RTX DOCUMENTATION

Group 23

**Nathan Woltman**

nhwoltma

**Justin Gagne**

jjgagne

**Cody Chung**

c34chung

**Evet Dinkha**

edinkha

Table of Contents

Chapter 1 – Introduction 3

Chapter 2 – Global Variables 4

2.1 Global Data Structures 4

2.1.1 Linked List 4

2.1.2 Generic Queue 4

2.1.3 Priority Queue 5

2.2 Global Variables 7

2.2.1 Variables in K\_memory.c 7

2.2.2 Variables in K\_process.c 8

2.2.3 Variables in i\_proc.c 9

2.2.4 Variables in sys\_proc.c 9

Chapter 3 – Kernel API 10

3.1 Process Switching 10

3.2 Memory Management 10

3.2.1 Requesting a Memory Block 10

3.2.2 Releasing a Memory Block 10

3.3 Process Priority 10

3.3.1 Getting the Process Priority 10

3.3.2 Setting the Process Priority 11

3.2 Interprocess Communications 11

3.2.1 Message Structure 11

3.2.2 Send Message Process 12

3.2.3 Receiving Message Process 12

3.3 Timing Services 12

Chapter 4 – Interrupts and Their Handlers/Processes 13

4.1 Interrupt I-Processes 13

4.1.1 The UART I-Process 13

4.1.2 Timer I-Process 14

Chapter 5 – System and User Processes 14

5.1 User Processes 14

5.1.1 24 Hour Wall Clock Display 14

5.2 User Tests 14

5.2.1 Set Priority Command Process 14

5.2.2 Stress Test 15

Chapter 6 – Initialization 15

Chapter 7 – Testing 15

Chapter 8 – Major Design Changes 15

Chapter 9 – Timing Analysis 15

Chapter 10 – Lessons Learned 15

10.1 Memory Management 15

10.2 Sharing Variables Across Different Files 16

10.3 Preemption 16

10.4 Pointer vs. Non-Pointer Queue 16

10.5 Getting Code onto the Cortex M3 16

10.6 Pointer Initialization 17

10.7 Message Structure 17

10.8 Issuing CRT Interrupts 17

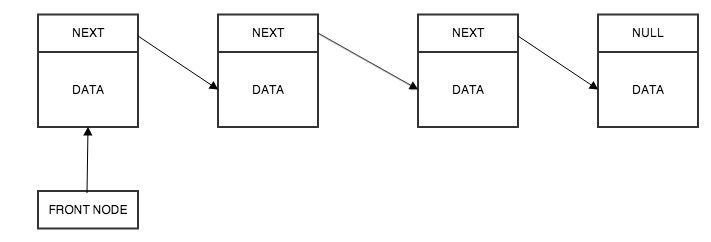
# Chapter 1 – Introduction

This report is written about the way our group implemented the OS Kernel. The purpose of this document is to help

# Chapter 2 – Global Variables

## 2.1 Global Data Structures

### 2.1.1 Linked List



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.

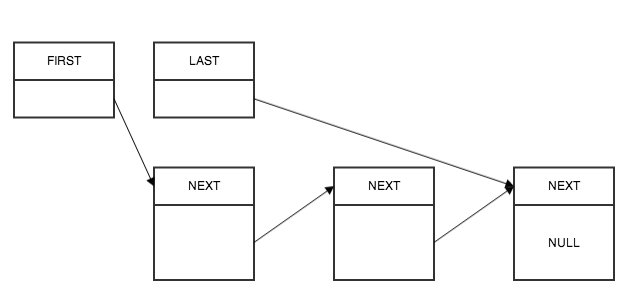
*3. ListNode\* pop\_front (ForwardList\* list);*

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



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.

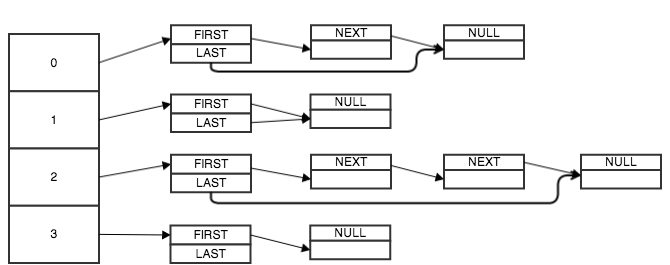
1. *void enqueue (Queue\* queue, QNode\* node);*

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



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 in K\_memory.c

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

The global variable needed to determine the last allocated stack low address. The variable holds 8 bytes of memory. Within the memory initiliazing, 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.

#### 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.

#### 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.

#### 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.

### 2.2.2 Variables in K\_process.c

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.

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

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.

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

The gp\_current\_process is a pointer to the current running 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 processes, otherwise it stays on the same process. The UART handler and the timer iprocesses 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 values set from the settestprocs().

#### 2.2.2.5 extern PCB\* timer\_proc;

The variable used so that the timer is able to have a reference to its own PCB.

### 2.2.3 Variables in i\_proc.c

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

#### 2.2.3.1 ForwardList\* delayed\_messages

It is used for any messages that are received during a timer interrupt. I\_proc will save the messages in the forward list, delayed\_messages. In the memory.c file, delayed\_messages is a variable used for the memory allocation of the variable.

#### 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.

### 2.2.4 Variables in sys\_proc.c

The file sys\_proc deals with all of the necessary calls to the system processes.

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

The registered\_commands variable is an array. It is used to register any commands in the KCD. If theses commands are every received by the KCD, they have to send an interrupt to the UART to display the approate message.

#### 2.2.3.2 int num\_reg\_commands

The variable used to store the number of registered commands.

# Chapter 3 – Kernel API

## 3.1 Process Switching

## 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.

## Process Priority

### 3.3.1 Getting the Process Priority

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

**int get\_process\_priority(int 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.

**int set\_process\_priority(int pid, int priority)**

## 3.2 Interprocess Communications

### 

### 3.2.1 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, 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;**

**U32 sender\_pid;**

**U32 destination\_pid;**

**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 msg\_envelope**

**{**

**int mtype;**

**char mtext[1];**

**} MSG\_BUF;**

### 3.2.2 Send Message Process

In the send\_message primitive, a process id and message was passed in.

**send\_message (int process\_id, void \*message)**

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.

### 3.2.3 Receiving Message Process

In the receive\_message primitive, a sender id is passed in.

**void \*k\_receive\_message(int\* sender\_id)**

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 removed from the process’s message queue and then returned.

## 3.3 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:

**int delayed\_send(int process\_id, void \*message\_envelope, int delay);**

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 passes in, 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.

**void c\_TIMER0\_IRQHandler(void)**

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

# Chapter 7 – Testing

# Chapter 8 – Major Design Changes

# 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 Sharing Variables Across Different Files

Initially, we struggled to share variables across different files. This was required so that we could access our ready and blocked queues from both k\_memory and k\_process source files. Searching online and looking through given code, we realized we needed to use the “extern” property to declare the variables to solve our problem.

## 10.3 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.

## 10.4 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.

## 10.5 Getting Code onto the Cortex M3

When we went to flash the code onto the Cortex M3, we were getting errors saying that the code could not be flashed successfully. Another student faced the same issue and recommended copying our source files into a new project and trying to flash the processor again. This approach fixed our problem.

## 10.6 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.

## 10.7 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.

## 10.8 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.