Project 2 Report

CSC 460

Group #2

Shreyas Devalapurkar - V00827994

Rickus Senekal - V00826364

List of Figures and Tables

Abstract

The foundation of this report is based on the details surrounding the implementation and improvement of a Time-Triggered scheler. A basic scheduler with a time-triggered architecture (TTA) is practical in time-sensitive embedded environments due to its qualities of concurrency, simplicity, determinism, lack of race conditions, and low overhead. The definitions of each of these qualities can be found in the Project 2 Specification document [1]. The improvements focused upon for the TTA scheduler included support for urgent/sporadic events, dynamic execution of tasks, creation and deletion of sporadic tasks, and task state independence.

Some I/O events are not periodic and TTA only deals with periodic tasks, the ability to schedule non time-based tasks was a limitation that was addressed through our project. With our updated scheduler, task frequency can now be modified whilst the workflow has started instead of having to schedule all tasks statically before starting the scheduler. The final improvement to the original TTA scheduler includes creation of multiple instances of the same function where each instance has its own state. Through the completion of the improved scheduler detailed, the advanced abilities mentioned above are made possible.

1. Introduction

This section will lay the foundation in describing the remainder of the report's contents. It will outline the purpose of the report while discussing the scope and overview of the project. All major terms to be used throughout the document will also be defined in this section of the report.

1.1. Purpose

The contents of this report will discuss our group's implementation of project two. The report's contents can be applied to the entire project and are not limited to a single feature of the project. The purpose of this documentation is to outline our architecture for the project, explain in detail our implementation process, and discuss any issues we ran into while working on this project.

1.2. Project Overview and Scope

For this project, we were tasked with building on top of our existing time-triggered architecture (TTA) and addressing some of the limitations of this very simple architecture. Our primary goal was to build an even more powerful TTA scheduler with more functionality and features, while preserving all the desirable attributes of our original TTA scheduler [2].

The scope of this project was well outlined by Dr. Mantis Cheng (https://webhome.csc.uvic.ca/~mcheng/460/spring.2019/p2.html) [1] and this outline was used throughout the development process to ensure completeness and correctness of our program.

1.3. Glossary of Terms

Term	Description
API	Application programming interface. Defines a contract for client and backend to communicate (similar to middleware)
Preemption	To run in advance of something or someone else
TTA	Time-triggered architecture. A scheduling architecture used for executing code asynchronously and at predetermined rates

2. Discussion

This section will explain our project implementation in detail. It will present a system architecture overview and with the use of code snippets, logic analyzer timing diagrams and images, will thoroughly explain how our system was designed and built. We will also explore and discuss some of the challenges we faced throughout our journey.

2.1. System Architecture and API Overview

Our basic system architecture was based on an API that we defined. This API serves as a contract between the user/client and the scheduler service. In other words, a new user using our scheduler can look at the API that we expose and understand exactly what functionality our scheduler supports and how to access that functionality. The functionality and endpoints our API exposes are discussed below.

2.1.1. Initializing the Scheduler

In order to provide timing functionality within other scheduler functions, the scheduler must be initialized. Initializing the scheduler can be done via the API call to:

```
void Scheduler_init();
```

A call to this endpoint does not take in any parameters and this function does not return anything.

2.1.2. Adding a Periodic Task to the Scheduler

To add a task you would like to run periodically, an API call to the following endpoint can be made:

```
void Scheduler_add_periodic_task(int16_t delay, int16_t period, task_cb task, bool state);
```

Each call to this endpoint requires the following input arguments:

- **Delay**: Time till task is first executed
- **Task period**: How often the task will be executed
- **Callback to the task function**: Pointer to the function to be executed
- **Ready state**: State determining whether or not a task is ready to run

2.1.3. Adding a Non Time-Critical Sporadic Task to the Scheduler

A non time-critical sporadic task is one which executes as soon as enough idle time between periodic tasks is allotted. This task will not interfere with the runtime of any periodic tasks. It is assumed that any non time-critical sporadic task will take an extremely minimal time to run to completion. To add a non time-critical sporadic task to the scheduler, an API call to the following endpoint can be made:

```
void Scheduler_add_non_time_critical_sporadic_task(task_cb task, bool state);
```

Each call to this endpoint requires the following input arguments:

- **Callback to the task function**: Pointer to the function to be executed
- **Ready state**: State determining whether or not a task is ready to run

2.1.4. Adding a Time-Critical Sporadic Task to the Scheduler

Unlike the non time-critical sporadic task, for a time-critical task, a set delay must be designated after which the task will first be executed. Task duration must be specified in order to ensure that no interruption to the flow of periodic tasks occurs. To add a time-critical sporadic task to the scheduler, an API call to the following endpoint can be made:

```
void Scheduler_add_time_critical_sporadic_task(int16_t delay, task_cb task, bool state, uint16_t duration);
```

Each call to this endpoint requires the following input arguments:

- **Delay**: Time till task is first executed
- **Callback to the task function**: Pointer to the function to be executed
- **Ready state**: State determining whether or not a task is ready to run
- **Duration**: Calculated/estimated runtime of the respective task

2.1.5. Starting the Scheduler

Once all desired tasks have been added, the scheduler can be started via an API call to the endpoint:

```
void Scheduler_start();
```

This function is responsible for dispatching all periodic and sporadic tasks. A call to this endpoint does not take in any parameters.

2.1.6. Dispatching Periodic Tasks

The scheduler requires dispatching of the added tasks. This is done within the scheduler, where all added periodic tasks are dispatched in accordance to their specified input parameters when it is time for them to run.

```
uint16_t Scheduler_dispatch_periodic_task();
```

This function is executed inside *Scheduler_start()*. A call to this function returns the *idle_time* based on the execution of the periodic tasks.

2.1.7. Dispatching Non Time-Critical Sporadic Tasks

All added non time-critical sporadic tasks are dispatched in accordance to their specified input parameters when it is time for them to run. An argument of the *idle_time* based on the period and runtime of the periodic tasks will be passed into this function. This value will be used to ensure that a sporadic task cannot break the execution cycle of a periodic task.

```
void Scheduler_dispatch_non_time_critical_sporadic_task(uint16_t idle_time);
```

This function is executed inside *Scheduler_start()*.

2.1.8. Dispatching Time-Critical Sporadic Tasks

All added time-critical sporadic tasks are dispatched in accordance to their specified input parameters when it is time for them to run. An argument of the *idle_time* based on the period and runtime of the periodic tasks will be passed into this function. This value will be used to ensure that a sporadic task cannot break the execution cycle of a periodic task.

```
void Scheduler_dispatch_time_critical_sporadic_task(uint16_t idle_time);
```

This function is executed inside *Scheduler_start()*.

2.1.9. Deleting a Sporadic Task

In order to remove an added sporadic task from the scheduler when its functionality is no longer desired, an API call to the following endpoint can be made:

```
void Scheduler_delete_sporadic_task(task_cb task);
```

Each call to this endpoint requires the following input arguments:

- **Task**: Pointer to the task whose functionality is no longer desired

2.1.10. Dynamically Modifying the Period of a Periodic Task

Once a periodic task has been started, changing how often it is executed may be required. The period of a periodic task can be modified via an API call to the following endpoint:

```
void Scheduler_modify_periodic_task_period(task_cb task, int16_t modified_period);
```

Each call to this endpoint requires the following input arguments:

- **Task**: Pointer to the task for which the period needs to be modified
- **Modified period**: The new period to determine how often the task will now run

2.2. Detailed Implementation Overview

In order to implement the API specified in section 2.1, we first constructed task data structures. We decided to use 3 data structures; one for periodic tasks, one for time-critical sporadic tasks, and one for non time-critical sporadic tasks. The details of these structs and their properties are shown below in Figure 1.0.

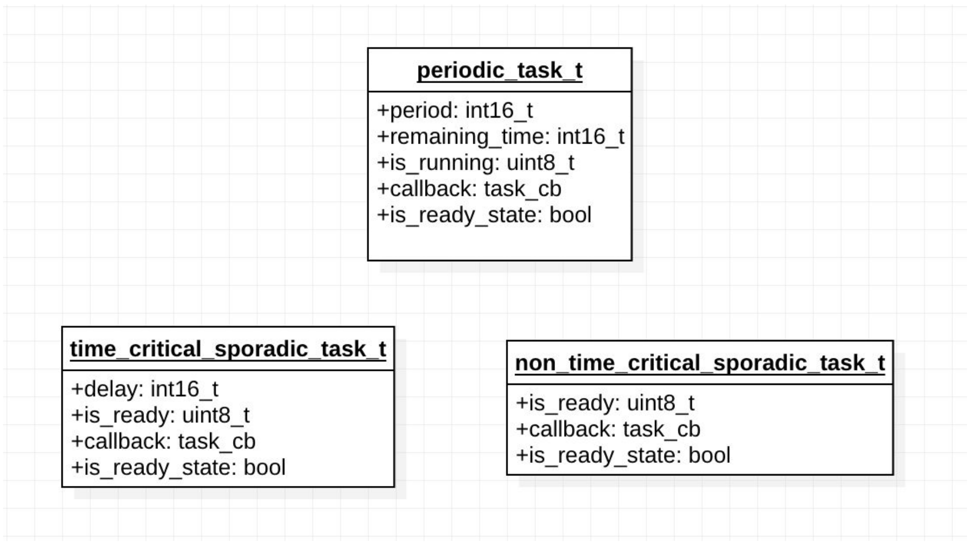


Figure 1.0 - Data structures for different types of tasks

These structs were necessary in order to store all the important information per different task type. This information was then used when it was time to execute or dispatch the respective tasks as will be discussed later on. Our architecture allowed for a maximum of 8 tasks of one type to be added at once. These tasks were stored in an array; one array per task type, each containing up to 8 tasks. The major pieces of functionality that we implemented for our advanced scheduler were as follows:

2.2.1. Initializing the Scheduler

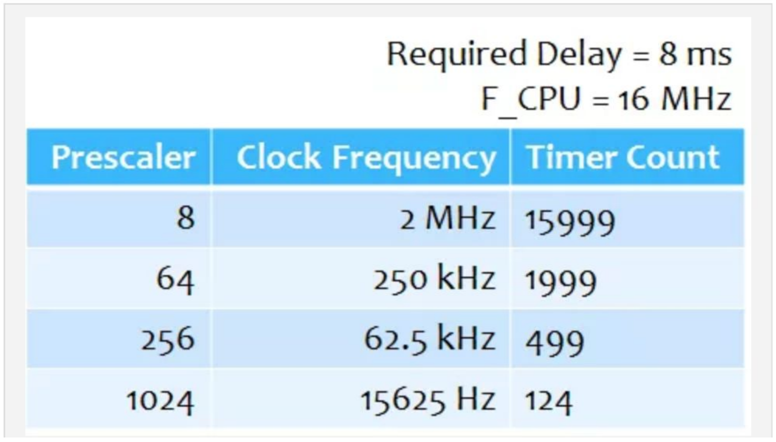
Timing resides at the center of the scheduler. In order for later operations to function correctly, two variables are initialized with the current execution time upon calling of the *Scheduler_init()* endpoint. The two variables include both a last runtime for period tasks, and a last runtime for sporadic tasks. In order to get the current execution time and count timing ticks at a particular rate, some calculations were done to determine our ideal prescaler value and timer count.

2.2.1.1. Calculating Timer Count

A very useful resource that we took advantage of in order to calculate what our exact timer count should be was an article by Mayank titled *AVR Timers: Timer 0* [3]. We wanted our required delay to be 10 ms. This would mean that if we schedule a task to run with a period of 100, it would effectively run every 10 * 100 = 1000 ms or 1 second. In order to achieve this delay we used the following formula:

Timer count = (Required delay / Clock time period) - 1

We were interested in calculating our necessary timer count. We decided to use Timer 0 (TCNT0) since we were most familiar with how it worked from past lab exercises. As specified in the article, when using timer 0 with an 8-bit counter (no prescaler), the maximum possible delay achievable is 8 ms [3]. Since we wanted to achieve a delay of 10 ms, we needed to implement a prescaler. Adding a prescaler allows the clock to effectively “run slower” and as a result, allows you to reach higher delay values. We decided to use a 1024 prescaler because the other options were above the maximum limit of an 8-bit counter as shown in figure 2.0.



Required Delay = 8 ms
F_CPU = 16 MHz

Prescaler	Clock Frequency	Timer Count
8	2 MHz	15999
64	250 kHz	1999
256	62.5 kHz	499
1024	15625 Hz	124

Figure 2.0 - Prescaler selection options

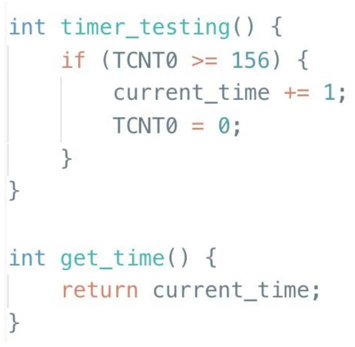
An 8-bit counter has a maximum limit of 255 and as can be seen in figure 2.0, the prescaler values of 8, 64 and 256 all have a timer count that exceeds that value. Thus, the 1024 prescaler value was used in order to achieve our 10 ms required delay. Now, our calculation was as follows based on the values in figure 2.0:

Timer count = (10 ms / 15625 Hz) - 1

= (10 ms / 0.064 ms) - 1

= **156**

This value was used to increment the clock time and get the current execution time within our tasks. The code for this can be seen in figure 3.0 below.



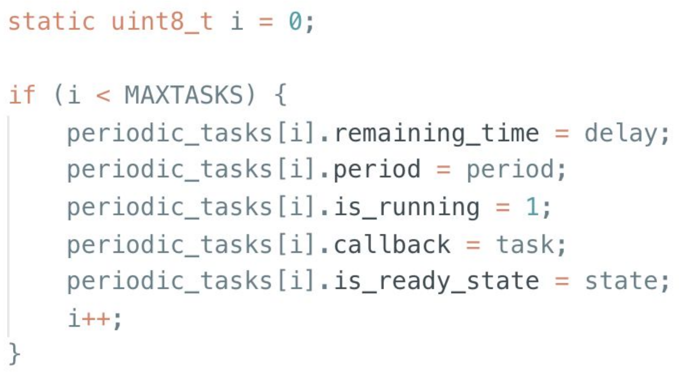
```
int timer_testing() {
    if (TCNT0 >= 156) {
        current_time += 1;
        TCNT0 = 0;
    }
}

int get_time() {
    return current_time;
}
```

Figure 3.0 - Getting current time based on timer count value

2.2.2. Adding a Periodic Task to the Scheduler

Scheduling of a single periodic task is done through the respective API endpoint as specified in Section 2.1.2. Each task is assigned a respective identification number from 0 to 7, depending on when it was added. As long as there are no more than 8 tasks of the same type currently scheduled, each periodic task is instantiated with the passed in arguments, as well as an *is_running* property, which is set to active. All this is shown in the code snippet presented in figure 4.0.



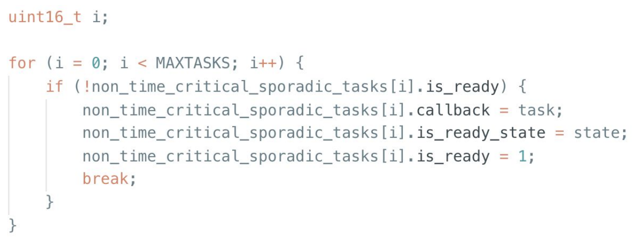
```
static uint8_t i = 0;

if (i < MAXTASKS) {
    periodic_tasks[i].remaining_time = delay;
    periodic_tasks[i].period = period;
    periodic_tasks[i].is_running = 1;
    periodic_tasks[i].callback = task;
    periodic_tasks[i].is_ready_state = state;
    i++;
}
```

Figure 4.0 - Adding a periodic task to the scheduler

2.2.3. Adding a Non Time-Critical Sporadic Task to the Scheduler

Scheduling of a non time-critical sporadic task is done through the respective API endpoint as specified in Section 2.1.3. Similar to the process of adding periodic tasks, up to 8 non time-critical tasks can be initialized with the passed in arguments as long as there is an available spot (8 non time-critical sporadic tasks are not already running). To prepare for dispatching of tasks, a property named *is_ready* is set to active. All this is shown in the code snippet presented in figure 5.0.



```
uint16_t i;

for (i = 0; i < MAXTASKS; i++) {
    if (!non_time_critical_sporadic_tasks[i].is_ready) {
        non_time_critical_sporadic_tasks[i].callback = task;
        non_time_critical_sporadic_tasks[i].is_ready_state = state;
        non_time_critical_sporadic_tasks[i].is_ready = 1;
        break;
    }
}
```

Figure 5.0 - Adding a non time-critical sporadic task

2.2.4. Adding a Time-Critical Sporadic Task to the Scheduler

Scheduling of a time-critical sporadic task is done through the respective API endpoint as specified in Section 2.1.4. For all time-critical sporadic tasks added via the API endpoint, if the task has not been started yet, it is initialized with the passed in arguments, as well as a property named *is_ready*, which is set to active. After the initial addition of a sporadic task, the for loop mentioned above is broken out of to prevent writing the same task into all 8 locations in the sporadic tasks array. All this is shown in the code snippet presented in figure 6.0.

```
uint16_t i;

for (i = 0; i < MAXTASKS; i++) {
    if (!time_critical_sporadic_tasks[i].is_ready) {
        time_critical_sporadic_tasks[i].delay = delay;
        time_critical_sporadic_tasks[i].callback = task;
        time_critical_sporadic_tasks[i].is_ready_state = state;
        time_critical_sporadic_tasks[i].is_ready = 1;
        time_critical_sporadic_tasks[i].duration = duration;
        break;
    }
}
```

Figure 6.0 - Adding a time-critical sporadic task

2.2.5. Starting the Scheduler

Once all desired tasks have been added to the scheduler, the timing counter is continuously incremented. The idle time between periodic tasks is calculated via dispatching of all periodic tasks. If idle time is greater than 0, meaning that there is time for sporadic tasks to be executed, the scheduler dispatches both time-critical and non time-critical sporadic tasks. All this is shown in figure 7.0.

```
while(1) {
    timer_testing();
    uint16_t idle_time = Scheduler_dispatch_periodic_task();

    if(idle_time) {
        Scheduler_dispatch_time_critical_sporadic_task(idle_time);
        Scheduler_dispatch_non_time_critical_sporadic_task(idle_time);
    }
}
```

Figure 7.0 - Starting the scheduler

2.2.6. Dispatching Periodic Tasks

As mentioned in Section 2.2.5, dispatching of tasks is done when the scheduler is started. For each periodic task in the respective array of tasks, the task's delay is reduced by the elapsed time. If the delay of a task reaches 0, a variable *t* is set as a pointer to the function passed in, a variable *state_val* is set to the *is_ready_state* of the respective task, and the delay of the task is reset to its defined period. If the periodic task is running, the idle time is set to 0 to ensure that no preemption occurs. If not, the idle time is overwritten with the minimum value of either the task remaining delay or current idle time. Once all tasks in the queue are updated, they are executed based on their currently active ready state as per the *state_val* variable. Finally, this function returns the idle time to be used by the sporadic task dispatchers. The detailed code implementation for this function can be viewed by checking out our source code as presented in the appendix.

2.2.7. Dispatching Non Time-Critical Sporadic Tasks

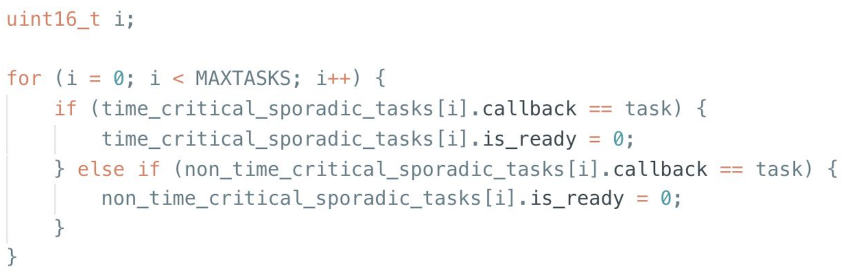
For each added non time-critical sporadic task, if the task has its *is_ready* state set to active, along with enough idle time available, a variable *t* is set as a pointer to the function passed in, a variable *state_val* is set to the *is_ready_state* of the respective task, and the task's *is_ready* state is set to 0. Once all tasks in the queue are updated, they are executed based on their currently active ready state as per the *state_val* variable. The assumption made here is that a non time-critical sporadic task will have a very short execution time/duration and thus should be able to run given any amount of idle time. The detailed code implementation for this function can be viewed by checking out our source code as presented in the appendix.

2.2.8. Dispatching Time-Critical Sporadic Tasks

Dispatching of time-critical sporadic tasks features a combined implementation from both periodic and non time-critical sporadic tasks. For each added time-critical sporadic task, the *is_ready* property is checked for confirmation, the idle time is ensured to be greater than 0, and a check is done to see if the sporadic task's duration is not longer than the current available idle time. This last check allows for the avoidance of preemption. With all requirements met, the task's delay is reduced by the elapsed time. Once the delay equals or goes below 0, the same updating procedure as per the dispatching of non time-critical sporadic tasks occurs. When all tasks in the queue have been updated, they are executed based on their currently active ready state as per the *state_val* variable. The detailed code implementation for this function can be viewed by checking out our source code as presented in the appendix.

2.2.9. Deleting a Sporadic Task

As detailed in Section 2.1.9, the task up for deletion is passed to the respective API endpoint. The sporadic task array is parsed until the matching task is found based on the task callback property, which is a pointer to the function. Once the desired task is found, its *is_ready* property is set to 0, which effectively opens up that slot in the array of sporadic tasks and sets it as a candidate for replacement when a new task is added to the task array. This is shown in figure 8.0 below.



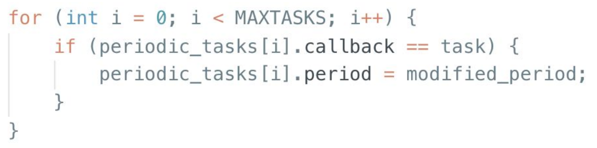
```
uint16_t i;

for (i = 0; i < MAXTASKS; i++) {
    if (time_critical_sporadic_tasks[i].callback == task) {
        time_critical_sporadic_tasks[i].is_ready = 0;
    } else if (non_time_critical_sporadic_tasks[i].callback == task) {
        non_time_critical_sporadic_tasks[i].is_ready = 0;
    }
}
```

Figure 8.0 - Deleting a sporadic task from the array of tasks

2.2.10. Dynamically Modifying the Period of a Periodic Task

Using the same method as deletion of sporadic tasks, how often periodic tasks are executed can be modified dynamically. Using the task function and desired period passed in as arguments, the periodic task array is parsed through, assigning the newly modified period to the matched task's period property. This is shown in figure 9.0 below.



```
for (int i = 0; i < MAXTASKS; i++) {
    if (periodic_tasks[i].callback == task) {
        periodic_tasks[i].period = modified_period;
    }
}
```

Figure 9.0 - Modifying the period of a periodic task

2.2.11. Adding Task Instances

In order to allow multiple tasks (instances) to execute the same code, we enable the passing of a state variable into our task functions. This state variable in our case is set to a boolean value and if the boolean value is true, the function executes when the task is called. Otherwise it does not. This allows for different types of behaviour from the same function that can be called by various different tasks.

2.3. Testing and Profiling

In order to test the functionality of our enhanced TTA scheduler, we designed and wrote many test programs. These test programs try to schedule a variety of different tasks and by examining the timing output of these tasks being run, we were able to verify whether or not our scheduler was behaving as expected. Some of the test cases we created are outlined below.

2.3.1. Scheduling a Single Periodic Task

In this test scenario, we wanted to ensure that our scheduler could handle scheduling a very basic periodic task. What we were looking for was that our task was being run at a specified frequency.

```
Scheduler_start_periodic_task(0, 100, pulse_pin3_task, true);
```

Figure 10.0 - Starting a single periodic task

As shown in figure 10.0, for this test, a single periodic task was started and its period was set to 1 second. The function *pulse_pin3_task* sets a digital pin to high and low in order for us to visualize how often the task was being run. The timing output is shown in figure 11.0.

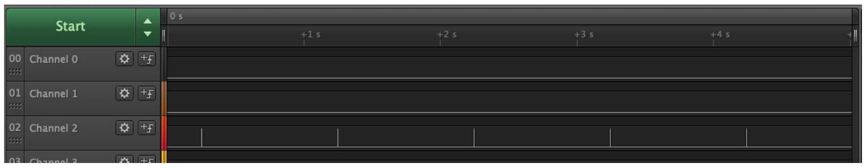


Figure 11.0 - Logic analyzer timing output for a single periodic task

As seen above in figure 11.0, channel 2 is the output for our periodic task and it can be seen that it runs consistently every second. Our periodic task scheduler works!

2.3.2. Scheduling Multiple Non-Overlapping Periodic Tasks

In this test scenario, we wanted to ensure that our scheduler could handle multiple periodic tasks being scheduled to run without overlapping with each other. This means that when it is time for them to run, they consistently run at different times.

```
Scheduler_start_periodic_task(0, 100, pulse_pin1_task, true);
Scheduler_start_periodic_task(10, 100, pulse_pin2_task, true);
```

Figure 12.0 - Scheduling two non-overlapping periodic tasks

As shown in figure 12.0, for this test, multiple periodic tasks were started. The first task started at time 0 and had a period of 1 second. The second task started with an offset of 100 ms and also had a period of 1 second. The functions *pulse_pin1_task* and *pulse_pin2_task* set digital pins to high and low in order for us to visualize how often the tasks were being run. The timing output is shown in figure 13.0.

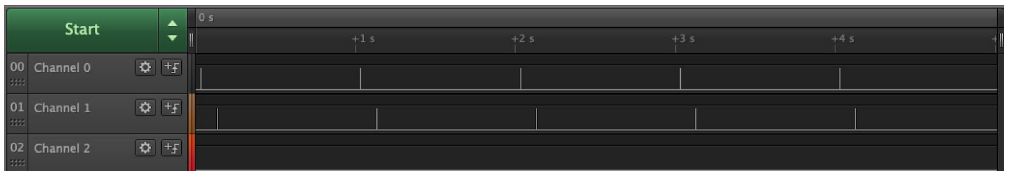


Figure 13.0 - Timing diagram of multiple non-overlapping periodic tasks

As seen above, our periodic tasks run consistently every second with their specified offsets and do not conflict with one another. Channel 0 indicates our first periodic task and channel 1 indicates our second task. It can be seen that the second task consistently runs 100 ms after the first one. Our periodic task scheduler works with multiple tasks!

2.3.3. Scheduling a Non-Time-Critical Sporadic Task

In this test scenario, we wanted to see whether we could successfully schedule a non time-critical sporadic task in between two periodic tasks when the sporadic task has enough time to execute. In order to simulate this environment, we started up two periodic tasks with a small offset from one another. Within the first periodic task, a non time-critical sporadic task was started up which tried to run as soon as it could find enough idle time to do so. Figures 14.0 and 15.0 show code snippets for our test setup.

```
Scheduler_start_periodic_task(0, 100, pulse_pin1_task, true);
Scheduler_start_periodic_task(10, 100, pulse_pin2_task, true);
```

Figure 14.0 - Starting up two periodic tasks, second one with an offset

```
void pulse_pin1_task(bool is_ready_state) {
    Scheduler_add_non_time_critical_sporadic_task(pulse_pin3_task, true);

    if (is_ready_state) {
        PORTG |= 0x20;
        PORTG &= ~0x20;
    }
}
```

Figure 15.0 - Functionality of the first periodic task

As shown in figure 15.0, when the first periodic task runs it adds a non time-critical sporadic task. All three functions *pulse_pin1/2/3_task* set digital pins to high and low in order for us to visualize how often the tasks were being run. The timing output is shown in figure 16.0.

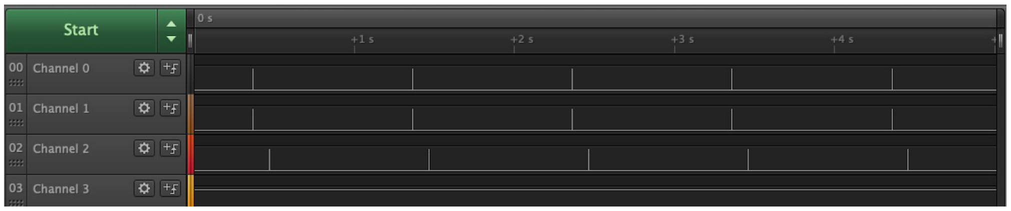


Figure 16.0 - Timing diagram of 2 periodic tasks

As seen in figure 16.0, our first periodic task (indicated by channel 1) runs first and then in the idle time of 100 ms that exists before the second periodic task can run (indicated by channel 2), our sporadic task runs (as shown in channel 0). This continues for every period since the sporadic task is being created every time the periodic one is. This proves that our non time-critical sporadic tasks are able to run as soon as they have enough idle time to run to completion, and that they work well with periodic tasks.

2.3.4. Scheduling Multiple One-Time Sporadic Tasks

In this test scenario, we wanted to ensure that we could create and dispatch multiple one-time sporadic tasks. To test this, we simply created two time-critical sporadic tasks that were planned to run after certain delays as shown in figure 17.0.

```
Scheduler_add_time_critical_sporadic_task(150, pulse_pin3_task, true, 5);
Scheduler_add_time_critical_sporadic_task(450, pulse_pin3_task, true, 5);
```

Figure 17.0 - Creating two time-critical sporadic tasks

The *pulse_pin3_task* function sets a digital pin to high and low in order for us to visualize how often the function and task was being run. The timing diagram is shown in figure 18.0.

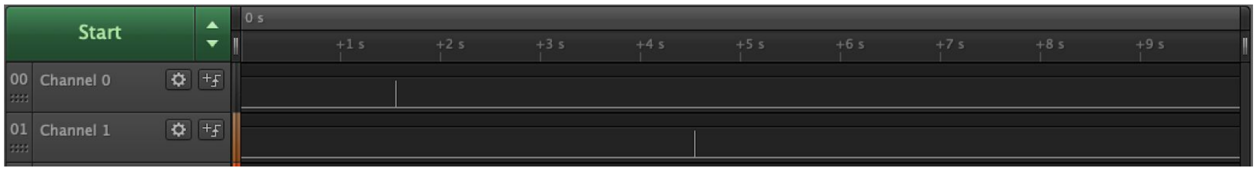
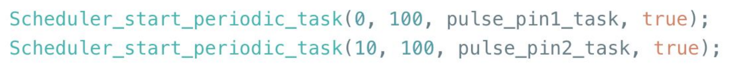


Figure 18.0 - Logic analyzer results for two sporadic tasks

As shown in figure 18.0, we see that both our sporadic tasks only run once. Our first sporadic task (channel 0) runs at its specified offset which is 1.5 seconds and the second task (channel 1) runs at its specified offset which is 4.5 seconds. This shows that we can schedule multiple one-shot sporadic tasks!

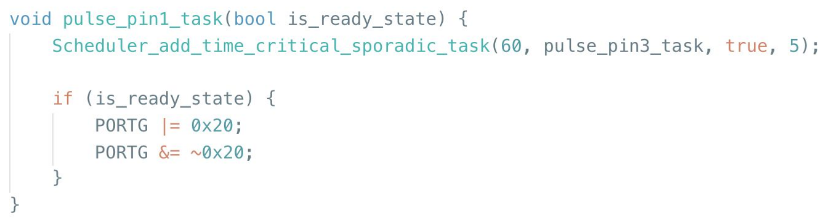
2.3.5. Deleting Sporadic Tasks

In this test scenario, we wanted to test that we can dynamically delete a sporadic task and disable it from being run once it has been created. To simulate this environment, we created two periodic tasks. The first periodic task would run immediately and the second would run in 100 ms time. Within the first periodic task, we added a time-critical sporadic task which would run in 600 ms, but within the second periodic task we deleted the sporadic task that was created. All this is shown in figures 19.0, 20.0, and 21.0 below.



```
Scheduler_start_periodic_task(0, 100, pulse_pin1_task, true);
Scheduler_start_periodic_task(10, 100, pulse_pin2_task, true);
```

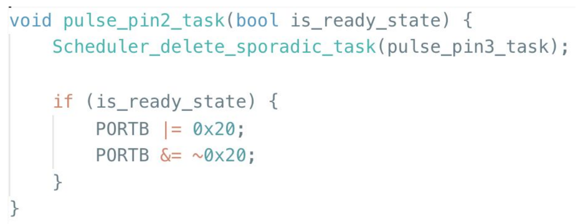
Figure 19.0 - Starting two periodic tasks



```
void pulse_pin1_task(bool is_ready_state) {
    Scheduler_add_time_critical_sporadic_task(60, pulse_pin3_task, true, 5);

    if (is_ready_state) {
        PORTG |= 0x20;
        PORTG &= ~0x20;
    }
}
```

Figure 20.0 - Adding a time-critical sporadic task to run in 600 ms within a periodic task



```
void pulse_pin2_task(bool is_ready_state) {
    Scheduler_delete_sporadic_task(pulse_pin3_task);

    if (is_ready_state) {
        PORTB |= 0x20;
        PORTB &= ~0x20;
    }
}
```

Figure 21.0 - Deleting the created sporadic task

Since the second periodic task will run before the sporadic task has a chance to run, the sporadic task will get deleted. Thus, we expected to see no output from the pin that the sporadic task was setting to high and low. The logic analyzer output can be seen in figure 22.0 below.

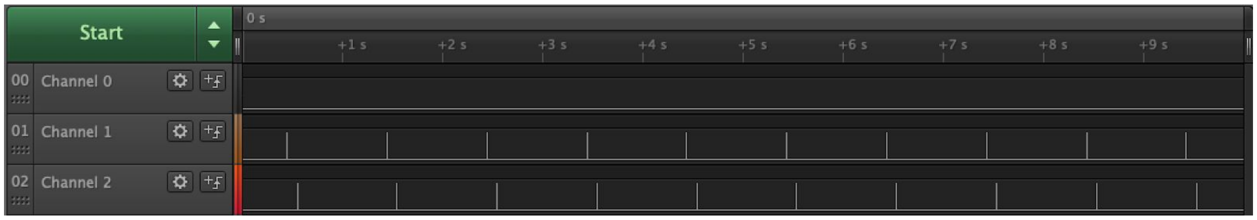


Figure 22.0 - No sporadic task execution occurs post deletion

Channel 0 represented the sporadic task and as can be seen above, it was never run since it was deleted. The periodic tasks; however, run perfectly fine in their specified periods as expected. Deleting sporadic tasks works!

2.3.6. Sporadic Task Taking Too Long to Run

This was a very important test scenario. Here, we wanted to make sure that our scheduler does not allow a sporadic task to consistently break the execution of a periodic task and cause a repeated timing error/jitter. To simulate such a scenario, we started up a periodic task with a period of 1 second, and then within the periodic task added a sporadic task with a duration of 4 seconds. It is very clear that if we let the sporadic task to run to completion, it will consistently cause the periodic task to miss its execution time because the periodic task needs to run every 1 second. The set up for this test case was as shown in figures 23.0 and 24.0.

```
Scheduler_start_periodic_task(0, 100, pulse_pin1_task, true);
```

Figure 23.0 - Starting a single periodic task with a period of 1 second

```
void pulse_pin1_task(bool is_ready_state) {
    Scheduler_add_time_critical_sporadic_task(200, pulse_pin3_task, true, 400);

    if (is_ready_state) {
        PORTG |= 0x20;
        PORTG &= ~0x20;
    }
}
```

Figure 24.0 - Adding a sporadic task within the periodic task with a long duration

The expected behaviour was for the sporadic task to never be allowed to run. This is because a periodic task should always take priority over a sporadic task and in this case, that property would be violated if the sporadic task was given permission to run. The timing output for this test case is shown in figure 25.0.

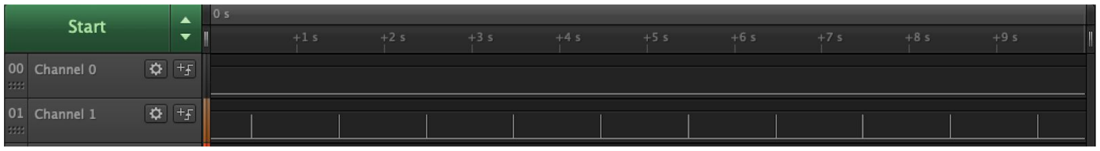


Figure 25.0 - Sporadic task does not run when it takes too long

As can be seen above, our periodic task runs every second (channel 1) but the sporadic task (channel 0) never runs. This shows that our scheduler honours the priority of periodic tasks and does not let a sporadic task break the periodicity of a periodic task, which is very important.

2.3.7. Dynamically Modifying the Period of a Task

In this test scenario, we were interested in checking that our scheduler supports setting a dynamic task schedule. In other words, we wanted to be able to manually update the period of a periodic task once it has begun execution. In order to achieve this, we set up a periodic task that would run every 100 ms. We also added a time-critical sporadic task which would run in 1.5 seconds time. This sporadic task's function was to modify the period of the periodic task from 100 ms to 1.5 seconds. The setup for this scenario is shown in figures 26.0 and 27.0.

```
Scheduler_start_periodic_task(0, 10, pulse_pin1_task, true);
Scheduler_add_time_critical_sporadic_task(150, sauce_func, true, 5);
```

Figure 26.0 - Adding a periodic and sporadic task

```
void sauce_func(bool is_ready_state) {
    if (is_ready_state) {
        Scheduler_modify_periodic_task_period(pulse_pin1_task, 150);
    }
}
```

Figure 27.0 - Function that sporadic task calls to modify the period of the periodic task

The logic analyzer output for this test case is shown below in figure 28.0 and it clearly shows how beautifully this test case worked out for us.

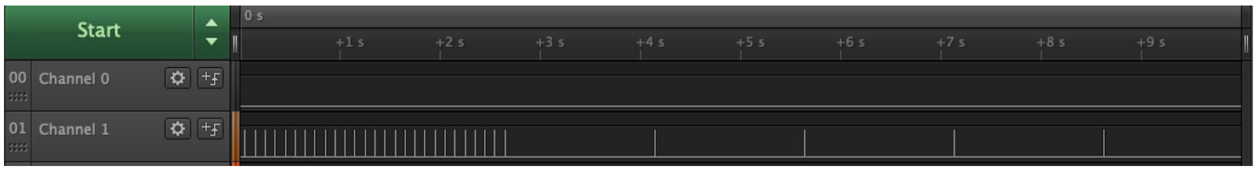


Figure 28.0 - Periodic task with a modified period

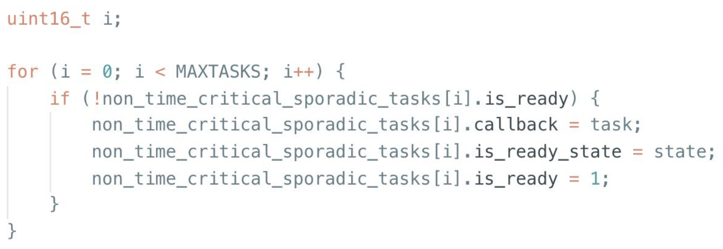
As you can see, initially the periodic task (channel 1) runs every 100 ms but once the sporadic task runs and causes the periodic task's period to change, the frequency at which the periodic task runs changes to once every 1.5 seconds. Dynamic task schedules work too! Yay!

2.4. Challenges and Blockers

We faced many challenges throughout the development of this project. Some of the main and interesting cases are documented below.

2.4.1. Inability to Create Multiple Sporadic Tasks

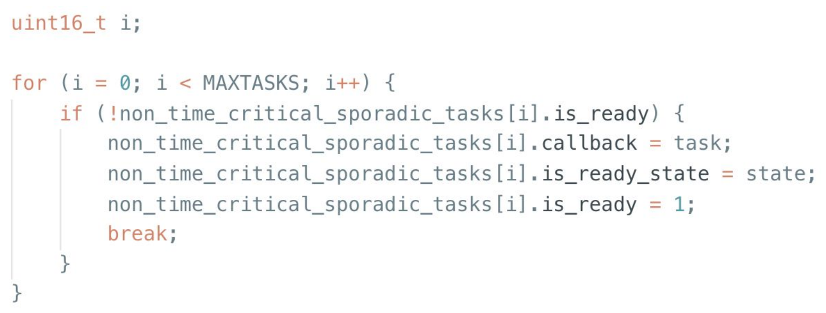
The way we implemented the addition of sporadic tasks initially was to just loop through our task array and identify a slot where a task's *is_ready* state is set to 0 (false) meaning that it is inactive and then add our new task to that slot. The issue we were seeing was that when we tried to add more than one sporadic task, none of the sporadic tasks would execute. This turned out to be due to a simple issue in our code and this is illustrated in figures 29.0 and 30.0.



```
uint16_t i;

for (i = 0; i < MAXTASKS; i++) {
    if (!non_time_critical_sporadic_tasks[i].is_ready) {
        non_time_critical_sporadic_tasks[i].callback = task;
        non_time_critical_sporadic_tasks[i].is_ready_state = state;
        non_time_critical_sporadic_tasks[i].is_ready = 1;
    }
}
```

Figure 29.0 - Code with a bug in sporadic task addition



```c
uint16_t i;

for (i = 0; i < MAXTASKS; i++) {
    if (!non_time_critical_sporadic_tasks[i].is_ready) {
        non_time_critical_sporadic_tasks[i].callback = task;
        non_time_critical_sporadic_tasks[i].is_ready_state = state;
        non_time_critical_sporadic_tasks[i].is_ready = 1;
        break;
    }
}
```

Figure 30.0 - Code with the bug fixed in sporadic task addition

If these two snippets of code are examined carefully, it is evident that they are almost identical. The only small difference is in the *break* which is present in the second case and not in the first case. What we were initially doing every time we wished to add a sporadic task was that we were looping through the entire array and assigning all empty slots to that task instead of only assigning the first empty slot we found. As a result, when multiple tasks were added, there were no empty slots for the new tasks even though we were expecting there to be some. Adding this single *break* statement allowed us to exit the loop we were in immediately after we identify an empty slot and add our task to the array successfully.

2.4.2. Ensuring a Non-Preemptive Behaviour for Sporadic Tasks

Initially our scheduler did not honour the priority of a periodic task in the sense that it would allow a sporadic task to preempt a periodic task. We were checking to ensure that there was idle time present for a sporadic task to run, but the issue was when the execution of a sporadic task took too long and overflowed into the execution time of a periodic task. In order to account for this tricky case, we added a parameter to our sporadic tasks called the duration. Now, the user can specify how long their sporadic task may take and based on that value, our scheduler will determine the ideal time to dispatch that task such that it does not cause any preemption. It took us some time to think of this test case and also to come up with a solution that honoured the priority of the periodic tasks.

3. Conclusion

Through this project, we were able to learn how a TTA scheduler works and what its limitations are. We were able to modify our existing scheduler to address these limitations and add new features. Some of such features included supporting the addition and deletion of sporadic or urgent tasks that are not based on a timed schedule, creating task functions to run as multiple different instances, and support for dynamic task schedules.

We thoroughly enjoyed working on this project despite its many tricky parts and we are proud of what we were able to accomplish. We look forward to the upcoming project 3 wherein we will be able to put both our project 1 and 2 work to use to build a cool robot that can defend its castle and throne. We would like to acknowledge Prashanti Priya Angara, our lab instructor (TA) for all her help and guidance. We are also thankful to Dr. Mantis Cheng for all his teachings. Finally, we are thankful to Adam Leung for all his help discussing project concepts and ideas with us, and for helping us find some interesting bugs in our code.

4. References

[1] M. Cheng, “CSC 460/560 Project 2,” Feb. 2019. [Online]. Available: https://webhome.csc.uvic.ca/~mcheng/460/spring.2019/p2.html. [Accessed Mar. 1, 2019].

[2] N. Macmillan, “Lab Guide: Time-Triggered Scheduling,” Feb. 2011. [Online]. Available: https://nrqm.ca/mechatronics-lab-guide/lab-guide-time-triggered-scheduling/. [Accessed Mar. 1, 2019].

[3] Mayank, “AVR Timers: Timer 0,” Jun. 24, 2011. [Online]. Available: http://maxembedded.com/2011/06/avr-timers-timer0/. [Accessed Mar. 2, 2019].

A. Appendix

This section will outline any additional information or resources necessary to supplement the contents of the report.

A.1. Source Code

The code for our enhanced TTA scheduler can be found at:
https://github.com/sdevalapurkar/legendary-roomba/tree/master/project2