SE 350

RTX DOCUMENTATION

Group 23

**Nathan Woltman**

nhwoltma

**Justin Gagne**

jjgagne

**Cody Chung**

c34chung

**Evet Dinkha**

edinkha

Table of Contents

Chapter 1 – Introduction 4

Chapter 2 – Global Variables 4

2.1 Global Data Structures 4

2.1.1 Linked List 4

2.1.2 Generic Queue 5

2.1.3 Priority Queue 6

2.2 Global Variables 7

2.2.1 Variables for Memory Management 7

2.2.2 Variables for Process Routines 8

2.2.3 Variables in Interrupt Process Commands 9

2.2.4 Variables in System Processes 9

Chapter 3 – Kernel API 10

3.1 Process Switching 10

3.2 Memory Management 11

3.2.1 Requesting a Memory Block 11

3.2.2 Releasing a Memory Block 11

3.3 Process Priority 12

3.3.1 Getting the Process Priority 12

3.3.2 Setting the Process Priority 13

3.4 Interprocess Communication 14

3.4.1 Message Structure 14

3.4.2 Send Message Process 14

3.4.3 Receiving Message Process 15

3.5 Timing Services 15

Chapter 4 – Interrupts and Their Handlers/ Processes 16

4.1 Interrupt I-Processes 16

4.1.1 The UART I-Process 16

4.1.2 Timer I-Process 18

Chapter 5 – System and User Processes 18

5.1 User Processes 18

5.1.1 24 Hour Wall Clock Display 18

5.2 User Tests 19

5.2.1 Set Priority Command Process 19

5.2.2 Stress Test 19

Chapter 6 – Initialization 19

6.1 Hardware Setup 19

6.2 Memory Initialization 20

6.3 Process Initialization 20

Chapter 7 – Testing 20

7.1 Manual Testing 21

7.2 Automated Testing 21

Chapter 8 – Major Design Changes 21

8.1 Changes from P1 Deliverable to P2 Deliverable 21

8.2 Changes from P2 Deliverable to P3 Deliverable 22

Chapter 9 – Timing Analysis 23

Chapter 10 – Lessons Learned 23

10.1 Memory Management 23

10.2 Sharing Variables Across Different Files 23

10.3 Preemption 23

10.4 Pointer vs. Non-Pointer Queue 24

10.5 Getting Code onto the Cortex M3 24

10.6 Pointer Initialization 24

10.7 Message Structure 24

10.8 Issuing CRT Interrupts 25

10.9 Understanding Interrupt-driven UART 25

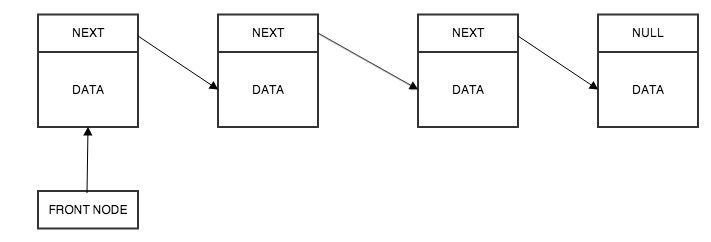
# Chapter 1 – Introduction

This report thoroughly documents our group’s SE 350 lab project RTX Kernel. General specifications can be found in the SE 350 Lab Manual. The report should give the reader a clear understanding of the way we chose to implement the RTX. Contents of the report include a description of the global variables used in the RTX, a description of all of the functions available in the kernel’s API, a description of how interrupts are handled in the kernel, descriptions of the RTX’s system and user processes, the initialization procedure of the RTX, a description of the methods used to test the system, major design changes between deliverables, a timing analysis of the system, and a lessons learned summary.

# Chapter 2 – Global Variables

## 2.1 Global Data Structures

### 2.1.1 Linked List



*Figure 1: Linked List Implementation*

A *forward linked list* was usedto keep track of the memory in the heap. The forward list structure itself contains only a “front” pointer that points to the first node in the list, which saves on memory from a normal list implementation. The list implementation has pointers to the front and back of the linked list. In the case of the heap, we use this pointer to easily request a new memory block (i.e. pop\_front), and to easily release a memory block (i.e. push\_front). Only four methods were needed for the implementation of the linked list:

*1. void init (ForwardList\* list);*

Initializes a forward list by setting its pointer to the front of the list to *NULL*.

*2. int empty (ForwardList\* list);*

Returns 1 if the list is empty, 0 otherwise.

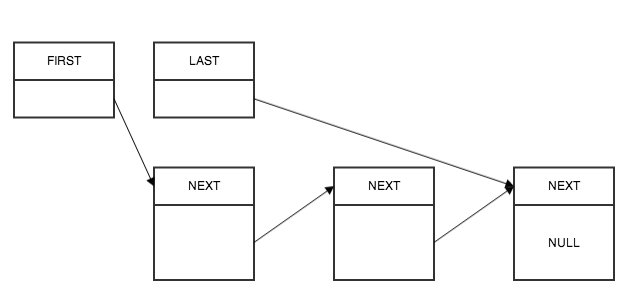
*3. ListNode\* pop\_front (ForwardList\* list);*

Removes and returns a pointer to the first node in the list.

1. *void push\_front(ForwardList\* list, ListNode\* node);*

Adds the input node to the front of the list.

### 2.1.2 Generic Queue



*Figure 2: Generic Queue Implementation*

Generic queues were used for the blocked queue and for each queue within the priority queue. As a typical queue, it is a FIFO structure. Our queue structure defines four methods:

*1. void init\_q (Queue\* queue);*

Initializes the queue by setting its pointers to the front and back of the queue to *NULL*.

1. *int q\_empty (Queue\* queue);*

Returns 1 if the queue is empty, otherwise, it returns a 0.

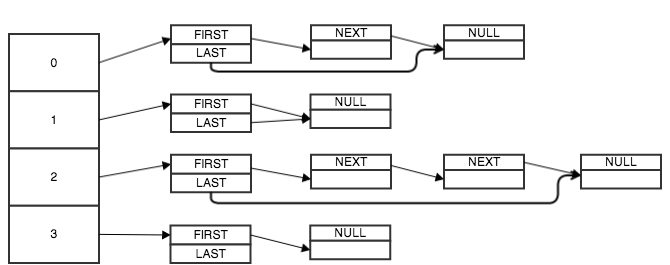
1. *void enqueue (Queue\* queue, QNode\* node);*

Adds the input node to the end of the queue.

1. *QNode\* dequeue (Queue\* queue);*

Removes and returns a pointer to the node at the front of the queue.

### 2.1.3 Priority Queue



*Figure 3: Priority Queue Implementation*

A priority queue structure is used for the ready queue. Our implementation of the structure is specific to the RTX project, in that there are only 4 priorities to keep track of in the system. Because there are a fixed number of priorities, the priority queue simply holds an array of 4 generic queues. Each of the generic queues in the ready queue will only contain PCBs that have a priority matching that queue’s position in the ready queue’s array. For example, all PCBs with the lowest priority will be stored in the ready queue’s 4th generic queue (i.e. in code: *queues[3]* because the lowest priority is 3). In the proc\_init function, each process (with a state of NEW) is added to its respective queue in the ready queue. There are four methods defined for the priority queue:

1. *void init\_pq (PriorityQueue\* pqueue);*

Initializes the priority queue by setting the *first* and *last* nodes of the queue

at each priority to *NULL*.

*2. QNode\* pop(PriorityQueue\* pqueue);*

Removes the highest-priority node (i.e. PCB) from the ready queue, and

returns a pointer to it.

*3. void push(PriorityQueue\* pqueue, QNode\* node, int priority);*

Adds the input node (i.e. PCB) to the end of the queue with the given priority.

*4. int remove\_at\_priority(PriorityQueue\* pqueue, QNode\* node, int priority);*

Removes a specific node (i.e. PCB) from the queue with the given priority. We use this function to remove a PCB from the ready queue before putting it back into the queue with a new priority when we call *set\_process\_priority()*.

## 2.2 Global Variables

Within our project, multiple global variables were used to manage calls made by functions, which were necessary to save the state of our kernel at a given time.

### 2.2.1 Variables for Memory Management

The file k\_memory is responsible for dealing with any memory type of calls. It holds the primitives needed for initializing the RAM when the kernel is first booted. It also holds the primitives necessary for processes requesting memory blocks or releasing blocks.

#### 2.2.1.1 U32 \*gp\_stack

This is the global variable needed to determine the last allocated stack low address. The variable holds 8 bytes of memory. Within the memory initializing, the \*gp\_stack is used to allocate the bytes for each memory block needed and for adjusting the exception stack frame.

#### 2.2.1.2 ForwardList\* heap

The heap is the structure responsible for storing all of the memory blocks that are assigned to the processes. It is in the form of a forward list because the memory blocks need to be accessed in the form of a list. The heap is necessary to be able to hold all of the memory blocks and make them easily accessible.

#### 2.2.1.3 PriorityQueue\* ready\_pq

The ready\_pq is responsible for holding all of the processes that are ready to be executed. However, it has to hold the processes with respect to their associated priorities and so a priority queue is used. The priorities are set to be 0, 1, 2, or 3, and each priority value has an associated queue full of processes. The ready\_pq is important since the interrupts are processed in a FIFO way, with respect to their respective priorities. When the next process is ready to execute, the kernel needs to know which process to execute and have it easily accessible.

#### 2.2.1.4 PriorityQueue\* blocked\_memory\_pq

Just like ready\_pq, blocked\_memory\_pq is responsible for holding all of the PCBs that are blocked on memory and cannot be executed. Each blocked memory queue is aligned with its respective priority value. The blocked\_memory\_pq is important to be able to determine which processes should be unblocked first when there are memory blocks available.

#### 2.2.1.5 PriorityQueue\* blocked\_waiting\_pq

Similar to the blocked\_memory\_pq, blocked\_waiting\_pq is responsible for holding all of the PCBs that are blocked while waiting for messages to arrive and so, they cannot be executed. Each blocked on waiting queue is aligned with its respective priority value. The blocked\_waiting\_pq is important to be able to determine which messages should be released first when there are processes receiving them.

### 2.2.2 Variables for Process Routines

The file k\_process is responsible for dealing with any of the adjusting of details of any of the PCBs. It contains the functions responsible for the initialization of the any of the processes, as well as the primitives responsible for switching between processes, changing process priorities, and sending messages.

#### 2.2.2.1 PCB \*\*gp\_pcbs

This is the global variable needed to store an array of the PCBs. It stores all of the features of a PCB, including the process id, the process state, whether the process is an iprocess, and holds a reference to the process’s message queue. The variable is important to the kernel because the correct process values need to be accessed when adjusting the memory or for any of the interprocess communications.

#### 2.2.2.2 PCB \*gp\_current\_process

The gp\_current\_process is a pointer to the current running process. This is one of the most used variables in our implementation, since we only want to be modifying the current process.

#### 2.2.2.3 U32 g\_switch\_flag

When an interrupt is received to the UART, the g\_switch\_flag is toggled. If the value is a 1, it means that the kernel has to switch the current process, otherwise it stays at the same one. The UART handler and the timer i-processes are the only processes that adjust this variable.

#### 2.2.2.4 extern PROC\_INIT g\_test\_procs[NUM\_TEST\_PROCS]

The process initialization table is determined in k\_process. The table is filled with the process values set from the set\_test\_procs().

### 2.2.3 Variables in Interrupt Process Commands

The file i\_proc deals with all of the i-processes used by the kernel.

#### 2.2.3.1 ForwardList\* delayed\_messages

It is used to temporarily hold messages sent with the *delayed\_send()* function. The timer i-process will save the messages it receives to the delayed\_messages list, and then send them when their send time arrives.

#### 2.2.3.2 volatile uint32\_t g\_timer\_count

The variable used to store the current time. The g\_timer\_count increments every millisecond when the timer interrupt is called.

#### 2.2.3.3. PCB\* timer\_proc;

The PCB used to store the timer process, since it is not initialized in k\_memory. The variable used so that the timer is able to have a reference to its own PCB. It is a pointer to the timer’s PCB.

### 2.2.4 Variables in System Processes

The file sys\_proc deals with all of the necessary calls to the system processes (the system processes being CRT and KCD).

#### 2.2.4.1 CMD registered\_commands[10]

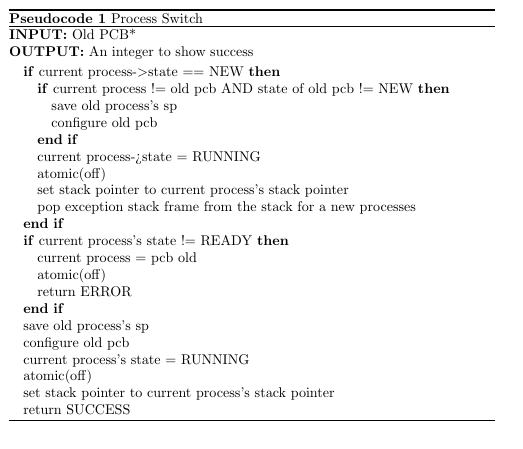
The registered\_commands variable is an array containing the data for each command that is registered with the KCD.

#### 2.2.3.2 int num\_reg\_commands

The variable used to store the number of registered commands and is updated when a new command is registered.

# Chapter 3 – Kernel API

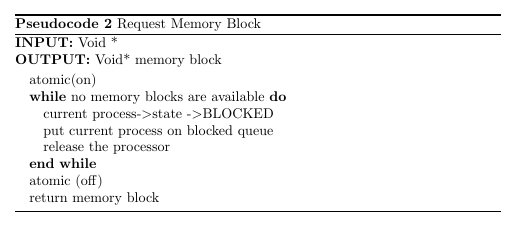
## 3.1 Process Switching



This function handles switching from the old process to the newly selected one. If the new process is in the NEW state, the old process is configured if it is not the same as the current process and is not NEW. Old processes are configured by updating their stack pointer and placing them in the ready queue if their current state is RUNNING or INTERRUPTED. Then, the new process’s state is set to RUNNING, and the stack pointer is updated to grab the new process’s stack state. Finally, *\_\_rte()* is called to pop the exception stack frame. The steps for process switching, when the new process’s state is not NEW, is almost identical. The only changes are that nothing happens if the new process is the same as the old one, and the exception stack frame does not need to be popped to run the new process.

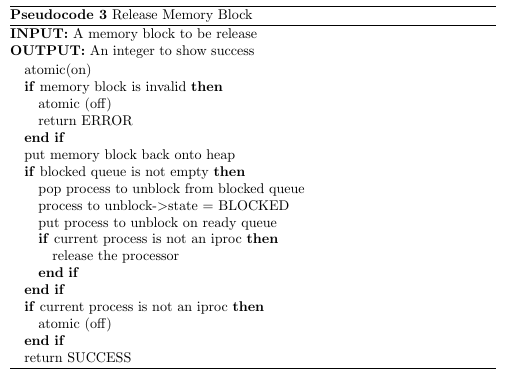
## 3.2 Memory Management

### 3.2.1 Requesting a Memory Block



If a process requests a memory block and there is a memory block available (i.e. the heap is not empty), the process can use the memory at the address returned by the result of popping the first block from the heap. Otherwise, the current process’s state is changed to BLOCKED, the process is added to the blocked queue, and the processor is released.

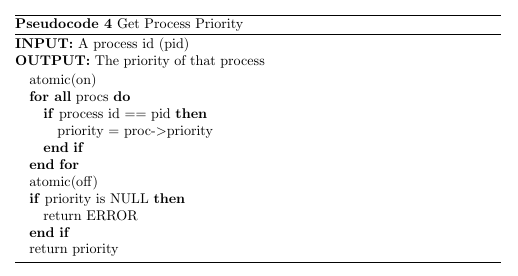
### 3.2.2 Releasing a Memory Block



When a process releases a memory block, the specified block is pushed back onto the heap. If there are any processes in the blocked queue, the first blocked process is popped off the blocked queue and placed back into the ready queue (because now there is memory for it to use, so it may continue where it left off). The processor is then released, giving the recently unblocked process a chance to run if its priority is high enough.

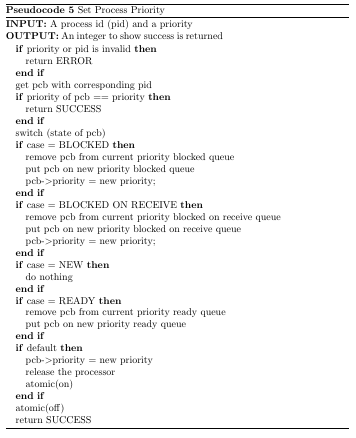
## 3.3 Process Priority

### 3.3.1 Getting the Process Priority

****

Returns the current priority of the process with the specified pid.

### 3.3.2 Setting the Process Priority



Allows user processes to change their own priority, or that of any other user process. The priority must be valid, and no unnecessary work is done if the new priority is the same as the current priority. If the process is currently in the ready queue, it is removed from the sub-queue associated with the old priority, then added to the sub-queue associated with the new priority. The process’s priority attribute is then changed and the processor gets released.

## 3.4 Interprocess Communication

### 3.4.1 Message Structure

The RTX supports a message-based Interprocess Communication (IPC). Messages are stored in envelope blocks, storing information about the sending and receiving processes, the next messages, the optional time to send the message (for delayed messaging), and the type and contents of the message. The implementation of a message envelope is as follows:

**typedef struct msg\_envelope**

**{ struct msg\_envelope \*next;**

**int sender\_pid;**

**int destination\_pid;**

**uint32\_t send\_time;**

**int mtype;**

**char mtext[1];**

**} MSG\_ENVELOPE;**

Due to the restrictive privileges set to the user process view, a message buffer was created to send and receive messages between two processes, with only the message type and contents being accessible. The implementation used is as follows:

**typedef struct msgbuf**

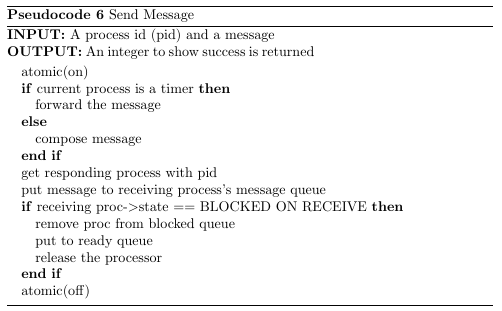
**{ int mtype;**

**char mtext[1];**

**} MSG\_BUF;**

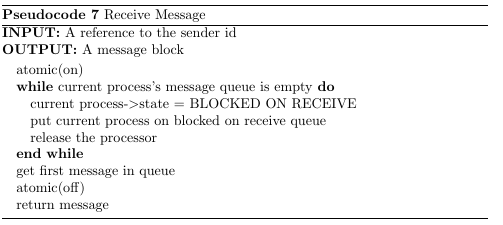
### 

### 3.4.2 Send Message Process



Since the process is only passed in the message buffer, the rest of the contents (the header of the message envelope) of the of the message envelop need to be saved for the kernel process view. The message is then added to the message queue of the receiving process. If the current process was blocked while waiting on a message, its state is changed to ready and the receiving procedure is added to the ready queue.

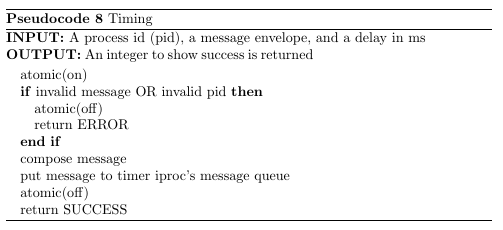
### 3.4.3 Receiving Message Process



The primitive checks while the message queue of the current process is empty for messages and blocks the incoming message. The process is then added to the blocked priority queue and the processor is released to deal with preemption. If the process is not blocked, the envelope is popped off the process’s message queue and then returned.

## 3.5 Timing Services

With the possibility of timer interrupts, the messages that are sent through the interrupt are saved in a delayed queue. When the time has expired, the messages are sent using this primitive:

****

In this primitive, the process id, the message envelope and the delayed time value is passed in. After the expiration (delay), the message is sent to the process\_id.

# Chapter 4 – Interrupts and Their Handlers/ Processes

## 4.1 Interrupt I-Processes

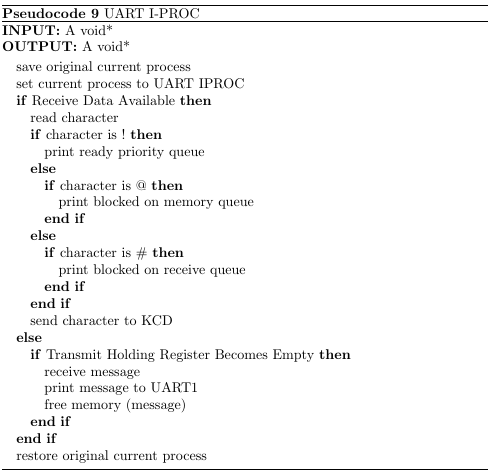
### 4.1.1 The UART I-Process

To consider UART0 interrupts, there are two handler primitives used. The first primitive is done using assembly.

**\_\_asm void UART0\_IRQHandler(void)**

In the UART0\_IRQHandler, the registers being used are saved and restored. Within this function, the c\_UART0\_IRQHandler is called, and that function deals with the rest of the irq handling.

**void c\_UART0\_IRQHandler(void)**



This UART I-Process forwards the characters passed to the KCD. Additionally, when the user inputs a character, an interrupt is called that calls the CRT function and the characters are echoed on the CRT display. Within the UART I-Process, there are hot keys for the user to click. The hotkeys implemented in our solution are:

1. “!” hotkey for printing the processes and priorities of those priorities from the ready queue
2. “@” hotkey for printing the processes and priorities of those processes from the blocked priority queue
3. “#” hotkey for printing the processes and priorities of those processes in the blocked on receiving messages priority queue

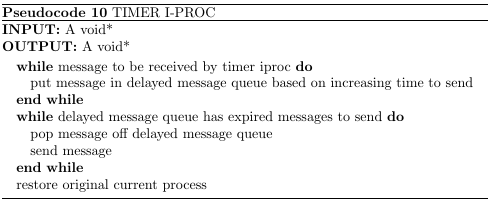
For each of these hotkeys, if they are ever pressed, the method leads to a print helper function.

**void print(PriorityQueue\* pqueue)**

In this print function, a priority queue is passed in (either the ready queue, blocked on memory queue, or blocked on receive queue), and while looping through the priorities, the corresponding processes and priorities are printed.

### 4.1.2 Timer I-Process

The timer I-Process is responsible for any hardware timer interrupts. After the time has expired, the timer I-Process has to deal with delayed messages.



In this process, any incoming messages are received and saved in a delay queue. The i-process then checks if the time has expired. When the timer expires, all of the messages are sent to their appropriate destination.

# Chapter 5 – System and User Processes

## 5.1 User Processes

### 5.1.1 24 Hour Wall Clock Display

The 24 Hour Wall Clock Display takes 3 types of commands:

1. %WR
2. %WS hh:mm:ss
3. %WT

When the usr\_procs begin, the 3 commands are registered to the KCD, so that if they are ever received, the appropriate command is written. After the registration, there is a WCProc that is constantly checking if the proc is sent a message. If it is, its correct output is verified and the correct output to be displayed is saved in a message and sent to the KCD to be displayed.

## 5.2 User Tests

### 5.2.1 Set Priority Command Process

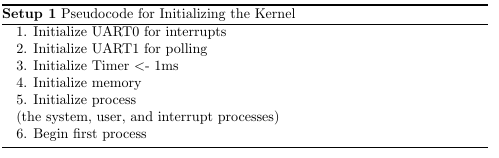
The priority command process takes in an input passed in by the user in the format **%C pid priority**. At any time that the user inputs the command, the pid being passed in points to the process that will have its priority changed. If the user ever inputs the wrong format for this command, they will be prompted with an error.

### 5.2.2 Stress Test

The stress tests comprise of 3 processes: process A, process B, and process C. Process A keeps checking for the user to input the **%**Z command. Once the user inputs that, process A keeps requesting memory blocks to send messages to process B. Process B receives the messages and sends the message it receives to process C. Process C receives all of the messages, but on every 20th message, it “sleeps” for 10 seconds, and when it “wakes” up, it sends the message to the CRT to display a message.

# Chapter 6 – Initialization

The RTX begins initialization in the k\_rtx\_init.c file. See for initialization pseudocode, and below for a more detailed description of the initialization procedure.



## 6.1 Hardware Setup

First, the interrupt-driven UART0 and polling UART1 are initialized. This initialization consists mainly of assembly code that sets certain bits at the hardware level so that the UART0 can be used for interrupt-driven user input and UART1 for printing debug statements by polling. Next, we initialize a timer to tick at every millisecond (again mostly done through assembly code). This allows us to send delayed messages and have a working wall clock. Once all of the hardware is initialized, the memory and processes can be set up.

## 6.2 Memory Initialization

Initializing memory consists of allocating memory for PCB pointers and stacks, all of the priority queues (the ready queue, the blocked on memory queue, the blocked on message receive queue, and the delayed message queue), and the heap used for requesting and releasing memory dynamically. Note that a macro defined memory block size of 128 bytes (found in k\_rtx.h) is used to allocate memory for our blocks in the heap. A special debug flag (DEBUG\_CUSTOM\_HEAP) has been implemented, which allows us to specify the number of blocks (NUM\_HEAP\_BLOCKS in k\_rtx.h) to allocate memory for in the heap. If the flag is not set in the target options, then the kernel will keep allocating memory for blocks until it runs out of space. This allows the kernel to request over 200 memory blocks at once, optimizing for maximum memory usage.

## 6.3 Process Initialization

Once the hardware and memory have been initialized, the kernel proceeds to initialize the processes. The first step is to call the function in the user test processes that sets up the g\_test\_procs global array of process initialization items (PROC\_INIT). This includes setting the user-defined priority, the process id, the user-defined stack size, and the initial stack pointer that points to the process’s function. The process table (also PROC\_INIT) is initialized next with the processes needed to be run first to register with the KCD. This means that the set priority command process, the wall clock process, and the stress test A process are placed first in the global g\_proc\_table array. Then, the user test processes are placed in the g\_proc\_table array, using information from the g\_test\_procs array. Next, the remaining processes are initialized in the g\_proc\_table (KCD, CRT, stress test B, stress test C, timer i-process, UART i-process, null process, in that order).

Then, the PCB for each process is configured, using the information from the global g\_proc\_table array. This includes setting the PID, priority, and state of each PCB, as well as whether or not the process is an i-process. Note that priorities, PIDs, and states are all macro defined in k\_rtx.h. Each process is also allocated a message queue in its respective PCB. The PCBs are placed on the stack and the stack pointers are updated. Finally, we place all of the processes but the null process and i-processes on the ready queue, and we set the i-process PCB states to READY. Now that everything has been initialized, the first process is run by calling k\_release\_processor.

# Chapter 7 – Testing

A mixture of both manual and automated tests was used to test the RTX as it was developed. The majority of testing was done manually, and a good testing framework was only implemented as of the P3 deliverable.

## 7.1 Manual Testing

The debugger was extensively used for testing, since Keil has several helpful tools such as the Watch Window and the breakpoints. Stepping through our code using various scenarios and examining the values of variables of interest helped us find the majority of our bugs. Once most of the API was implemented, a better testing system was implemented.

## 7.2 Automated Testing

The six user test processes were used to automate the testing procedure using unit tests. The first process (PID\_P1) was used to print testing results and to call the other five user to test the get and set priority functions (PID\_P2), the request and release memory block functions (PID\_P3), the send and receive message functions (PID\_P4), the delayed send message function (PID\_P5) and the set process priority command (PID\_P6). The unit tests worked by setting flags and ensuring that the API calls returned RTX\_OK rather than RTX\_ERR. All user processes began in the ready queue. PID\_P1 releases the processor so that PID\_P2 can run, which eventually sets priorities such that preemption occurs and PID\_P3 through PID\_P6 all get run, set a flag, and get blocked on message receive. PID\_P2 completes execution and also gets blocked on message receive. Once PID\_P1 prints the results of the priority tests, it sends messages to PID\_P3 through PID\_P6 when appropriate to test the different areas of the code. Just before the results of the tests are printed, the relevant test process is blocked on receive message so that we avoid returning to it. Note that the wall clock and user input (i.e. most of UART) were tested manually, as well as the hotkeys. Also note that during the P3 demo, it was learned that it would have been best to continuously release the processor once the test processes had finished running, rather than have them all blocked on receive message. This would have simplified testing the ready queue hotkey testing.

# 

# Chapter 8 – Major Design Changes

## 8.1 Changes from P1 Deliverable to P2 Deliverable

The P1 demo helped to identify some major flaws in the RTX. First of all, the initial implementation assumed that the blocked on memory queue did not need to be a priority queue. During the demo, it was learned that a priority queue is needed so that the highest priority process is always the first to receive an available memory block. Consequently, the blocked on memory queue was changed so that it used a priority queue rather than a simple queue, and changes were made to the set priority and release memory block primitives to reflect the new structure. Additionally, functions to check if the priority queue is empty and to retrieve the highest priority process in the priority queue were added to our generic priority queue structure to complement the implementation change.

The provided user test cases for preemption proved to be helpful. Comparing the actual output with the expected output, it was possible to tailor our preemption handling (in our scheduler function) to match the expected behaviour. The biggest flaw was that if the highest priority process released the processor and every process in the ready queue had lower priority, the lower priority processes had the chance to run. Because the RTX should always be choosing the highest priority process to run, this was incorrect behaviour, and needed to be fixed.

Another major design change from the first deliverable to the second was the restructuring of the RTX memory system. Because the only use for memory blocks in the RTX was to send and receive messages, the request and release memory block primitives were changed so that they returned a pointer to a message body instead of the beginning of a memory block. It was assumed that a system level envelope is essentially a memory block, and that we had a system level-only header, as well as a user-defined body. Thus, it was important to use pointer arithmetic to provide the user only with the information it should be able to access, all the while allowing the RTX to manipulate the header variables (i.e. the next pointer, the sender and receiving process ids, and the send time).

Finally, some checks were added to the release memory block and send message primitives to handle i-process preemption differently than regular process preemption. The RTX uses a global switch flag which is checked in the i-process IRQ assembly handlers, so the processor should not be released early if the current running process is an i-process (the processor will be released once the IRQ has completed execution).

## 8.2 Changes from P2 Deliverable to P3 Deliverable

Before the P2 demo, the RTX was using a global delayed message queue. During the demo, the TA suggested placing this queue in the timer’s PCB to avoid the user accessing the queue and to practice better coding style. Thus, the design was changed accordingly. Also, during the demo, an issue was encountered where the user would type something but the input would not be displayed on UART0 because the RTX was out of memory. To account for this, UART polling was used to display user input when there were no more memory blocks available for messaging.

Because all of the processes were implemented in P3, the file structure was changed to be cleaner and more representative of what each file contains in terms of code. This was not so much of a functionality or design change, but more of a good coding style. Also because all of the processes were implemented, the global initialization process table was changed so that all of the processes were included and the null process is placed at the end of the table rather than at the beginning. This doesn’t seem like a major change, but caused a significant bug where the incorrect process was being scheduled when the ready queue was empty. Yet another change that was made once all the processes were implemented was a total revamp of the user test cases (i.e. unit tests). Before this, the RTX had mostly been tested manually. Since everything was in place and better coding style and testing practices are always the best option, it was time to write some good tests. See Chapter 7 – Testing for more details on how the unit tests were implemented.

Finally, the UART i-process had to be changed from the P2 deliverable to the P3 deliverable due to the addition of the stress tests. It is important to note that we realize now that the RTX UART i-process was poorly implemented, and that this was a major lesson learned (see section 10.9 for more detailed information). The UART i-process was mostly hacked together to simply work, and the order of code execution had to be fiddled around with so that the RTX did not end up in the incorrect place in the UART i-process (i.e. the line that prints “Should not get here!” should never be executed). The UART i-process design is likely the biggest flaw of the RTX, and if we had to start over, more effort would be put in to asking TAs for help with it.

# Chapter 9 – Timing Analysis

# Chapter 10 – Lessons Learned

## 10.1 Memory Management

Our first task for coding the RTX was to build a memory management system that would make it easy to allocate and deallocate memory. It didn’t take long for us to decide to use a forward linked list as a data structure for our heap, but it was challenging to make it generic so that any object could be added to the list. Due to the lack of generics in C, we had to come up with a way to generalize the code. We could have used macro definitions, but we decided instead to use type casting to cast input objects (i.e. memory blocks) to list nodes that could be added to the list. It also took some thought as to whether we should allocate 128 bytes or 132 bytes for each memory block. The manual stated that each block should be a minimum of 128 bytes, but we were unsure if the 128 bytes accounted for the 4 byte pointer indicating the beginning of each memory block or not. We decided to use 128 bytes for each memory block, with 4 of those bytes being used to locate the remaining 124 bytes of memory space.

## 10.2 Preemption

One of the issues that we came across in Part 1 was determining how to implement preemption without interrupts. Eventually, it became clear that releasing the processor would cause preemption if implemented correctly. Thus, we needed to release the processor after changing a process’s priority, after requesting a memory block and blocking a process, and after freeing a memory block and unblocking a process. In each of these situations, preemption may occur to allow higher priority processes to execute.

## 10.3 Pointer vs. Non-Pointer Queue

We encountered two issues with our generic queue data structure, both related to using copies of data structures instead of copies of pointers to those structures. The problem was that in some functions we had created local variables and modified them, thinking that this changed the Queue. However, the local variables were only copies of Queues and QNodes, and after some debugging, we realized we needed to use pointers to update the actual data structures.

## 10.4 Pointer Initialization

When we ran our program on the simulator, everything was fine; however, running our code on the hardware caused Hard Faults. The issue ended up being that we assumed pointers were initialized to NULL. It turns out that the simulator initializes pointers to NULL, but the hardware does not. Thus, we had to add initialization functions to our data structures to initialize all pointer values to NULL, else face the dire consequences of trying to access invalid pointers.

## 10.5 Message Structure

When implementing the IPC primitives, one of the problems faced was how to implement the structure of the message envelope. The issue was whether to split the data into a kernel and user view or to include all of the details in the message envelope structure but hide the certain parts from the kernel view. The latter was what was chosen. The message envelope structure includes all of the necessary details mentioned in Section 2.1.1. Essentially, it is split into two sections: the header, the content only accessible to the kernel, and the message content, accessible to the user and the kernel. To deal with the accessibility, the header part of the message envelope can only be accessed through the use of offsets of the addresses. When implementing the stress tests, process c needed to have the messages in message envelopes instead of message buffers, since it needed to be able to store them in a queue. However, the user processes do not have access to the message headers, and so they cannot send and receive message envelopes. The work around to this problem was to have two converter functions from messages to envelopes and envelopes to messages. All of the stress processes would be sending messages through the message buffer, but when process C needed to add or remove messages from its local queue, it would call the appropriate converter and add or remove that to its queue.

## 10.6 Issuing CRT Interrupts

When the user inputs commands, it was unclear whether or not to wait until they finish their input (carriage return) to output their message or if it was necessary to output each character. After asking on the class discussion board, it was clarified that the expected output is that the character should be immediately echoed back to the user. To implement this expectation, the char was sent to the CRT was sent to the console through a message to be echoed back to the console.

## 10.7 Understanding Interrupt-driven UART

Understanding how the interrupt-driven UART should work was probably the biggest and hardest problem we encountered while designing the RTX. It took some time to understand that UART0 should only be used for user input and feedback to the user, and that UART1 should be used for all of the debugging statements. However, the bigger issue was understanding how a message passed from the CRT process could be displayed on UART0 without polling. Using the debugger, we could see that popping the stack after the IRQ handler had run would cause execution to resume at the beginning of the IRQ, causing the IRQ handler to run with no interrupt bit set, ultimately printing “Should not get here!” to the console. A good number of ~~hours~~ days were spent trying to debug this, and the biggest lesson learned here is that we should have gone to office hours or asked for help during a lab help session instead of trying to just hack a fix.