**UNIVERSITY OF WATERLOO**

Faculty of Engineering

**Software Design Description for MTE 241Real Time Operating System Project**

Prepared by

Zhuojun Li [20359456]

Bongkyun Park [20340649]

Jawad Ateeq [20327861]

Yifei Cheng [20344112]

2B Mechatronics Engineering

Contents

[1 Introduction 4](#_Toc306819038)

[2 Global Information 5](#_Toc306819039)

[2.1 RTX.h 5](#_Toc306819040)

[2.2 UserAPI.h 10](#_Toc306819041)

[2.3 Keyboard.h 10](#_Toc306819042)

[2.4 CRT.h 10](#_Toc306819043)

[2.5 Queue.h 11](#_Toc306819044)

[3 Primitives 11](#_Toc306819045)

[3.1 Kernel Primitives 11](#_Toc306819046)

[3.2 User Primitives 16](#_Toc306819047)

[4 Processes 16](#_Toc306819048)

[4.1 User Processes A, B, and C 17](#_Toc306819049)

[4.2 Keyboard Process 17](#_Toc306819050)

[4.3 CRT Process 18](#_Toc306819051)

[4.4 Wall Clock Process 20](#_Toc306819052)

[4.5 Command Console Interface Process 21](#_Toc306819053)

[4.6 Interrupt Handling 24](#_Toc306819054)

[5 Kernel Entry 30](#_Toc306819055)

[6 Initialization 30](#_Toc306819056)

[6.1 OS data structure construction 30](#_Toc306819057)

[6.2 Initialization Table 31](#_Toc306819058)

[6.3 Message Envelopes 33](#_Toc306819059)

[6.4 Process creation 33](#_Toc306819060)

[7 Process selection 36](#_Toc306819061)

[8 Proccess\_switch 36](#_Toc306819062)

[9 Plan for Implementation 37](#_Toc306819063)

[10 Reference 38](#_Toc306819064)

**List of Figures**

No table of figures entries found.

**List of Tables**

[Table 1: Header files and their function prototypes 10](#_Toc306819657)

# Introduction

The purpose of this document is to illustrate the complete design description of a small real time executive (RTX).

The software design description (SDD) shows all data structures, *struct'*s and their relationships, each individual function and major internal procedure with input/output parameters, return values and, where appropriate, a description of its functionality and sufficiently complete pseudo code.

The inter-process and internal primitive interaction diagrams as well as a brief section outlining the planned division of implementation and testing responsibilities among the group members are also presented [1].

The following are the structure and class diagram of the RTX:

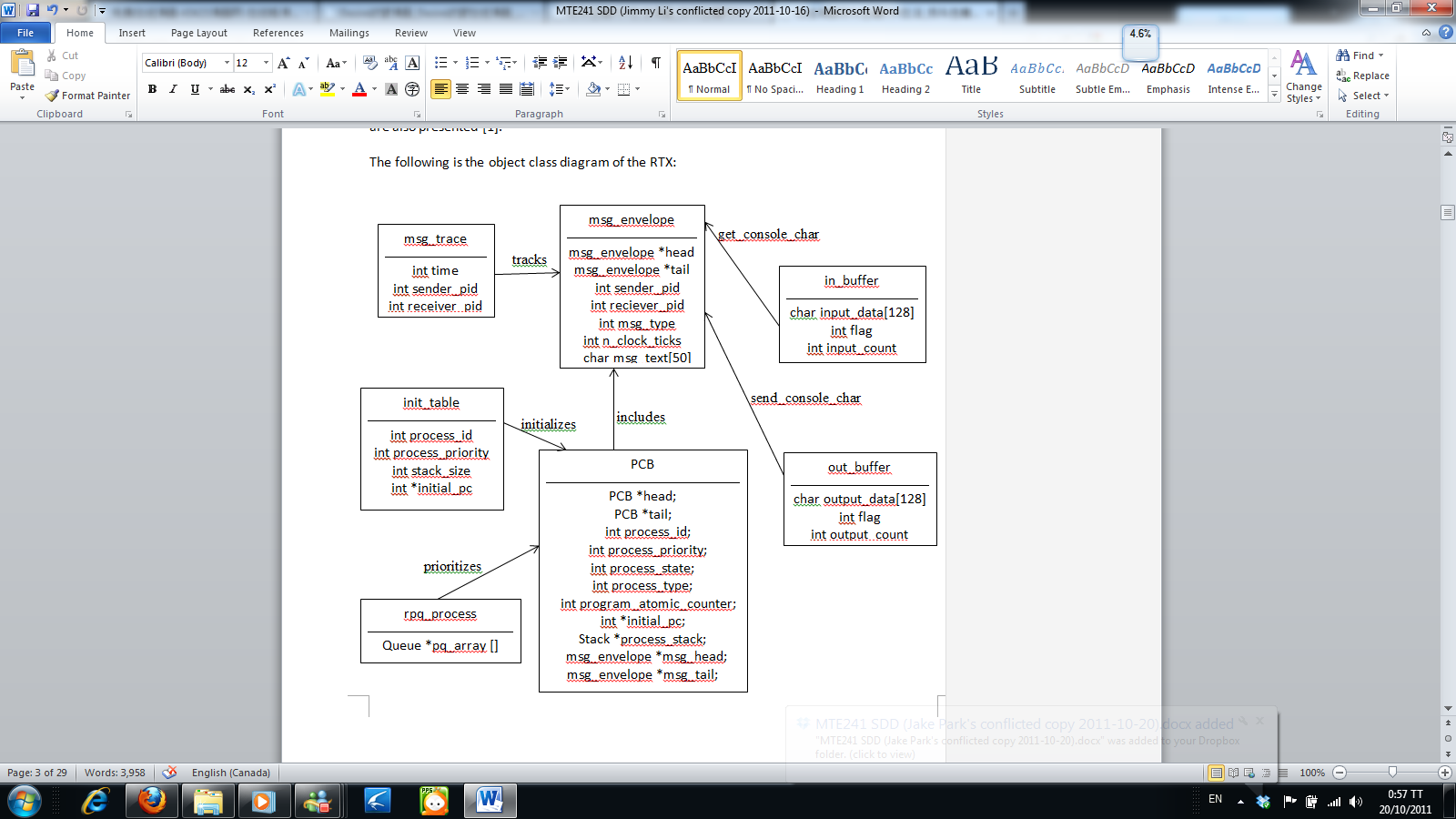


Figure 1: Class Diagram

# 

Figure 2: Structure Diagram

# Global Information

## RTX.h

RTX.h provides definition for kernel APIs and kernel data field and objects. The following code defines the constants, process IDs (PIDs), Process states, IPC variables, message types, error codes and user signals.

/\*\*Constants\*\*/

//RTX Processes

#define NUM\_OF\_USER\_PROC 5;

#define NUM\_OF\_IPROC 3;

//Process IDs

#define PROC\_A 0;

#define PROC\_B 1;

#define PROC\_C 2;

#define PROC\_CCI 3;

#define PROC\_CLK 4;

#define IPROC\_KBD 5;

#define IPROC\_CRT 6;

#define IPROC\_TIMER 7;

#define PROC\_NULL 8;

//Process priorities

#define IPROCESS 0 //used for iprocesses

#define High 1

#define MEDIUM 2

#define LOW 3

//Process state

#define READY 0;

#define EXECUTING 1;

#define INTERRUPTED 2;

#define BLOCKED\_ON\_ENVELOPE 3;

#define BLOCKED\_ON\_RECIEVE 4;

//IPC

#define MAX\_NUM\_MSG\_ENV 150;

#define MAX\_CHAR = 128;

//Message types

#define DISPLAY\_ACK 0;

#define CONSOLE\_INPUTE 1;

#define TERMINATED 2;

#define INCREMENT\_CLOCK 3;

#define CHANGE\_CLOCK 4;

#define STOP\_CLOCK 3;

//Error Codes

#define ERROR\_BAD\_MEMORY\_UNMAP -1;

#define ERROR\_BAD\_FILE\_CLOSE -2;

#define ERROR\_INVALID\_PID -3;

#define ERROR\_INVALID\_MID -4; //invalid message id

#define ERROR\_FAIL\_TO\_MALLOC -5;

#define ERROR\_INVALID\_PRIORITY -6;

//Error Codes

#define PROC\_A\_PRIORITY

#define ERROR\_BAD\_FILE\_CLOSE -2;

#define ERROR\_INVALID\_PID -3;

#define ERROR\_INVALID\_MID -4; //invalid message id

#define ERROR\_FAIL\_TO\_MALLOC -5;

#define ERROR\_INVALID\_PRIORITY -5;

//Buffer Size for TX and RX:

#define BUFFER\_SIZE = 128; //bytes

The following code describes the various data structures that are private to the RTX. These include the message envelope, initialization table, PCB, TX and RX memory maps, the message trace buffers, queues and ready process queues.

For message envelopes, the message text field should be less than or equal to 128 bytes because the size of the buffer used in RX and TX shared memory map is 128 bytes.

typedef struct msg\_envelope {

msg\_envelope \*head;

msg\_envelope \*tail;

int sender\_pid;

int reciever\_pid;

int msg\_type;

int n\_clock\_ticks;

char msg\_text[128];

} msg\_envelope;

typedef struct init\_table {

int process\_id;

int process\_priority;

int stack\_size;

int \*initial\_pc;

}

typedef PCB{

PCB \*head;

PCB \*tail;

int process\_id;

int process\_priority;

int process\_state;

int program\_atomic\_counter;

int \*initial\_pc;

char \*process\_stack;

jmp\_buffer context;

msg\_envelope \*msg\_head;

msg\_envelope \*msg\_tail;

} PCB;

//struct of input buffer of RX memory mapped space:

typedef in\_buffer {

char input\_data[128]; // 128 byte buffer stream

int flag; //flag to indicate that input has been buffered. 1 indicates input is ready; 0

otherwise.

int input\_count; counter of buffered input

} input\_buffer

//struct of output buffer of TX memory mapped space:

typedef out\_buffer{

char output\_data[128] // 128 byte buffer stream

int flag;//flag to indicate that output has been buffered.

int output\_count; //counter of buffered output

} output\_buffer

//the send trace buffer

Struct send\_trace\_buffer{

msg\_trace send\_trace\_buffer\_array[16];

}send\_trace\_buffer

//the receive trace buffer

Struct receive\_trace\_buffer{

msg\_trace receive\_trace\_buffer\_array[16];

} receive\_trace\_buffer;

//this structure is used as a record in the send and receive trace buffers

struct msg\_trace{

int time;

int sender\_pid;

int receiver\_pid;

} msg\_trace

struct Queue{

PCB \*head;

PCB \*tail;

}

struct rpq\_process{

Queue \*pq\_array[4];

}

Kernel data fields used by RTX processes are shown below.

PCB \*PCB\_ptr current\_process; //stores the current process context

Queue free\_env\_Q;

Queue blocked\_on\_resource\_Q;

Queue sorted\_timeout\_lst;//used by timer services to store timeout requests

int size\_of\_stack = 16kb;

PCB \*pcb\_pointer\_tracker[NUM\_OF\_PROCESSES]; //Process list

rpq\_process rpq;

int kernel\_time = 0; //internal clock time

in\_buffer \*in\_mem\_ptr; // Pointer to input buffer

out\_buffer \*in\_out\_ptr; // Pointer to output buffer

The Kernel API functions implement the actual functionality that is required by the project description. The following code snippet gives a quick overview of what primitives are implemented.

/\*\*Kernel API\*\*/

int k\_send\_message( int dest\_process\_id, MsgEnv \* msg\_envelope );

MsgEnv \* k\_receive\_message( );

MsgEnv \* k\_request\_msg\_env( );

int k\_release\_ msg\_env ( MsgEnv \* memory\_block );

int k\_release\_processor( );

int k\_request\_process\_status( MsgEnv \* memory\_block );

int k\_terminate( );

int k\_change\_priority(int new\_priority, int target\_process\_id);

int k\_request\_delay( int time\_delay, int wakeup\_code, MsgEnv \* message\_envelope );

int k\_send\_console\_chars(MsgEnv \* message\_envelope );

int k\_get\_console\_chars(MsgEnv \* message\_envelope );

int k\_get\_trace\_buffers( MsgEnv \* message\_envelope);

The Interrupt handling interface is as follows:

void atomic (bool on) //'on' is be defined as 0 for true

void cleanup();

void die(int signal);

void kb\_handler();

void crt\_handler();

void tick\_handler(); //handles the SIGALARM signal

void exception handler(int signum); //calls the correct handler for the signal received

The ready priority queue interface includes the following functions:

Int enqueue\_PCB (PCB \*proc\_to\_enqueue);

Int dequeue\_PCB();

Int rpq\_enqueue(PCB \*proc\_to\_enqueue);

int rpq\_dequeue();

int enqueue\_msg(MsgEnv \*msg\_to\_enqueue);

# 

# Primitives

The primitives provided in this section are used by user processes. Since user processes cannot enter the kernel by itself, most of the actual implementation is done by kernel primitives.

Most of the kernel primitives here follow the logic provided in the documents “Tutorial\_Week3\_RTX\_Details” and “An Example Design of a Non-preemptive Real-Time Kernel”.

## Kernel Primitives

This section describes the kernel primitives that can work in the kernel’s supervisor mode without interruption.

The K\_send\_message and K\_receive\_message primitives are used to communicate data and process statuses between processes through the act of passing envelopes. K\_send\_message sends a message through configuring the fields in the envelope passed to it, and then enqueuing that envelope onto another process’s message queue.

int K\_send\_message ( int dest\_process\_id, MsgEnv \* msg\_envelope ) {

verify that the destination process exists, if not return ERROR\_INVALID\_PID

get sender\_id and dest\_id and store them into envelope;

use dest\_id to look up the target process PCB pointer;

enqueue envelope onto the message queue of the target process;

store the details of this send transaction on the send\_trace\_buffer

if (target process state is blocked\_on\_receive) {

set state to ready;

enqueue blocked process to ready queue;

}

return 0;

}

MsgEnv \* K\_receive\_message() {

while ( process own message queue is empty) {

change process state to block\_on\_receive;

process\_switch();

}

dequeue the head envelope from msg queue;

store the details of this receive transaction on the receive\_trace\_buffer

return a pointer to this envelope;

}

Since no envelopes can be created after initialization, in order for a process to perform the send\_message operation, it must first acquire a empty message envelope. This is done through K\_request\_msg\_env. K\_request\_msg\_env checks the free envelope queue for an unused envelope, if yes then that envelope is dequeued and returned to the calling process. If not then the process become blocked.

MsgEnv \* K\_request\_msg\_env() {

while (free\_env\_Q is empty) {

set process state to blocked\_on\_resource;

process\_switch();

}

dequeue a blank envelope from free\_env\_Q;

return pointer to this envelope;

}

Often times, after a process receives an envelope and performs all required work to handle it, for effective memory management it should be returned to the operating system. K\_release\_msg\_env allows a process to voluntarily surrender an envelope by dequeuing it to the free evenlope queue. This primitive can also cause a blocked process to become ready.

MsgEnv \* K\_release\_msg\_env(Msg \* msg\_env\_ptr) {

remove all fields in the msg env;

enqueue onto free\_env\_Q;

if (blocked\_on\_resource queue is not empty) {

set state of first pcb on blocked\_on\_resource queue to ready;

dequeue from blocked\_on\_resource queue;

}}

The K\_release\_processor primitive surrenders the control of the CPU by a process. This is done through putting the executing process’s PCB onto the ready queue, and then calling process\_switch to select the next process to be executed.

int K\_release\_processor () {

set state of process to ready;

enqueue onto ready queue;

process\_switch();

return 0;

}

In order to display the priorities and statuses of all processes in the RTX, a 1D array called pcb\_pointer is used to record all PCB pointers. This process list centralizes the location of all PCB pointers so that acquiring process statuses and Ids is an inexpensive operation. At initialization, this array is instantiated and all PCB pointers are loaded into it. K\_request\_process\_status loops through pcb\_pointer, collects all required information from each PCB, and stores them in a single message envelope. An example output is provided below:

int K\_request\_process\_status( MsgEnv \* msg\_env\_ptr ) {

msg\_env\_ptr->msg\_text = “Total processes: “ + pcb\_pointer\_tracker.size() + “; (Process\_ID, Priority, State);”;

for (each entry i in pcb\_pointer\_tracker array) {

if (pcb\_pointer\_tracker[i]->process\_state != TERMINATED) { //we do not want to include processes that are terminated

msg\_env\_ptr->msg\_text += “(” + pcb\_pointer\_tracker[i]->process\_id + “,” + pcb\_pointer\_tracker[i]->priority + “,” + pcb\_pointer\_tracker[i]->process\_state + “) ;”;

}}

return SUCCESS;

}

The terminate process ensures that any shared memory is released before the RTX shuts down. All child processes are killed before termination.

int k\_terminate(){

retCode = 0;

kill KB and CRT child processes

unmap memory maps for both processes

if (bad unmaps)

retCode = BAD\_UNMAP;

close temp memory map files and unlink them (i.e delete them)

if (bad file closes)

retCode = ERROR\_BAD\_FILE\_CLOSE;

kill rtx process and return control to Linux

return retCode;

}

The following primitive accesses the PCB of the target process passed to it as a parameter and changes its priority to the required priority.

int k\_change\_priority(int new\_priority, int target\_process\_id){

return ERROR\_INVALID\_PID or ERROR\_INVALID\_PRIORITY if process or priority are illegal

if (target\_process\_id and new\_priority are valid){

PCB \* apcb = getPCBbyPID (target\_process\_id)

apcb -> priority = new\_priority

retCode = 0;

invoke scheduler so that process is enqueued onto appropriate queue

}

return retCode;

}

The k\_request\_delay primitive works by sending a message to the timing service with the delay peroid enclosed in the message text.

int k\_request\_delay( int time\_delay, int wakeup\_code, MsgEnv \* message\_envelope )

{

if (time\_delay, wakeup\_code and message\_envelope != NULL){

message\_envelope -> n\_clock\_ticks = time\_delay; // popoulate num of clock ticks desired inside message

message\_envelope -> message\_text = wakeup\_code; // write message with code timing service will reply with after delay is over

int retCode = K\_send\_message(TSERVICE\_ID, message\_envelope);

return retCode

}}

The k\_send\_console\_chars primitve sends a message containing the output to be displayed to the CRT handler process. It then calls the crt\_handler process so it can check its message queue.

int k\_send\_console\_chars(MsgEnv \* message\_envelope ){

int retCode = 0;

if (message\_envelope != NULL){

retCode = k\_send\_message(PROC\_CRT\_ID, message\_envelope);

crt\_handler();

}

else

retCode = ERROR\_INVALID\_MID;

return retCode;

}

The k\_get\_console\_chars primitve sends an empty envelope to the kB handler process to be filled with the keyboard output. It then calls the kb\_handler process so it can check its message queue.

//run by i-process handling the signal from UART receiver

int k\_get\_console\_chars(MsgEnv \* message\_envelope ){

int retCode = 0;

if (message\_envelope != NULL){

retCode = k\_send\_message(PROC\_KB\_ID, message\_envelope);

kb\_handler(); //make sure kb process does not also send a signal to iproc

}

else

retCode = ERROR\_INVALID\_MID;

return returnCode;

}

This routine accesses the send\_trace\_buffer and receive\_trace\_buffer

int k\_get\_trace\_buffers( MsgEnv \* message\_envelope){

int retCode = 0;

if (message\_envelope != NULL){

//get all 16 elements of send and receive trace buffers and copy them into the message envelope

get the contents of the send\_trace\_buffer and receive\_trace\_buffer trace buffers and store them in the message text

//return the envelope to the invoking process

int dest\_pid = message\_envelope -> source\_pid;

int return\_code = send\_message (dest\_pid, message\_envelope);

return return\_code;}

else

retCode = ERROR\_INVALID\_MID;

return retCode;

}

**Note:**

In order to display the priorities and statuses of all processes in the RTX, a 1D array called pcb\_pointer is used to record all PCB pointers. This process list centralizes the location of all PCB pointers so that acquiring process statuses and Ids is an inexpensive operation. At initialization, this array is instantiated and all PCB pointers are loaded into it. K\_request\_process\_status loops through pcb\_pointer, collects all required information from each PCB, and stores them in a single message envelope. An example output is provided below:

Total processes: 4; (Process\_ID, Priority, State); (1, 0, 2); (2, 3, 0); (3, 3, 0); (3, 2, 1)

## User Primitives

Every kernel primitive described in section 3.1 has an associated user primitive. User primitives invoke their kernel counterparts while ensuring atomicity. User primitives and their corresponding kernel primitives share the same parameters and return values. An example of a user primitive in the user API is shown below.

int terminate()

{

int retCode;

atomic(on);

retCode = k\_terminate();

atomic(off);

return retCode;

}

## Kernel Entry

User processes access kernel APIs indirectly by invoking user visible APIs. When the user APIs are invoked, the atomic (boolean on) function is used to mask all interrupt signals from other processes. Then, corresponding kernel APIs are invoked to execute whatever is requested from processes, with full access to kernel data field and objects.

# Processes

This section of the report aims to provide pseudo code for the various processes that run during the RTX execution.

## User Processes A, B, and C

These processes are provided in the project description document. They are user processes that utilize the user API to interact with the RTX.

## Keyboard Process

Keyboard process uses RX shared memory map to deliver user’s input to RTX. While the process is getting initialized as a child process of the RTX, it will use the arguments passed on from the parent to establish connection to RX shared memory map. This done by making its local pointer to point to the RX shared memory map created by the RTX.

The keyboard process scans for input from the user while it is running and stores it into RX shared memory. Once there is no more input from user, it will set a flag on the RX shared memory and send a Unix signal to RTX. The process runs until RTX sends signal to terminate. The implementation of the keyboard process is given in the example design document for this course.

//this function is invoked before the keyboard process exits (system shutdown) to deallocate the shared memory

void die(int signal) {

exit(0);

}

int main(int argc, char \* argv[] ) {

int pid, fid; // parent process id and file id of RX shared map memory

ccadr\_t mmap\_ptr; //pointer to the shared map memory

in\_buffer \*in\_mem\_ptr; // C standard pointer to the shared map memory

int loop\_index;

//Parent signals this process to terminate

Sigset(SIGINT, die);

//Retrieve parent id and file id from the argument of this process

sscanf(argv[0], “%d”, &pid);

sscanf(argv[1], “%d”, &fid);

//Establish connection to RX shared memory map

mmap\_ptr = map((ccadr\_t) 0, buffer\_size, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fid, (off\_t) 0);

if (mmap\_ptr == NULL) {

//Failed to allocate memory map

die(0);

}

in\_mem\_ptr = (input\_buffer \*) map\_ptr; //The pointer to a shared memory

//Now we start to catch input from keyboard

input\_buffer\_ptr->flag = 0; //setting “not-ready” flag

intput\_buffer\_ptr->input\_counter = 0;

loop\_index =0;

do {

kbd\_input = getchar(); //Get user input

if (kbd\_input is not a new line) {

if (loop\_index < MAXCHAR\_IN\_ENVENVELOPE -1) {

//We store the character inside the input\_buffer

in\_buf\_ptr->in\_data[loop\_index] = kbd\_input;

in\_buf\_ptr->input\_counter++;

loop\_index++;

} else {

kill(kbd\_ihander\_process\_id, SIGURS1); //send a Unix signal to keyboard interrupt

handler process

in\_buf\_ptr->flag = 1; // This sets the flag to be ready, which is represented by integer 1

loop\_index = 0; set loop counter to zero so we can restart

intput\_buffer\_ptr->input\_counter = 0;

while (in\_buf\_ptr->flag == 1) {

usleep(100000) //Sleep while kbd handler is processed and waits for other input

}

}while(1) //Execute this process until the process is signalled to terminate by RTX

}

}

}

## CRT Process

The CRT process behaves as a counterpart of keyboard process, but it shares similar internal structure such as data field. It is responsible for outputting any message from RTX to the user, through the screen display. The CRT process is also a child of RTX process. Once it’s created it uses the argument passed from its parent process to make its local pointer to point to TX shared memory map.

CRT continuously checks for the flag set by RTX in TX shared memory map to output messages. Once CRT notices that the flag has been set, it will take the message stored in the shared memory map. Once it has finished outputting the entire message, it will send a signal back to RTX to notify the message is now displayed on the screen. The process runs until RTX signals to terminate.The CRT process prints the stored output from TX shared memory map on the screen – its implementation is shown below.

void die(int signal)

{

exit(0);

}

int main(int argc, char \* argv[] )

{

int pid, fid; // parent process id and file id of TX shared map memory

ccadr\_t mmap\_ptr; //pointer to the shared map memory

out\_buffer \*out\_mem\_ptr; // C standard pointer to the shared map memory

int loop\_index;

//Parent signals this process to terminate

Sigset(SIGINT, die);

//Retrieve parent id and file id from the argument of this process

sscanf(argv[0], “%d”, &pid);

sscanf(argv[1], “%d”, &fid);

//Establish connection to RX shared memory map

mmap\_ptr = map((ccadr\_t) 0, buffer\_size, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fid, (off\_t) 0);

if (mmap\_ptr == NULL) {

//Failed to allocate memory map

die(0);

}

out\_buf\_ptr = (out\_buffer \*) mmap\_ptr; // Shared memory pointer is now created

loop\_index = 0;

out\_buf\_ptr->output\_count=0;

out\_buf\_ptr->flag =0;

//Now we print the output stored in the memory mapped file

do {

while (output\_buf\_ptr->output\_count==0) {

usleep(100000) //Nothing to display, wait for the output to arrive in the TX memory mapped file

}

while (output\_buf\_ptr->output\_count > 0) {

loop\_index =0; //set loop index to zero to start printing from the beginning of the array

output = output\_buf\_ptr->output\_data[loop\_index];

if (output != empty character) {

printf(“%c”,output); //print the output retrieved from tx memory mapped file

loop\_index++;

out\_buf\_ptr->output\_count--; //Decrement the output count since it has been

displayed

}}

kill(CRT\_ihandler\_process, SIGUSR2); Signal CRT interrupt handler that all output has been

displayed and it is ready to continue.

}while(1) // run until this process is killed by RTX

}

Null process

void main()

{

while(1) {

release\_processor();

}

} // call release\_processor() API to make currently executing process to yield

## Wall Clock Process

The wall clock process updates the clock on the CRT display every second. Time is kept as a local variable called wall\_clock within the process. The wall\_clock’s value is updated by the timing service iprocess, which sends a message with status “INCREMENT\_CLOCK” to the wall clock process everytime it is invoked (see section 4.6). Whether the wall clock should be displayed is controlled by a local variable call mode which can be ON or OFF. When mode is ON, send\_console\_chars will be called to output the clock onto the monitor. The value of mode can be changed by the CCI which sends messages with statuses TURN\_ON\_CLOCK or STOP\_CLOCK.

Void wall\_clock() {

int wall\_clock = KERNEL\_TIME; //when process is run for the first time

int mode = "off";

do {

msg = receive\_message();

if (msg != null) {

if (msg has "INCREMENT\_CLOCK" type) {

if (wall\_clock != 999999) {

increment wall\_clock by 1 second;

} else {

wall\_clock = 000000;}

release\_msg\_env(msg);

} else if (msg has "CHANGE\_CLOCK" type) {

wall\_clock = msg>msg\_text;

release\_msg\_env(msg);

} else if (msg has "STOP\_CLOCK" type) {

mode = "off";

release\_msg\_env(msg); }

else if (msg has "TURN\_ON\_CLOCK" type) {

mode = "on";

}}

if (mode == "on") {

msg->msg\_text = wall\_clock;

send\_console\_char(msg); }

release\_processor();

} while (1);

}

## Command Console Interface Process

The main functionalities of the CCI are described in the MTE 241 Project Description. The user commands arrive at the CCI in the form of messages, and are either dequeued from PCB’s message queue, or directly called via get\_console\_chars. The switch-case statement checks the message text and matches it with one of the cases. Generally, CCI handles each case by calling the appropriate user primitives, and then calls send\_console\_chars to output the results of user primitives. However, for wall clock commands (c, cd, ct), a decision was made to call send\_console\_chars inside the wall clock process. This is to prevent messages containing time getting confused with messages containing user commands inside the message queue of CCI’s PCB. The while (1) { } loop wraps around the entire process to allow CCI to continuously fetch the latest user inputs.

command\_console\_interpreter() {

while (1) {

if( head envelope of msg\_queue has display\_ack type ) {

release that envelope back to free\_env\_Q;}

initialize variables: input\_msg, error;

if ( the process's msg queue is not empty ) {

input\_msg = dequeue msg from msg queue;}

else {

input\_msg = request\_msg\_env();

set message envelope's source\_id to this process;

do) {

error = get\_console\_chars(input\_msg);

} while (error == GET\_CONSOLE\_CHAR\_ERROR)

}

if (input\_msg->msg\_text not empty) {

else

switch(input\_msg->msg\_text;) {

case "s": {

input\_msg->sender\_id = this process's id;

clear all other fields in inputMsg;

send\_message(user\_process\_A\_pid, input\_msg);

} break;

case "ps": {

line is a pointers to envelopes

request\_process\_status(input\_msg);

line = request\_msg\_env();

line->source\_pid = this process's pid

line->message\_text = input\_msg->msg\_text;

send\_console\_chars(line);

release\_msg\_env(input\_msg);

} break;

case "cd": {

input\_msg->msg\_type = "TURN\_ON\_CLOCK";

send\_message(id of wall clock, input\_msg);

} break;

case "ct": {

input\_msg->msg\_type = "STOP\_CLOCK";

send\_message(id of wall clock, input\_msg);

} break;

case "b": {

do {

error = get\_trace\_buffer(input\_msg);

//the get\_buffer\_content method inside get\_trace\_buffer pre-arranges the buffer contents into an output-friendly format;

} while (error == ERROR\_INVALID\_MID);

do {

error = send\_console\_chars(input\_msg);

} while (error == ERROR\_INVALID\_MID);

} break;

case "t": {

terminate();

} break;

default:{

if (input\_msg->msg\_text's first character is "c" && the format of the next 9 chars is in the format of " hh:mm:ss") {

input\_msg->msg\_type = "CHANGE\_CLOCK";

input\_msg->msg\_text = the 6 digit number representing the time to be changed to

send\_message(id of wall clock, input\_msg);}

else if (input\_msg->msg\_text's first character is "n" and the next 4 chars is in the format of " n nn" or " n n") {

separate the priority and process id from the argument, store them into variables tar\_priority and tar\_pid;

if ("tar\_pid is the null process") {

input\_msg->msg\_text = "cannot change priority of null process";}

if (tar\_priority > 3 || tar\_pid > NUM\_OF\_PROCESSES || tar\_priority < 0) {

input\_msg->msg\_text = "invalid priority level or process id";}

outcome = change\_priority(tar\_priority, tar\_pid);

if (outcome is successful){

input\_msg->msg\_text = "priority change success";}

else {

input\_msg->msg\_text = "priority change failed";}

do {

error = send\_console\_chars(input\_msg);

} while (error == ERROR\_INVALID\_MID); }

else{

input\_msg->msg\_text = "command not recognized";

do {

error = send\_console\_chars(input\_msg);

} while (error == ERROR\_INVALID\_MID);

}}}}

else {//input\_msg is blank, nothing happens

release\_msg\_env(input\_msg);

}}}

## Interrupt Handling

The implementation of the atomic and cleanup functions used below is given in the example design document on UW ACE. This is section describes the implementation for the keyboard, crt and tick handler iprocesses. An exception handler calls the appropriate process.

//routine to call before the keyboard or crt handler processes exiting

//this routine is called when the SIGINT signal is received

void die(int signal){

cleanup();

printf("\n\n Signal Received. Leaving demo..\n");

exit(0);

}

void kb\_handler(){

MsgEnv \* msg = receive\_message(); // ensure iproc does not block

if (msg != NULL)

enqueue the message onto local message queue

MsgEnv \* current\_msg = first message in queue

inputbuf command; //the command that was entered by user at kb

if (message queue is not empty){

if (in\_mem\_p->indata[0] != '\0') //if shared memory is not empty{

strcpy (command.indata, in\_mem\_p->indata); //copy buffer contents

in\_mem\_p->input\_count = 0; //set number of chars to read to 0, so receiver knows micro is done reading

current\_msg->msg\_text = command.indata;

change message type to ‘console\_input’

store current pid in msg, and insert KB\_HANDLER \_ID into msg as source id

return the message to the invoking process

}

}

else

set process state to idle }

void crt\_handler(){

//check for any new messages

MsgEnv \* msg = receive\_message(); //ensure the service does not block i\_process

if (msg != NULL)

enqueue the message onto the local message queue

if (there are still messages on queue) //there are still messages waiting for service{

if (out\_mem\_p.flag == 1){

MsgEnv \* current\_msg = dequeue the current message from local msg queue

strcpy(out\_mem\_p.output\_data, current\_msg->msg\_text);

out\_mem\_p.output\_count = sizeof(current\_msg->msg\_text);

store current pid in msg, and insert CRT\_HANDLER \_ID into msg as source id

change message type to ‘display\_ack’ for returning to CCI

return envelope to CCI as acknowledgement once message has been displayed

}

}

else

set state variable to idle

}

void tick\_handler(){

timeout\_iprocess();

}

//given in code - MODIFY according to our constants and primitives!

//this is launched everytime a signal is intercepted by the RTX

void exception handler(int signum){

//pre-handling code

pcb\* temp;

pcb\* savePCB = RTX.current\_process;

switch(signum){

case SIGINT:

die()

break;

case SIGALARM:

tick\_handler(signum);

break;

case SIGUSR1: // CRT handler

crt\_handler(signum);

break;

case SIGUSR2: // KB handler

kb\_handler(signum)

break;

}

RTX.current\_process = savePCB;

//post handling code

return;

}

void timeout\_iprocess()

{

msgWallClock = request\_msg\_env();

set msg\_type to INCREMENT\_CLOCK;

K\_send\_message(msgWallClock, ID of wallclock process);

if (KERNEL\_TIME is at 999999) {

KERNEL\_TIME = 000000;

} else {

KERNEL\_TIME++; //increment counter everytime process is launched

}

MsgEnv \* msg = k\_receive\_message(); //receive pending messages but do not block

while(msg is not null){

sort\_list(msg);

msg = k\_receive\_message(); //see if any more msgs left

}

if (timeout\_list is not empty){

decrement the tick count of 1st message in the timeout\_list

while(timeout\_list.head->n\_clock\_ticks is zero){

dequeue the first timeout request

save it's current source id

set TSERVICE\_ID as source id

k\_send\_message(envelope, destination\_id); //return envelope

}

}

void sort\_list(Msg \*msg){

int delay\_time = msg->n\_clock\_ticks;

int temp = timeout\_list.head->n\_clock\_ticks;

if delay\_time is greater than temp{

subtract delay\_time and temp

store result in msg->n\_clock\_ticks

enqueue the new msg in the correct pos in list

}

else

enqueue the new msg as head of list

}

}

# 

# Initialization

System initialization includes the following operations:

* OS data and field structure construction
* Process creation
* Selection of the first process to begin

## OS data structure construction

Internal data structures or objects that kernel APIs have to use are created and initialize at the RTX initialization phase. For this project, the data structure that is primarily used is queue. Queues are an essential part of managing and scheduling process. There are two types of queue used: a priority queue and generic queue. The following diagram shows the structure that is selected for implementing priority queue:

H

T

H

T

Priority queue can be implemented as an array of queues. The index of array will indicate the priority of processes. In Queue implementation code, rpq\_enqueue and rpq\_dequeue functions must be implemented to support this data structure. Priority queue will enqueue the process in terms of its priority; the process with higher priority will be executed by the CPU first. This design is to achieve the effectiveness and efficiency in multiprogramming, while ensuring the RTX allows its processes to perform their tasks successfully.

The generic queue is used in various parts of RTX. They can be found in message envelope queue, timeout request queue, and free envelope queue. The enqueue and dequeue function of each queue should be able to work with different types of data structures defined in RTX. Thus, different types of enqueue and dequeue functions are expected, but the design concept remain the same. The generic queue has a header pointer and tail pointer like a priority queue but, it will enlist the element in first in first out order.

All these objects must be initialized for RTX to support is fundamental functionalities. Following pseudo code initializes the data field of RTX:

int initialize\_data() {

pcb\_pointer\_tracker = (PCB[] \*) malloc(NUM\_OF\_PROCESS\*sizeof(PCB\*));

blocked\_on\_resource\_Q = (Queue \*) malloc(sizeof(Queue)); //Allocate memory for

free\_env\_Q= (Queue \*) malloc(sizeof(Queue)); //Allocate memory for free\_env\_q

rpq = (rpq\_process \*) malloc(sizeof(rpq\_process)); //Allocate memory for rpq

}

## Initialization Table

During initialization, initialization table must be created and populated before processes are created. Initialization table stores basic information about each process that is required by the RTX to create PCBs. Only a fixed amount of initialization table gets created since RTX runs only a fixed number of processes for this project. This constraint is provided in project description. The pseudo code for this procedure is as follows:

int initialize\_IT() {

initialization\_table init\_table[NUM\_OF\_PROCESS]; //Declare an array of initialization\_table

/\*\*User processes\*\*/

set init\_table[0]->process\_id as PROC\_A\_ID

set init\_table[0]->process\_priority as PROC\_A\_PRIORITY

set init\_table[0]->stack\_size as STACK\_SIZE

set init\_table[0]->initial\_pc as (void\*)process\_A

set init\_table[1]->process\_id as PROC\_B\_ID

set init\_table[1]->process\_priority as PROC\_B\_PRIORITY

set init\_table[1]->stack\_size as STACK\_SIZE

set init\_table[1]->initial\_pc as (void\*)process\_B

set init\_table[2]->process\_id as PROC\_C\_ID

set init\_table[2]->priority as PROC\_C\_PRIORITY

set init\_table[2]->stack\_size as STACK\_SIZE

set init\_table[2]->initial\_pc as (void\*)process\_C

set init\_table[3]->process\_id as CCI\_PROC

set init\_table[3]->priority as CCI\_PROC\_PRIORITY

set init\_table[3]->stack\_size as STACK\_SIZE

set init\_table[3]->initial\_pc as (void\*)process\_CCI

set init\_table[4]->process\_id as PROC\_NULL\_ID

set init\_table[4]->process\_priority as PROC\_NULL\_PRIORITY

et init\_table[4]->stack\_size as STACK\_SIZE

set init\_table[4]->initial\_pc as (void\*)process\_Null

set init\_table[4]->process\_id as PROC\_CLK

set init\_table[4]->process\_priority as PROC\_NULL\_PRIORITY

set init\_table[4]->stack\_size as STACK\_SIZE

set init\_table[4]->initial\_pc as (void\*)process\_Null

/\*\*\*\*/

/\*\* Iprocesses \*\*/

set init\_table[5]->process\_id as PROC\_TIMER

set init\_table[5]->priority as PROC\_C\_PRIORITY

set init\_table[5]->stack\_size as STACK\_SIZE

set init\_table[5]->initial\_pc as (void\*)process\_timer

set init\_table[6]->process\_id as KBD\_IPROC

set init\_table[6]->priority as KBD\_IPROC\_PRIORITY

set init\_table[6]->stack\_size as STACK\_SIZE

set init\_table[6]->initial\_pc as (void \*)kbd\_iprocess

set init\_table[7]->process\_id as CRT\_IPROC

set init\_table[7]->priority as CRT\_IPROC\_PRIORITY

set init\_table[7]->stack\_size as stack\_size

set init\_table[7]->initial\_pc as (void \*)crt\_iprocess

/\*\*\*\*/

}

## Message Envelopes

Message envelopes are essential part of interprocess communication. Processes store message into the message envelopes and then send to other processes to communicate with them. Message envelopes also store valuable information such as clockticks and message type, which are used by certain processes to perform their tasks.

RTX creates a fixed number of envelopes at the initialization process in order to avoid run-time over-heads. For this project, it is determined that 120 message envelopes are required to be instantiated to achieve this effect. These envelopes are then enqueued in free envelope queue until they are required by other processes. Pseudo code for this section is as follows:

void create\_env() {

msg\_env \*temp\_env = NULL;

int I =0;

for (i=0; I <NUM\_OF\_MBLOCKS\_FOR\_ENVS; i++) {

Assign temp\_env with a pointer returned by create\_msg\_env

If (temp\_env is not null) {

Enqueue temp\_env to free\_env\_Q

} else {

Break out of the loop

}

}

}

## Process creation

User processes and iprocesses are created by creating PCBs that represent those processes using initialization table (hence the initialization-table driven approach to create processes). PCBs store the context of those processes in a data structure called jmp\_buf. PCBs also store stack pointers. Stack pointers are retrieved by allocating a pointer to a memory segment, then offsetting. This is to ensure that stack grows downward, rather than growing upward. Once the PCB for user process has been allocated and its data field are initialized, they are enqueued in ready process queue. It is important to

It is important to note that iprocesses are not stored in ready process queue. Thus, a slight modification is required in creating PCBs of iprocessses.

Keyboard and CRT processes are also created at this stage. They are children of RTX. RX and TX memory map files are created before these children processes are created. Then the file ids of these memory map files are passed onto those children processes so they can also share those memory map files with RTX.

Most of code is given in the example design document in section 13.1. Pseudo code for this section is as follows:

int initialize\_process() {

int i = 0;

jmp\_buf kernel\_init\_buf;

int stack\_offset = 8; //Better offset is to be determined

char \*jmpsp;

PCB \*temp\_pcb = NULL;

for (i =0; i < NUM\_OF\_USER\_PROC; i++) {

temp\_pcb = (PCB \*) malloc (sizeof (PCB)); // allocate memory for PCB

If (temp\_pcb is not null) {

Assign temp\_pcb ->process\_id as init\_table[i]->process\_id

Assign temp\_pcb->process\_priority as init\_table[i]->process\_priority

Assign temp\_pcb->process\_state as STATE\_READY

Assign temp\_pcb->program\_atomic\_counter as 0

Assign temp\_pcb->initial\_pc as init\_table[i]->initial\_pc

Assign temp\_pcb->msg\_head as NULL

Assign temp\_pcb->msg\_tail as NULL

Assign temp\_pcb->process\_stack as ((char\*)malloc(init\_table[i].stack\_size))+size-stack\_offset;

pcb\_pointer\_tracker = temp\_pcb; //acts like a process list

} else {

return ERROR\_FAIL\_TO\_MALLOC; }

rpq\_enque(temp\_pcb);

current\_process =temp\_pcb;

if (setjmp (kernel\_buf) == 0) {

jmpsp = temp\_pcb->process\_stack;

\_\_\_asm\_\_ (“mov1 %0, %%esp”: “=m” (jsmp));

\_set\_sp(jmpsp);

if (setjump (temp\_pcb->context) ==0){

longjmp (kernel\_buf, 1);

} else {

current\_proess->state = EXECUTING;

//process starts here}

for (i = NUM\_OF\_USER\_PROC; i < NUM\_OF\_IPROC; i++) {

temp\_pcb = (PCB \*) malloc (sizeof (PCB)); // allocate memory for PCB

if (temp\_pcb != NULL) {

Assign temp\_pcb ->process\_id as init\_table[i]->process\_id

Assign temp\_pcb->process\_priority as init\_table[i]->process\_priority

Assign temp\_pcb->process\_state as STATE\_READY

Assign temp\_pcb->program\_atomic\_counter as 0

Assign temp\_pcb->initial\_pc as init\_table[i]->initial\_pc

Assign temp\_pcb->msg\_head as NULL

Assign temp\_pcb->msg\_tail as NULL

A

Assign temp\_pcb->process\_stack as ((char\*)malloc(init\_table[i].stack\_size))+size-stack\_offset;

pcb\_pointer\_tracker = temp\_pcb; //acts like a process list

} else {

return ERROR\_FAIL\_TO\_MALLOC; }}

return 0;

}

Also the keyboard and CRT processes must be forked during initialization. The following pseudo code provides the details:

void initialize\_kbd\_process() {

int fid = open(“in\_buf”, O\_RDWR | O\_CREAT | O\_EXCL, (mode\_t) 0755); //Create input

buffer file

ftruncate(fid, BUFFER\_SIZE); //Making the size of the file the same as the buffer

int pid = getpid(); // parent id to pass on to keyboard process

char [20] arg\_for\_child1;

char [20] arg\_for\_child2; // Two char array to store pid and fid

spritnf(arg\_for\_child1, %d, pid);

sprintf(arg\_for\_child2, %d, fid); //Convert the id into character array

int current\_id = fork() //Create kbd child process

if (current\_id ==0) { //Check if this is child process

excl(“./keyboard”, “keyboard”, arg\_for\_child1, arg\_for\_child2); // execute keyboard

process and pass in the pid and fid. };

//Parent process continues to create input memory map

caddrt \*mmap\_ptr; Pointer to shared memory map

mmap\_ptr = mmap((ccard\_t) 0, BUFFER\_SIZE , PROT\_READ | PROT\_WRITIE, MAP\_SHARED, fid, (off\_t) 0); //Create shared map region between RTX and keyboard

in\_mem \_ptr = (in\_buffer \*) mmap\_ptr; //Standard C RX memory map pointer;

if (input\_mem\_ptr == null) {

//Failed at allocating shared map pointer

exit(0);

}

}

void initialize\_crt\_process() {

int fid = open(“out\_buf”, O\_RDWR | O\_CREAT | O\_EXCL, (mode\_t) 0755); //Create out

buffer file

ftruncate(fid, BUFFER\_SIZE); //Making the size of the file the same as the buffer

int pid = getpid(); // parent id to pass on to keyboard process

char [20] arg\_for\_child1;

char [20] arg\_for\_child2; // Two char array to store pid and fid

spritnf(arg\_for\_child1, %d, pid);

sprintf(arg\_for\_child2, %d, fid); //Convert the id into character array

int current\_id = fork() //Create kbd child process

if (current\_id ==0) { //Check if this is child process

excl(“./CRT”, “CRT”, arg\_for\_child1, arg\_for\_child2); // execute keyboard

process and pass in the pid and fid. };

//Parent process continues to create output memory map

caddrt \*mmap\_ptr; Pointer to shared memory map

mmap\_ptr = mmap((ccard\_t) 0, BUFFER\_SIZE, PROT\_READ | PROT\_WRITIE, MAP\_SHARED, fid, (off\_t) 0); //Create shared map region between RTX and keyboard

out\_mem \_ptr = (out\_buffer \*) mmap\_ptr; //Standard C TX memory map pointer

if (out\_mem\_ptr == null) {

//Failed at allocating shared map pointer

exit(0);

}

}

# Process selection

For the purposes of the RTX, the ‘scheduler’ is an entity that, when invoked, selects the highest priority process and then invokes the dispatcher to handle the process switch. The following priority levels are defined in section 2.1.

#define IPROCESS 0 //used for iprocesses

#define HIGH 1

#define MEDIUM 2

#define LOW 3

All of the processes running under the RTX are prioritized based on these priority levels. At initialization all processes are enqueued onto the ready queue with the highest priority level process at the head of the queue.

PROC\_A\_PRIORITY = MEDIUM;

PROC\_B\_PRIORITY = MEDIUM;

PROC\_C\_PRIORITY = MEDIUM;

PROC\_CCI\_PRIORITY = HIGH;

PROC\_CLK\_PRIORITY = HIGH;

IPROC\_KBD\_PRIORITY = IPROCESS;

IPROC\_CRT\_PRIORITY = IPROCESS;

IPROC\_TIMER\_PRIORITY = IPROCESS;

PROC\_NULL\_PRIORITY = LOW;

# 

# Proccess\_switch

When an interrupt handling occurs or a user request is processed by a kernel primitive, a process switch will be requested which blocks the currently executing process in order to release CPU for the new process to run. The blocked process then restarts and returns to its next line of code before the call to process switch.

Process switch calls context switch which performs the saving and restoring of the process context.

process\_switch( )

{

//suspend 'current' process to release CPU

pcb\_ptr \*next\_pcb; //hold pointer to the next process pcb

next\_pcb = rpq\_dequeue( ); //pointer points to the next highest priority ready process

context\_switch( current, next ); //switch the context of 'current' process to 'next' process

// set next->status to 'executing'

current\_process = next\_PCB // set 'current\_process' to point to next's PCB

return;

}

context\_switch (jmp\_buf \*previous, jmp\_buf \*next)

{

return\_code = setjmp(\*previous); // obtain return\_code, save previous process context

if (return\_code == 0) // if setjmp is successful

{

longjmp(\*next,1); // restore next process context

}

return;

}

# 

# Plan for Implementation and testing

## RTX implementation plan

The work division amongst the group is shown in the following table, along with the expected deadlines. Note that elements listed after the ‘modificatoin’ stage are part of the full implementation.

|  |  |  |
| --- | --- | --- |
| **Group member** | **Responsibilities** | **Due Date** |
| Bongkyun Park | Create Initialization table, data structures (queues, linked lists), RTX fields | Oct 29 |
| Create and enqueue message envelopes, initialize queues, create PCBs | Oct 31 |
| Create processes, create shared memory, fork KBD and CRT processes | Nov 2 |
| PI Test plan 1 | Nov 4 |
| Modifications | **Nov 6** |
| Jawad Ateeq | Exception handler | Oct 23 |
| KB process | Oct 26 |
| KB handler | Oct 29 |
| CRT handler | Nov 1 |
| PI Test plan 2 | Nov 3 |
| Modifications | **Nov 6** |
| Release\_msg\_env(), Get\_trace\_buffers(), | Nov 11 |
| Zhuojun Li | Receive\_message(), Send\_message() | Oct 31 |
| Send\_console\_chars(), Get\_console\_chars(), | Oct 29 |
| PI Test plan 3 | Nov 2 |
| Modifications | Nov 4 |
| Terminate(), Change\_priority() | Nov 7 |
| Request\_msg\_env(), Request\_delay() | Nov 11 |
| Yifei Cheng | CRT process, make changes for proc P | Oct 29 |
| Process\_ Switch() | Nov 2 |
| PI Test plan 4 | Nov 4 |
| Modifications | **Nov 6** |
| Request\_process\_status(), Release\_processor() | Nov 11 |

Table : Partial RTX implementation work distribution

## RTX full implementation plan

The following table describes the work division within the group for the full implementation as well as the expected deadlines.

|  |  |  |
| --- | --- | --- |
| **Group member** | **Responsibilities** | **Due Date** |
| Bongkyun Park | Wall Clock | Nov 14 |
| CCI (c set time, cd and ct) | Nov 18 |
| FI Test plan 1 | Nov 20 |
| Modifications | **Nov 22** |
| Jawad Ateeq | Timing services | Nov 16 |
| FI Test plan 2 | Nov 19 |
| Modifications | **Nov 21** |
| Update Final design report | Nov 25 |
| Zhuojun Li | CCI (s, ps, b) | Nov 14 |
| CCI (t and n) | Nov 17 |
| FI Test plan 3 | Nov 19 |
| Modifications | **Nov 21** |
| Yifei Cheng | Null process | Nov 12 |
| Process A, B and C | Nov 15 |
| FI Test plan 4 | Nov 17 |
| Modifications | **Nov 19** |

Table : Full RTX implementation plan

## Test plan

The test plan has been crafted as the following:

Test scripts should be written and test plans should be constantly updated for all parts of the RTOS as the code is being written.

A test script should be written for each major and sub-major component of the RTOS, including: initialization table, queues, scheduler, IPC primitives, CRT, keyboard, interrupt handling, trace buffer, and process termination.

Once each individual unit of the RTX passes all tests, a master test plan can be written which will largely simulate the sequence of tasks that must be performed for the final demonstration.

The code is to be developed and tested in portions – bulk development is to avoided to reduce debugging time.

GDB will be used as the debugging tool.

**Partial Implementation test plan**

Each of the numbers below refer to a specific group member (please see Table 1 and Table 2)

**PI Test plan 1:**

* To test the priority queues and other data structures, create a process that uses the structure interfaces and outputs results. These results should indicate, for example, that data was properly enqueued/dequeued from the queus.
* The functionality of message envelopes and PCBs can be verified using by filling in data and then observing the final data structure with its values.

**PI Test plan 2:**

The P process can be used to test whether keyboard input is received by the RTX or not. The process would need to be modified to ouput data to standard output since the CRT infrastructure would not be in place. Once the CRT has been coded, the original P process can be used.

**PI Test plan 3:**

The KB and CRT processes would invoke the coded primitives, so they can be used to test the functionality of all four primitives.

**PI Test plan 4:**

* The CRT testing is included in test plan 2 using the P process.
* Process switching can be verified by creating a dummy process Q, and switching between P and Q (P can be modified by adding an invocation to release\_processor()). Each process will have its own priority to test scheduling.

**Final Implementation test plan**

**FI Test plan 1:**

Testing the wall clock does not need any additional test processes. Once the relevant CCI commands are in place they can be used to verify its functionality.

**FI Test plan 2:**

The timing service can be tested by creating a test process that requests a timeout. The trace buffer can be analyzed to figure out if the correct messages are being sent to and forth.

**FI Test plan 3:**

The CCI can be tested without any additional test processes since all the relevant processes should be in place by now.

**FI Test plan 4:**

The processes can be tested by invoking process A from the CCI, and ensuring that processes B and C are invoked as defined in the project description.

# Reference

[1] MTE Project Description [Documentation], (2011), *Deliverables*, retrieved Oct 2011 from

<https://uwangel.uwaterloo.ca/uwangel/section/default.asp?id=UW%2DMCL%2DC%2D110819%2D134422>