency of the RTOS tick interrupt. The tick rate should not be set too high compared to the CPU frequency as there will be some CPU overhead every time the tick occurs.

If more than one task has the same priority, the RTOS scheduler will switch between these tasks at every tick if `configUSE_TIME_SLICING` is set to 1. Using a higher tick rate will, therefore, cause the CPU time for each task to be smaller. For example, if a tick rate of 1000 Hz is used, the time slice for each task will be T=1/f=1/1000 Hz, which is 1 ms.

3.2 Configuring Memory

FreeRTOS has two memory allocation schemes: Static and dynamic allocation. Static memory allocation requires the application itself to allocate and deallocate memory. This can be more difficult to implement but provides more control. When using the dynamic allocation scheme, the memory allocation occurs automatically within the API functions. The application only has to call the create and delete functions. To use dynamic allocation, set the `configSUPPORT_DYNAMIC_ALLOCATION` define in the configuration file to 1.

With dynamic allocation, FreeRTOS needs to have a portion of RAM allocated for an area called the heap. FreeRTOS functions ending with “Create” allocates memory on the heap. When creating a task by calling `xTaskCreate()`, the memory will be allocated on the heap for that particular task consisting of a stack and a task control block (TCB). The task stack size is defined when creating the task. Queues, mutexes, and semaphores will also allocate heap memory when created. For tips on defining the right stack size, see the “Debugging in FreeRTOS” section. Refer to the figure below for an illustration of how the FreeRTOS memory works.

When the dynamic allocation is used, `configTOTAL_HEAP_SIZE` needs to be sufficiently large. In other words, it must be large enough to fit all the task stacks and other allocated memory. In the demo example, the heap size is set to 0x800 (2048 bytes). To optimize the heap size, `xPortGetFreeHeapSize()` can be used. This function returns the amount of unallocated heap. See the “Debugging in FreeRTOS” section for tips on how to find the correct heap size.

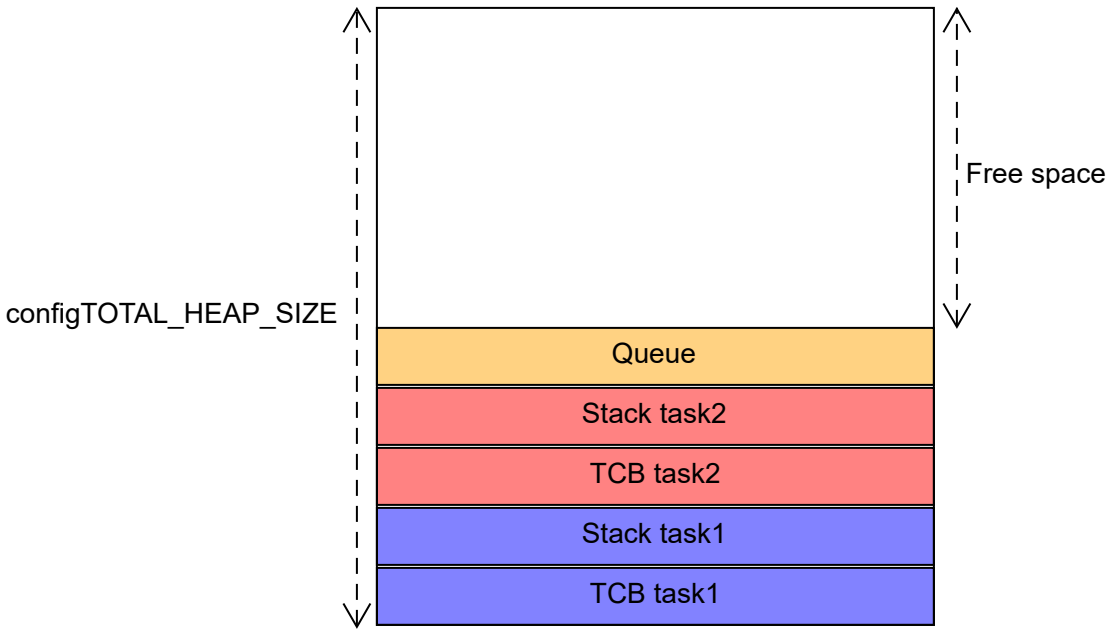
There are five different heap implementations, named heap 1-5, available in FreeRTOS. To choose which heap allocation to use, change the `USE_HEAP` define in the `heap.c` file found in the `freeRTOS/portable/MemMang` folder.

The Atmel | START example uses the heap 1 implementation. Heap 1 is similar to static allocation in the way that once the memory is taken, it cannot be freed or reallocated. Heap 1 is easy to debug but requires that tasks and other FreeRTOS objects such as queues, semaphores, and mutexes are kept on the heap throughout the life of the application, as creating and destroying these objects will make the application run out of memory.

The heap 2 scheme does, unlike heap 1, allow memory to be freed, but should be used with care if the memory allocated is of random size because it may cause the available free memory to become fragmented, which can result in allocation failures.

Heap 3, 4 and 5 are more advanced schemes. For more information, refer to https://www.freertos.org/a00111.html.

Figure 3-1. FreeRTOS Memory



3.2.1 Configuring the Stack

In FreeRTOS, each task has its own, separate stack. The task stacks are allocated on the heap when using dynamic memory allocation. Except for the idle task, the stacks are allocated when calling `xTaskCreate()`, where one of the input parameters is the stack size for the task.

The idle task is created when calling `vTaskStartScheduler()` to ensure that there is always at least one task that is able to run. The idle task's stack size is given by `configMINIMAL_STACK_SIZE` in the configuration file.

It is important that the sum of allocated stack memory, as well as allocated memory from queues, semaphores, etc., does not exceed the heap size defined in `configTOTAL_HEAP_SIZE`.

4. Thinking Like an RTOS Developer

Real-time programming and its strict time constraints change the way a developer thinks. Instead of using the typical state machine implementation, the application is partitioned into multiple tasks. One task satisfies a portion of the application, whereas another task has other responsibilities. For instance, in the Atmel | START example, there is one task that blinks the LEDs, one task that writes the time to an OLED screen, and several other tasks with different responsibilities. In a real-time system, these tasks run concurrently, and a portion of their code will be executed every time the scheduler allows them to. This is different from the typical linear application code where everything is evaluated sequentially and always in the same order.

If there are tasks that are more important than others and require faster response time, the RTOS can be configured to allow higher priority tasks to interrupt lower priority ones. This is called preemption.

For the application to run correctly, it is important to make sure that the tasks do not interrupt each other at times where it can cause harm. An example of this might be if a task reads the value of a shared resource before another task is finished doing computations on it. Then the read value might be wrong, which again might cause other problems. It is important to know how to use task communication to ensure mutual exclusion to prevent such errors. This is discussed in the Task Communication section.

The figures below outlines a simple program sequence for homebrewing and what a transition from linear programming to parallel or concurrent tasks would look like.

Figure 4-1. Linear Programming

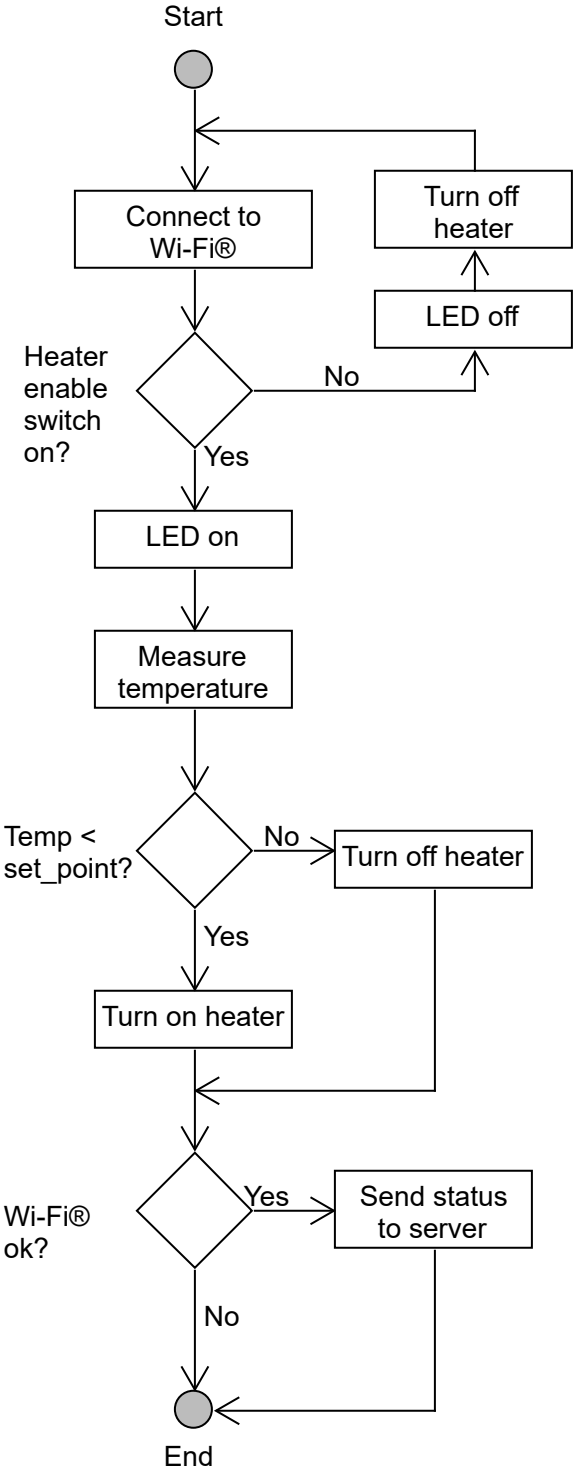
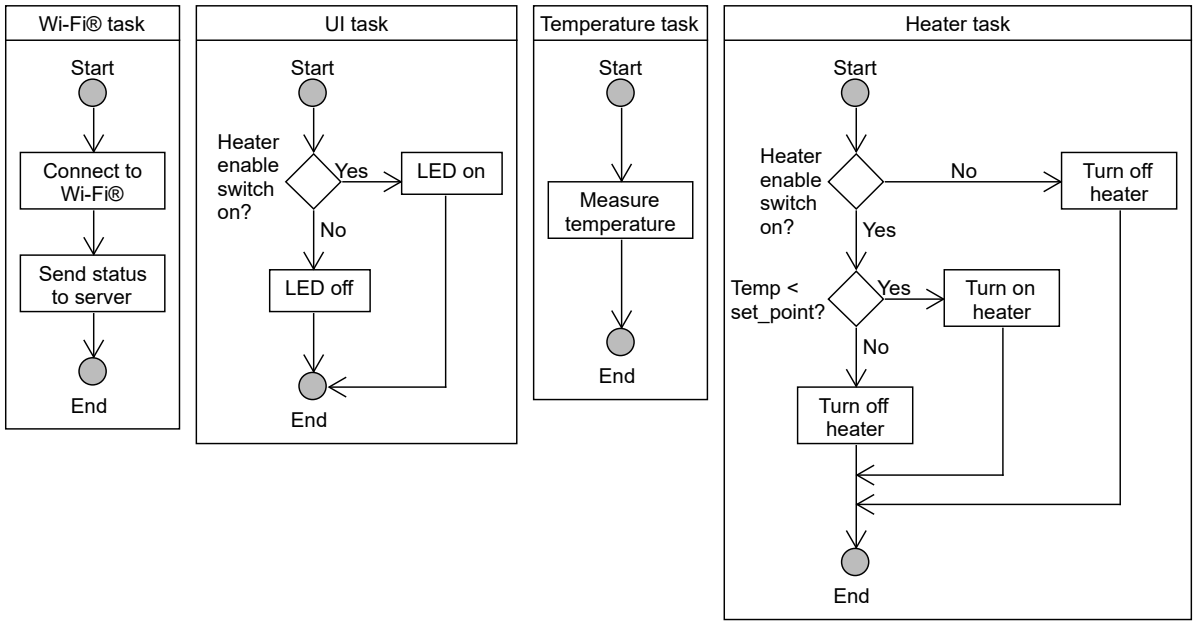


Figure 4-2. Task Driven Code



4.1 Tasks

A task is created by calling the function `xTaskCreate(taskName, pcName, stackDepth, pvParameters, priority, pxCreatedTask)`. Tasks are typically implemented as an infinite loop. See the example below on how to implement a task.

```
xTaskCreate(exampleTask, "example", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

void exampleTask(void *pvParams){
    while(1){
        //do task work
    }
}
```

4.2 Blocking Versus Non-Blocking Functions

A blocking function blocks a task from continued execution. When a task gets blocked, the RTOS will switch execution to a different task. This optimizes the CPU utilization because there will never be cycles where the CPU is doing nothing.

Typically tasks will block when they call a delay function, when they are waiting for communication or resources used by other tasks, or when an interrupt (ISR) is called. When calling `vTaskDelay(ticksToDelay)`, the task will block for as many ticks as specified. The actual time the task will block depends on the tick rate which is specified in the configuration file.

When tasks share resources, they must wait for each other to finish using the resource before they can start using it themselves. Waiting for a resource like this is also called blocking. In order to create such behavior, semaphores and mutexes can be used. A more thorough explanation of shared resources and synchronization is found in the next section.

4.3 Task Communication

FreeRTOS offers five primary inter-task communication mechanisms: queues, semaphores, mutexes, stream buffers, and message buffers. Common for all of these is that they can cause tasks to block if the resource or data is not available. For example, if a task wants to insert an element into a full queue, it will end up waiting (blocking) for some time, and then again check if there is any available space in the queue.

4.3.1 Queue

Queues offer inter-task communication of a user-definable fixed length. The developer specifies the message length when creating the queue. This is done by calling `QueueHandle_t queueName = xQueueCreate(queueLength, elementSize)`. The input parameter `queueLength` specifies the number of elements the queue can hold. `elementSize` specifies the size of each element in bytes. All elements in the queue must be of equal size. When the queue is created, tasks can communicate with each other by sending and receiving data through the queue. The queue is of FIFO structure (first in/first out) such that the receiver will always receive the item that was first inserted.

The functions for sending to and receiving from a queue are:

```
xQueueSend(queueName, *itemToQueue, ticksToWait)
xQueueReceive(queueName, *buffer, ticksToWait)
```

The last argument, `ticksToWait`, specifies how long the sending task should block when waiting for available space in a full queue. Similarly, it specifies how long a receiving task will block when waiting to receive any elements from an empty queue. The define `portMAX_DELAY` can be used here. If `INCLUDE_vTaskSuspend` in the configuration file is set to 1, then `portMAX_DELAY` will cause the task to block indefinitely (without a time-out). If set to 0, the blocking time will be 0xFFFF.

If the send and receive functions are to be called form an ISR, `xQueueSendFromISR()` and `xQueueReceiveFromISR()` must be used.

4.3.2 Semaphore

Semaphores are used for synchronization and to control access to shared resources between tasks. A semaphore can either be binary or counting and is essentially just a non-negative integer count.

A binary semaphore is initialized to 1 and can be used to guard a resource that can only be handled by one task at a time. When a task takes the resource, the semaphore is decremented to 0. If another task then wants to use the resource and sees that the semaphore is 0, it blocks. When the first task is finished using the resource, the semaphore is incremented and is thus available to other tasks. A binary semaphore can be created with `SemaphoreHandle_t semaphoreName = xSemaphoreCreateBinary(void)`.

A counting semaphore works in the same manner, but for resources that can be used by multiple tasks at once. For example, if you have a parking garage with room for 10 cars, you can allow 10 semaphore access. Every time a car enters, the semaphore will be decremented by 1 until it reaches 0 and no one is allowed entrance before someone leaves. A counting semaphore should be initialized to the number of tasks that can have concurrent access to the resource and is created with `SemaphoreHandle_t semaphoreName = xSemaphoreCreateCounting(maxCount, initialCount)`.

When a task wants a resource protected by a semaphore, it calls the function `xSemaphoreTake(semaphoreName, ticksToWait)`. If the semaphore evaluates to 0, the task will block for the time specified in `ticksToWait`. When a task is finished using the semaphore, the function `xSemaphoreGive(semaphoreName)` is called.

4.3.3 Mutex

A mutex is a lot like a binary semaphore, but in addition, it provides a priority inheritance mechanism. If a high priority task gets blocked from accessing a resource that is already taken by a lower priority task, the lower priority task will inherit the priority of the high priority task until it has released the mutex. This ensures that the blocking time of the high priority task is minimized since the low priority task now cannot be preempted by other medium priority tasks.

A mutex is created by using the `semaphoreHandle_t mutexName = xSemaphoreCreateMutex(void)` function.

Mutexes should not be used from an interrupt because the priority inheritance mechanism only makes sense for a task, and not for an interrupt.

4.3.4 Stream Buffer

Stream buffers transfer data between two tasks, or from an ISR to a task. Unlike queues, the stream buffer assumes that there is only one reader and only one writer. When creating a stream buffer, the maximum size of the buffer is specified. As the name implies, a stream buffer allows a stream of bytes to be transferred between a writer and a reader. The byte stream can be of an arbitrary length as long as it is within the size of the buffer.

A stream buffer is created using the function `xStreamBufferCreate(BufferSizeBytes, TriggerLevelBytes)`. The first input argument specifies the total number of bytes the buffer is able to hold. The second argument, the trigger level, specifies the number of bytes that must be in the buffer before a blocked receiver is moved out of the blocked state.

The sender may block if the buffer is full. How long the sender task should block while waiting for data is given by a time-out that is set in the send function. Similarly, a receiver will block if the buffer is empty.

Unlike a queue, the sender is not required to have the complete message ready before putting it in the buffer.

The send and receive functions are shown below.

```
xStreamBufferSend(streamBuffer, *pvTxData, dataLengthBytes, ticksToWait);
xStreamBufferReceive(streamBuffer, *pvRxData, bufferLengthBytes, ticksToWait);
```

4.3.5 Message Buffer

Message buffers work a lot like stream buffers, but instead of having the receiver request a certain amount of data, the first part of the message specifies the message length. Additionally, a message can only be read out as the length specified, and not as individual bytes.

Message buffers are implemented using stream buffers, so the assumption of only one reader and one writer applies here as well. The message buffer also works in the same way as the stream buffer such that a sender will block if the buffer is full, and a receiver will block if the buffer is empty.

A message buffer is created with the function `xMessageBufferCreate(bufferSizeBytes)`. The size specified should equal the total number of bytes (not messages) the buffer should be able to contain.

Sending and receiving data is done by using the functions below.

```
xMessageBufferSend(messageBuffer, *pvTxData, xDataLengthBytes, ticksToWait);
xMessageBufferReceive(messageBuffer, *pvRxData, bufferLengthBytes, ticksToWait);
```

4.4 Scheduling

Scheduling is the software deciding which task to run at what time. FreeRTOS has two modes of operation when it comes to handling task priorities: With and without preemption. Which mode to use can be set in the configuration file. When using the mode without preemption, also called cooperative mode, it is up to the developer to make tasks that yield the CPU through the use of blocking functions and the `taskYIELD()` function.

When using a preemptive scheduler, a task will automatically yield the CPU when a task of higher priority becomes unblocked. However, there is one exception: When a higher priority task blocks from an ISR, the `taskYIELD_FROM_ISR()` function has to be called at the end of the ISR for a task switch to occur.

If `configUSE_TIME_SLICING` is set to 1, the scheduler will also preempt tasks of equal priority at each time the tick occurs. Time slicing is not available in cooperative mode.

In both modes, the scheduler will always switch to the highest priority unblocked task. If there are multiple tasks unblocked with the same priority, the scheduler will choose to execute each task in turn. This is commonly referred to as round robin scheduling.

In the preemptive mode, higher priority tasks are immediately switched to when they get unblocked. In the cooperative mode, the release of a semaphore might unblock a higher priority task, but the actual task switch will only happen when the currently executing task calls the `taskYIELD()` function or enters a blocking state.

For more information on scheduling, see the “Scheduling Algorithms” section in the Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide.

5. Debugging in FreeRTOS

This section will cover some of the most common errors and provide some information on how to solve them.

5.1 Heap Debugging

Problems concerning the heap might occur due to the allocated heap memory size being too small, or the choice of heap implementation doesn't suit the application.

The function `xPortGetFreeHeapSize()` returns the total amount of heap space that remains unallocated when called. If this is very small, increasing the heap size might be considered.

Choosing a heap implementation that is not suiting the application can also cause trouble. As mentioned in the "Configuring Memory" section, there are five different heap implementations. Some of them do not free allocated memory, while others have problems with allocating memory of random size. The heap requirements for the specific application must be considered. See https://www.freertos.org/a00111.html for more information on each heap implementation.

5.2 Checking for Stack Overflow

A stack overflow is a very common source of support requests, but FreeRTOS provides some features to help debug such issues.

Stack overflow occurs when a task tries to push more elements on the stack than there is room for. The stack size is specified when a task is created. To find out how close a task is to overflowing its stack, the `uxTaskGetStackHighWaterMark(TaskHandle_t xTask)` function can be used. It returns the minimum of unused stack in words. If an 8-bit MCU is used, a return value of 1 will mean 1 byte. For a 32-bit MCU, 1 word means 4 bytes. To use this function, the `INCLUDE_uxTaskGetStackHighWaterMark` must be set to 1 in the configuration file.

It is also possible to check for stack overflow during run-time. By setting the `configCHECK_FOR_STACK_OVERFLOW` to 1, the kernel checks that the stack pointer remains within the valid stack space. If not, the stack overflow hook function is called. A hook function is a function that allows the application to react when something happens and provide different behavior. For example, an LED can be turned on when a stack has overflowed. Another option might be to print an error message and reboot the system. The `vApplicationStackOverflowHook()` function must be provided by the application.

See https://www.freertos.org/Stacks-and-stack-overflow-checking.html for more information about FreeRTOS and stack overflow.

5.3 Trace

Tracealyzer, previously known as FreeRTOS+Trace, is an analysis tool that collects data on how the embedded application is behaving. This data is valuable when troubleshooting or optimizing the performance of the application. Tracealyzer provides a graphic visualization of when which of the tasks get to run, and this can be helpful to find problems such as starvation. Refer to the application note FreeRTOS™ Using Percepio® Trace on ATmega4809 for an explanation of how to use Tracealyzer in Atmel Studio. For even more information, see https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_Trace/RTOS_Trace_Instructions.shtml.

6. Demo

The FreeRTOS example for the ATmega4809 in Atmel | START is available here: http://start.atmel.com/#examples/ATmega4809/FreeRTOS. The required hardware is the ATmega4809 Xplained Pro, and an OLED1 Xplained Pro connected via EXT3. The demo application blinks an LED while displaying the time on the OLED display. Pushing a button will also light up the corresponding LED and simultaneously update the OLED display to reflect the last button event.

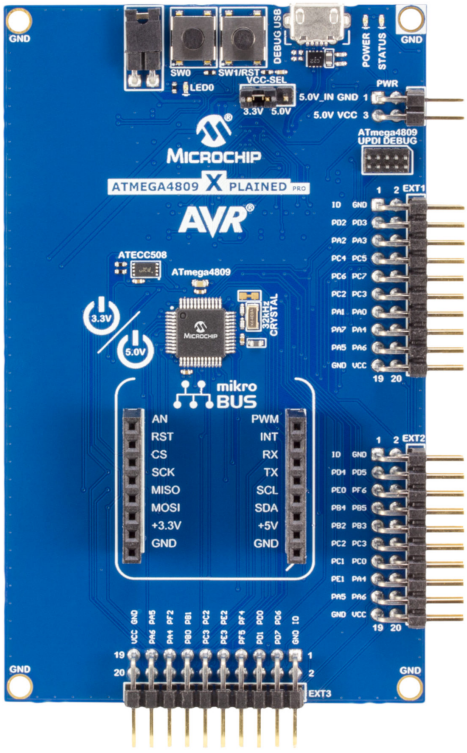
6.1 Required Hardware

Required hardware for the Atmel | START example project is described below.

6.1.1 ATmega4809 Xplained Pro Evaluation Kit

The ATmega4809 Xplained Pro evaluation kit is a hardware platform for evaluating the ATmega4809 AVR® microcontroller (MCU).

Figure 6-1. ATmega4809 Xplained Pro

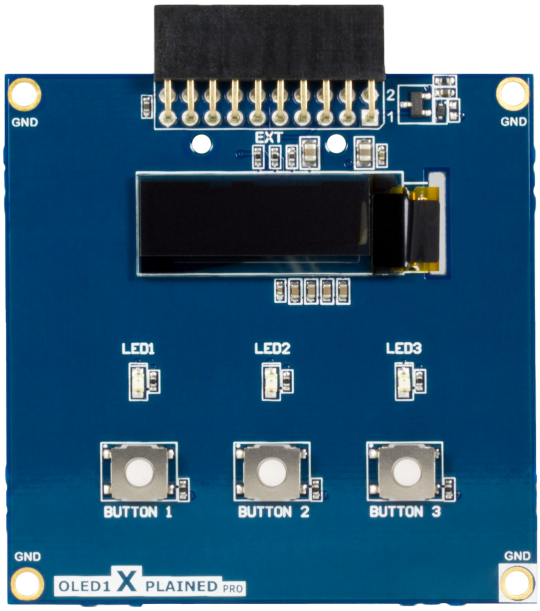


Webpage: https://www.microchip.com/developmenttools/ProductDetails/atmega4809-xpro.

6.1.2 OLED1 Xplained Pro Extension Kit

OLED1 Xplained Pro is an extension kit with a 128x32 OLED display, three LEDs and three push buttons. It connects to the extension headers of any Xplained Pro evaluation kit.

Figure 6-2. OLED1 Xplained Pro



Webpage: https://www.microchip.com/developmenttools/ProductDetails/ATOLED1-XPRO.

6.2 Partitioning Into Tasks

The overall goal of the example application is to blink an LED, handle button pushes and display the time on an OLED screen. Revisit the way of thinking described in section "Thinking like an RTOS developer". Accordingly, the different jobs are partitioned into individual tasks.

When thinking about what the application should do, it seems useful to have one task that is responsible for the LED blinking every 250 ms. The handling of the buttons pushed should also be an individual task. There is also a need for some sort of clock handling and a way to write to the OLED without causing any conflict because of simultaneous writing.

The example uses seven tasks, two queues, a mutex, a stream buffer and a message buffer. The section "Implementation" will go into detail on how these tasks are implemented. Below is a short description of each task and the communication methods they use.

Status LED Task

Blinks the LED0 on the ATmega4809 Xplained Pro every 250 ms. Uses no communication.

Keyboard Task

Detects if the state of one of the keys (buttons) changes, that is, if a button is pushed or released. Sends the updated key state to a queue called `key_queue`.

Main Task

Receives the updated key states through the `key_queue`, and creates a string to be printed on the OLED screen. Before writing, the task asks for a mutex protecting the screen called `oled_semaphore`. The task also sends info about pushed buttons to `led_queue`.

LED Task

Receives information about the latest button events through `led_queue` and sets the state of the LEDs accordingly.

Clock Task

Increments the time every second and writes to the OLED screen after taking the `oled_semaphore`. Can also set a new time if requested.

Terminal Transmit Task

Receives information on a message buffer called `terminal_tx_buffer`, and then puts this to the USART TX buffer.

Terminal Receive Task

Receives information through a stream buffer called `terminal_rx_buffer` and can, based on the received message, request the clock task to set a new time.

6.3 Shared Resources

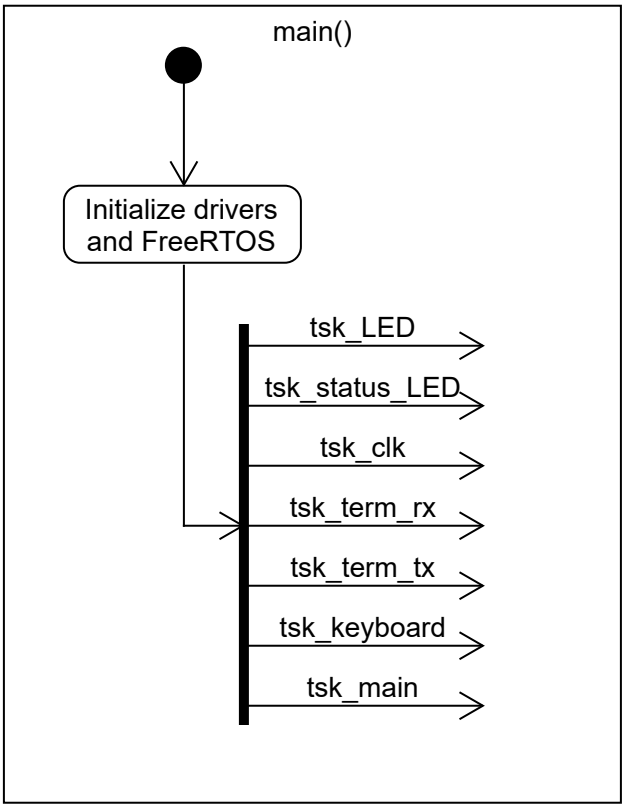
A shared resource in this application is the OLED screen. Both the main task and the clock task want to write to the display, but only one can do so at a time. If two tasks try to write to the screen simultaneously, the program might crash, or we could get some unpredictable behavior. The process of writing to the OLED is, therefore, protected by a mutex.

6.4 Implementation

This section provides a short explanation of the implementation of each task. UML diagrams are also provided for each task. UML stands for Unified Modeling Language and provides a standard way to visualize the design of a system. For the diagrams used in this document, parts that have to do with FreeRTOS are colored gray.

The initialization of drivers and FreeRTOS, as well as the creation of the tasks, is done in main, as shown in the figure below.

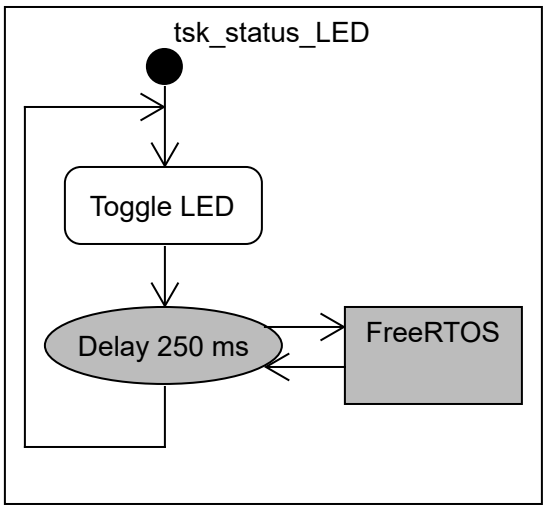
Figure 6-3. Initialization of Tasks and Drivers



6.4.1 Status LED Task

This is perhaps the simplest task of the application. It does only consist of an LED toggling function and a delay of 250 ms. The delay function takes the number of ticks as an input parameter. To convert ms into ticks, the function `pdMS_TO_TICKS(250)` is used. The UML diagram is shown in the figure below.

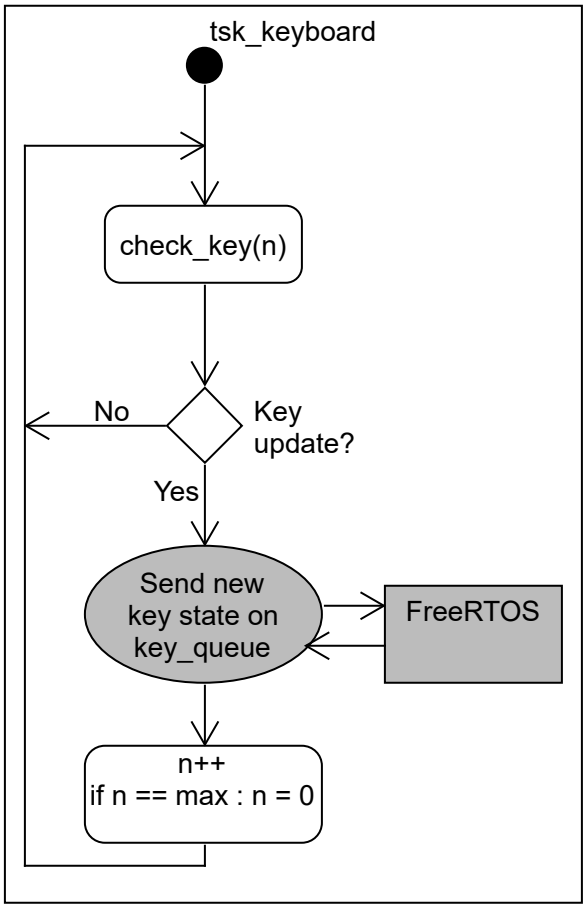
Figure 6-4. Status LED Task



6.4.2 The Keyboard Task

The keyboard task `tsk_keyboard()` is the task that handles the button pushes. If the state of one of the buttons is changed, the new state is put into the `key_queue`. The receiver of the `key_queue` is the main task which is described in the next section. Refer to the figure below for the UML diagram.

Figure 6-5. Keyboard Task

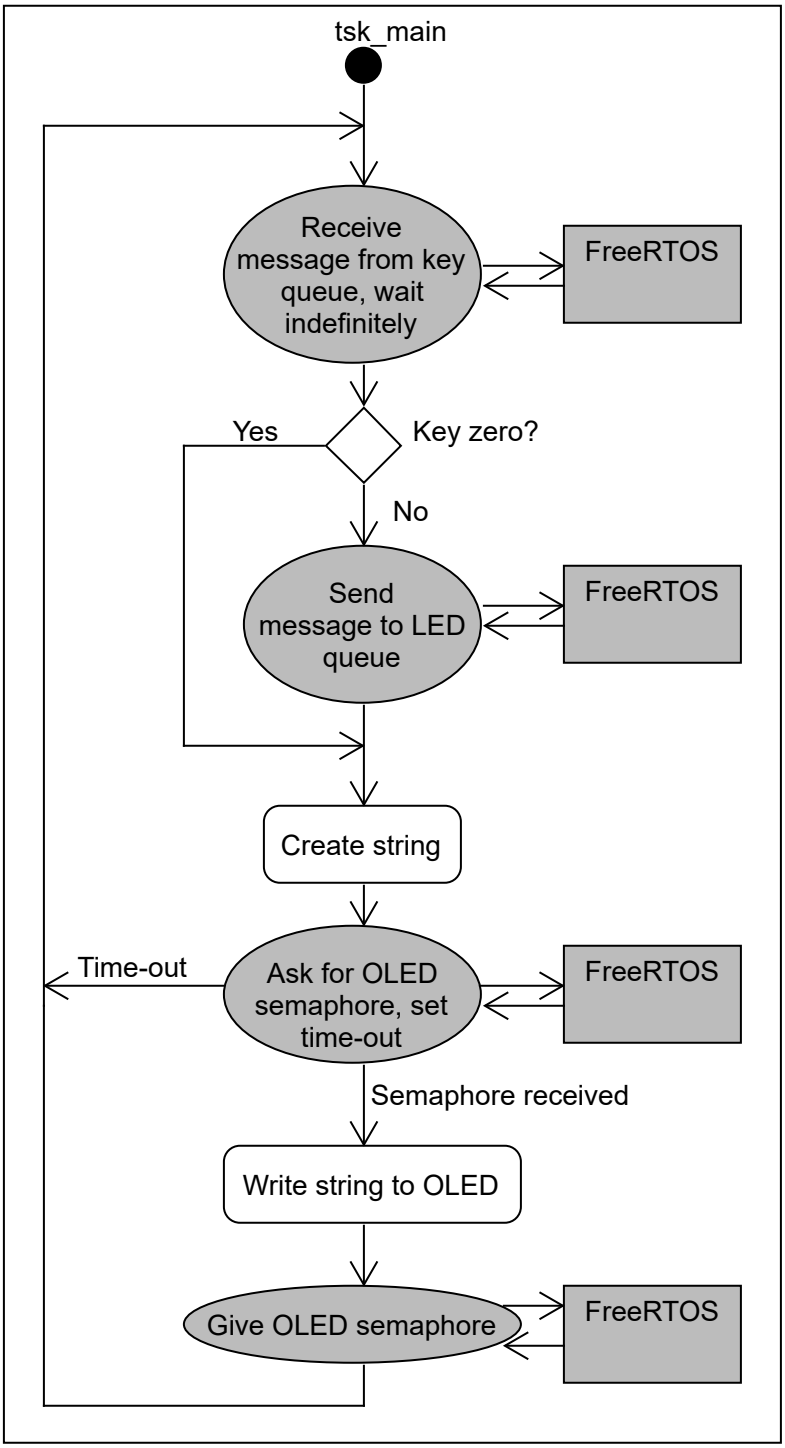


6.4.3 The Main Task

The `tsk_main()` task receives what is put on the `key_queue` by the keyboard task. Based on the received info it creates a string that reflects the last button event. The string is then written to the OLED. Examples of such strings are “Button 2 Released” and “Button 1 Pushed”.

The main task also sends the info about the pushed buttons to the `led_queue` such that the corresponding LED will be lit. See the figure below for the UML diagram of the main task.

Figure 6-6. Main Task

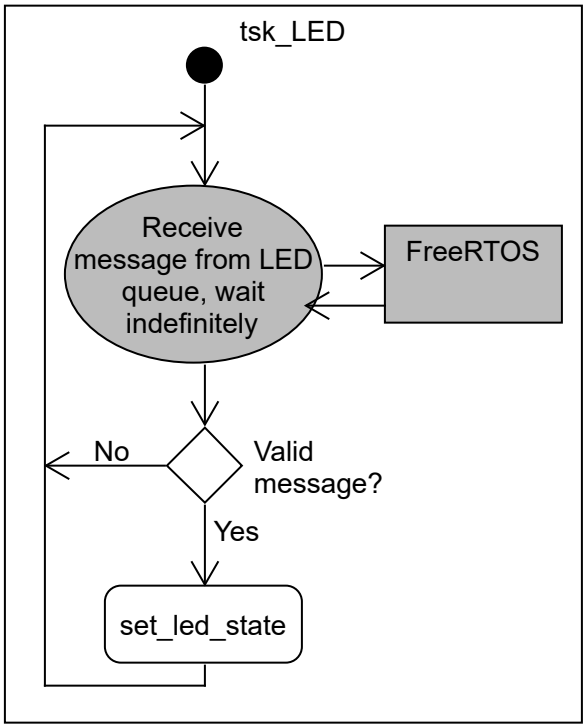


6.4.4 The LED Task

The responsibility of the LED task is to light up the LEDs that belong to the pushed buttons and to turn off the LEDs of the buttons that are not pushed. The LED will be lit for as long as the button is held down. The information about

the recent button events is received through the `led_queue`. As mentioned in the previous section, it is the main task that sends to this queue. Refer to the figure below for the UML diagram.

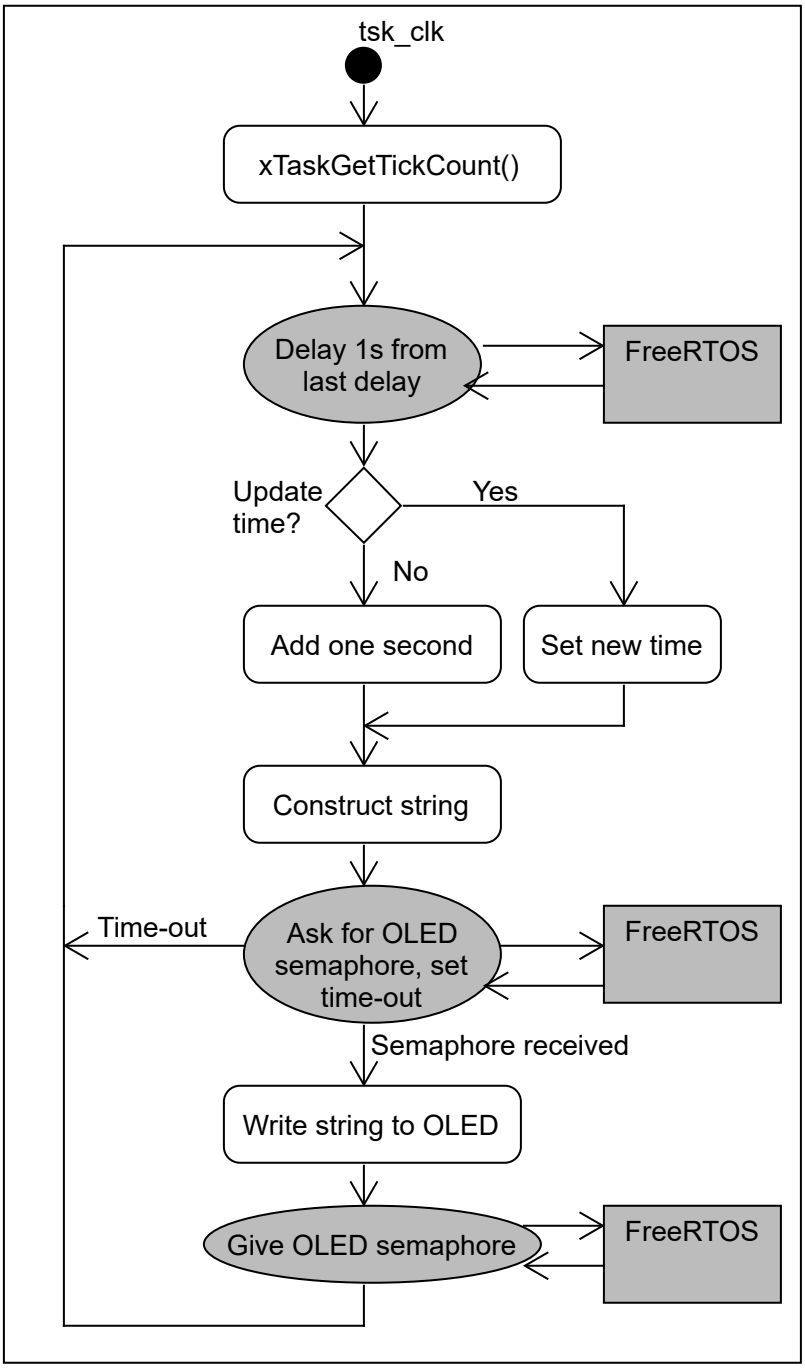
Figure 6-7. LED Task



6.4.5 The Clock Task

The clock task is responsible for writing the time to the OLED every second. This is done by taking the `oled_semaphore` when writing. The clock task is also responsible for incrementing the time every second. See the figure below for the UML diagram.

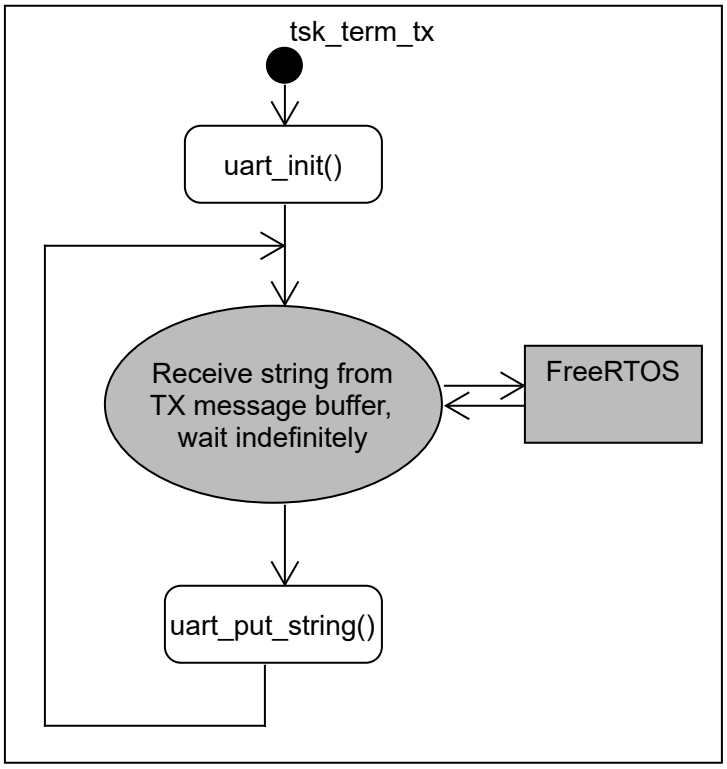
Figure 6-8. Clock Task



6.4.6 The Terminal Transmit Task

The terminal transmit task is responsible for sending what it receives on a message buffer called `terminal_tx_buffer` via the USART, as shown in the figure below.

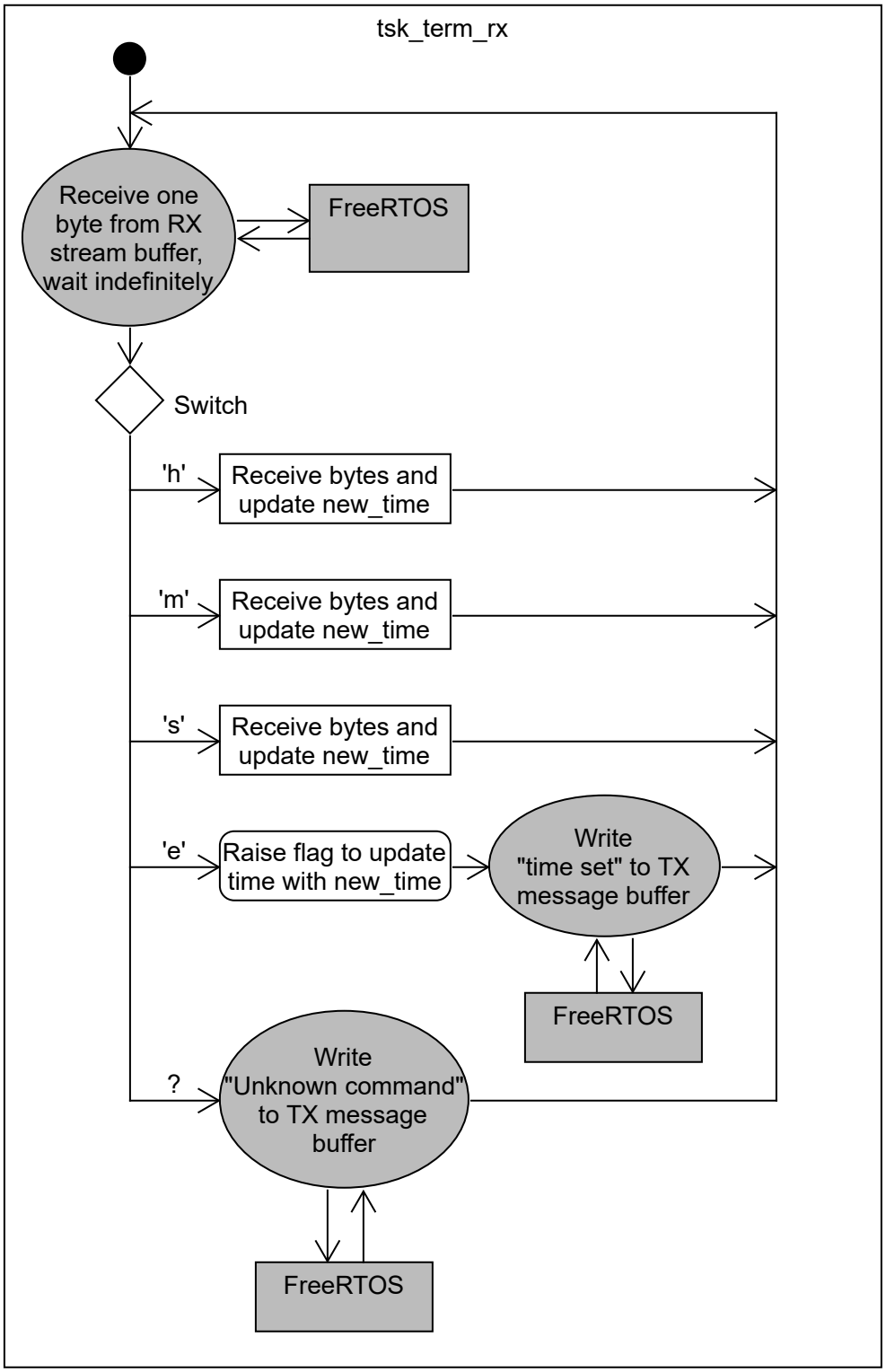
Figure 6-9. Terminal Transmit Task



6.4.7 The Terminal Receive Task

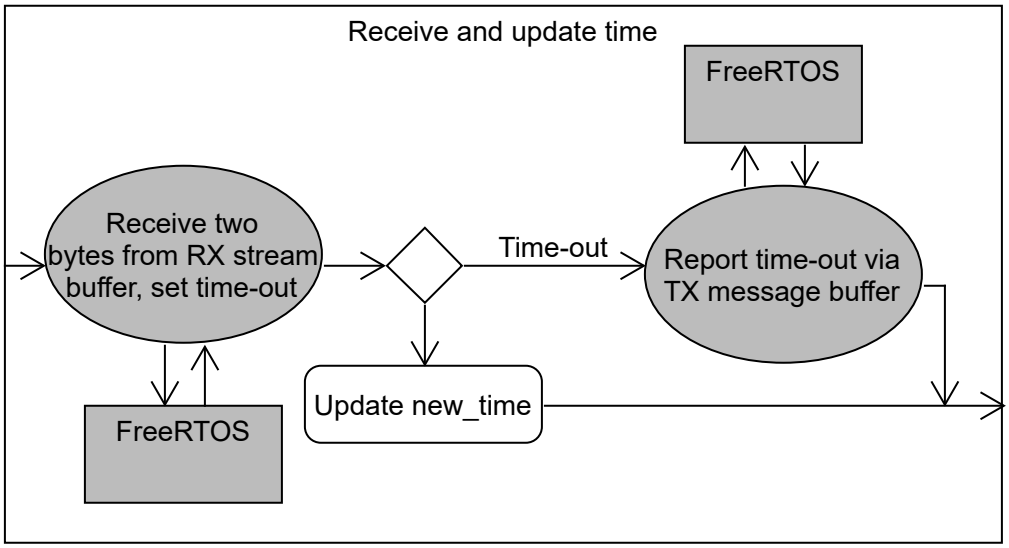
The ISR puts the received USART data on a stream buffer called `terminal_rx_buffer`. The terminal receive task is then taking the received data on the `terminal_rx_buffer`, and based on the received message, calls an update time function unless a time-out has occurred. The terminal receive task reports back to the terminal transmit task via the `terminal_tx_buffer` about the latest event, whether the time was set or if the request timed out. Refer to the figure below for the UML diagram.

Figure 6-10. Terminal Receive Task



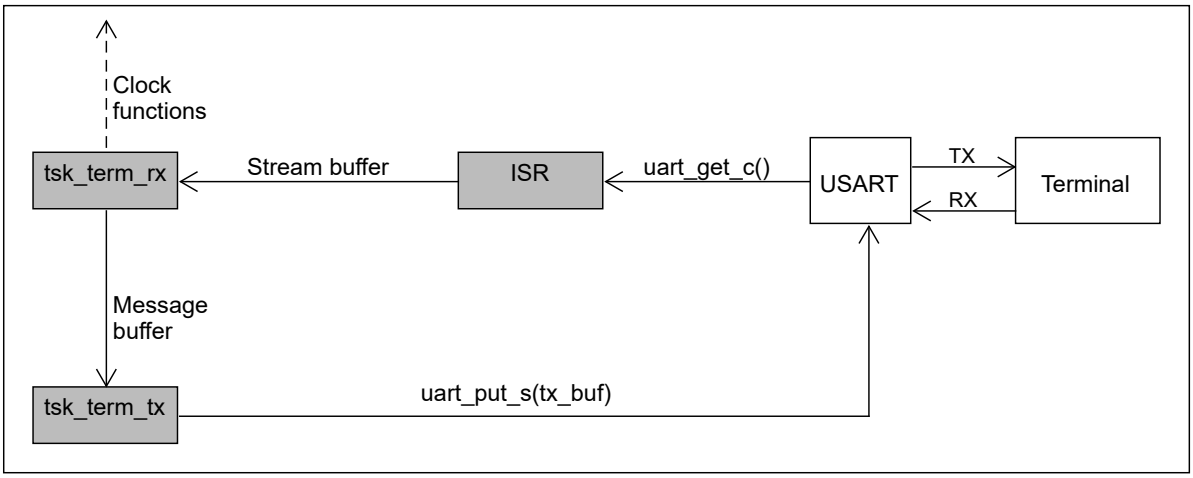
The figure below shows a more in-depth description of the “receive bytes and update new_time” block.

Figure 6-11. Receive Bytes and Update Time



The communication between the two terminal tasks as well as the USART communication is described in the figure below.

Figure 6-12. Communication Between the Terminal Transmit Task and the Terminal Receive Task



7. Get Source Code from Atmel | START

The example code is available through Atmel | START, which is a web-based tool that enables configuration of application code through a Graphical User Interface (GUI). The code can be downloaded for both Atmel Studio and IAR Embedded Workbench® via the direct example code-link below or the *Browse examples* button on the Atmel | START front page.

The Atmel | START webpage: http://start.atmel.com/

Example Code

ATmega4809 FreeRTOS Example
Click *User guide* in Atmel | START for details and information about example projects. The *User guide* button can be found in the example browser, and by clicking the project name in the dashboard view within the Atmel | START project configurator.

Atmel Studio

Download the code as an `.atzip` file for Atmel Studio from the example browser in Atmel | START by clicking *Download selected example*. To download the file from within Atmel | START, click *Export project* followed by *Download pack*.

Double click the downloaded `.atzip` file, and the project will be imported to Atmel Studio 7.0.

IAR Embedded Workbench

For information on how to import the project in IAR Embedded Workbench, open the Atmel | START User Guide, select *Using Atmel Start Output in External Tools*, and *IAR Embedded Workbench*. A link to the Atmel | START User Guide can be found by clicking *Help* from the Atmel | START front page or *Help And Support* within the project configurator, both located in the upper right corner of the page.

8. Revision History

Doc. Rev.	Date	Comments
B	12/2019	Removed tiny0 and 1 from Relevant Devices section. Updated flow charts in Thinking Like an RTOS Developer section.
A	04/2018	Initial document release.

The Microchip Website

Microchip provides online support via our website at http://www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip design partner program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Product Change Notification Service

Microchip's product change notification service helps keep customers current on Microchip products. Subscribers will receive email notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, go to http://www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: http://www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with

your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, chipKIT, chipKIT logo, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PackeTime, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TempTrackr, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, FlashTec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet-Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, Vite, WinPath, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, EtherGREEN, In-Circuit Serial Programming, ICSP, INICnet, Inter-Chip Connectivity, JitterBlocker, KleerNet, KleerNet logo, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, SAM-ICE, Serial Quad I/O, SMART-I.S., SQI, SuperSwitcher, SuperSwitcher II, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, and Symmcom are registered trademarks of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

© 2019, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

ISBN: 978-1-5224-5219-5

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit http://www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: http://www.microchip.com/support Web Address: http://www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Australia - Sydney** Tel: 61-2-9868-6733 **China - Beijing** Tel: 86-10-8569-7000 **China - Chengdu** Tel: 86-28-8665-5511 **China - Chongqing** Tel: 86-23-8980-9588 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 **China - Hong Kong SAR** Tel: 852-2943-5100 **China - Nanjing** Tel: 86-25-8473-2460 **China - Qingdao** Tel: 86-532-8502-7355 **China - Shanghai** Tel: 86-21-3326-8000 **China - Shenyang** Tel: 86-24-2334-2829 **China - Shenzhen** Tel: 86-755-8864-2200 **China - Suzhou** Tel: 86-186-6233-1526 **China - Wuhan** Tel: 86-27-5980-5300 **China - Xian** Tel: 86-29-8833-7252 **China - Xiamen** Tel: 86-592-2388138 **China - Zhuhai** Tel: 86-756-3210040	**India - Bangalore** Tel: 91-80-3090-4444 **India - New Delhi** Tel: 91-11-4160-8631 **India - Pune** Tel: 91-20-4121-0141 **Japan - Osaka** Tel: 81-6-6152-7160 **Japan - Tokyo** Tel: 81-3-6880- 3770 **Korea - Daegu** Tel: 82-53-744-4301 **Korea - Seoul** Tel: 82-2-554-7200 **Malaysia - Kuala Lumpur** Tel: 60-3-7651-7906 **Malaysia - Penang** Tel: 60-4-227-8870 **Philippines - Manila** Tel: 63-2-634-9065 **Singapore** Tel: 65-6334-8870 **Taiwan - Hsin Chu** Tel: 886-3-577-8366 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 **Thailand - Bangkok** Tel: 66-2-694-1351 **Vietnam - Ho Chi Minh** Tel: 84-28-5448-2100	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4450-2828 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-72400 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-72884388 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820