

# RTX Project Report

Xiang, Dian  
20431601  
dxiang@uwaterloo.ca

Justin McGirr  
20413625  
jmcgirr@uwaterloo.ca

Adrian Cheung  
20421743  
a32cheun@uwaterloo.ca

Aaron Morais  
20413440  
aemorais@uwaterloo.ca

April 6, 2014

## Abstract

# Contents

<b>I</b>	<b>Introduction</b>	<b>6</b>
<b>II</b>	<b>Awesome RTX</b>	<b>7</b>
<b>1</b>	<b>Global Variables</b>	<b>8</b>
1.1	Description . . . . .	8
1.2	Heap Data Structures . . . . .	8
1.3	Process Scheduler Data Structures . . . . .	9
1.3.1	Ready and Blocked Priority Queues . . . . .	10
1.3.2	Priorities . . . . .	11
1.3.3	Message Passing . . . . .	12
<b>2</b>	<b>Kernel API</b>	<b>13</b>
2.1	Description . . . . .	13
2.2	Memory Management . . . . .	13
2.2.1	Description . . . . .	13
2.2.2	Requesting Memory . . . . .	14
2.2.3	Releasing Memory . . . . .	15
2.3	Processor Management . . . . .	16
2.4	Interprocess Communication . . . . .	16
2.4.1	Send Messages . . . . .	17
2.4.2	Receive Messages . . . . .	18
2.4.3	Delayed Messages . . . . .	19
2.5	Process Priority . . . . .	19
2.5.1	Set Process Priority . . . . .	20
2.5.2	Get Process Priority . . . . .	20

<b>3</b>	<b>Interrupts and Their Handler and Processes</b>	<b>21</b>
3.1	Description . . . . .	21
3.2	UART . . . . .	21
3.3	TIMER . . . . .	21
<b>4</b>	<b>System and User Processes</b>	<b>22</b>
4.1	System Processes . . . . .	22
4.1.1	Description . . . . .	22
4.1.2	Null Process . . . . .	22
4.1.3	CRT Display Process . . . . .	23
4.1.4	Keyboard Command Decoder Process . . . . .	23
4.2	User Processes . . . . .	25
4.2.1	Wall Clock Process . . . . .	25
4.2.2	‘funProcess’ . . . . .	25
4.2.3	‘schizophrenicProcess’ . . . . .	25
4.2.4	‘fibProcess’ . . . . .	25
4.2.5	‘memoryMuncherProcess’ . . . . .	25
4.2.6	‘releaseProcess’ . . . . .	25
<b>5</b>	<b>Initialization</b>	<b>26</b>
5.1	Description . . . . .	26
5.2	Memory Layout . . . . .	26
5.3	Process Mailbox . . . . .	26
<b>6</b>	<b>Major Design Changes</b>	<b>27</b>
6.1	Description . . . . .	27
6.2	Heap . . . . .	27
6.3	Scheduler . . . . .	27
6.4	What We Learned . . . . .	27
<b>III</b>	<b>Testing and Analysis</b>	<b>28</b>
<b>7</b>	<b>Testing</b>	<b>29</b>
7.1	Description . . . . .	29
7.2	Theoretical Analysis . . . . .	29
7.3	Measurements . . . . .	29

<b>8</b>	<b>Timing Analysis</b>	<b>30</b>
8.1	Requesting and Releasing Memory . . . . .	30
8.1.1	Single Request Single Release . . . . .	30
8.1.2	Cumulative Memory Requests . . . . .	31
8.2	Sending and Receiving Messages . . . . .	33
8.2.1	Single Send, Single Receive . . . . .	33
8.2.2	Cumulative Message Sending . . . . .	34
8.2.3	Effects of Message Length . . . . .	35
<b>A</b>	<b>Raw Measurement Data</b>	<b>37</b>
A.1	Code Used To Collect Timing Data . . . . .	37
A.2	Data Analysis For Single Request Single Release . . . . .	40
A.2.1	Request . . . . .	40
A.2.2	Release . . . . .	41
A.3	Data Analysis for Single Send Single Receive . . . . .	42
A.3.1	Send . . . . .	42
A.3.2	Receive . . . . .	43

# List of Algorithms

# List of Figures

8.1	Request Memory over 5 machines . . . . .	31
8.2	Release Memory over 5 machines . . . . .	32
8.3	Release Memory over 5 machines . . . . .	33
8.4	Top: Sending messages; Bottom: Receiving messages . . . . .	34
8.5	Cumulative message sending . . . . .	35
8.6	Sending messages with variable message length . . . . .	36

# **Part I**

## **Introduction**



# **Part II**

## **Awesome RTX**

# Chapter 1

## Global Variables

### 1.1 Description

There are various amounts of variables used throughout the RTX. This section describes<sup>13</sup> the global variables that are used to accomplish tasks in the RTX. Other global variables such as the global process IDs can be seen in the appendix TODO. There are three major sections that required global variables and data structures: memory management in the heap, scheduler, and user processes.

### 1.2 Heap Data Structures

The RTX provides functionality to request and release memory from the heap, which is shared and stored in the RAM of the board. There is one main data structure that stores each memory block.

```
1 typedef struct HeapBlockHeader {  
    int source_pid;  
3    int dest_pid;  
    unsigned int send_time;  
5    struct HeapBlock* p_next;  
    struct HeapBlock* p_next_usr;  
7 } HeapBlockHeader;  
9 #define HEAP_BLOCK_SIZE 128
```

```
11 typedef struct HeapBlock {  
    HeapBlockHeader header;  
    byte data[HEAP_BLOCK_SIZE  
    ];  
} HeapBlock;
```

The HeapBlock structure stores a header and the content of the block. When a memory is requested, users are given the data with a adjustable size of HEAP\_BLOCK\_SIZE and do not have knowledge of the header. Helper functions in the RTX turn the user block back into kernel block by adjusting the pointer of the block. Each header contains information used for message passing. This global data structure is used in the following sections:

1. **Process Message Passing** - Uses this data structure to pass messages between processes. Message envelopes are implemented as HeapBlocks.
2. **Timer I-Process** - The process uses message passing to send delayed messages.
3. **UART I-Process** - The process uses message passing to process input and

output characters.

4. **CRT I-Process** - The process receives messages and passes message for terminal output.
5. **KCD Process** - The KCD uses message passing to CRT for display and user processes for processing.

## 1.3 Process Scheduler Data Structures

The RTX has a fixed-priority based scheduler that acts as a uniprocessor system. Context switching is required and the following data structures are used for this purpose.

### 1.3.1 Ready and Blocked Priority Queues

On initialization, each in memory process is given a process control block (PCB). The PCB contains data on the process such as its process ID, stack pointer, process state, and priority that the kernel will use for scheduling. The blocked and ready priority queues of PCBs keep track of which processes are blocked and ready for execution. Each process is in one of the 5 states at all time listed in ProcessState. More details about the states can be found in section TODO. Processes that are in state PROCES\_STATE\_READY are on the ready queue. Processes in state PROCESS\_STATE\_BLOCKED are in the blocked queue. There is also an implicit queue for

processes blocked on message but we won't go into that in this section.

```
typedef enum {  
    PROCESS_STATE_NEW           =  
    0,  
    PROCESS_STATE_READY        =  
    1,  
    PROCESS_STATE_RUNNING      =  
    2,  
    PROCESS_STATE_BLOCKED      =  
    3,  
    PROCESS_STATE_BLOCKED_ON_MESSAGE =  
    4,  
} ProcessState;  
  
typedef struct PCB {  
    // Stack pointer  
    U32* sp;  
  
    ProcessID pid;  
    ProcessState state;  
    ProcessPriority priority;  
  
    struct PCB* p_next;  
    // Incoming messages, waiting to be  
    // processed.  
    HeapBlock* message_queue;  
} PCB;  
  
PCB* g_ready_process_priority_queue[  
    PROCESS_PRIORITY_NUM] = {NULL};  
PCB* g_blocked_process_priority_queue[  
    PROCESS_PRIORITY_NUM];
```

PCBs and the priority queues are used in the following sections:

1. **Memory Management** - When a process requests memory without available memory blocks, the memory management subsystem must add the process to a blocked queue. When a memory block is released, the process with the highest

priority in the blocked queue (if any) is moved into the ready queue. Thus, the memory management subsystem must have access to the PCBs, and blocked and ready queues in order to accomplish these task.

2. **Process Management** - The process management subsystem is the one that schedules processes and keep track of the process priorities. Ready processes are taken from the ready queue to be ran if the current process is blocked or finished its execution. The process management subsystem also takes care of updating the two priority queues when the priority of a process has been changed.
3. **UART Process** - The UART Process needs access to the PCB. The PCBs also contain a mailbox for messages. The UART displays any incoming messages to the terminal display. Thus, it needs to know the PCB structure in order to gain access to the mailbox.

KCD, CRT, timer I-process, and UART I-process.

```
typedef enum {
    PROCESS_PRIORITY_INVALID      = 0 ,
    PROCESS_PRIORITY_SYSTEM_PROCESS = 1 ,
    PROCESS_PRIORITY_HIGH        = 2 ,
    PROCESS_PRIORITY_MEDIUM      = 3 ,
    PROCESS_PRIORITY_LOW         = 4 ,
    PROCESS_PRIORITY_LOWEST      = 5 ,
    PROCESS_PRIORITY_NULL_PROCESS = 6 ,
    PROCESS_PRIORITY_UNSCHEDULABLE = 7 ,

    PROCESS_PRIORITY_NUM          = 8
} ProcessPriority;

typedef enum {
    USER_PROCESS_PRIORITY_HIGH    = 0 ,
    USER_PROCESS_PRIORITY_MEDIUM = 1 ,
    USER_PROCESS_PRIORITY_LOW     = 2 ,
    USER_PROCESS_PRIORITY_LOWEST  = 3 ,

    USER_PROCESS_PRIORITY_NUM     = 4 ,
} UserProcessPriority;
```

### 1.3.2 Priorities

The RTX is based on a fixed-priority scheduler with 5 user process priorities and 2 system process priorities given below. Users are given priorities PROCESS\_PRIORITY\_HIGH to PROCESS\_PRIORITY\_LOWEST. PROCESS\_PRIORITY\_NULL\_PROCESS is given to the null process and PROCESS\_PRIORITY\_SYSTEM\_PROCESS are given to critical processes such as the

This priority structure is used by the process management unit to schedule and block processes based on their priorities. The RTX provides users with UserProcessPriority while keeping an internal structure of ProcessPriority.

### 1.3.3 Message Passing

Message passing is a way for interprocess communication provided by the RTX. Messages are created in the form of a msgbuf

structure, which includes the type and content. Some processes such as the CRT and KCD will require a certain type of message to be sent before the message can be processed correctly.

```

1  typedef enum {
    MESSAGE_TYPE_KCD_KEYPRESS_EVENT
        = 0,
3
    MESSAGE_TYPE_KCD_COMMAND_REGISTRATION
        = 1,
    MESSAGE_TYPE_CRT_DISPLAY_REQUEST
        = 2,
5    MESSAGE_TYPE_WALL_CLOCK
        = 3,
    MESSAGE_TYPE_COUNT_REPORT
        = 4,
7    MESSAGE_TYPE_WAKEUP_10
        = 5,

9    MESSAGE_TYPE_NUM
        = 6,
} MessageType;
11
12 struct msgbuf {
13     MessageType mtype:
    char mtext[HEAP_BLOCK_SIZE - sizeof(
        MessageType)];
15 };

```

The message passing data structure is evident in the KCD, wall clock process, and all I-processes. It can also be used in all user processes.

# Chapter 2

## Kernel API

### 2.1 Description

This section describes the kernel API that is available to users in the RTX. It will only go into details of how to use each API function call and states of different scenarios. Details of the implementation can be found in section TODO or from the raw code.

### 2.2 Memory Management

#### 2.2.1 Description

The RTX provides the utility of simple memory management of the heap. The main memory on the Keil MCB1700 is divided into sections of the RTX Image, PCB data, the heap, and process stacks seen in figure blah. \*\*\* TODO insert figure blah here \*\*\*

The Keil MCB1700 does not have the necessary hardware to support virtual memory. Thus, a fixed memory management scheme is used. The OS kernel is always loaded into main memory in its entirety. When the OS boots, a user stated number of PCBs and

process stack are allocated. The OS also allocates memory for system processes such as the Keyboard Command Decoder Process (KCD) and the CRT Display Process. After allocation of PCBs and process stacks, the remaining memory is used for the heap, which is shared between all processes. This section will be focused on the memory management of the heap.

The heap is further divided into a variable number of blocks. Each block contains a header (HeapBlockHeader) and 128 bytes of data. Depending on the number of user and system processes, the number of available heap memory will vary. The size of the data can vary from 128 bytes but must be set at compile time. The OS supports two kernel calls that gives access to these memory blocks.

#### 2.2.2 Requesting Memory

```
void *request_memory_block();
```

The first functionality supported by the OS is the ability to request memory. This call

gives back a pointer to a memory block in the heap. The size of the memory block is defined by the constant `HEAP_BLOCK_SIZE`, which defaults to 128 bytes. These blocks are used for storing local variables or as envelopes for interprocess communication (described in section TODO). The user must cast the memory block to the proper type for his or her own use. An example of the usage is provided below, which stores the numbers 0-31 in an array of size 32.

```

1 void user_process() {
2     int size = 32;
3
4     int* array = (int*)
5     request_memory_block();
6
7     for (int i = 0; i < size; i++) {
8         array[i] = i;
9     }
10
11     // ...
12
13     release_memory_block((void*)array)
14     ;
15 }

```

All processes will share the same heap memory pool. Thus, this primitive will block the process if there are no remaining free memory blocks. In that case, it will only be unblocked there is a new memory block available and if it has the highest priority on the list of processes waiting for a memory block. When using the memory block, the user must be aware of writing past the heap block size. The OS does not check this in any way. Thus, undefined behavior may occur. Also, the user must remember to release the memory block

or memory leakage will occur.

Requesting memory is reliant on calls to the heap to find an available memory block to allocate. The heap keeps a list of the status of each memory block to determine whether or not a block of memory can be requested. The following is the implementation used by the heap:

```

function FINDFREEBLOCK
    for  $i = 0$  to TotalNumberOfBlocks
    do
        if FreeSpaceBitmap[i] is a free block
        then
            return i
        end if
    end for
    return not found
end function

```

The lookup time of the FreeSpaceBitmap is  $O(1)$  as it is a simple array lookup and a boolean evaluation. Finding a free block has its dependencies on the total number of blocks. The best case for requesting memory is  $O(1)$  when the first block in the heap is free. Since this is a linear search, on average the complexity is  $O(n)$  where  $n$  is the total number of blocks.

Although this is worse in theta complexity than a simple free-list implementation, it was chosen because it allows almost all memory errors to be caught immediately, significantly easing debugging. We verify heap block alignment, check for double-frees, and verify that the pointer lies within the heap, each of which has helped us find bugs that would otherwise be invisible while developing the RTX.

### 2.2.3 Releasing Memory

```
1 int release_memory_block(void *  
    memory_block);
```

The second functionality supported by the OS is the ability to release memory. This is a non-blocking (but potentially pre-empting) call that returns the memory block back to the OS. This should be called when a message is received and not passed on or when the process is done using the memory block. If any process is blocked on memory, it will be unblocked and put to the ready queue. If the current process has lower priority than the unblocked process, then the current process will be preempted and the higher priority process will be executed instead. An example of this can be seen in figure CODE1 TODO on line 13.

In order to release the memory, `release_memory_block` will need to rely on the implementation of the heap. As stated in 2.2.2, the heap keeps a list of all available memory blocks. In order to release a memory block, the heap will set that block in the list as a free block. Releasing the block thus has  $O(1)$  complexity as it only relies on an array lookup. The implementation is as follows:

```
function                                RELEASEMEM-  
    ORY(MemoryBlock)  
    FreeSpaceBitmap[position of Memory-  
    Block]  $\leftarrow$  true  
end function
```

## 2.3 Processor Management

```
1 int release_processor();
```

The OS manages processes as though it is on a uniprocessor. A priority based scheme with context switching is used for scheduling processes. A process can voluntarily release the processor to the OS at any time during its execution. If there are errors in the call, it will return an error without releasing the processor to the OS. If there are no errors and the invoking process is ready to execute, it is put to the end of the ready queue of its priority. If there are no other processes of equal or higher priority, the process will be chosen to execute again. However, if there is, another process will be chosen for execution. Below is an example which prints an increasing number and releases the processor at every turn.

```
1 void user_process() {  
    static num = 0;  
    while(1) {  
        print(num++);  
        release_processor();  
    }  
}
```

## 2.4 Interprocess Communication

Communication between processes is primarily done through message-based interprocess communication (IPC), although static variables can also be used in some cases. De-



tails of the internal layout of process mailboxes can be found in section TODO. The RTX gives three primitives to carry out this<sup>7</sup> task, one for sending, one for delayed sending, and one for receiving.

### 2.4.1 Send Messages

```
1 int send_message(int process_id, void
    *message_envelope);
```

A process can compose a message envelope to be sent to another process. Memory for envelope message must be requested from the RTX using the request\_memory\_block() routine. The envelope consists of a type (mtype) and the message data (mtext) which must be filled in as seen below. The predefined message types are used for the KCD, CRT, and Wall Clock process, which are built into the RTX (see section TODO for more information). The message data has a predefined size which is smaller than the HEAP\_BLOCK\_SIZE. Thus, any message that are longer will exhibit undefined behavior. The process ID of the receiving process must also be known ahead of time in order to use send\_message().

```
1 typedef enum {
    MESSAGE_TYPE_KCD_KEYPRESS_EVENT
        = 0,
3    MESSAGE_TYPE_KCD_COMMAND_REGISTRATION
        = 1,
    MESSAGE_TYPE_CRT_DISPLAY_REQUEST
        = 2,
5    MESSAGE_TYPE_WALL_CLOCK
        = 3,
```

```
    MESSAGE_TYPE_USER_DEFINED
        = 4,
    MESSAGE_TYPE_NUM
        = 5,
9 } MessageType;

11 struct msgbuf {
    MessageType mtype;
    char mtext[HEAP_BLOCK_SIZE -
13    sizeof(MessageType)];
};
```

The primitive returns a status which validates the, message, receiver process ID, and ready queue. User processes is allowed to send a message to any user processes or system processes such as the KCD. If the receiving process is currently blocked on receiving message, this call will add the message to the process' message box and put it back on the message queue. The current process continues to execute unless the receiving process has a higher priority. In that case, the current process will be preempted and put to the back of the ready queue. Thus, this primitive may effect the execution of the process.

In order to keep track of messages that have been sent, the messages are stored in a queue, specifically a linked list. Messages are added to the end of the linked list resulting in an  $O(n)$  insertion complexity. It is possible to have this run in  $O(1)$  time by using a doubly-linked list and keeping track of the first and last elements in the list, but it seemed more important to provide a working prototype first, and optimize the runtime when (if) it became an issue.  $O(n)$  is likely to be quite small even in the worst

case, given the limited size of the RTX heap.<sup>9</sup>

## 2.4.2 Receive Messages<sup>11</sup>

```
void *receive_message(int *sender_id);
```

A process can use this primitive to receive<sup>13</sup> messages from other processes. Unless the sender\_id is NULL, the sender of the message will be written to the sender\_id param-<sup>15</sup>eter. The current process will check its mailbox for any incoming messages. If there are no messages in its mailbox, the routine will<sup>17</sup> block the process and select another process for execution. Execution of the process will<sup>19</sup> occur again if a message is sent from another process and this process has the high-<sup>21</sup>est priority in the ready queue. The message\_envelope is heap memory underneath<sup>23</sup> Thus, it is required that unless the process passes the message onto another process, the final receiving process must release the mes-<sup>25</sup>sage\_envelope block after its usage. An example of send\_message and receive\_message is shown below.

```
1 void process_send() {27  
    while(1) {  
3        // Request one block for  
        process_receive to use29  
        struct msgbuf*  
        message_envelope31  
5        = (struct msgbuf*)  
        request_memory_block();  
        message_envelope->mtype =  
        MESSAGE_TYPE_USER_DEFINED.1;  
7        strcpy(message_envelope->mtext  
        , "process message");  
        send_message(  
        PROCESS_RECEIVE_ID,  
        message_envelope);
```

```
        // Send another block for  
        process_receive to send to CRT11  
        struct msgbuf*  
        message_envelope13  
        = (struct msgbuf*)  
        request_memory_block();  
        message_envelope->mtype =  
        MESSAGE_TYPE_USER_DEFINED.2;  
        strcpy(message_envelope->mtext15  
        , "print this");  
        send_message(  
        PROCESS_RECEIVE_ID,  
        message_envelope);  
    }  
}  
  
void process_receive() {  
    while(1) {  
        struct msgbuf* message21  
        = (struct msgbuf*)  
        receive_message(NULL);  
        if( message->mtype =  
        MESSAGE_TYPE_USER_DEFINED.2 ) {  
            // Send to CRT for  
            printing; do not release memory23  
            message->mtype =  
            MESSAGE_TYPE_CRT_DISPLAY_REQUEST;  
            send_message(  
            PROCESS_ID_CRT, message);  
        } else {  
            // Do something with the  
            message then release it25  
            release_memory_block( (  
            void*)message );  
        }  
    }  
}
```

## 2.4.3 Delayed Messages

```
int delayed_send(int process_id, void  
*message_envelope, int delay);
```

This primitive is very similar to the primitive of `send_message` in section 2.4.1 with the addition of a delay. Instead of sending the message immediately, the message will be sent delay number of milliseconds. The `message_envelope` is constructed in the same way. The `process_id` will be the receiving process. The execution of the process may be preempted after a the delay period if the receiving process was blocked on `receive_message` and has a higher priority.

## 2.5 Process Priority

The RTX schedules processes based on a fixed priority scheme. Thus, the RTX provides the utility to change priorities and to get the process priority of any user processes.

### 2.5.1 Set Process Priority

```
1 int set_process_priority(int
    process_id, int priority);
```

This primitive allows user processes to set the priorities of other user processes. User processes are not allowed to set the priorities of any system processes. System processes, however, are allowed to set the priorities of other system or i-processes. The primitive validates if the current process is allowed to set the priority of the process with `process_id` and whether it's a valid `process_id`. It also checks the validity of the priority. An error status of `RTX_ERR` is given back if the validation does not pass. If properly set, a status of `RTX_OK` is returned. The caller of this primitive is not blocked but can be pre-

empted if the priority of the set process is higher. Otherwise, the process continues to execute. An example usage is shown in section 2.5.2

### 2.5.2 Get Process Priority

```
1 int get_process_priority(int
    process_id);
```

Given a `process_id`, this primitive gives back the priority of any processes including any system or i-processes to a user process. A invalid `process_id` will result in a `RTX_ERR` status. Otherwise, a `RTX_OK` is returned to the caller. An example usage of both get and set process priority is shown below.

```
1 // Assume this process has a process
  // ID of 3
  // and there is another user process
  // with ID of 1
3 void user_process() {
    int process_priority_3 =
    get_process_priority(3);
    if( process_priority_3 ==
    USER_PROCESS_PRIORITY_MEDIUM ) {
        // Make process 1 have a
        // higher priority
        // This call will preempt
        int status =
        set_process_priority(
            1,
            USER_PROCESS_PRIORITY_HIGH );
        if( status == RTX_ERR ) {
            release_processor();
        }
    }
}
```

# Chapter 3

## Interrupts and Their Handler and Processes

3.1 Description

3.2 UART

3.3 TIMER

# Chapter 4

## System and User Processes

### 4.1 System Processes

#### 4.1.1 Description

The RTX consists of system processes that sit on top of the kernel. These system processes usually have higher priority than user processes because they carry out tasks for the OS that the kernel cannot do otherwise. System processes in this RTX includes the null process, and I/O processes.

#### 4.1.2 Null Process

The design of the RTX requires that a process should always be executing. Thus, if there are no user or other system processes, the null process executes. The only job of the null process is to keep the processor running. It has the lowest priority of any process using the RTX. If any process is added into the ready queue during the execution of this process, a preemption will occur and the new process will take control of the processor. The final implementation follows the following form.

```
function NULL_PROCESS
```

```
    while true do  
        end while  
end function
```

#### 4.1.3 CRT Display Process

The RTX supports output to terminal display. CRT is a system process that the RTX uses to accomplish the task of display. The CRT process depends on the UART I-Process (details of the UART can be found in section TODO). The CRT is implemented using the following algorithm.

```
function CRT_PROCESS  
    while true do  
        message = RECEIVE_MESSAGE(NULL)  
        if message is not a CRT display request then  
            continue  
        end if  
        SEND_MESSAGE(PROCESS_ID_UART, message)  
        Set interrupt bits  
    end while  
end function
```

The CRT display process accepts message requests from any process of type MESSAGE\_TYPE.CRT\_DISPLAY\_REQUEST. This tells the CRT to send the message for display to the UART I-process, which interacts with the hardware registers to output to terminal. Since the UART process is an interrupt process, the interrupt bit must be set in order for the process to run. Thus, after sending the message to UART's mailbox, the CRT process enables the interrupt bits for the I-process to trigger. The UART I-process passes the message character by character to the hardware to display to the terminal.

#### 4.1.4 Keyboard Command Decoder Process

The RTX provides functionality to forward user input commands to a particular process. The keyboard command decoder process (KCD) handles two tasks: giving input to the CRT for terminal display and forwarding certain inputs to different processes for further processing. For the first task, all keyboard inputs are filtered by the KCD. The UART I-process is called when a key is first pressed. The UART I-process (explained further in section TODO) then passes a message to the KCD telling of type key press. In the first case, the KCD will take this message and pass it on to the CRT for terminal display.

In the latter case, processes can register a command with the KCD. Each command registration consists of a single capital letter of the alphabet. If a command is taken, the

KCD will not register the command and exit with an error. After a valid command has been registered, user input with the pattern "%" followed by a upper case character will be parsed by the KCD. If user types in "%" followed by a registered character, the KCD will buffer the input until the user presses the return key. In that scenario, the KCD will pass it onto the registered process. The following algorithm of the KCD explains this in more detail.

```
function PROCESS_INPUT(message)
  if There a "%" character in the buffer
  then
    if message.Content is a command character then
      add to buffer
    else
      clear buffer
    end if
  else if message.Content is "%" then
    add to buffer
  else if message.Content is enter and buffer is not empty then
    send buffer to registered command
  end if
  Send message to CRT to display
end function
```

```
function PROCESS_REGISTER(message)
  if message.Command has not been taken then
    register message.Command
  else
    error
  end if
end function
```

**function** KCD\_PROCESS

    initialization

**while** true **do**

        message = RECEIVE\_MESSAGE(NULL)

**if** message type is key press input **then**

            PROCESS\_INPUT(message)

**else if** message type is command registration **then**

            PROCESS\_REGISTER(message)

**end if**

**end while**

**end function**

on enter.

## 4.2 User Processes

### 4.2.1 Wall Clock Process

### 4.2.2 ‘funProcess’

### 4.2.3 ‘schizophrenicProcess’

### 4.2.4 ‘fibProcess’

### 4.2.5 ‘memoryMuncherProcess’

### 4.2.6 ‘releaseProcess’

An example of this usage is the wall clock process (described in more detail in section 4.2.1). The wall clock process registers the character ‘W’ as a command. The registration will be done by passing a message to the KCD of command registration type. This message is received by the KCD. If the ‘W’ command has not been registered, it becomes registered by the wall clock process. Meanwhile, the user types in random characters. Any character that is not ‘%’ is passed by the KCD to the CRT for display. Once a ‘%’ is received, the KCD checks the next character to see if it’s a registered command. If its not a registered command, the KCD clears the buffer and starts at the beginning. If it is a registered command, the KCD keeps a buffer of all the input characters. This buffer fills with data and is sent to the wall clock process for processing after the user presses enter. The buffer is only 20 characters long. Thus, any data for a process after 20 characters will not be sent to the registered process

# Chapter 5

## Initialization

### 5.1 Description

### 5.2 Memory Layout

### 5.3 Process Mailbox



# Chapter 6

## Major Design Changes

6.1 Description

6.2 Heap

6.3 Scheduler

6.4 What We Learned

# Part III

## Testing and Analysis

# Chapter 7

## Testing

### 7.1 Description

In order to test the operating system there were 3 main methods of testing: Unit Tests, Testing Processes, and Manual Testing. The latter was used for the CRT and KCD. Unit tests were implemented for the various data structures in the operating system. For the different queues that were implemented, the unit tests would check the push and pop functionalities. The heap was also unit tested. In this case there were unit tests in place to validate expected behaviour for memory blocks being continuously freed, validation for misaligned memory, freeing invalid blocks, and double freeing. All unit tests set up a specific state, called a specific method, and had its results verified against the expected.

At each iteration of development, processes were designed to test the newly added features. For instance, after the first iteration 6 processes were implemented to automatically test the functionality of setting priority, releasing processor, releasing memory, requesting memory, blocking, and unblocking. This was accomplished by having the processes use the new functionality, and keeping track of

the number of times each process was running and the order in which they ran. As the processes ran, certain test flags were set to see the progress of the testing suite. At the second and third iteration the processes were designed in the same way but for the following functionality: sending messages, receiving messages, and delayed messages. When all the processes finished running, the results of the tests would be displayed indicating the number of tests that passed and which ones.

While testing the CRT and KCD, the majority of the tests were manual. These processes are involved with keyboard inputs and visual feedback. As such they were tested with manual input and validations.

# Chapter 8

## Timing Analysis

### 8.1 Requesting and Releasing Memory

All data used and processed data can be found in the Appendix in [A](#)

#### 8.1.1 Single Request Single Release

In the first analysis, the request\_memory and release\_memory methods were timed. Single request, single release means that a single memory block was requested and the time taken was recorded. Similarly for releasing a memory block. By doing so we are able to keep the heap in the same state, eliminating the effects of cumulative memory requests (which will be tested in section [8.1.2](#)). The method to time this is shown in the following algorithm. Data was collected across 5 machines, with 1000 calls to each function.

```
function                               SINGLE-  
GLE_REQUEST_SINGLE_RELEASE  
  for  $i = 0$  to 1000 do  
    RequestStartTime[i]  $\leftarrow$  Curent  
    Time
```

```
    memoryBlock  $\leftarrow$  RE-  
QUEST_MEMORY_BLOCK  
    RequestEndTime[i]  $\leftarrow$  Curent Time  
    ReleaseStartTime[i]  $\leftarrow$  Curent  
    Time  
    RELEASE_MEMORY_BLOCK(memoryBlock)  
    ReleaseEndTime[i]  $\leftarrow$  Curent Time  
  end for  
end function
```

Based on the analysis, the mean run time for release memory is 405.44ns. The standard error (marked in the green mean diamonds on the figure) is very small, giving a 95% confidence interval ranging from [404.99, 405.84] to [405.1, 405.95]. The F ratio is incredibly small (0.0474), meaning that there is little to no variance among the different machines and as such it can be concluded that the hardware does not greatly affect the performance of the method.

The results of releasing memory are very similar to that of requesting memory. The mean time taken to release memory is 1229.237ns. The variations among different machines were very small, giving an F Ratio of 0.0328. This shows that the performance

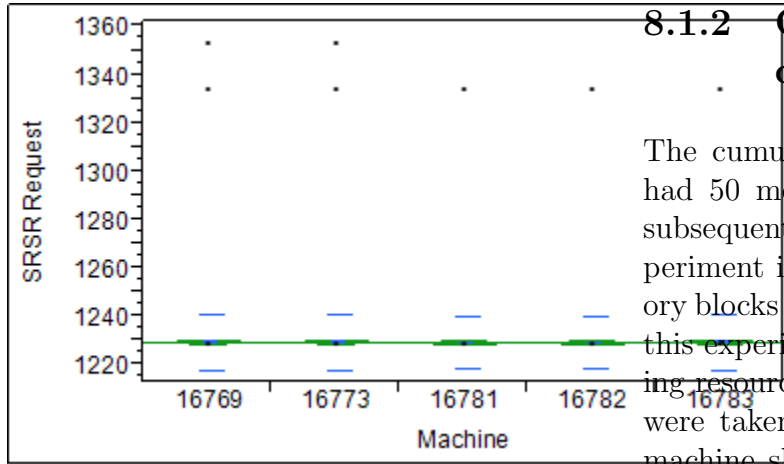


Figure 8.1: Request Memory over 5 machines

is consistent across different hardware. The performance was as well consistent within itself, the fastest machine had a 95% confidence interval of  $[1228.5, 1229.9]$ , and for the slowest  $[1228.6, 1230.0]$ . Releasing memory takes on average a little under three times as long as requesting memory

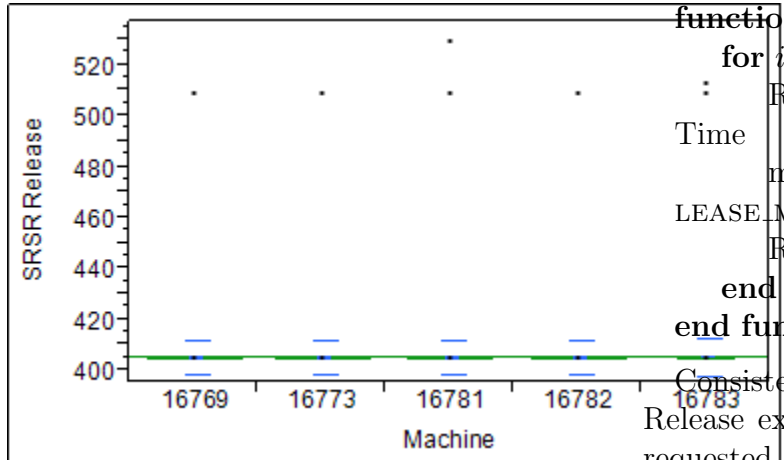


Figure 8.2: Release Memory over 5 machines

### 8.1.2 Cumulative Memory Requests

The cumulative memory requests condition had 50 memory blocks requested and then subsequently released. This differs from experiment in 8.1.1 where the requested memory blocks do not accumulate. The purpose of this experiment is to see the effect of depleting resources on run time. Once again, trials were taken over 5 machines, however, each machine showed the same performance. To achieve this, the following two method calls were made.

**function CUMULATIVE\_REQUEST**

```

for  $i = 0$  to 50 do
    RequestStartTime[i]  $\leftarrow$  Current Time
    memoryBlock  $\leftarrow$  REQUEST_MEMORY_BLOCK
    RequestEndTime[i]  $\leftarrow$  Current Time
end for
end function

```

**function CUMULATIVE\_RELEASE**

```

for  $i = 0$  to 50 do
    ReleaseStartTime[i]  $\leftarrow$  Current Time
    memoryBlock  $\leftarrow$  RELEASE_MEMORY_BLOCK
    ReleaseEndTime[i]  $\leftarrow$  Current Time
end for
end function

```

Consistent with the Single Request, Single Release experiment, the first memory block requested took 1229ns. A linear trend line is overlaid on top showing an  $R^2$  value of 1 (a perfect trend). The trend shows that for

each additional memory block requested, the run time increase by 17ns. This linear trend is in accordance with the implementation of the request memory method which has  $O(n)$  complexity described in 2.2.2.

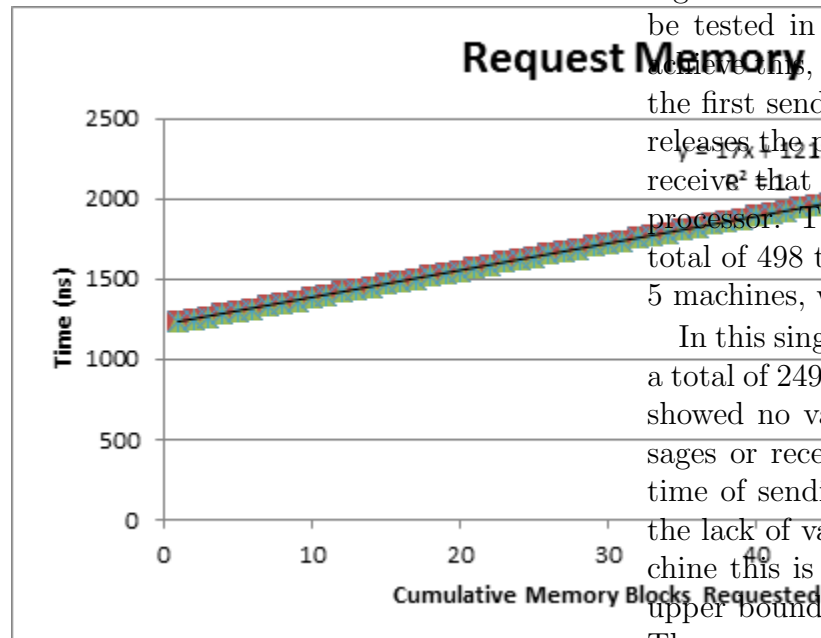


Figure 8.3: Release Memory over 5 machines

## 8.2 Sending and Receiving Messages

### 8.2.1 Single Send, Single Receive

In this experiment, the `send_message` and `receive_message` methods were timed. A single message was sent and the time taken was recorded. Similarly, the time taken to receive

that message was timed, specifically the time for the function call and not the delay between the time the message was sent and the time the message was received. By doing so we are able to eliminate the effects of having inconsistent memory available (that will be tested in the following experiment). To achieve this, two user processes were defined, the first sends a message of a fixed size then releases the processor, the second would then receive that message and again release the processor. This process would then repeat a total of 498 times. Data was collected across 5 machines, with 498 calls to each function.

In this single send, single receive condition, a total of 2490 trials were taken, and the data showed no variation for either sending messages or receiving messages. The mean run time of sending a message is 399ns. Due to the lack of variation on or between each machine this is also the value of the lower and upper bound of the 95% confidence interval. The same applies for receiving the message, which is twice as fast at 182ns.

### 8.2.2 Cumulative Message Sending

The cumulative message sending condition, as the name implies, had messages accumulate before any were received. The purpose of this is to see the relation between the number of messages sent (and not yet received) with the time it takes to send a new message. In order to test this, 50 messages were sent from within a single user processes. Once finished, another process would take

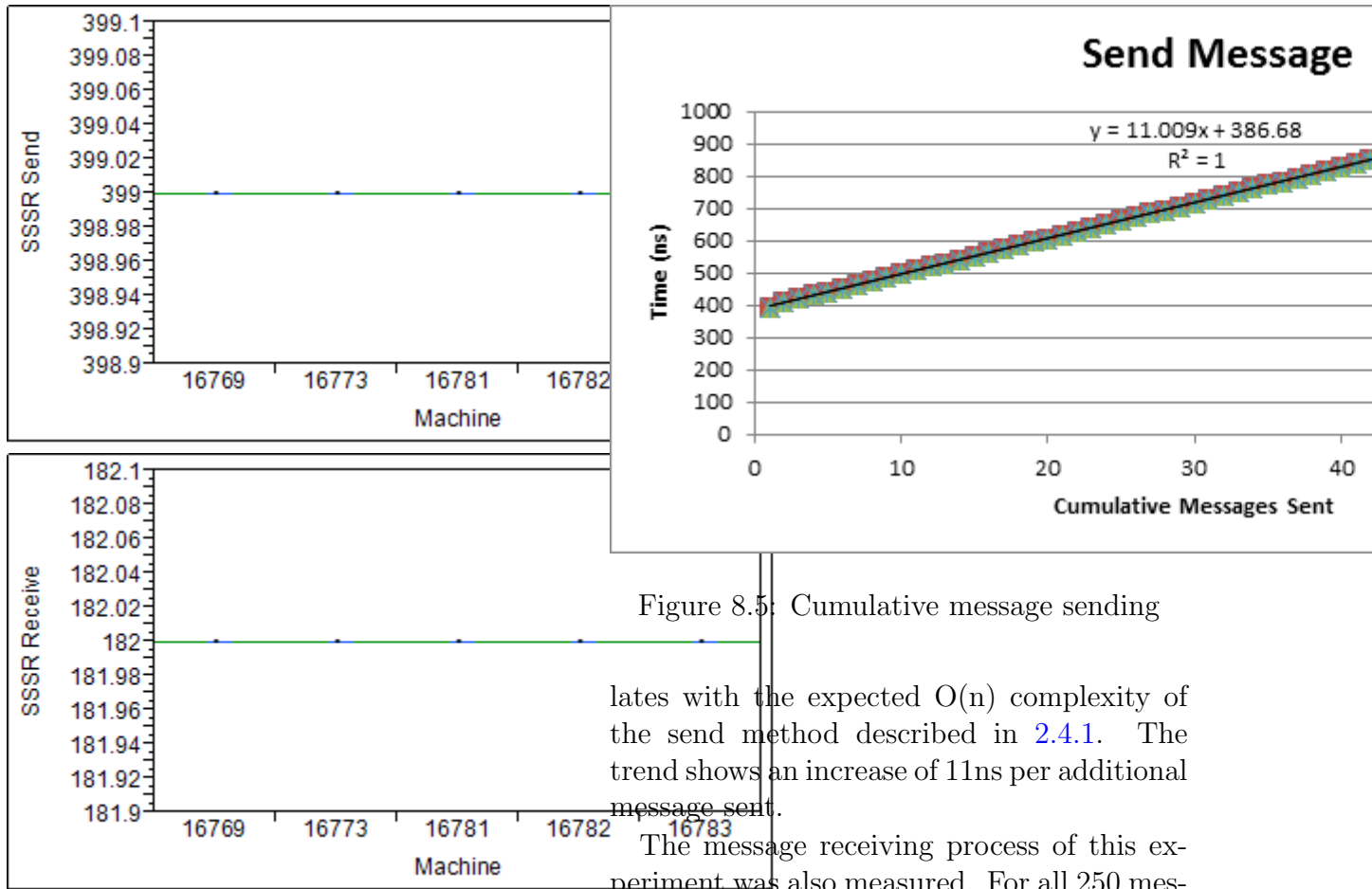


Figure 8.5: Cumulative message sending

Figure 8.4: Top: Sending messages; Bottom: Receiving messages

lates with the expected  $O(n)$  complexity of the send method described in 2.4.1. The trend shows an increase of 11ns per additional message sent.

The message receiving process of this experiment was also measured. For all 250 messages received, the time taken was consistent at 182ns. This is as well expected due to the  $O(1)$  complexity of the receive message method describe in 2.4.2.

over and receive those 50 messages. The results from 5 different machines are displayed in the Figure.

Similar to the single send, single receive condition in 8.2.1, the results were the same across all machines. A linear trend line is superimposed onto the results. The trend displayed has an  $R^2$  value of 1 showing that this linear trend is very significant. This corre-

### 8.2.3 Effects of Message Length

Another factor that was tested is the length of the message being sent. Depending on the implementation for the kernel it is possible that in some cases a longer message will take longer to send or receive. To test this, messages were continuously sent and received in

the same manner as the single send, single receive condition in 8.2.1 with the addition of an accumulating message (ie the first message sent was a, the second message aa and so forth). This was tested up until a message of 50 characters was sent. As shown in the two figures, the message length had no effect on the time taken to send or receive messages.

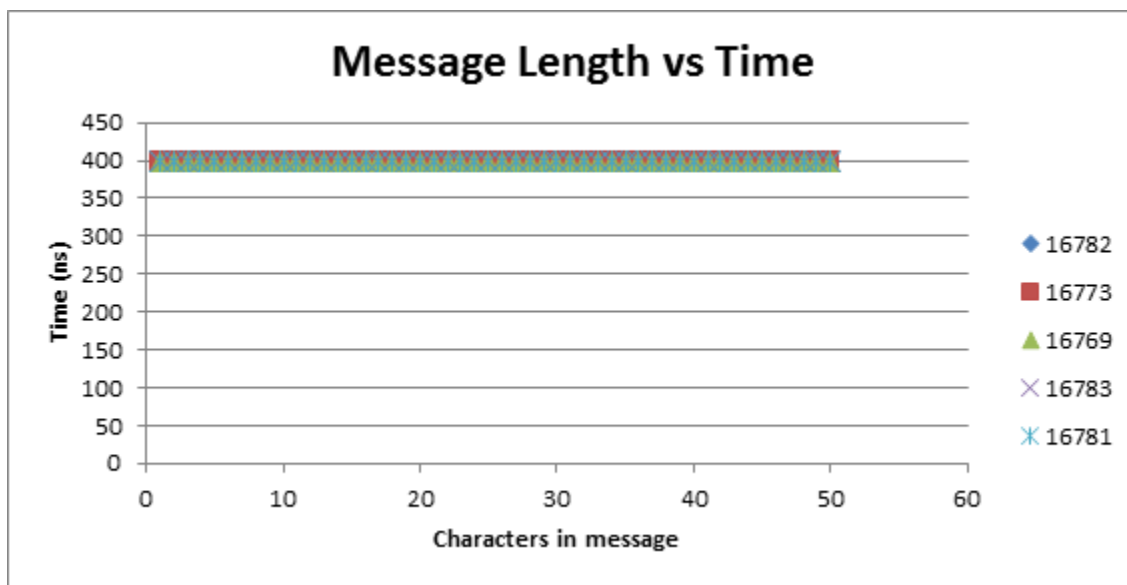


Figure 8.6: Sending messages with variable message length



# Appendix A

## Raw Measurement Data

### A.1 Code Used To Collect Timing Data

```
static void procl(void)
{
    timer_test_init();
    static const int memory_block_count
        = 50;
    static void* memory_blocks[
        memory_block_count];

    int start_time = 0;
    int end_time = 0;
    LOG("Started single request single
        release");
    for (int i = 0; i < 1000; i++) {
        start_time = s_test_timer->TC;
        memory_blocks[0] =
        request_memory_block();
        end_time = s_test_timer->TC;
        LOG("Request: %d", end_time -
            start_time);
        start_time = s_test_timer->TC;
        release_memory_block(memory_blocks
            [0]);
        end_time = s_test_timer->TC;
        LOG("Release: %d", end_time -
            start_time);
    }
    LOG("Finished single request single
        release");
```

```
LOG("Started requesting");
for (int i = 0; i <
    memory_block_count; i++) {
    start_time = s_test_timer->TC;
    memory_blocks[i] =
    request_memory_block();
    end_time = s_test_timer->TC;
    LOG("%d", end_time - start_time);
}
LOG("Finished requesting");
test_results[REQUEST_MEMORY_TEST] =
    1;
LOG("Started releasing");
for (int i = 0; i <
    memory_block_count; i++) {
    start_time = s_test_timer->TC;
    release_memory_block(memory_blocks
        [i]);
    end_time = s_test_timer->TC;
    LOG("%d", end_time - start_time);
}
LOG("Finished releasing");

test_results[RELEASE_MEMORY_TEST] =
    1;
set_process_priority(1,
    USER_PROCESS_PRIORITY_LOWEST);
set_process_priority(2,
    USER_PROCESS_PRIORITY_HIGH);
while (1) {
    s_iteration_count++;
    release_processor();
```

```

    }
46 }
48 //...
50 static int const required_messages =
    50;
52 static void proc3(void)
{
54     LOG("Started receiving messages");
    for (int i = 0; i <
        required_messages; i++) {
56         int start_time = 0;
        int end_time = 0;
58         start_time = s_test_timer->TC;
        void* message = receive_message(
            NULL);
60         end_time = s_test_timer->TC;
        LOG("Receive: %d", end_time -
            start_time);
62         release_memory_block(message);
    }
64     LOG("Finished receiving messages");
    set_process_priority(5,
        USER_PROCESS_PRIORITY_HIGH);
66     set_process_priority(4,
        USER_PROCESS_PRIORITY_LOWEST);
    set_process_priority(3,
        USER_PROCESS_PRIORITY_LOWEST);
68     release_processor();
}
70
static void proc4(void)
72 {
    LOG("Started sending messages");
74     for (int i = 0; i <
        required_messages; i++) {
        struct msgbuf* message_envelope =
            (struct msgbuf*)
            request_memory_block();
76         message_envelope->mtype = 10;
        strcpy(message_envelope->mtext,
            test_phrase);
78         int start_time = 0;
        int end_time = 0;
80         start_time = s_test_timer->TC;
        send_message(3, message_envelope);
82         end_time = s_test_timer->TC;
        LOG("Send: %d", end_time -
            start_time);
84     }
    LOG("Finished sending messages");
86     set_process_priority(3,
        USER_PROCESS_PRIORITY_HIGH);
    set_process_priority(4,
        USER_PROCESS_PRIORITY_LOWEST);
88     release_processor();
}
90
int sample_size = 500;
static void proc5(void)
{
92     set_process_priority(6,
        USER_PROCESS_PRIORITY_HIGH);
    void* message = receive_message(NULL);
94     release_memory_block(message);

    while (true) {
96         int start_time = 0;
        int end_time = 0;
98         start_time = s_test_timer->TC;
        void* message = receive_message(
            NULL);
100         end_time = s_test_timer->TC;
        LOG("Receive: %d", end_time -
            start_time);
102         release_memory_block(message);
        release_processor();
104     }
}
106
static void proc6(void)
{
108     LOG("Started send-receive together")
        ;
    for (int i = 0; i < sample_size; i
        ++){
110
112

```

```

114 struct msgbuf* message_envelope =
    (struct msgbuf*)
    request_memory_block();
    message_envelope->mtype = 10;
116 strcpy(message_envelope->mtext,
    test_phrase);
    int start_time = 0;
118 int end_time = 0;
    start_time = s_test_timer->TC;
120 send_message(5, message_envelope);
    end_time = s_test_timer->TC;
122 LOG("Send: %d", end_time -
    start_time);
    release_processor();
124 }
    LOG("Finished send-receive together"
    );
126
    LOG("Started variable message length
    together");
128 for (int i = 0; i < sample_size; i
    ++){
        struct msgbuf* message_envelope =
            (struct msgbuf*)
            request_memory_block();
            message_envelope->mtype = 10;
130 for (int j = 0; j < i; j++){
                message_envelope->mtext[j] = *a
                ";
            }
134 int start_time = 0;
            int end_time = 0;
            start_time = s_test_timer->TC;
136 send_message(5, message_envelope);
            end_time = s_test_timer->TC;
138 LOG("Send: %d", end_time -
            start_time);
            release_processor();
140 }
    LOG("Finished variable message
        length together");
142 }

```

## A.2 Data Analysis For Single Request Single Release

### A.2.1 Request

Summary of Fit

Rsquare	2.63E-05
Adj Rsquare	-0.00077
Root Mean Square Error	11.43196
Mean of Response	1229.237
Observations (or Sum Wgts)	5000

Analysis of Variance

Source	DF	Sum of Square	Mean Square	F Ratio
Machine	4	17.14	4.285	0.0328
Error	4995	652795.5	130.69	-
C. Total	4999	652812.7	-	-

Means of Oneway Anova

Machine	Number	Mean	Std Error	Lower 95%	Upper 95%
16769	1000	1229.29	0.36151	1228.6	1229.9
16773	1000	1229.29	0.36151	1228.6	1229.9
16781	1000	1229.17	0.36151	1228.5	1229.8
16782	1000	1229.17	0.36151	1228.5	1229.8
16783	1000	1229.27	0.36151	1228.6	1229.9

Means and Std Deviations

Machine	Number	Mean	Std Dev	Std Err Mean	Lower 95%	Upper 95%
16769	1000	1229.29	11.7342	0.37107	1228.5	1229.9
16773	1000	1229.29	11.7342	0.37107	1228.5	1229.9
16781	1000	1229.17	11.0616	0.3498	1228.5	1229.8
16782	1000	1229.17	11.0616	0.3498	1228.5	1229.8
16783	1000	1229.27	11.5476	0.36517	1228.5	1229.9

## A.2.2 Release

Summary of Fit

Rsquare	3.79E-05
Adj Rsquare	-0.00076
Root Mean Square Error	6.8087
Mean of Response	405.4416
Observations (or Sum Wgts)	5000

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio	Prob > F
Machine	4	8.79	2.1968	0.0474	0.9958
Error	4995	231560.2	46.3584		
C. Total	4999	231569	-		

Means for Oneway Anova

Machine	Number of Trials	Mean	Std Error	Lower 95%	Upper 95%
16769	1000	405.416	0.21531	404.99	405.84
16773	1000	405.416	0.21531	404.99	405.84
16781	1000	405.436	0.21531	405.01	405.86
16782	1000	405.416	0.21531	404.99	405.84
16783	1000	405.524	0.21531	405.10	405.95
	16783		498	399	0

Means And Std Deviations

Machine	Number	Mean	Std Dev	Std Err Mean	Lower 95%	Upper 95%
16769	1000	405.416	6.56765	0.20760	399.405	405.82
16773	1000	405.416	6.56765	0.20760	399.405	405.82
16781	1000	405.436	6.90519	0.21836	399.405	405.86
16782	1000	405.416	6.56765	0.20760	399.405	405.82
16783	1000	405.524	7.39649	0.23390	399.405	405.98

## A.3 Data Analysis for Single Send Single Receive

### A.3.1 Send

Summary of Fit

Rsquare	-
Adj Rsquare	-
Root Mean Square Error	0
Mean of Response	399
Observations (or Sum Wgts)	2490

Analysis of Variance

Source	DF	Sum of Square	Mean Square	F Ratio
Machine	4	0	0	-
Error	2485	0	0	-
C. Total	2489	0	-	-

Means for Oneway Anova

Machine	Number	Mean	Std Error	Lower 95%	Upper 95%
16769	1000	405.416	0.21531	404.99	405.84
16773	1000	405.416	0.21531	404.99	405.84
16781	1000	405.436	0.21531	405.01	405.86
16782	1000	405.416	0.21531	404.99	405.84
16783	1000	405.524	0.21531	405.10	405.95
	16783		498	399	0

Means and Std Deviations

Machine	Number	Mean	Std Dev	Std Err Mean	Lower 95%	Upper 95%
16769	1000	405.416	6.56765	0.20760	399.405	405.82
16773	1000	405.416	6.56765	0.20760	399.405	405.82
16781	1000	405.436	6.90519	0.21836	399.405	405.86
16782	1000	405.416	6.56765	0.20760	399.405	405.82
16783	1000	405.524	7.39649	0.23390	399.405	405.98

### A.3.2 Receive

Summary of Fit

Rsquare	-
Adj Rsquare	-
Root Mean Square Error	0
Mean of Response	182
Observations (or Sum Wgts)	2490

Analysis of Variance

Source	DF	Sum of Square	Mean Square	F Ratio	Prob > F
Machine	4	0	0	-	-
Error	2485	0	0	-	-
C. Total	2489	0	-	-	-

Means of Oneway Anova

Machine	Number	Mean	Std Error	Lower 95%	Upper 95%
16769	498	182	0	182	182
16773	498	182	0	182	182
16781	498	182	0	182	182
16782	498	182	0	182	182
16783	498	182	0	182	182

Means and Std Deviations

Machine	Number	Mean	Std Dev	Std Err Mean	Lower 95%	Upper 95%
16769	498	182	0	0	182	182
16773	498	182	0	0	182	182
16781	498	182	0	0	182	182
16782	498	182	0	0	182	182
16783	498	182	0	0	182	182