RTOS Design and Implementation

1.0 Overview

For this project we designed and implemented a simple real-time operating system to fulfill the specifications provided to us. This report will describe the design, implementation, testing, and profiling of this active kernel style system, as well as any insights or recommendations we have. The target hardware platform for this project was again the ATmega2560.

Within this simple operating system, there exists three types of tasks, System, Periodic, and Round Robin. Systems tasks are of the highest priority, execute on a first come first serve basis, and always run to completion before yielding. Periodic tasks, which are the second highest priority, are time critical tasks which are executed based on a strict timing schedule provided by the engineer, defined by a start tick, period, and worst case execution time. Finally, Round Robin tasks are the lowest priority, are first come first serve, but only execute for a single tick before yielding the processor to the next Round Robin task.

One of the key requirements of the system was to ensure preemption of tasks by any tasks with a higher priority. For example, if a Round Robin task were to create a System level task, as will be discussed later in detail, the creation of the task would result in the Kernel switching to execute this new System task, before returning to execute the Round Robin task.

In addition to prioritized and periodic tasks, the OS also supports a simple synchronization mechanism known as a Services. Services allow a non-periodic task to subscribe to the publishing of a value by another task, the task which subscribes will wait until another task publishes a value to that Service. This mechanism will be discussed in detail below.

In the sections which follow we will discuss the Runtime environment of the OS and the data structures it uses, the life cycle of tasks and how they are scheduled, our testing strategies and test cases, and finally, a profile of the execution times and overhead of the OS. Lastly, we will present any notes or recommendations we have.

2.0 RTOS Environment

The sections which follow provide detailed descriptions of the runtime environment and memory space of the operation system. We designed the system to contain only statically allocated data within the kernel, although this requires definition of the limits of stack and number of tasks at compile time, it ensures that memory usage by the kernel is consistent from run to run, regardless of when during execution tasks are created and destroyed.

2.1 C Run Time

Although we were provided with examples of c runtime instructions to initialize the system, we chose to use the included crt.s that comes bundled with AVR. The decision was made to keep the implementation simple and more portable as the initialization is written in C; execution starts with the OS' main method, which then calls kernel_init to initialize the needed data structures.

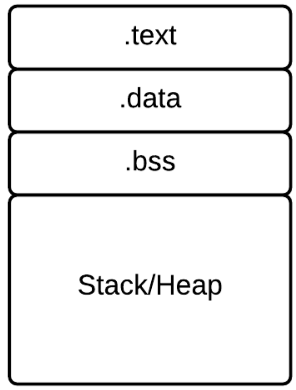

Figure 1 - High Level Memory allocation after C Runtime Initialization

The .text portion of memory contains all program code, including the Kernel and User Application. The *.data* portion of memory contains all variables declared and initialized, either by the user application or the kernel. For the kernel, task contexts including stacks, periodic task metadata, current task descriptors are stored here, frankly, most of the data related to the kernel can be found in this section. The .bss section contains all variables declared but not initialized, again either by the user application or our kernel, as a result, it contains the remaining data of the kernel, such as the current task pointer, and kernel stack pointer value.

The remaining free space in memory is then used for the Heap and Stack. Heap is used for dynamic allocation of resources, the kernel, is it is strictly statically defined, will never make use of this space, however an ill-informed user may write an application using this space, often we try to steer clear of dynamic memory in embedded systems however. The Stack space is then normally used as the stack of the process, however, applications written using our OS, will not make use of this space, as its stack space for each task is defined in static memory space in the *.data* section. If a developer were to find there was excessive free space in the stack, one may choose to increase the workspace of the tasks.

2.2 RTOS Data Structures

The subsections below discuss in detail the data structures and allocation schemes of contexts and task descriptors.

2.2.1 Task Descriptors and Periodic Meta Data

Tasks within the system are defined using the task_descriptor_t type, shown below, which contains all components to a tasks execution but the code itself, which as mentioned previously is located in .text.

<INSERT CODE SNIPPET>

As one would expect the we find the task's stack, stack pointer, priority, and state (more on States in section 2.1.2). In addition to these intuitive elements, task descriptors also contain a pointer to a periodic_task_metadata_t structure, a pointer to an integer which is used to publish values to the task if it has subscribed, and finally, a pointer to another task_descriptor_t, details of these three additional fields will be discussed below.

Unlike System and Round Robin tasks, Periodic tasks have additional information that defines them; this information is stored in a periodic_task_metadata_t, shown below, which contains the period, worst case execution time, and next, a value which indicates the next tick the task is to be executed. In addition, the structure also contains a pointer to a task descriptor, as well as another periodic metadata structure. The use of these fields will be discussed below.

<INSERT CODE SNIPPER OF META DATA>

2.2.2 Task Queues

To store and queue tasks for execution, the kernel makes use of 5 queues. To avoid the use of dynamic memory these queues and the maximum number of tasks and periodic tasks are defined at compile time and allocated statically. Previously we mentioned the that periodic metadata and task descriptors each contain pointers to themselves and the other. This allows us to create queues in place, without the need to allocate nodes as the descriptors themselves are the nodes in the linked list.

At startup, the kernel initializes all task descriptors and periodic metadata in place, and creates two ‘dead’ queues which contain all the unused descriptors for tasks and periodic metadata structures. As tasks are created, takes are moved to an appropriate queue based on their priority. As tasks are killed, their task descriptors are cleared for reuse, and the descriptor is added back to the dead pool.

Below in Figure 2, we have defined the system to have a maximum of 4 tasks, up to 2 of which can be periodic. As a result, we statically allocate an array of 4 task_descriptor_t structures, and another array of 2 periodic_metadata_t. After initialization, the structures are all linked in order, and the queue for the dead tasks points to the first and last entry.

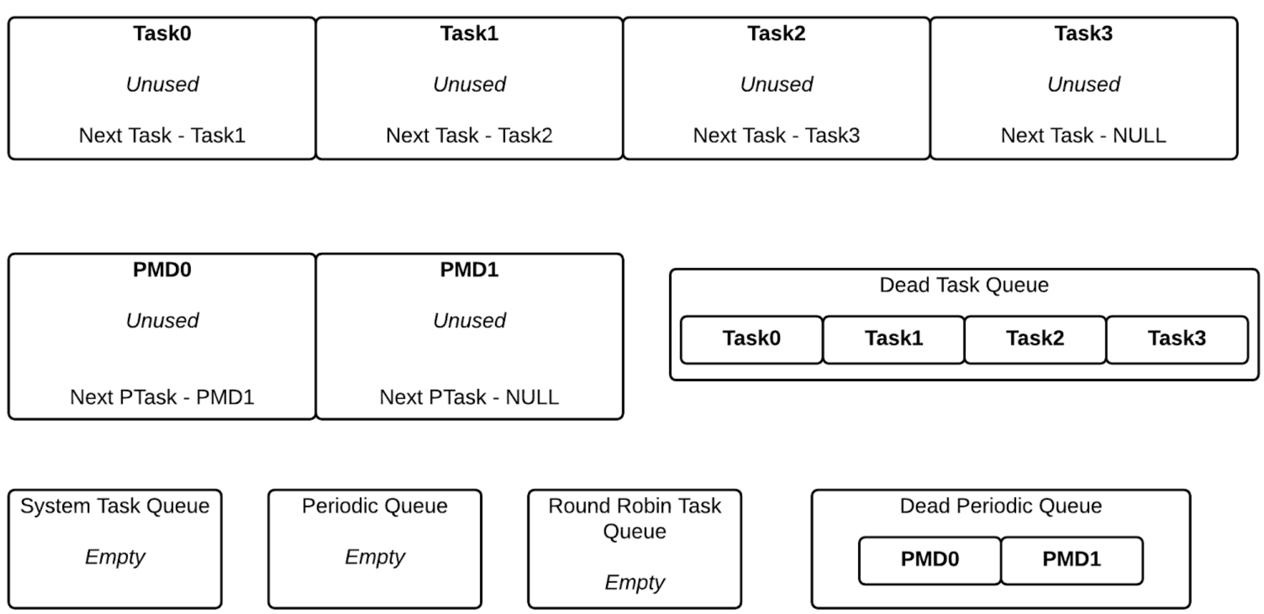

Figure 2 - Task Descriptors and Periodic Meta Data Arrays and Queues after Initialization

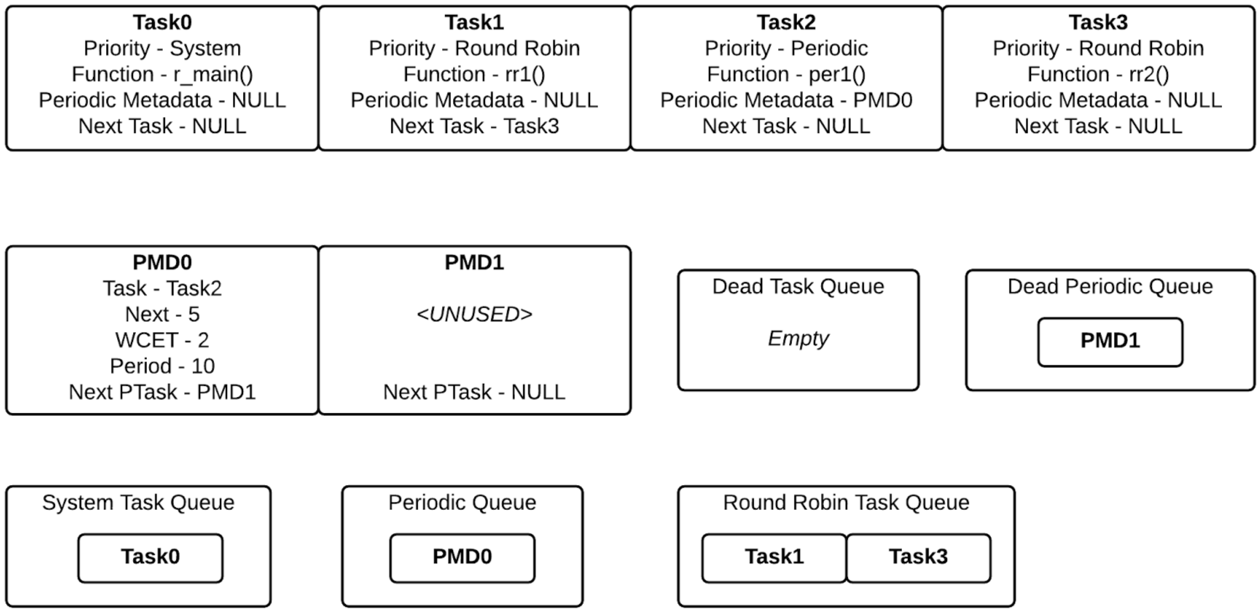

Figure 3 - Task Descriptors and Periodic Metadata Arrays and Queues after task creation

In Figure 3 above, the system has created a system task, two round robin tasks, and a single periodic task. As we can see, the system task and round robin tasks have been moved from the dead queue and into their respective schedule queue. The single periodic task however, is not placed in a queue using its task descriptor, instead it is placed in a queue based on its metadata, which contains a reference to the task descriptor for use by the scheduler. Figure 3 below shows the state of the queues after the creation of these tasks.

Section 4.0 discusses task scheduling in detail and provides additional details on how queues are utilized for scheduling and task management.

2.2.3 Context Switching

As we decided to implement the OS using an active kernel, API calls require a switch between User space and Kernel space, and this requires a change of execution context. In addition, this context switch is needed to change back to the executing task once the kernel has completed its work. The context switch operation requires that you replace the current execution context of the running task, with that of the kernel or a user task.

The execution context of a task includes its stack pointer, all 32 general purpose registers, the SREG register and the EIND value, which stores an additional bit used in function addresses in combination with the two ZH and ZL registers. To accomplish the swap of these values with that of the target task, we made use of the provided RESTORE_CTX() and SAVE_CTX() macros, with a modification to include the missing EIND value, required. This additional bit also changes the size of the stack pointer and return address, to compensate, we had to add two additional bytes to the initial stack of the task when constructing its first execution context.

Switching occurs by first pushing the executing context of the current task onto the task’s stack, the execution of the target task, or kernel, is then restored by popping it off its stack, where it would have been stored when changing tasks. This approach requires that we must manually build the first execution context by hand when creating a task.

3.0 OS API

Detailed documentation for the OS interface can be found in os.h. Below in figure X we should the interfaction of OS calls with the kernel. Many of the calls require a context change to the kernel, but calls to Get_Arg() and Now() do not, instead they can fetch directly into memory for the values they are looking for.

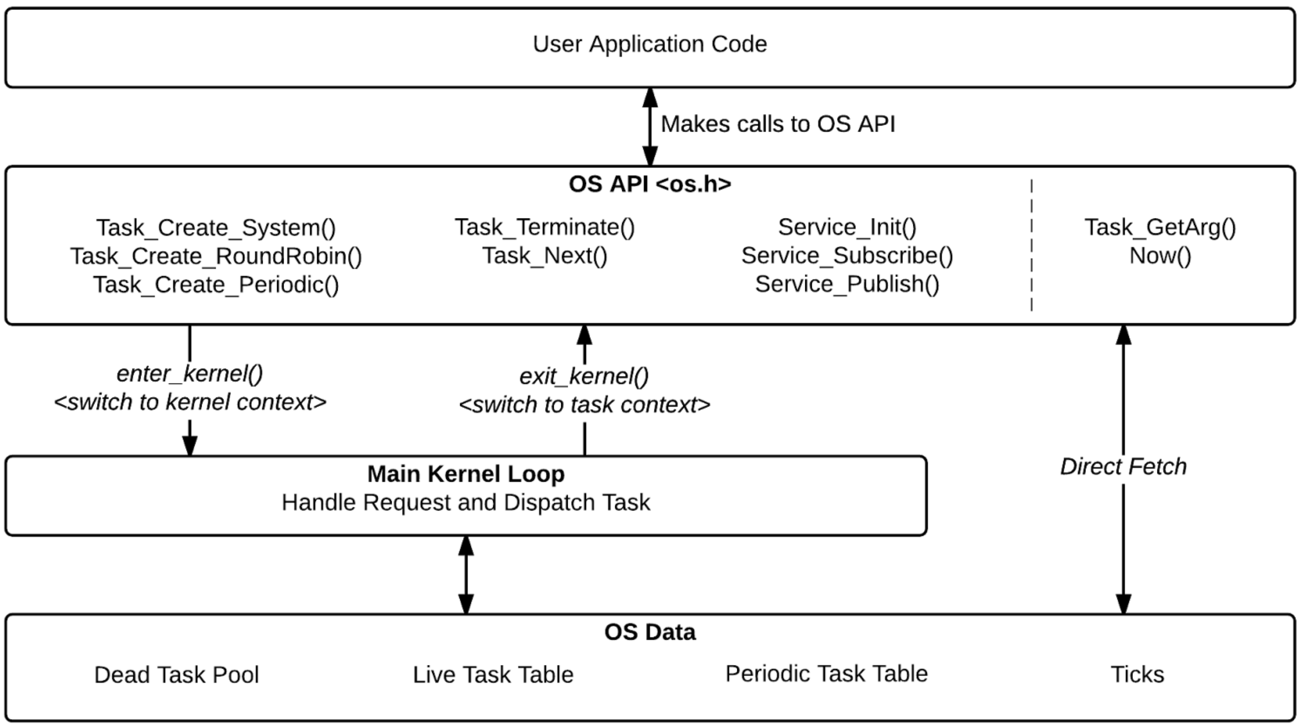

Figure 3 - OS API and Context Boundaries

4.0 Task Scheduling and OS Requests

After the system is initialized, a single System level task is created before the OS starts, that tasks points to the user space r_main, defined by the user application. This r_main should then create tasks as needed and otherwise configure the application on the user side. We chose to run this r_main as a System task to ensure that it is not preempted by a task created during user space initialization. This section will cover in detail the scheduling and preemption schemes implemented in our RTOS.

Task Life Cycle

Tasks have a finite number of potential states which they can occupy. Then not in user, we refer to the task as being DEAD. After initial creation, and when the thread is not waiting, it resides in the READY state, indicating it can be run at any time. The current running task is the sole task with the state RUNNING, while any task waiting on a service will reside in the WAITING state.

Below in Figure A, the life cycle of a task is given as a finite state machine. Each of the operations used as transition triggers is described briefly below, more detailed information on services can be found below in Section XXX.

Task_Next - Task voluntarily relinquishes its remaining time.

Task_Create - Task creates a new task of any priority. It is possible that a call to this OS method, for example by a Round Robin task to create a System task will result in immediate preemption, placing the current task back into a ready state, and in this case, placing the Round Robin task at the front of the Round Robin Queue.

Task_Terminate - This is called automatically by tasks which simply return from their functions, but can also be called explicitly to terminate the task, clearing its descriptor and periodic metadata, it is then added to the dead queues.

Service_Subscribe - The task will wait on a service to publish before returning to a READY state. Note, periodic tasks may not call this method as they may not be bound to synchronization mechanisms, as per the RTOS specification.

Service_Publish - Task publishes a value to the service, waking any tasks currently waiting on that service. This call can result in preemption if a task of high priority is awoken.

Kernel Request Handling and Dispatch

Services

As specified in os.h, services are used to transfer data safely between tasks, as well as allow tasks to wait for certain data to be published. This has many uses, such as waiting for a section of memory to become available, or waiting for radio communication to finish with a device. As per the documentation, services were split into three distinct OS API calls: Service_Init, Service_Subscribe, and Service_Publish.

The service data structure is built up of only a task queue, the same sort of queues used for system tasks and round robin tasks. This queue simply holds all tasks that subscribe to a given service.

<Add service data structure>

Each OS API function ends up making a call to the kernel. As seen in Figure 3, all Service functions create a new kernel task to complete, enter the kernel, and allow the kernel to handle all appropriate service functionality. Moving all service functionality into the kernel allows us to separate the service and task logic that exists in the kernel only from the user's task context. It also had the benefit of easily being able to preempt tasks that awoke when publishing data to a service. The following sections will go into more detail describing each call.

Initializing

As described in the given documentation, Service_Init will return a pointer to a new service type. First, the user makes a call to the OS API Service_Init, which will disable interrupts and set the kernel request to 'SERVICE_INIT'. It then switches context and enters kernel mode, where the kernel will proceed to create a new service descriptor found in the services list in the OS data. It will then exit the kernel, and return the service descriptor to the user. There is a maximum amount of services that can be created in the RTOS, which in this case is 10. If a user creates more than the maximum amount, the RTOS will abort and send the appropriate error message.

The services list is an array of services that exists in the OS data.

Subscribing

The functionality of Service_Subscribe follows the given documentation. A task may provide a service descriptor and the address to where the task expects the publisher to publish the data. Once the task subscribes, it will enter a waiting state until a task publishes to the service. A task will call the OS API Service_Subscribe, which will proceed to create a kernel request called SERVICE_SUB. Then, the task will switch contexts to the kernel, which will subscribe the specified task to the service descriptor. The task is added to the service's task queue, and it's state is changed to WAITING. Then, the kernel will attempt to dispatch a new task, since the current task is set to WAITING.

The maximum amount of tasks that can subscribe to a service is related to the amount of memory the ATMega2560 contains, as it is capped by the amount a queue_t type can store. If the user provides an invalid service descriptor, the RTOS will abort and send the appropriate error message.

There is a restriction on the type of tasks that can subscribe to a service. Only system and round robin tasks may subscribe. If a periodic task attempts to subscribe to a service, the RTOS will abort and send the appropriate error message to the user.

Publishing

As described in the documentation, Service_Publish will take a service descriptor and an int16_t to publish to every task found in the services task queue. All tasks found in the service change their state to READY, and the data is written to the expected address given when the tasks subscribed. More specifically, a call to the OS API Service_Publish will create a kernel request called SERVICE_PUB. It will then switch contexts to the kernel, and publish the data to the given service descriptor. Then, for each task in the service's task queue, we will set their state to READY, and add them back into the appropriate task queue depending on their priority (SYSTEM or ROUND_ROBIN). It will then check and see if there are any tasks that preempt the currently running task, and if so, will change the current task.

If an invalid service descriptor is provided to the Service_Publish function, the RTOS will abort and send the appropriate error message to the user.

There were various problems attempting to implement the publish routine. Initially, every publish call would go to completion, even if there was a system task that was awoken when publishing to the service. This was quickly fixed by adding the tasks back into their appropriate queues. Another problem we found was handling interrupts that published to services. The desired functionality was to prevent preemption on an interrupt, and let them run to completion. This was to avoid interrupt handlers being preempted and not resetting various handler flags which would prevent other interrupts from happening.

<Add Pub Sub architecture or call diagram>

Error Handling

The sample RTOS code came bundled with a number of runtime and initializations errors that could be encountered by the system. We were fans of the 'pulse x times to indicate error code x' and extended it into our solution. When an error occurs, we set the error_msg variable to the appropriate error code, and call OS_Abort(), effectively stopping the system. We removed and added various error codes, which are listed here as well as in error_code.h.

ERR_1_INIT_FAILURE
An initialization error happened during the function kernel_init when first starting the RTOS.

ERR_RUN_0_USER_CALLED_OS_ABORT
A runtime error meant to catch any instance of the user calling OS_Abort without actually setting the error message. In a production environment, this error should never happen.

ERR_RUN_1_TOO_MANY_TASKS
A runtime error stating the user is attempting to create too many tasks. The maximum allowed tasks is currently set in the MAXPROCESS constant.

ERR_RUN_2_TOO_MANY_PERIODIC_TASKS
A runtime error that catches if the user is creating specifically too many periodic tasks. As periodic tasks exist in their own priority queue built especially for them, they have stricter constraints over other task queues.

ERR_RUN_3_PERIODIC_INVALID_CONFIGURATION
The user specified a configuration that cannot work for periodic tasks. Specifically, this error will be thrown when the user specifies a period of 0 or they set the worst case execution time to be longer than the period.

ERR_RUN_4_PERIODIC_TOOK_TOO_LONG
A runtime error found when a periodic task runs longer than its expected worst case execution time.

ERR_RUN_5_PERIODIC_TASKS_SCHEDULED_AST
A runtime error that occurs when the user schedules two periodic tasks to run at the same time. It is a requirement that the user ensures this never happens, not the RTOS.

ERR_RUN_6_ILLEGAL_ISR_KERNEL_REQUEST
A runtime error when an interrupt service routine attempts to make a kernel request when it is not supposed to.

ERR_RUN_7_RTOS_INTERNAL_ERROR
A runtime that occurs when the current task in the RTOS is null. An error that should never happen as long as the idle_task exists.

ERR_RUN_8_SERVICE_CAPACITY_REACHED
A runtime error found when a user attempts to create more services that the max amount of services specified under the constant MAXSERVICES.

ERR_RUN_9_INVALID_SERVICE
A runtime error found when the user gives the functions Service_Subscribe or Service_Publish an invalid service descriptor.

ERR_RUN_10_PERIODIC_SUBSCRIBE
A runtime error thrown when a periodic task attempts to subscribe to a service. There is a strict requirement that subscribing tasks must not be periodic.

ERR_RUN_11_PERIODIC_FOUND_WHEN_PUBLISHING
A runtime error thrown when a periodic task is awoken when a service is published to. Since services don't accept periodic subscribers, there is no way this task should happen unless the user misuses the service descriptor without running the OS API functions.

ERR_RUN_12_TASK_WITHOUT_PRIORITY
A runtime error found when a task that is awoken from a published service has no priority specified. This should never happen unless the user misuses the service descriptor.

Testing

Rigorous testing was applied to the RTOS to ensure we caught every possible task state and combination, as well as testing all our possible error states. The testing framework used is taken mostly courtesy of the test framework used by Justin Tanner and Scott Craig during their initial creation of the RTOS.

The test framework uses trace.h and uart.h to print messages to the trace buffer, and output it to the serial monitor. To allow us to view the trace messages, we ended up opening the Arduino IDE and checking that the messages found in the serial monitor were identical to the expected messages found in each test file.

Every test is a specific file found in the tests folder of the project. Each test has a description in its filename to specify the type of test that is being run, and there is a comment at the top of each test displaying the expected string to be found in the serial monitor. Since every test should run with a newly started RTOS, it was necessary for each test to define its own r_main. Then, a constant was used to specify which test's r_main was compiled. This meant that each tested needed us to recompile the entire RTOS, and flash it onto the boards memory every time.

Each test was built around the same concept: Write a test to produce a specific functionality, and use trace to ensure that everything ran in the appropriate order. Generally we ended up writing the numbers 1 - 10 to trace and ensuring they appeared in chronological order. For tests that had multiple tasks and services, it was beneficial to write chronological trace statements to prove the order of initialization and completion.

<?Diagram representing the call order: User writes test → test calls OS → OS runs tasks → test writes to trace → trace to uart → uart to serial monitor>

Test Cases

Here is a list of all test cases run on the RTOS, and a short description of their use c*ase.*

test000_sanity.cpp
The basic sanity check test. It ensures we can create a system task that will write a message to the trace buffer. If we can't see any output from this, then either our test framework or our RTOS is broken and all other tests will provide useless information.

test001_NowCorrectTime.cpp

Ensures that the Now function returns the correct time in ms. This test will create a loop that writes the result from Now to the trace buffer, and the waits 100ms. Visually, we inspect and ensure the values being written are in increments of 100.

test002_create_system_task.cpp
Creates a system task, and ensures that it is scheduled and runs to completion. This is the basic system task test. If this fails, then all subsequent tests that use system tasks should fail as well.

test003_multiple_system_task_order.cpp
Creates multiple systems tasks to be scheduled. Ensures that system tasks run in the order they are created.

test004_system_task_preemption.cpp
Creates two system tasks, and ensures that one system task doesn't preempt the other. System tasks should run in the order they are scheduled. Very similar to test003.

test005_system_preempt_nonsystem_task.cpp
Creates a round robin task that then creates a system task. The system task should preempt the round robin task as it has more priority over it. This is verified using the order of the trace messages that is written by each task. Ensures our basic preemption code is working correctly.

test006_create_periodic_task.cpp
Creates a periodic task that should get scheduled and will then print a message to the trace buffer. This is the basic periodic task test. If this test fails, then all subsequent periodic tests will also fail.

test007_periodic_worst_case_time_larger_period.cpp
This test will create a periodic task that has a worst case execution time larger than its period. We expect this test to throw the error ERR_RUN_3_INVALID_CONFIGURATION and flash the red LED on the board 3 times.

test008_periodic_zero_period.cpp
This test will create a periodic task where the period is specified to be 0. We expect this test to throw the error ERR_RUN_3_INVALID_CONFIGURATION and flash the red LED on the board 3 times.

test009_multiple_periodic_tests.cpp
Creates multiple periodic tasks that do not collide, and ensures they are both running at the correct times. This is verified by printing both the TICK they are running, as well as a trace message corresponding to that task. This test ensures we are scheduling periodic tasks correctly, and that we aren't forgetting to reschedule periodic tasks when they call Task_Next.

test010_periodic_tasks_collide.cpp
This test will create two periodic tasks that start at the exact same time. We expect this test to throw the error ERR_RUN_5_PERIODIC_TASKS_SCHEDULED_AST and flash the red LED on the board 5 times.

test011_periodic_task_not_completed_on_time.cpp
This test will create a periodic task with a period of 5 ms. The task itself is specified to delay for 50 ms. We expect this test to throw the error ERR_RUN_4_PERIODIC_TOOK_TOO_LONG and flash

the red LED 4 times. This test ensures our logic that checks the amount of time a periodic task has run for is working as expected.

test012_periodic_system_preemption.cpp
This test will create a periodic task that will then create a system task. The periodic task should then be preempted by the system task. This is verified by the order in which they write messages to the trace buffer. This test ensures that our logic for system preemption works also for periodic tasks.

test013_create_round_robin.cpp
This test creates a round robin test, and ensures it gets scheduled and runs to completion. This is the basic round robin test case, and if it fails then every other test that uses round robin tasks will also fail.

test014_multiple_round_robin.cpp
This test will create multiple round robin tasks, and ensure they are scheduled and run in the correct order by viewing the order in which they wrote to the trace buffer.

test015_create_service.cpp
This test will call the Service_Init function, and ensure we return a service descriptor. This test is to make sure our service initialization calls aren't throwing errors for any reason.

test016_create_max_services.cpp
This test ensures we can create up to MAXSERVICES. It will loop, and call Service_Init MAXSERVICES times.

test017_create_over_max_services.cpp
This test will attempt to create more than MAXSERVICES services. We expect this test to throw the error ERR_RUN_8_SERVICE_CAPACITY_REACHED and blink the red LED on the board 8 times.

test018_system_subscribe_service.cpp
This test ensures that a system task can subscribe to a service. A system task is created, that will then subscribe to a service. We ensure that the task is waiting by checking the trace messages and ensuring the system task trace message was not written to the buffer.

test019_periodic_subscribe_service.cpp
This test will attempt to create a periodic task that will then subscribe to a service. As periodic tasks should not be able to subscribe, we expect this test to throw the error ERR_RUN_10_PERIODIC_SUBSCRIBE and blink the red LED 10 times.

test020_round_robin_subscribe_service.cpp
This test will ensure a round robin task can subscribe to a service, and will then wait. It will create a round robin task that will immediately subscribe to a service. We ensure it is waiting by verifying that it did not write to the trace buffer.

test021_publish_round_robin.cpp
This test verifies that round robin tasks can publish to a service and wake up other round robin tasks. We also ensure the order they are scheduled in is correct.

test022_publish_preempt_system.cpp
This test verifies that system tasks awoken from a service preempt the running round robin task. A round robin task will publish to a service that contains a system task. This system task should then preempt the round robin task that published, and run to completion before running the round robin task.

test023_subscribe_invalid_service.cpp
This test will attempt to subscribe to a service using an invalid service descriptor. We expect it to throw the error ERR_RUN_9_INVALID_SERVICE and blink the red LED on the board 9 times.

test024_service_interrupted.cpp
This test ensures that an interrupt service routine does not get preempted when publishing to a service. If an ISR publishes to a service that contains a system task, we expect the ISR to complete its task before switching contexts to the system task.

test025_publish_periodic.cpp
This test ensures that a periodic task can publish to a service. It will also ensure that tasks will be rescheduled in the appropriate order.

test026_rr_task_next.cpp
This test ensures that a round robin task can call Task_Next and switch contexts properly.

Profiling

Profiling was run on the RTOS to measure the overhead added by the new OS API functions. As shown in Figure 3, every new function added except the Now function switches contexts into the kernel to complete the request. It then becomes necessary for us to measure the timing issues this could cause.

Most of the profiling code was taken courtesy of Mike Krazanowski. The original source code can be found here and the report detailing his profile findings can be found here.

To measure the amount of time each new function took, we included various macros to send a HIGH or LOW signal to specific pins. To test any function, right before calling it we would send a HIGH signal to a pin. Then, once the function completed, we would send the LOW signal to the pin. By using a logic analyzer, we were able to measure how long a certain pin produced a HIGH signal.

Function Call	**Time Elapsed (microseconds)**
OS_Init	47.1975
Task_Create_System	43.9375
Task_Create_Periodic	52.6875
Task_Create_RoundRobin	43.8125
Task_Next	33.8125

Init	26.8750
Subscribe	34.9375
Publish	44.8750

Timed every call/significant event in the RTOS
Context Switching
Kernel Overhead

Conclusion

As per the documentation made available to use regarding old projects, we quickly found the problem while porting the RTOS to the ATmega2560, the inability to address all 256K of memory space. To remedy the problem we added the additional byte needed to address functions into our context switching code to ensure that it was also included in the switch.