Graduation Project for Computer Systems Engineering II: A Real-Time Kernel in C

Kristian Andersson, Filip Säterberg

Abstract

The primary purpose if this report is to document the work performed during Computer Systems Engineering II, a course for computer engineering students held at Halmstad University in 2022. The purpose of the course work is to complete a partially developed Real-Time Kernel for the Atmel SAM3x8e microcontroller.

1 Introduction

The purpose of the course work was to improve an incomplete kernel provided to the students, with the goal of delivering a functional Real-Time Micro-Kernel, hence to be abbreviated as RTMK. The following sections of this report include a summarized overview of the developed project, a detailed description of the components of RTMK, any assisting libraries developed as well as an overview of thereto related unit- and integration tests.

The produced result of the project work is to be found in the following files:

- kernel_functions.c
- linked_list_functions.c
- mailbox_functions.c
- test_mailbox.c
- test_integration.c
- test_linked_list.c.

2 Project overview

The contents of the project, and the delivered RTMK, can be summarized as follows:

1. Improving an incomplete kernel
2. Main kernel components
3. Task Administration
4. Inter-Process Communication
5. Task Scheduling
6. Task lifetime and context switching

These topics are explained in brief in the proceeding section.

2.1 Improving an incomplete kernel

The incomplete kernel was delivered to the students together with relevant assembly code for exception and registry handling as well as the necessary data structures for the kernel. The project work consisted of implementing kernel functions in addition to which linked list libraries and mailbox libraries were implemented to support the functions of the RTMK itself as well as it's three main components.

2.2 Main kernel components

The three main components of the Real-Time Kernel are Task Administration, Inter-Process Communication, and Task Scheduling. The project work was in large part focused on the implementation of these three components, the construction of each constituting a separate phase of the project work. As mentioned, supporting libraries were developed to provide the necessary data structure in support of these functions. Both of the libraries are linked lists by by structure, albeit with different behaviour and utility. The Inter-Process Communication relies heavily on both of these libraries, while the other two components - Task Administration and Task Scheduling - primarily utilise the functions contained in the linked list library.

Along with the components mentioned, the core functions of the kernel itself are implemented in the kernelfunctions library. These functions primarily concern kernel initialization together with the necessary data structures, and a function to run the kernel.

2.2.1 Task Administration

On the one hand, Task Administration concerns the creation and termination of tasks, as well as the sorting of tasks in a scheduled order according to deadline. Task creation concerns the creation of the task control block, TCB, essentially containing everything that the task needs to run.

Another purpose of Task Administration is the queueing of tasks. This is done with a sorted double linked list: ReadyList, where all tasks ready to be executed are located and accessed.

2.2.2 Inter-Process Communication

Inter-Process Communication concerns the delivery of data between tasks. Messages may be sent and received synchronously or asynchronously and are delivered through mailboxes, double linked FIFO lists.

Synchronous communication might result in the task being blocked, in which case it is placed in an unsorted double linked list: WaitingList. A blocked task becomes unblocked and returned to the ReadyList when the message is delivered.

2.2.3 Task Scheduling

Task Scheduling concerns the assignment and update of deadlines, incrementing the system ticker, and placing idle tasks in a sleeping mode. Sleeping tasks are placed in an unsorted double linked list: TimerList. The scheduling function TimerInt is called at every SysTick Interrupt to extract from TimerList any tasks with an expired deadline and/or sleep duration, as well as extracting from WaitingList any tasks with an expired deadline.

2.3 Task lifetime and context switching

Throughout it's lifetime, a task is located in either of the lists ReadyList, WaitingList, TimerList, depending on it's state: Ready, Blocked, or Sleeping, respectively. It it is worth noting again that the only other state of a task - running - is reserved for the task at the head of the ReadyList. Whenever a task has been moved, a possible context switch occurs. This involves loading and unloading the TCB to and from the stack. This is one of the functions delivered as part of the incomplete kernel.

3 The operation of the RTMK

3.1 Initialization

Upon initialization, the kernel is put into "INIT" mode. Necessary data structures are initialized and memory is allocated; ReadyList, WaitingList, and TimerList. An idle task is created and put into the ReadyList. It has a negative deadline, meaning it will always be at the tail of the ReadyList and runs whenever there is no other task ready.

As long as the kernel is in "INIT" mode, any task created will be placed in the ReadyList. The kernel has a "run function" to set the kernel mode to "RUNNING". This function then prepares to load the TCB of the first task in the ReadyList. The actual loading is performed by a function delivered as part of the incomplete kernel.

The stages described are illustrated with a sequence diagram in Figure 1.

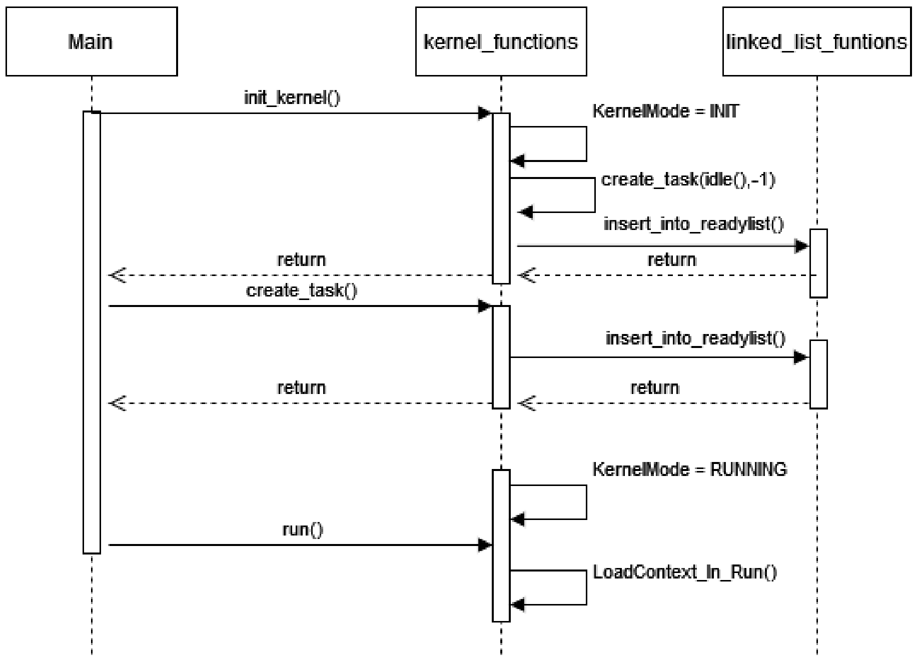

Figure 1: A sequence diagram detailing the process of initializing and running the kernel.

3.2 Task Administration and Scheduling

While the kernel is formally comprised of ask administration and scheduling as two different components, the illustration of both is more comprehensive when included in a single context. As should be apparent from the flowchart in Figure 2, in the completed RTMK the components are inter-dependent.

Task Administration contains a function to create any task. This involves the initialization and memory allocation of it's Task Control Block. The TCB contains deadline, message, a stack memory containing environment variables, program counter. Functions to save and restore the TCB was delivered as part of the incomplete kernel, and are called upon at every context witch as well as SysTick Interrupt.

A function to terminate tasks is also contained within the Task Administration. Only the currently running task may call for termination. It will then be extracted from the ReadyList and have it's allocated memory freed and data structures removed.

As previously mentioned, the SysTick Interrupt handler was also delivered as part of the incomplete kernel. This is illustrated in the flowchart as a separate process. It is upon this process that the scheduler is able to maintain deadlines as well as wake any sleeping task in due time. If a the deadline of a task is changed, a possible context switch will occur. This is because the task with lowest deadline is always at the front of the ReadyList. This function along with some others have been omitted from the flowchart in Figure 2 for clarity purposes.

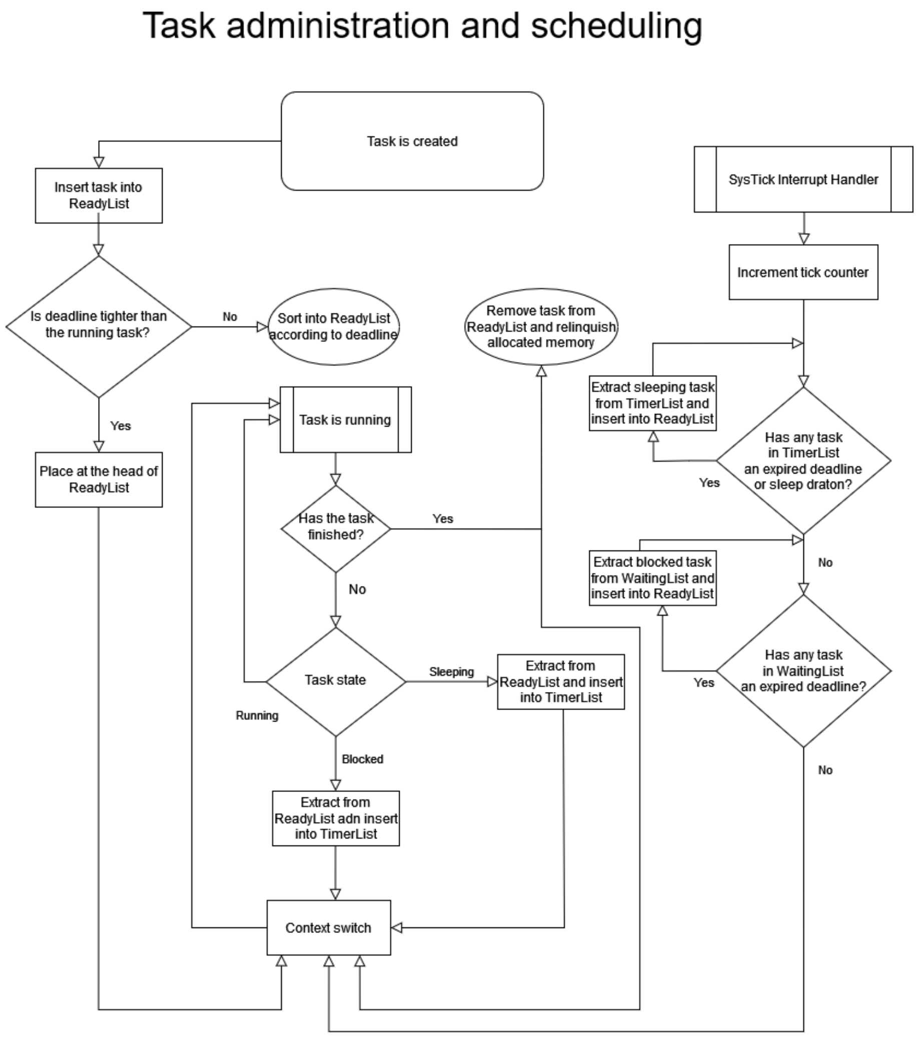
Task administration and scheduling
Task is created
Insert task into ReadyList
Is deadline tighter than the running task?
No
Sort into ReadyList according to deadline
Yes
Place at the head of ReadyList
Remove task from ReadyList and relinquish allocated memory
Task is running
Has the task finished?
Yes
No
Task state
Sleeping
Extract from ReadyList and insert into TimerList
Running
Blocked
Extract from ReadyList adn insert into TimerList
Context switch
SysTick Interrupt Handler
Increment tick counter
Extract sleeping task from TimerList and insert into ReadyList
Has any task in TimerList an expired deadline or sleep draton?
Yes
No
Extract blocked task from WaitingList and insert into ReadyList
Has any task in WaitingList an expired deadline?
Yes
No

Figure 2: A flowchart detailing the lifetime of a task from creation to termination, as it is being handled by the scheduler and task administration.

3.3 Inter-Process Communication

Together with a FIFO mailbox, the Inter-Process Communication is fulfilled by four separate function calls:

- send_wait
- receive_wait
- send_no_wait
- receive_no_wait

Any task has precisely one message stored in its TCB. This message is either a send or receive message. Within memory constraints, the kernel will hold any number of mailboxes. A mailbox can contain either asynchronous or synchronous messages (blocked or unblocked), but not both at the same time.

A task using asynchronous communication will attempt to receive or send a message and then continue. An asynchronous receive will fail however if there is no sending message at the front of the mailbox. Synchronous send or receive messages will cause the task to be blocked until it is unblocked by an incoming receive or send message respectively. These may be either asynchronous or synchronous.

When a task is blocked during communication, it is moved to the WaitingList and a context switch will occur. The reverse is not always true when a task is unblocked and moved from WaitingList to ReadyList. Note that due to this context switch, the asynchronous sending function may end up with an expired deadline even if there is already a receive message present in the mailbox. This depends first and foremost on the receiving tasks deadline; if it is shorter than the sending task, the context switch will result in the receiving task continuing to run instead. Ultimately this may cause the sending function to expire past it's deadline, even though it remains in the ReadyList. One possible scenario for this is such that the receiving task would go on to create further tasks, each with shorter deadlines than the sending function.

On this note, it could be argued that the task scheduler in the delivered RTMK should, in a future update, be made to contain the necessary switching functions to ensure that this scenario does not occur, and that tasks get a fair time slice to operate in.

Figure 3 illustrates the task administration and scheduling.

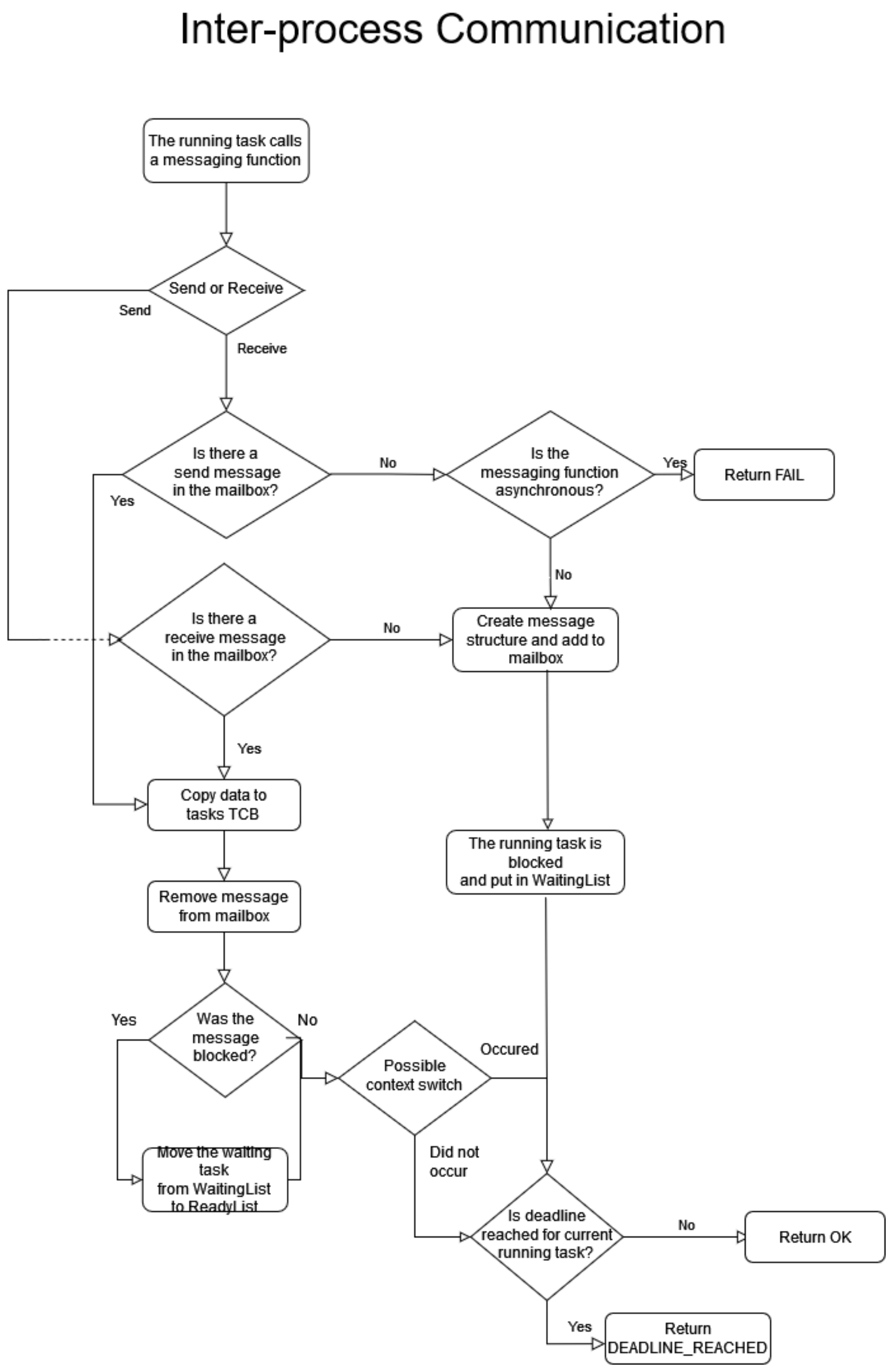
Inter-process Communication
The running task calls a messaging function
Send or Receive
Send
Receive
Is there a send message in the mailbox?
Yes
No
Is the messaging function asynchronous?
Yes
Return FAIL
No
Is there a receive message in the mailbox?
No
Create message structure and add to mailbox
Yes
Copy data to tasks TCB
The running task is blocked and put in WaitingList
Remove message from mailbox
Yes
Was the message blocked?
No
Possible context switch
Occured
Move the waiting task from WaitingList to ReadyList
Did not occur
Is deadline reached for current running task?
No
Return OK
Yes
Return DEADLINE_REACHED

Figure 3: A flowchart detailing how data transfer is handled between processes.

4 Assisting Libraries

In this section the assisting libraries implemented will be given an overview in brief.

4.1 Linked List Functions

The assisting library Linked List Functions is divided between supporting functions for ReadyList, WaitingList and TimerList, respectively. These are presented in brief below.

4.1.1 ReadyList Functions

Displayed below are the assisting functions for the ReadyList as contained in linked_list_functions.c:

```
exception    insert_into_readylist(listobj *NewObject)
```

Inserts a list object into the ReadyList. The list object will be placed in the list according to deadline, with the shortest deadline arriving first in the list.

```
listobj*     extract_from_readylist()
```

Removes the specified list object from the ReadyList and returns it.

```
void         reschedule_readylist()
```

This function extracts the running task from the ReadyList and inserts it again. This function is currently called only when a deadline is updated.

4.1.2 WaitingList Functions

Displayed below are the assisting functions for the WaitingList as contained in linked_list_functions.c:

```
void     insert_into_waitinglist()
```

Extracts the (now blocked) running task from ReadyList and inserts it into WaitingList.

```
void     remove_from_waitinglist(listobj* WaitingObject)
```

Removes the specified list object and inserts it into ReadyList.

4.1.3 TimerList Functions

Displayed below are the assisting functions for the TimerList as contained in linked_list_functions.c:

```
void     insert_into_timerlist()
```

Extracts the running task and inserts it into ReadyList. Currently this function is called only when a task is put to sleep by the kernel.

```
void     remove_from_timerlist(listobj* SleepingObject)
```

Removes the specified list object and inserts it into ReadyList.

4.2 Mailbox Functions

The supporting library for the mailbox and its necessary lfunctions are presented below.

```
mailbox*     create_mailbox(uint nMessages, uint nDataSize)
```

Creates a mailbox with storage capacity nMessages. The size of the messages is specified by nDataSize, for example the size of an integer or char.

```
int          no_messages(mailbox* mBox)
```

Returns the number of unblocked messages stored in the specified mailbox.

```
int          no_blocked_messages(mailbox* mBox)
```

Returns the number of blocked messages stored in the specified mailbox.

```
exception    message_status(mailbox* mBox)
```

Returns the status of the pending message at the front of the mailbox. The status i.e. the type of message is either Receiver or Sender.

```
int          is_wait_type(msg* message)
```

Returns an effective 1 or 0 if the message is blocked or unblocked, i.e. if it's synchronous or asynchronous.

```
void         add_message(mailbox* mBox, msg* message)
```

Adds the specified message to the specified mailbox and increments the unblocked message-counter, if and only if the mailbox contains no blocked messages.

```
void         add_blocked_message(mailbox* mBox, msg* message)
```

Adds the specified message to the specified mailbox and increments the blocked message-counter, if and only if the mailbox contains no unblocked messages.

```
exception    remove_mailbox(mailbox* mBox)
```

Removes the specified mailbox if it is empty.

```
void         remove_message(mailbox* mBox, msg* message)
```
Removes the specified message from the specified mailbox if it exists, and decreases the relevant message-counter.

5 Testing

This section will introduce and explain the tests delivered together with the RTMK.

5.1 Unit tests

Unit tests were developed to test and assert the functions of the assisting libraries.

5.1.1 Linked List Test

Displayed below are the testing functions for the assisting linked library. These functions are located in test_linked_list_functions.c:

```
test_readylist_insert_emptyOBJ()
```

Asserts that the ReadyList insert works as expected i.e. that the insert is properly sorted according to the task deadline.

```
test_readylist_front_insert_obj()
```

Asserts that a task with shorter deadline than the currently running task is inserted at the front of the ReadyList.

```
test_readylist_tail_insert_n_objects()
```

Asserts that n tasks with a deadline higher than that of any object currently in the ReadyList are inserted last in the list i.e. before the idle task at the tail.

```
test_readylist_front_head_insert()
```

Assert that consecutive inserts of tasks with a lower deadline than that of any object currently in the ReadyList are inserted at the front of the ReadyList, i.e. at the head.

```
test_readylist_extract_from_readylist()
```

Asserts that the extracting function returns the head node of the ReadyList.

```
test_waitinglist_insert_n_objects()
```

Asserts that insertion of n objects into the WaitingList works as intended.

```
test_waitinglist_insert_()
```

Asserts that the insertion of an object into the WaitingList works as intended.

```
test_waitinglist_remove_from_waitinglist()
```

Asserts that the remove function returns the specified list object and that the object is removed properly from the WaitingList.

```
test_timerlist_insert()
```

Asserts that the specified object is inserted properly into the TimerList.

```
test_timerlist_insert_n_objects()
```

Asserts that n list objects are inserted properly into the TimerList.

```
test_timerlist_remove_from_timerlist()
```

Asserts that the remove function returns the specified list object and that the object is removed properly from the TimerList.

5.1.2 Mailbox Test

Displayed below are the assisting functions for the TimerList as contained in linked_list_functions.c:

```
test_create_mailbox()
```

Asserts that a mailbox is created properly with storage capacity nMessages and the specified data size.

```
test_add_message()
```

Asserts that unblocked messages are inserted correctly into the mailbox and that the unblocked message counter is incremented correctly. Also asserts that insertion of blocked messages is illegal.

```
test_add_blocked_message()
```

Asserts that blocked messages are inserted correctly into the mailbox and that the blocked message counter is incremented correctly. Also asserts that insertion of unblocked messages is illegal.

```
test_remove_message()
```

Asserts that messages are removed correctly and that the relevant counter is decremented properly.

```
test_no_messages()
```

Asserts that the no_messages() function properly returns the number of unblocked messages for any specified mailbox.

```
test_no_blocked_messages()
```

Asserts that the no_blocked_messages() function properly returns the number of unblocked messages for any specified mailbox.

```
test_remove_mailbox()
```

Asserts that any specified mailbox is removed properly and that the removal of any mailbox not empty is illegal.

```
test_message_status()
```

Asserts that the message_status() function properly returns the status of the message at the head of any mailbox, i.e. RECEIVER or SENDER.

```
test_is_wait_type()
```

Asserts that the is_wait_type() function properly returns an effective 1 or 0 if the message at the head of any mailbox is either blocked or unblocked.

5.2 Integration tests

Integration tests were developed to assert that the resulting work of Phase Two and Phase Three respectively work as intended. Provided as part of the course work were three integration tests for each phase, the passing of which were part of the course curriculum. The integration tests presented below were developed in addition to these, to test for functionality and edge cases not included in the tests provided.

5.2.1 Inter-Process Communication: test_integration.c

This integrated test asserts that asynchronous-synchronous communication works as intended, i.e. that an asynchronous send message attempt will properly unblock an synchronous receive message, and vice versa.

The test also asserts that task scheduling works properly when deadlines expire, i.e. that sleeping tasks are returned to the ReadyList and terminated in due order.

Also included in the test is the assertion that float value type messages are not garbled in communication, something not tested for previously during the curriculum.

5.2.2 Inter-Process Communication: test_integration2.c

This integrated test asserts that the wait() function included in kernel_functions.c works as intended. It also asserts that the TimerInt function in the kernel works properly. Also included in the test is the assertion that the set_deadline() function in the kernel works properly, something not tested for previously during the curriculum.

6 Conclusion

The purpose of this document was to present the work done to develop and establish a functional RTMK from an incomplete kernel. This necessitated the development of supporting libraries for relevant data structures, for which unit tests were created to confirm that they function as intended. At the end of the two major phases of the project work, integration tests were developed to confirm that the RTMK behaves as expected during running mode, as well as to test for any edge cases not included during the course curriculum.

As of the delivery of this project, the RTMK may still be improved with regards to fair time slicing. Furthermore, there are without any doubt bugs still present in the code. Time allowing, integration testing could be improved further. Another area for improvement is algorithm efficiency and memory handling.

References

[1] Program in C https://www.youtube.com/watch?v=tas0O586t80 [Accessed: 11 March 2022]

[2] Per Brinch Hansen. 1970. The nucleus of a multiprogramming system. Commun. ACM 13, 4 (April 1970), 238–241. DOI:https://doi.org/10.1145/362258.362278

[3] Furber, Steve. (2017). Microprocessors: The engines of the digital age. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science. 473. 20160893. 10.1098/rspa.2016.0893.