CSC 460 Project 2

Spring 2019

Team 13

Hamzah Mansour	V00812108
Christina Araya	V00781339

Table of Contents

List of Figures

List of Tables

1 Abstract

The content of this report describes the implementation, usage and testing of a TTA scheduler implemented for project 2 of the CSC 460 course offered in University of Victoria in the Spring term. The TTA scheduler has three designated task types, periodic, system and low-priority tasks. The periodic tasks run at a set period and the others only run once. The program takes into consideration the applications of the tasks, including arguments to differentiate tasks run with the same callback function. It also considers several error scenarios, including misuse of the program, and minor recoverable errors. In section 4.1 test cases are enumerated and results are explained. There are six test cases in total that are addressed in the report, a successful run that has periodic, system and low priority tasks running smoothly without interruption. A scenario in which a system or low priority task executed longer than expected, and started to interfere with periodic scheduling. As well as a test in which a system task was missed, and one where a low priority task was missed. There were also two tests involving timing conflicts, a major and minor time conflict of periodic tasks.

The aim of the project was to implement an improved TTA scheduler, urgent events were handled by adding separate scheduling and queuing for different task priority. Interrupt handlers could not modify data shared with time based tasks and were kept to a minimum. Time based tasks were considered the highest priority and were therefore not interruptible. The temporary or one-shot tasks could be created on the fly, but there was no specified interface to then delete the one-shot task. However, if given access to the list, the user may remove tasks from the front of the list and therefore delete oneshot tasks. Also callbacks can be run as multiple instances under the current implementation.

Difficulties encountered during the making of the TTA scheduler involved differing development environments, over complications with memory management systems leading to unpredictable behaviour and schedule designation errors. Some compiler features that were originally expected were found to be unavailable and required explicit implementation rather than simple library inclusions. There was also some additional work that could be resolved involving the delayed scheduling of oneshot tasks that was never completed.

2 Project Introduction

The objective of the project[1] is to implement a Time-Triggered Architecture Scheduler (TTA scheduler) to be used on an ATMega2560 microcontroller, unlike the TTA scheduler provided in project 1 this one will have some improvements. The TTA scheduler will have one shot events/interrupts, parameters can be added to the tasks and there are priorities for different tasks. These will be in addition to the baseline implementation for the original project 1 TTA scheduler which include a periodic time based scheduler that assigns tasks ahead of time with low CPU overhead. The scheduler should allow for no race conditions and periods should be

deterministic. The scheduling should be simple and robust, misuse of the scheduler should be addressed appropriately depending on the scenario. It should also be possible to create multiple task-like functions.

The goal of the project 2 scheduling program is to address the limitations of the project 1 implementation and create a program that meets all the above specifications. There also are a number of questions that need to be addressed in the construction of the scheduler:

1. How can we add interrupts(or urgent events into our TTA scheduler?
2. Should time-based tasks be interruptible?
3. Could interrupt handlers be allowed to modify data shared with time-based tasks?
4. Could a one-shot (i.e. executed once) task be created/deleted on the fly without a predefined timing schedule?
5. Could a task function be created to run as multiple instances?

Additionally, to make sure that the proposed scheduler is robust against any misuse of the software, both recoverable and non-recoverable errors must be addressed. Some examples of these can include, timing conflict between oneshot and periodic tasks, jitter of time based tasks caused by interrupts, periodic task collisions, excessive oneshot tasks and tasks preventing periodic tasks from running.

3 Procedure

3.1 Periodic task specifications

For a TTA scheduler periodic tasks are scheduled at set intervals, in this project these tasks have periods, remaining time to execute, a callback to the function and a list of arguments. This can be seen in the structure used:

```
typedef struct // periodic tasks
{
      int32_t period;
      int32_t remaining_time;
      uint8_t is_running;
      task_cb callback;
      LinkedList<arg_t> args;
} task_t;
```

Where the arguments are contained in a linked list, whose specifications are visited later. Additionally the struct contains an integer is_running that can be used to stop the execution of the periodic task.

3.2 One-shot task specifications

One-shot tasks (both system tasks and low-priority tasks) are defined the same way and in the execution of the system work in a similar fashion as well. One-shot tasks, like periodic tasks, have a linked list of arguments, a callback function, an indicator showing it is running, and remaining time. However, the is_running specifier cannot be used to turn off the one-shot task, and a max time is included to set the maximum running time of the system. In order to differentiate the specification of a system or low-priority task, where the system task has a higher priority than the low-priority task, there is a priority specified in the struct. When the priority is set to one, it is used as a system task, otherwise it is a low-priority task.

```
typedef struct { // one-shot tasks
      int32_t remaining_time;
      int32_t max_time;
      uint8_t is_running;
      task_cb callback;
      int priority;
      uint32_t id;
      LinkedList<arg_t> args;
} oneshot_t;
```

3.3 Argument specifications

Arguments are used by all of the task specifications, where a linked list of them is used to pass information to the different functions. The structure of each of the data nodes in the task argument lists is simple. In order to identify what the input or output data pins are used for the tasks, a port letter and integer pin number is in the argument struct. This way if a task is run multiple times with different output pins, the pins can be specified in the argument for the task, therefore differentiating tasks that run the same callback function.

```
typedef struct {
      char port;
      uint8_t pin;
} arg_t;
```

3.4 Initialization

Setting up the project for execution requires a few functions in the main file, as well as some function calls in a specific order. One of the first things that the main function should call in order to setup the scheduler is Scheduler_Init(), which sets up the interrupt timer used for the schedule timing. Each interrupt tick takes approximately 1ms and each time the interrupt is hit a global current_tick which represents the system time. Periodic tasks and oneshot tasks may be scheduled in the setup after initialization, and following this an infinite loop that calls the periodic dispatch function is required. The loop should also call an idle function that calls the oneshot task dispatch function.

The main file excluding tasks and scheduling calls can be defined like this:

```
void idle(uint32_t idle_time)
{
	// It should return before the idle period (measured in ms) has
expired.
	Scheduler_Dispatch_Oneshot();
}

void setup()
{
	Scheduler_Init();
	//start offset in ms, period in ms, function callback
	//call to schedule here
}

void loop()
{
	uint32_t idle_time = Scheduler_Dispatch_Periodic();
	if (idle_time)
	{
		idle(idle_time);
	}
}

int main(){
	setup();
	for (;;){
		loop();
	}
	for (;;);
	return 0;
}
```

The Scheduler_Init() function is implemented as follows:

```
// setup our timer
void Scheduler_Init()
{
	current_tic = 0;
	Disable_Interrupt();
	//Clear timer config.
	TCCR1A = 0;
	TCCR1B = 0;
	//Set to CTC (mode 4)
	TCCR1B |= (1<<WGM12);

	TCCR1B |= (1<<CS10);
	TCCR1B |= (1<<CS12);

	//Set TOP value 10ms
	OCR1A = 16;

	//Enable interupt A for timer 3.
	TIMSK1 |= (1<<OCIE1A);

	//Set timer to 0 (optional here).
	TCNT1 = 0;
	Enable_Interrupt();

}
```

3.4.1 Periodic tasks

Initializing periodic tasks is done by calling Scheduler_StartPeriodicTask which takes a delay, period, task and a list of arguments. There are a maximum of 8 periodic tasks available at a time, periodic tasks cannot be removed from the periodic task list. The delay variable refers to the offset from the initialization of the task until the task is called. The period refers to the number of ticks needed to pass between executions of the callback function. The task variable specified is a callback defined in the following way:

```
///A task callback function
typedef void (*task_cb)(LinkedList<arg_t>&);
```

Scheduling for periodic tasks will start running immediately after the execution of this function usually placed in the setup() call:

```
//start offset in ms, period in ms, function callback
```

```
Scheduler_StartPeriodicTask(0, 200, taskA, obj);
```

3.4.2 One-shot tasks

Initializing one-shot tasks requires a call to Schedule_OneshotTask which takes the parameters remaining_time, max_time, callback, priority, and a list of arguments. There is no maximum number of one-shot tasks; one-shot tasks are assigned to two different linked lists and are assigned according to priority. The remaining time variable was meant to be used as a delayed start for a one-shot task, however it wasn't a critical feature and at the moment, doesn't affect the project. The max_time variable is meant to capture the number of ticks needed to execute the function, and is used to limit when one-shot tasks are called as to not overlap with periodic tasks. The callback variable refers to the same kind of variable as defined in the periodic scheduling function. The schedulting for the oneshot tasks will start after the execution of this function:

```
Schedule_OneshotTask(10,10,taskC,0,obj1 );
```

3.4.3 Scheduling decisions

At the start of every idle loop, a check is performed using the function Scheduler_Dispatch_Periodic to call a periodic task if available and update the available idle time. Then in the idle task a check is performed every loop to check for and run oneshot tasks. The one-shot tasks are scheduled as shown in the following figure:

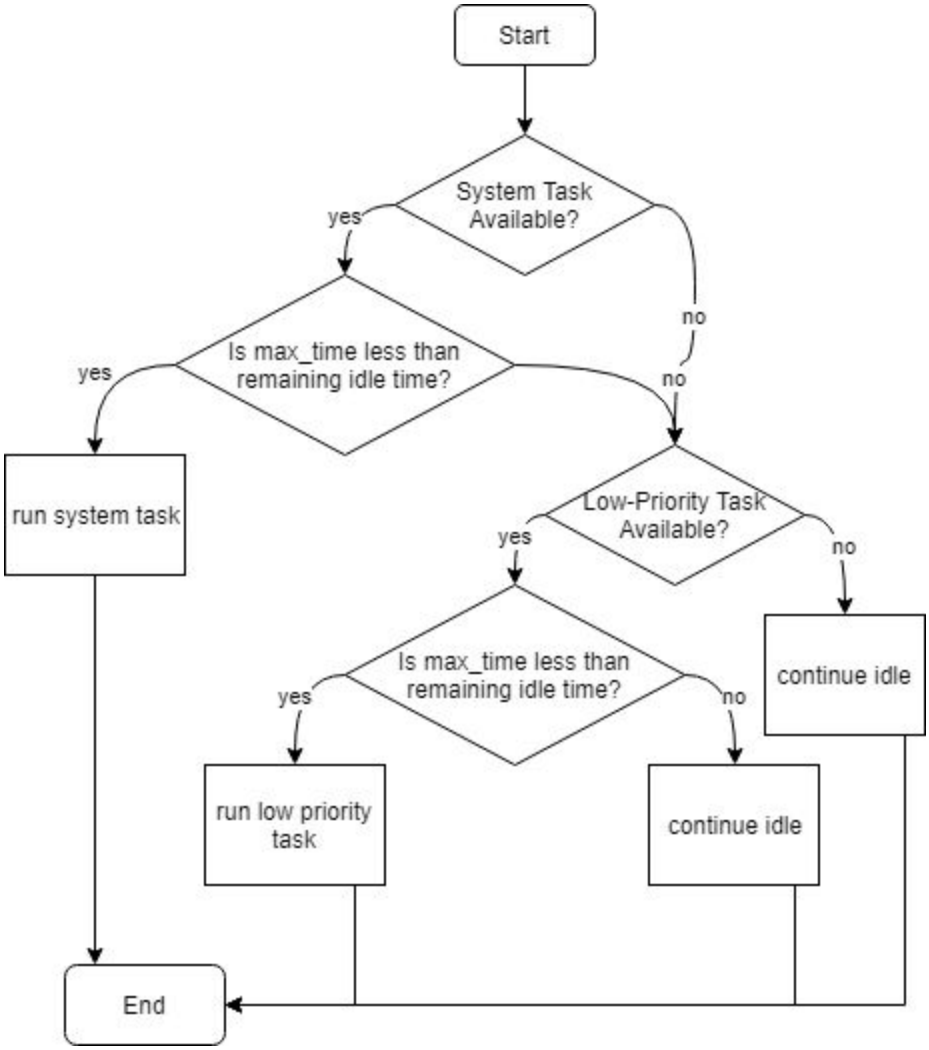

Figure 1: Scheduling Flowchart

Periodic tasks will always be at a higher priority than one-shot tasks, and if the available remaining idle time (remaining time before a periodic task) is insufficient to run a task then the task won't be run.

3.4.4 Error Handling

For this project a number of possible errors were considered, including possible misuse of the system, and handling of minor scheduling conflicts. Depending on the severity of the errors present in the execution the scheduler either exits execution or corrects for the error.

The first error considered was when the max_time variable being set to a smaller interval than the actual run time. This was considered to be a case of misuse of the system, and in this case the one-shot task could overlap with the periodic task execution. Because this was a case of misuse, when this kind of scheduling would occur, if the periodic tasks weren't affected the tasks were left alone. However, when the one-shot task started to overlap with the timing of the

periodic task the scheduler terminates. The check for this error occurs in the timing interrupt and is checked every tick.

Another severe error could be produced when the periodic tasks were given an impossible schedule. This case is also considered to be an example of misuse of the TTA scheduler. For this case when a timing conflict occurs between periodic tasks a time conflict counter is incremented and if the counter exceeds a certain value the scheduler terminates. The check for this error occurs once every tick as well in the interrupt.

When the one-shot tasks were scheduled there were two errors that could occur depending on the priority of the associated task. If a system task failed to run enough times during the execution of the schedule the system exits. This is achieved by counting a variable every time a periodic task is scheduled, if this value exceeds a certain amount it is considered a critical failure and the scheduler is stopped. The other error is when a lower priority task is scheduled and it fails to execute after a few periodic tasks have finished. If this occurs similarly to the system tasks a counter is used to check how many periods have passed, if too much time has passed the low priority task is popped off the list. After the offending task is removed, the system can continue execution.

If the periodic tasks have a minor scheduling conflict that can be resolved, the execution of the periodic tasks get offset and the system can continue execution as desired.

4 Results/Experiments

Testing the scheduler was done by creating set test cases that would create a different number of tasks with varying arguments, changing the offset, period, and the tasks being called. To do this, ports on the ATMega2560 were selected as output pins, as shown in “LED_Test.c”. The values of these ports was then set to high at the start of a task’s execution, then low once the task had fully finished execution. By doing so, the execution of all of the tasks, the timing of the tasks (their total running time), as well as how the scheduler prioritized tasks could be seen clearly by using a logic analyzer to study the output of all of the pins at once.

4.1 Testing

4.1.1 Test cases

The following test cases were developed to be a way to exhaustively test the functionality and behaviour of the scheduler.

Table 1: Test cases

Test #	Test case function	Description
1	`test_success`	Tests that the scheduler correctly handles multiple periodic and one-shot tasks (both system tasks and low-priority tasks)
2	`test_toolong`	Tests that the scheduler exits with an error indicating that a scheduled one-shot task is taking too long to complete execution
3	`test_misssystem`	Tests that the scheduler will eventually exit with an error if a system task misses execution too many times (>10)
4	`test_missoneshot`	Tests that a low-priority one-shot task is taken off the task queue if it misses execution too many times (>5)
5	`test_impossibleschedule`	Tests that the scheduler will exit with an error if there are overlapping and impossible-to-schedule periodic tasks
6	`test_timeconflict`	Tests that the scheduler will correctly schedule periodic tasks with overlapping or conflicting execution times

4.1.2 Test analysis

4.1.2.1 test_success

This test was written to schedule two periodic tasks (TaskA and TaskB), that would run at the same period, with a reasonable offset. TaskA also would schedule three one-shot tasks (TaskC twice and TaskD once) where one of the TaskC tasks created was a system task. TaskD would then call two low-priority one-shot TaskC tasks.

In order to easily visualize the execution of these tasks, each was written to the same busy-loop, as a way to simulate real-world tasks that this scheduler would be used for in future projects.

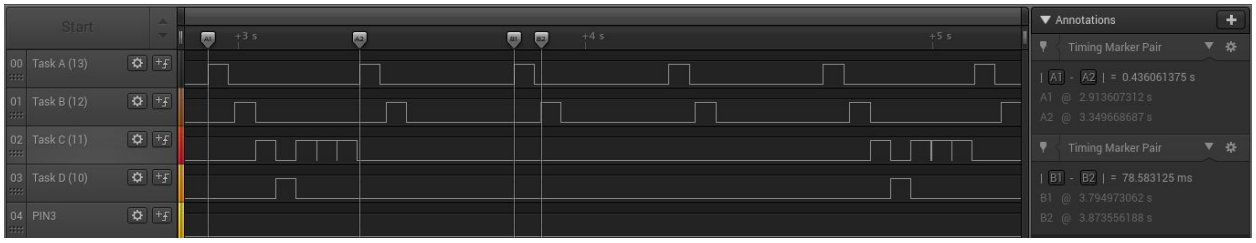

Figure 2: test_success results screen capture

The results of the test show that the busy-loops cause each task to run for approximately 58.5ms to complete. We can also see that the scheduler correctly ran the two periodic tasks at the right period (400 ticks≅436ms) and with the correct offset (70 ticks≅78ms).

The scheduler also correctly first scheduled the system task TaskC, then the low-priority TaskD and TaskC, followed by the two TaskC tasks that were called by TaskD. This cycle repeated every four iterations, since TaskA created these one-shot tasks every 4 times it would run.

4.1.2.2 test_toolong

This test was written similarly to test_success, with the difference that TaskA2 now calls TaskD2, which has a busy-loop ten times longer, leading to an expected execution time of 585 ticks, while the task was still created with a max_time parameter of 60 (ticks).

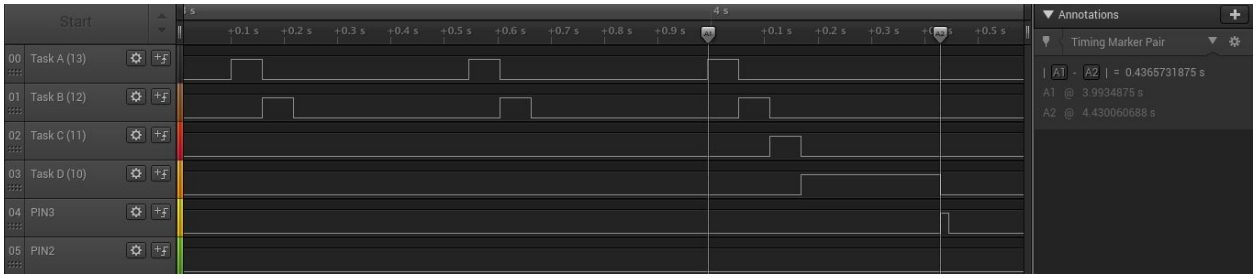

Figure 3: test_toolong results screen capture

The results of this test shows that once the interrupt finds that a one-shot system task is taking too long (longer than the period of 400 ticks≅436ms), it will result in an unrecoverable error, indicating by the short spike on PIN3.

4.1.2.3 test_misssystem

This test was written similarly to the previous test cases, with the difference that TaskA3 now calls a TaskC system task with a max_time parameter of 300 ticks, which cannot be scheduled with the overall period of 400 ticks and two periodic tasks each taking 58.5ms to complete execution.

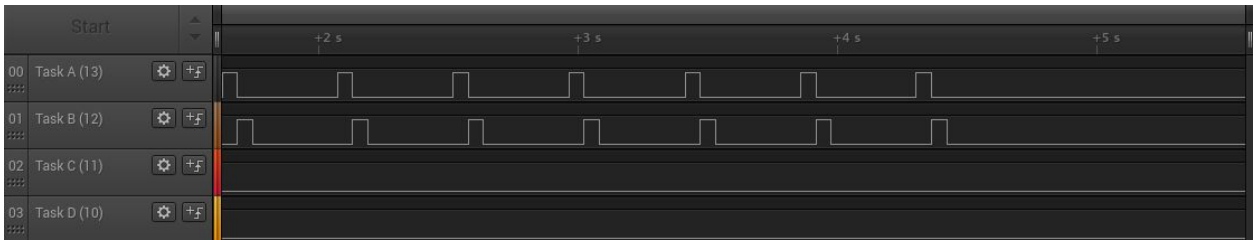

Figure 4: test_misssystem results screen capture

The results of this test shows that once the system task misses being run over 10 times, the scheduler returns exits due to this unrecoverable error.

4.1.2.4 test_missoneshot

This test was written similarly to the previous test cases, with the difference that TaskA4 now calls a TaskC low-priority one-shot task with a max_time parameter of 400 ticks, which cannot be scheduled with the overall period of 400 ticks and two periodic tasks each taking 58.5ms to complete execution.

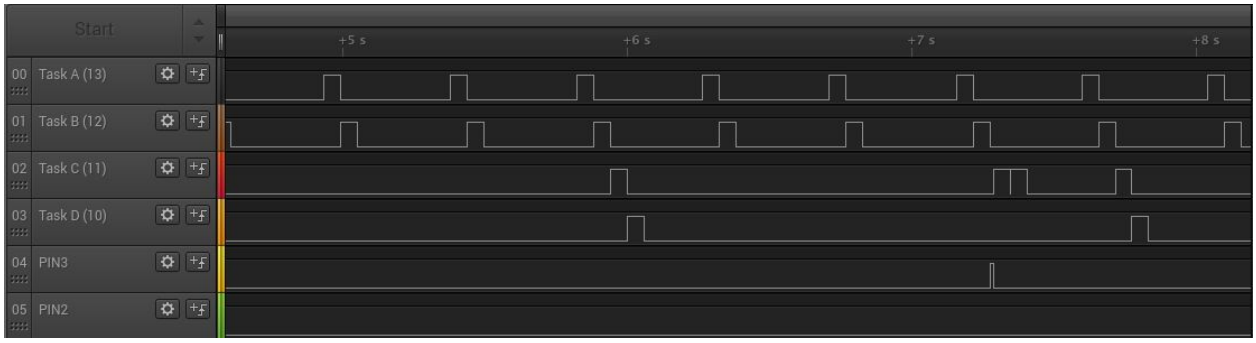

Figure 5: test_missoneshot results screen capture

The results of this test shows that the TaskC one-shot task doesn't run (the task runs only 3 times instead of the schedule 4), and every time it misses execution 5 times, it get taken off of the task queue, as indicated by the spike on PIN3.

4.1.2.5 test_impossibleschedule

This test was written similarly to the previous test cases, with the difference that TaskA and TaskB are given periods such that the scheduling of them is not possible. TaskA is scheduled with a period of 20 ticks and TaskB is scheduled with a period of 40 ticks.

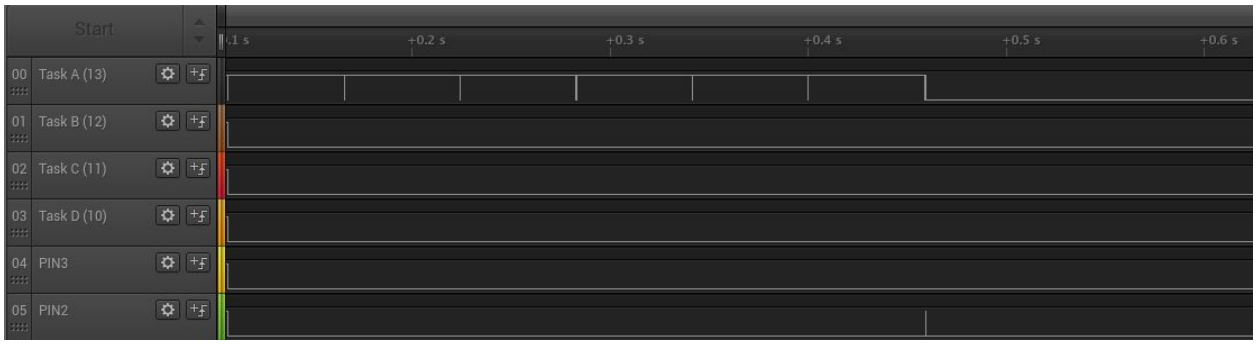

Figure 6: test_impossibleschedule results screen capture

The results of this test show that the scheduler attempts to schedule these two tasks, but exits once more than 10 periodic tasks have been attempted to be executed while there is an impossible schedule given for the periodic tasks. This failure can be seen by the short spike at PIN2, which occurs once the variable time_conflict_count reaches a value > 10.

4.1.2.6 test_timeconflict

This test was written similarly to the initial success test case, with the difference that TaskA and TaskB are not given offsets. Both TaskA and TaskB are scheduled with an offset parameter of 0 ticks.

Figure 7: test_timeconflict results screen capture

The results of this test show that the scheduler scheduled the periodic tasks TaskA and TaskB with an offset roughly equal to the execution time of these tasks (58.66ms).

4.2 Observations

One of the main challenges that we faced in project 2 was the inclusion of some compiler features that were being used. These included the linked list implementation was initially expected to be something that could just be included without much setup. However, this proved to be difficult and required an explicit implementation as seen in the “scheduler.h” file. This linked list implementation was necessary for creating a task queue. Additionally, it was later found that the new and delete keywords originally expected in the implementation weren’t available and an explicit implementation of this functionality was required in the Linked list class as well. The implemented new and delete alternatives were actually simply wrappers over malloc and free calls for simplicity.

Because of the nature of the project, the implementation of the callbacks and arguments were expected to be free form. However, since both the task lists and the argument lists for the different tasks used the linked list functionality, calls to pop tasks from the scheduler would attempt to destruct argument lists. This wasn't desired functionality and because the resources allocated to these lists were small, the original destructor was abandoned. This issue could have theoretically been resolved by adding reference counting to the linked list class that could keep track of the number of references to a particular object, and decrement the reference count instead of destruct it in the case of multiple references. However, this would be more difficult when considering that the linked list causing the issue was contained in an object that was being referenced in the other list.

Other minor issues were found with the calculation of the remaining idle time, in order for the one-shot tasks to get appropriately allocated. However, beyond this there were relatively few difficulties in the implementation of the scheduler.

4.3 Future work

Most of the expected behaviour and functionality of the scheduler was observed during the informal testing of the system as well as through the developed test cases shown previously. However, there was still some functionality left that was left not fully implemented.

An example of this was the one-shot task remaining time implementation, which would allow a one-shot task to be scheduled with a parameter that would specify how much time would be remaining before the task was to be executed. This could delay the execution of one-shot task when it is scheduled by using a timer to trigger an interrupt at some point in the future. Currently, the delay parameter is not used at all when scheduling a one-shot task, as these tasks are only scheduled as soon as possible.

5 Conclusion

As part of our design of the TTA scheduler[3], urgent events were handled by adding a regular interrupt occurring roughly every 1ms. Time-based tasks were chosen to be uninterruptible, as well as one-shot tasks. To avoid race conditions and more complex timing and scheduling issues, all of these tasks were designed to run to completion. A one-shot task could be created or deleted on the fly, to execute only once, without a predefined timing schedule. A task could be created to run as multiple instances, given multiple instances of a single task were not modifying the same shared data. There was one unimplemented feature left in the specification of the one-shot task that, if implemented would allow the user to delay the execution of a task.

Problems encountered in the implementation of the TTA scheduler included unexpected compiler version issues, memory management and some trouble with scheduling.

6 References

[1] M. Cheng, "Project 2", Webhome.csc.uvic.ca, 2019. [Online]. Available: https://webhome.csc.uvic.ca/~mcheng/460/spring.2019/p2.html . [Accessed: 13- Mar- 2019].
[2] A. Polyakov and J.Griffis, "Project 3", Webhome.csc.uvic.ca, 2019. [Online]. Available: .https://webhome.csc.uvic.ca/~mcheng/samples/polyakov/project3.html#scheduling . [Accessed: 13- Mar- 2019]
[3] github.com, 2019. [Online]. Available: https://github.com/HamzahMansour/csc460/tree/master/project2 . [Accessed: 13- Mar- 2019]