



---

BUILDING A TALENT TRUST

# The Door Operating System

---

**ECE354**  
Design description

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>  | <b>4</b>  |
| <b>2</b> | <b>Global Information and Data Structures</b>                                  | <b>5</b>  |
| 2.1      | Data Structures . . . . .  | 5         |
| 2.1.1    | Process Control Block . . . . .  | 5         |
| 2.2      | Message envelope . . . . .   | 5         |
| 2.3      | Queues . . . . .   | 6         |
| 2.3.1    | Ready Process Queue . . . . .  | 6         |
| 2.3.2    | Memory Waiting Queue . . . . .   | 6         |
| 2.3.3    | Message waiting Queue . . . . .  | 6         |
| 2.4      | Global Variables & Constants . . . . .   | 6         |
| 2.4.1    | Global Variables . . . . .   | 6         |
| 2.4.2    | Constants . . . . .  | 6         |
| 2.5      | Memory . . . . .   | 7         |
| 2.5.1    | Memory Map . . . . .   | 7         |
| 2.5.2    | Memory Management . . . . .  | 7         |
| 2.6      | OS State Description . . . . .   | 7         |
| <b>3</b> | <b>Primitives</b>  | <b>9</b>  |
| 3.1      | int send_message (int process_ID, void * MessageEnvelope ) . . . . .           | 9         |
| 3.2      | char * receive_message () . . . . .  | 10        |
| 3.3      | char * request_memory_block () . . . . .                                       | 10        |
| 3.4      | int release_memory_block (char* MemoryBlock) . . . . .                         | 10        |
| 3.5      | int release_processor () . . . . .   | 11        |
| 3.6      | int delayed_send (int process_ID, void * MessageEnvelope, int delay) . . . . . | 11        |
| 3.7      | int set_process_priority (int process_ID, int priority) . . . . .              | 12        |
| 3.8      | int get_process_priority (int process_ID) . . . . .                            | 13        |
| 3.9      | int process_switch() . . . . .   | 13        |
| 3.10     | int context_switch(next_proc) . . . . .  | 13        |
| <b>4</b> | <b>Interrupts</b>  | <b>15</b> |
| 4.1      | Software Interrupts . . . . .  | 15        |
| 4.1.1    | System call code . . . . .   | 15        |
| 4.2      | Hardware Interrupts . . . . .  | 17        |
| 4.2.1    | Timer i-process . . . . .  | 17        |
| 4.2.2    | UART i-process . . . . .   | 18        |
| <b>5</b> | <b>Processes</b>   | <b>20</b> |
| 5.1      | System processes . . . . .   | 20        |
| 5.1.1    | Null Process . . . . .   | 20        |
| 5.1.2    | Keyboard Command Decoder Process . . . . .                                     | 20        |
| 5.1.3    | CRT Display Process . . . . .  | 21        |

---

|       |  |    |
|-------|--|----|
| 5.2   | User Processes . . . . .               | 22 |
| 5.2.1 | Set Priority Command Process . . . . . | 22 |
| 5.2.2 | Wall Clock Display Process . . . . .   | 22 |
| 5.2.3 | Test Processes . . . . .               | 22 |
| 5.3   | Initialization . . . . .               | 25 |
| 5.4   | Process Initialization Table . . . . . | 25 |
| 5.5   | Implementation/Test Plan . . . . .     | 25 |
| 5.5.1 | Storage of the code . . . . .          | 25 |
| 5.5.2 | Compilation of the code . . . . .      | 25 |
| 5.5.3 | Tools . . . . .                        | 25 |
| 5.5.4 | Testing on the board . . . . .         | 26 |
| 5.5.5 | Using an oscilloscope . . . . .        | 26 |
| 5.6   | Task Division . . . . .                | 27 |

# List of Figures

|     |  |    |
|-----|--|----|
| 2.1 | Memory map for the RTX . . . . .                   | 8  |
| 2.2 | Structure of the memory . . . . .                  | 8  |
| 2.3 | Process States . . . . .                           | 8  |
| 4.1 | System call (soft interrupt) mechanism . . . . .   | 15 |
| 4.2 | Graphical description of timer interrupt . . . . . | 18 |

# List of Tables

|     |   |    |
|-----|---|----|
| 2.1 | Description of the process control block . . . . .        | 5  |
| 2.2 | Message Enveloppe . . . . .                               | 6  |
| 2.3 | Global variables used in the RTX . . . . .                | 7  |
| 2.4 | Constants of the RTX . . . . .                            | 7  |
| 4.1 | Correspondance between traps and primitives . . . . .     | 16 |
| 5.1 | Description of the Process Initialization Table . . . . . | 25 |
| 5.2 | Description of tasks division . . . . .                   | 27 |
| 5.3 | Description of milestone . . . . .                        | 27 |

# Chapter 1

## Introduction

We will be coding a real time, preemptive operating system with five priorities. Various features will be implemented :

- Multiprogramming environment
- Interprocess Communications
- The operating system has to support debugging
- The target hardware will be a MFC5307 board

This design document will demonstrate how this operating system will be implemented.

## Chapter 2

# Global Information and Data Structures

## 2.1 Data Structures

### 2.1.1 Process Control Block

All control block data is described here:

- **ID number** : The id of a process is an unique integer value.
- **process state** : The process state is an enumeration of constants which describes the current state of the process. Please refer to section 2.4.2 for the list of constants.
- **process priority** : The process priority is an integer which describes the priority of the process.
- **process stack pointer** : This is a memory pointer which points to the memory zone of the stack.
- **stack size** : This is an integer value which represents the size of the stack.
- **next pcb** : This is a pointer to the next Process Control Block.

| Name          | Type of datum                        |
|---------------|--------------------------------------|
| ID            | Integer                              |
| state         | Enumeration                          |
| riority       | Enumeration                          |
| stack_pointer | Pointer to a memory word             |
| stack_size    | Integer                              |
| send_message  | Pointer to the send message queue    |
| recv_message  | Pointer to the receive message queue |
| next_pcb      | Pointer to next PCB                  |

Table 2.1: Description of the process control block

## 2.2 Message enveloppe

The table 2.2 page 6 presents the data structure used for the messages envelope.

Actual data to be used by the destination process

| Field           | Type       | Description  |
|-----------------|------------|--|
| next_msg        | struct ptr | Points to the next message   |
| sender_pid      | Integer    | Process ID of the source   |
| destination_pid | Integer    | Process ID of the destination  |
| type            | Integer    | Type of message  |
| delay           | Integer    | Used by the <code>delayed_send</code> primitive. Message is delivered to destination after the delay period expires. |
| data            | Character  | actual data to be used by the destination process  |

Table 2.2: Message Enveloppe

## 2.3 Queues

Here, we are going to implement various queues for the core of the RTX. Queues are very important in the design of the RTOS because this is the data structure most often used when a context switch occurs. For this reason, we want the access time of the queues to be  $O(1)$ .

### 2.3.1 Ready Process Queue

We will be implementing Round Robin preemptive scheduling. The best data structure for this scheduling policy is a FIFO<sup>1</sup> queue. Here, we will use one FIFO queue per priority so that retrieving priority process information runs in  $O(1)$ . Each one of these FIFO queues are linked to one another to permit very fast access to data.

### 2.3.2 Memory Waiting Queue

Processes waiting for memory will require a special queue. This queue will grow up only when the system does not have enough memory. When a process arrives in this queue, it goes to the `MEMORY_WAIT` state.

### 2.3.3 Message waiting Queue

Processes waiting for message will require a special queue. When a process arrive in this queue, it takes the `MESSAGE_WAIT` state.

## 2.4 Global Variables & Constants

### 2.4.1 Global Variables

Here are the global variables used in the system. Please refer to the memory map of the RTX to find their mapping.

### 2.4.2 Constants

The list of constants is descibed in Table 2.4.

---

<sup>1</sup>First In First Out

| Variable Name                 | Description   |
|-------------------------------|---|
| current_process               | Pointer to the running PCB  |
| ticks                         | Integer : number of timer ticks   |
| queues_READY [MAX_PRIO]       | Table of pointers to the queue of PCBs in the <b>ready</b> state. The index of the table is the priority of the processes |
| queue_MEMORY_WAIT [MAX_PRIO]  | Table of pointers to the queue of PCBs in the <b>memory waiting</b> state   |
| queue_MESSAGE_WAIT [MAX_PRIO] | Table of pointers to the queue of PCBs in the <b>message waiting</b> state  |
| queue_INTERRUPTED_ [MAX_PRIO] | Table of pointers to the queue of PCBs in the <b>interrupted</b> state  |

Table 2.3: Global variables used in the RTX

| Constant         | Description                                   |
|------------------|---|
| RUNNING          | The process is running.                       |
| READY            | The process is ready to run.                  |
| INTERRUPTED      | The process was interrupted.                  |
| MESSAGE_WAIT     | The process is blocked for a message.         |
| MEMORY_WAIT      | The process is waiting for a memory block.    |
| HIGHEST_PRIORITY | The highest priority for a process (Level 0). |
| LOWEST_PRIORITY  | The lowest priority for a process (Level 4).  |

Table 2.4: Constants of the RTX

## 2.5 Memory

### 2.5.1 Memory Map

We will be using the VON NEUMANN architecture, which means that we have to keep in mind that the code and the data are loaded in the same memory. At the beginning of the memory is the code of the RTX kernel (dispatcher and so on...). We can divide this code in two sections : the executable code and the data structures (globals variables and various process control blocks). Following the first section is the various process code and data. In this RTX, the local processes stack is going to be used to backup the context of the process during context switch. In this way, we will be able to use hardware features for context switch. Indeed, most of the architectures have built in "return from subroutines" procedures. So we will just have to set the stack register (D7 for the 68k for example) and call the return from subroutine function. Located after the code for data structures is executable code for processes, and next stacks for each process. The remaining memory space will be available for dynamic memory allocation.

### 2.5.2 Memory Management

Our operating system has to provide dynamic memory management. From the RTX description, we know the memory will be aligned with a block size of **128 Bytes**. The free memory will be available with a uni-directional linked list.

## 2.6 OS State Description

The states a process may take are outlined in Figure 2.3.

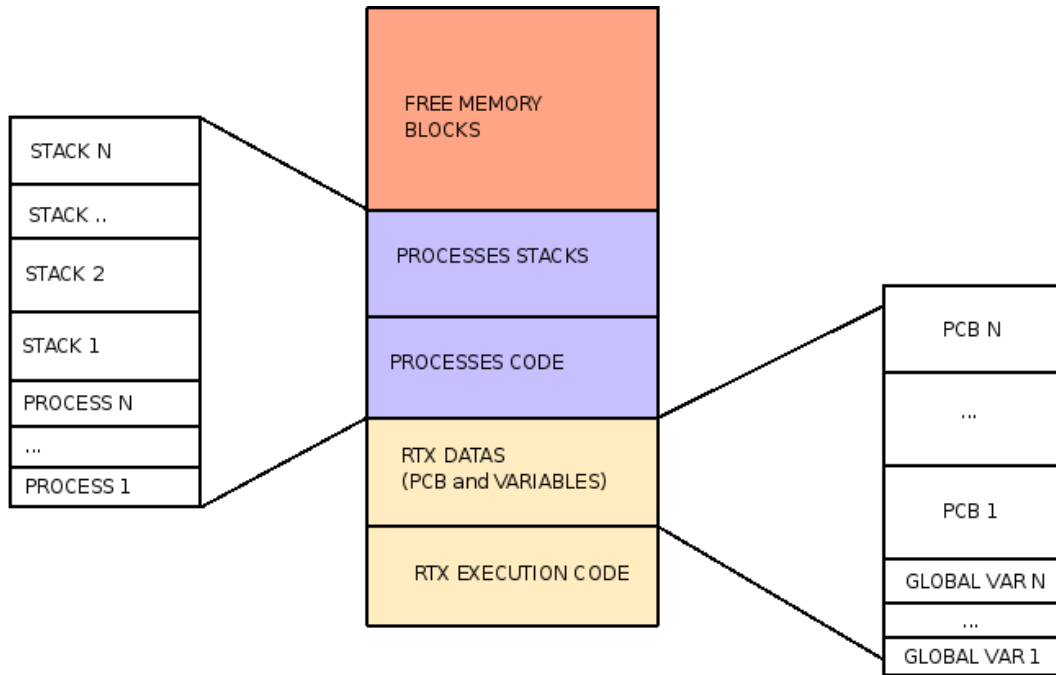


Figure 2.1: Memory map for the RTX

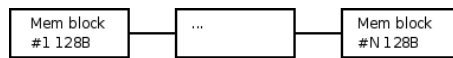


Figure 2.2: Structure of the memory

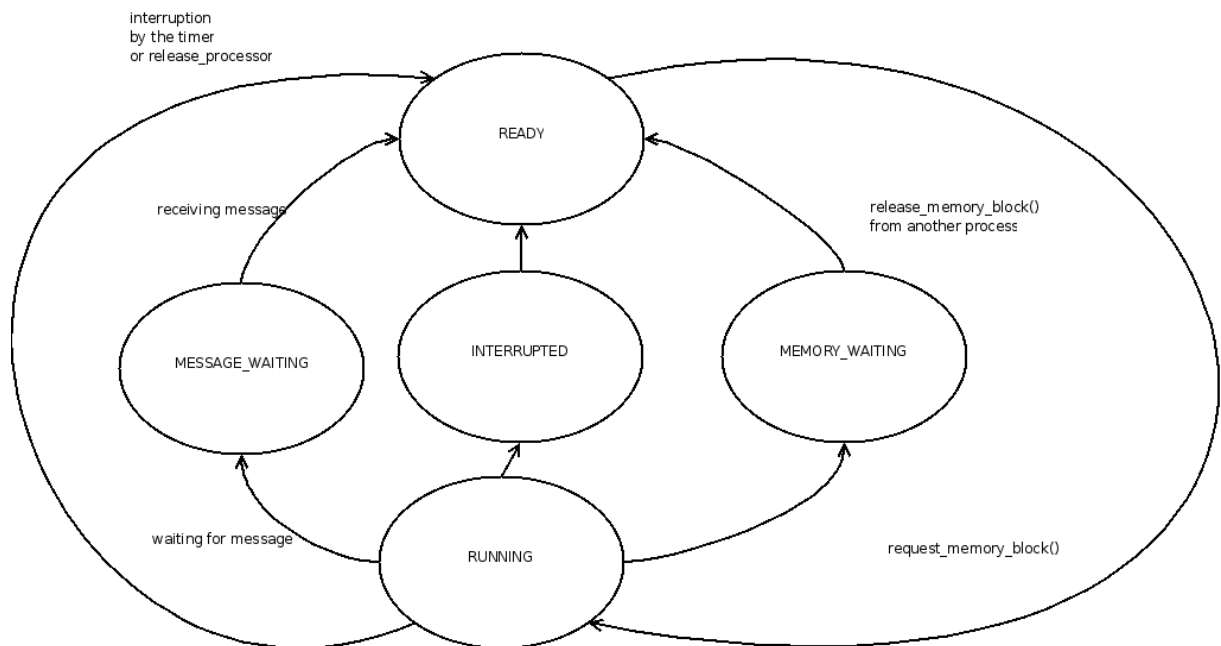


Figure 2.3: Process States

## Chapter 3

# Primitives

### 3.1 `int send_message (int process_ID, void * MessageEnvelope )`

Delivers a message to the target process through a message envelope (a memory block). Changes the state of target process to READY if appropriate. The sending process is preempted if the receiving process was blocked waiting for a message and has higher priority, otherwise the sender continues executing. The header of the message will have the layout given in the course overheads. It also fills in the `sender_process_id` and `destination\_process_id` fields in the message envelope. The `sender_process_id`, `destination_process_id` and `message_type` fields are all of type `int`.

```
1 int send_message(int process_ID, char* messageEnvelope)
2 {
3     sourcePID = currentPCB.ID
4     destinationPID = process_ID
5
6     if (destinationPID doesn't exist) {
7         release_memory_block (messageEnvelope)
8         return INVALID_DESTINATION
9     }
10
11 destinationPCB = get PCB by destinationPID
12 sourcePCB = get PCB by sourcePID
13
14 enqueue messageEnvelope to destinationPCB.messageQueue
15
16 if (destinationPCB.state is BLOCKED_ON_RECEIVE){
17     enqueue destinationPCB to READY queue
18     dequeue destinationPCB from BLOCKED_ON_RECEIVE queue
19     If destinationPCB.priority > sourcePCB.priority {
20         process_switch()
21     }
22 }
23
24 return SUCCESS
25 }
```

## 3.2 char \* receive\_message ()

This is a blocking receive. If there is a message waiting, a pointer to the message envelope will be returned to the caller. If there is no such message, the calling process blocks and another process is selected for execution.

```

1 char* receive_message()
2 {
3
4 while (currentPCB.messageQueue is empty) {
5     currentPCB.state = BLOCKED_ON_RECEIVE
6     enqueue currentPCB to BLOCKED_ON_RECEIVE queue
7     process_switch()
8 }
9
10 messageEnvelope = dequeue currentPCB.messageQueue
11
12 return messageEnvelope
13 }
```

## 3.3 char \* request\_memory\_block ()

The primitive returns a pointer to a memory block to the calling process. If no memory block is available, the calling process is blocked according to its priority and is served on a first-come-first-served basis. If several processes are waiting for memory blocks and a block becomes available, it will be given to the highest priority waiting process.

```

1 char* request_memory_block()
2 {
3     memory_block = dequeue FREE_MEMORY queue
4
5     if (memory_block is null) {
6         set current process state to BLOCKED_ON_MEMORY
7         enqueue currentPCB to BLOCKED_ON_MEMORY queue
8         process_switch()
9     }
10
11     return memory_block
12 }
```

## 3.4 int release\_memory\_block (char\* MemoryBlock)

This primitive returns a memory block to the RTX. If there are processes waiting for a block, the block is given to the highest priority process, which is then unblocked. The caller of this primitive never blocks, but could be preempted. Thus, it may affect the currently executing process.

```

1 int release_memory_block(char* MemoryBlock)
2 {
3     if (MemoryBlock belong to the region of memory allocated for processes) {
4         return INVALID_MEMORY_BLOCK
5     }
6
7     enqueue MemoryBlock to FREE_MEMORY queue
8
9     if (BLOCKED_ON_MEMORY queue is not empty) {
```

```

10     highest_priority_blocked_on_memory_PCB = dequeue BLOCKED_ON_MEMORY           queue
11     highest_priority_blocked_on_memory_PCB.state = READY
12     enqueue highest_priority_blocked_on_memory_PCB to READY queue
13
14     if (highest_priority_blocked_on_memory_PCB.priority > currentPCB.priority){
15         process_switch()
16     }
17 }
18
19 return SUCCESS
20 }

```

### 3.5 int release\_processor ()

Control of the CPU is transferred to the RTX (the calling process voluntarily releases the processor). The invoking process remains ready to execute. However, another process may be selected for execution.

```

1 int release_processor(){
2     currentPCB.state = READY
3     //currentPCB will be enqueued to the end of the READY queue at the appropriate priority as
4     //READY queue is implemented as a FIFO structure. Which means, if there is another
5     //process at //the same priority as the current process in the READY queue, it will be
6     //selected to run. However //if there is no other process in the READY queue at the
7     //same priority than the current process, //process_switch() will select the
8     //current process to be run again
9     enqueue currentPCB to READY queue
10    process_switch()
11 }

```

### 3.6 int delayed\_send (int process\_ID, void \* MessageEnvelope, int delay)

The invoking process does not block. The message (in the memory block pointed to by the second parameter) will be sent to the destination process (process\_ID) after the expiration of the delay (The timeout given in units of msec).

```

1 int delayed_send (int process_ID, char* MessageEnvelope, int delay){
2     if (delay < 0){
3         return INVALID_DELAY
4     }
5
6     if (delay == 0){
7         send_message(process_ID, MessageEnvelope)
8     }
9
10    if (delayed_send_queue is empty){
11        enqueue MessageEnvelope to delayed_send_queue
12        return SUCCESS
13    }
14
15    ptr = delayed_send_queue.head
16    prevPtr = NULL
17

```

```

18 while (ptr != NULL && delay > ptr.delay) {
19     delay = delay - ptr.delay
20     prevPtr = ptr
21     ptr = ptr.nextMessage
22 }
23
24 //account for the case when delay is smaller than the first message envelope's delay
25 if (prevPtr == NULL)
26 {
27     ptr.delay = ptr.delay - delay
28     MessageEnvelope . delay = delay
29     MessageEnvelope . nextMessage = ptr
30     delayed_send_queue . head = MessageEnvelope
31 }
32 //account for the case when delay is larger than the last message envelope's delay
33 else if (ptr == NULL)
34 {
35     MessageEnvelope . delay = delay
36     MessageEnvelope . nextMessage = NULL
37     prevPtr . nextMessage = MessageEnvelope
38 }
39 else
40 {
41     ptr . delay = ptr . delay-delay
42     MessageEnvelope . delay = delay
43     prevPtr . nextMessage = MessageEnvelope
44     MessageEnvelope . nextMessage = ptr
45 }
46
47 return SUCCESS
48 }

```

### 3.7 int set\_process\_priority (int process\_ID, int priority)

This primitive sets the priority of the process with process\_ID to the value given in priority. A process may change its own priority. The priority of the null process may not be changed from level 4 and it is the only process that can be assigned to level 4. The caller of this primitive never blocks, but could be preempted. This preemption may affect the currently executing process.

```

1 int set_process_priority (int process_ID, int priority) {
2
3     if (process_ID == 0 || process_ID doesn't exist){
4         return INVALID_PID
5     }
6
7     if (priority == 4){
8         return INVALID_PRIORITY
9     }
10
11     PCB = get PCB of process_ID
12
13     if (PCB == currentPCB){
14         If (priority > currentPCB . priority ){
15             currentPCB . priority = priority
16         }else{
17             currentPCB . priority = priority
18             process_switch()

```

```

19 }
20 } else if (PCB.state == BLOCKED_ON_MEMORY){
21     PCB.priority = priority
22     dequeue PCB from BLOCKED_ON_MEMORY
23     enqueue PCB to BLOCKED_ON_MEMORY
24 } else if (PCB.state == BLOCKED_ON_RECEIVE){
25     PCB.priority = priority
26 }else {
27     PCB.priority = priority
28     if (priority > currentPCB . priority){
29         process_switch()
30     }
31 }
32
33 return SUCCESS
34
35 }

```

### 3.8 int get\_process\_priority (int process\_ID)

This primitive returns the priority of the process specified by the process\_ID parameter. For an invalid process\_ID, the primitive returns -1.

```

1 int get_process_priority (int process_ID) {
2     PCB = get PCB by process_ID
3
4     if (PCB == NULL){
5         return INVALID_PID
6     }
7
8     return PCB.priority
9 }

```

### 3.9 int process\_switch()

```

1 int process_switch (){
2     highest_priority = getHigestPriority(READY queue)
3
4     if (highest_priority > currentPCB . priority){
5         PCB = dequeue READY queue
6         PCB . state = RUNNING
7         currentPCB . state = READY
8         enqueue currentPCB to READY queue
9         context_switch(PCB)
10    }
11
12    return SUCCESS
13 }

```

### 3.10 int context\_switch(next\_proc)

```
1 int context_switch (next_proc) {  
2   push(data_registers)  
3   push(address_registers)  
4   currentPCB . SP = SP  
5   currentPCB = next_proc  
6   SP = currentPCB . SP  
7   assembly(return from subroutine)  
8 }
```

## Chapter 4

# Interrupts

### 4.1 Software Interrupts

We use software interrupts to hide all the kernel functions from the user by using an interface. If the user wants to run a kernel primitive it will do so by calling the trap function with a number as parameter (e.g. trap number 12). The trap function is a special command commonly found in most standard CPU instruction sets including the CPU of the Coldfire board. Invoking the trap will save the context of current user process onto the CPU stack and registers. Then, a trap handler (similar to an interrupt handler) will take over. This trap handler uses the trapped vector number to call the appropriate kernel primitive (a system call).

Once the kernel finished running the appropriate primitive, it restores the context of the original user process to memory and the process will continue from where it left off. Please see Figure 4.1 for a high level depiction of software interrupts.

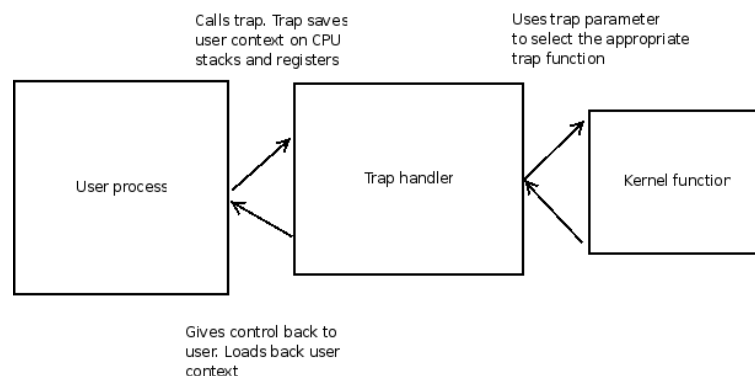


Figure 4.1: System call (soft interrupt) mechanism

#### 4.1.1 System call code

The following assembly code outlines how trap instructions will be executed:

```

1 system_call_XX()
2 {
3   assembly_code(move.B #Opcode, D0) // move the intrustion code into a data register
4   assembly_code(Trap #Kernel_trap_vector)

```

5 }  
}

Upon invoking the trap, process will switch context (see section on context switching) and the trap handler will make the appropriate system call in supervisor mode. On return, the ISR (trap handler) then will return the return value (if any) to a pre-defined memory location on the process PCB. The following is the pseudo code for kernel operations corresponding to the trapped vector number :

```

1 Mask all the interrupts // atomic(on)
2 Switch process context (save registers on CPU stack)
3 Go to trap handler (switch to supervisor mode)
4 Call the corresponding kernel primitive using trap number
5 Load registers (switch back)
6 Unmask interrupts // atomic(off)
7 Process context switches
8 return

```

| Trap number | Primitive  |
|-------------|--|
| 0           | int send_message (int process_ID, void * MessageEnvelope )           |
| 1           | char * receive_message ()  |
| 2           | char * request_memory_block ()                                       |
| 3           | int release_memory_block (char* MemoryBlock)                         |
| 4           | int release_processor ()   |
| 5           | int delayed_send (int process_ID, void * MessageEnvelope, int delay) |
| 6           | int set_process_priority (int process_ID, int priority)              |
| 7           | int get_process_priority (int process_ID)                            |
| 8           | int process_switch()   |
| 9           | int context_switch(next_proc)  |

Table 4.1: Correspondance between traps and primitives

## 4.2 Hardware Interrupts

This section outlines the hardware interrupt design of the RTX. The two devices on the MCF5307 board which may generate hardware interrupts are the timer and the UART. The timer interrupt allows processes to receive timeout notices based on the requested expiry time. On the other hand, the UART interrupt handles the important tasks of input from the keyboard, and output to the CRT.

When a hardware interrupt occurs, the ISR will disable interrupts and save the context of the currently running process. It then invokes the corresponding i-process to service the device. Interrupts are disabled to maintain a simple design and to prevent the scenario where older interrupts must wait for the CPU while a newer interrupt is being serviced.

There are two i-processes to handle the required operations to service the timer and the UART. The i-processes are assigned the highest priority (0) to prevent them from being preempted by a higher priority process. In addition, primitives that are called by the i-processes are written to ensure the i-process cannot be blocked. At the end of an i-process, interrupts are re-enabled and `release_processor()` is called to hand the CPU back to the RTX. However, the subsequent process executed may or may not be the original process that was interrupted. This is because another process may have gained a higher priority while the interrupt was serviced.

### 4.2.1 Timer i-process

The Timer i-process provides a timeout functionality that is utilized by other processes. These processes may request for a timeout notification by sending this i-process a message containing the source process id and the relative timeout period. Once a timeout has expired, the timer i-process will send the original request messages back to the sender to notify the process of the event. The pseudo code of the timer i-process is provided below with the sequence of events outlined in Figure 4.2.

```

1 timer_i_process
2 {
3   env <- receive_message() //receive first request
4   while (env != null)
5   {
6     Add request to timeout_list
7     //timeout_list must be sorted ascending by relative timeout duration
8     env <- receive_message() //receive next request
9   }
10
11   Increment timeout_counter
12
13   if (timeout_list is not empty)
14   {
15     while (timeout_list.head.expiry_time == timeout_counter)
16     {
17       //return envelope to requestor process
18       env <- timeout_list.dequeue()
19       destination_process_id <- env.sender_process_id
20       env.sender_process_id <- timer_i_process.process_id
21       send(destination_process_id, env)
22     }
23   }
24   Set register(s) to re-enable interrupts
25   release_processor()
26 }
```

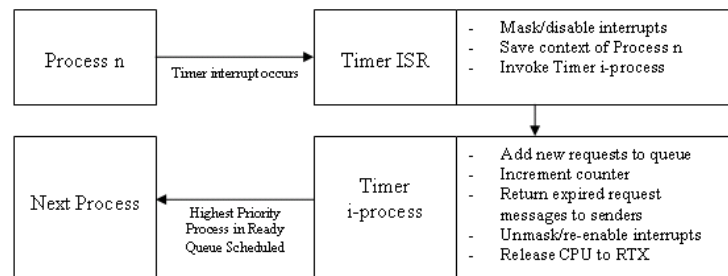


Figure 4.2: Graphical description of timer interrupt

### 4.2.2 UART i-process

The UART i-process is responsible for handling the user I/O. Its first function is receiving characters from the keyboard through the serial interface and passing it to the Keyboard Command Decoder process. Its second function is transmitting characters from the CRT Display Process to the CRT display. In addition, the UART i-process must also handle the hot keys feature modifies various debug settings. Note that the UART i-process is not directly responsible for echoing the keyboard input to the CRT display. This is handled by the Keyboard Command Decoder process, which sends a message to the CRT Display Process before it is received by the UART i-process.

Unlike the timer i-process, the UART i-process has to handle more than one type of request. Therefore, it must be able to distinguish between input and output requests and perform the required task. The pseudo code of the UART i-process is provided below.

```

1  uart_i_process
2  {
3    if (UART data ready register)
4    {
5      input_char = Character read from UART data register
6      if (input_char == Hot Key 1)
7      {
8        Process Hot Key 1
9      }
10     else if (input_char == Hot Key 2)
11     ...
12     else
13     {
14       env <- request_memory_block()
15       If (env != null)
16       {
17         env.sender_process_id = uart_i_process.process_id
18         env.destination_process_id = KCD.process_id
19         env.data = input_char
20         send_message(KCD.process_id, env)
21       }
22     }
23     else
24     {
25       Indicate error, key press is lost
26     }
27   }
28
29   env <- receive_message()
30   While (env != null)
  
```

```
31 {
32   output_char = env.data
33   Set UART registers for output
34   UART output register = output_char
35   release_memory_block(env)
36   env <- receive_message()
37 }
38 Set register(s) to re-enable interrupts
39 release_processor()
40 }
```

# Chapter 5

## Processes

### 5.1 System processes

#### 5.1.1 Null Process

This process runs as the lowest priority process (level 4) in the RTX.

```
1 loop forever
2 end loop
```

#### 5.1.2 Keyboard Command Decoder Process

The Keyboard Command Decoder process is responsible for decoding various user input commands. A keyboard command begins with the

```
1 loop {
2   message = receive_message()
3
4   if (message.type != KEYBOARD_INPUT && message.type == COMMAND_REGISTRATION) {
5     //COMMANDS_REGISTRATION queue contains the process IDs and the associated //
6     //commands
7     enqueue COMMANDS_REGISTRATION queue
8   }
9
10  else if (message.type == KEYBOARD_INPUT){
11    character = message.data
12    send character to CRT
13
14    if (character == %){
15      buffer_flag = 1
16    }
17
18    if (buffer_flag) {
19      if (character == '\n'){
20        command = merge buffer
21        search COMMAND_REGISTRATION queue for the command and send command to the
22        appropriate process using send_message()
23        empty buffer
24      }else{
25        buffer character
26      }
27    }
28  }
29 }
```

```
26 }  
27 }  
28 }
```

### 5.1.3 CRT Display Process

The sole purpose of the CRT Display Process is to pass output characters received from other processes to the UART i-process. When the CRT Display Process receives a CRT display request, it takes each character of the output string and forwards it to the UART i-process in an envelope so that it can be transmitted to the CRT display. The pseudo code of this process is outlined below.

```
1 loop{  
2   message = receive_message()  
3  
4   if (message.type is CRT_DISPLAY_REQUEST) {  
5     for each character in message.data {  
6       messageEnvelope = request_memory_block()  
7       messageEnvelope.data = character  
8       send_message( UART I-process, messageEnvelope )  
9     }  
10  
11    release_memory_block(message)  
12  }  
13  
14  else { //invalid message type  
15    release_memory_block(message)  
16  
17  }  
18 }
```

## 5.2 User Processes

### 5.2.1 Set Priority Command Process

The Set Priority Command Process allows users to modify the priority of another process through the %C command. This command has two parameters: the id of desired process and its new priority. Upon receiving the request, the parameters are verified to ensure the target process exists. Once verified, the target process priority is updated immediately, which may alter its position in any queues its in. Passing any invalid parameters will result in an error message and the request being ignored. The pseudo code of this process is shown below.

```

1 register set_priority_command_process as the handler for 'C' with KCD
2 loop {
3   message <- receive_message()
4   get process_id and new_priority from message . data
5   if (process_id is valid and new_priority is valid){
6     set_process_priority(process_id, new_priority)
7   }
8   else{
9     print "illegal parameters" on console using CRT Display process
10  }
11  release_memory_block(message)
12 }

```

### 5.2.2 Wall Clock Display Process

The Wall Clock Display Process is responsible for handling the clock feature. The clock may be started using the %WShh:mm:ss command which results in the time being displayed on the CRT. To stop the clock, the %WT command is used. Shown below is the pseudo code of this process.

```

1 register wall_clock_display_process as the handler for 'W' with KCD
2 loop {
3   message <- receive_message()
4   if (message.type is KCD_REQUEST){
5     if (message.data = "WS.*"){
6       time <- parse time from message.data
7       send_message (CRT display process, time)
8       delayed_send (wall clock display process, 'refresh clock', 1000)
9     }
10    else if (message.data = "WT.*"){
11      terminate display
12    }
13    } else if (message.type is 'refresh clock') {
14      time <- message.data + 1 second
15      send_message(CRT display,time)
16      delayed_send (wall clock display process, 'refresh clock', 1000)
17    }
18  }
19  release_memory_block(message)
20 }

```

### 5.2.3 Test Processes

This section contains pseudo code of the various test processes which will be run on the RTX. The purpose of this is to verify the functionality and behaviour of the system under various scenarios.

**Process A**

```

1  p <- request_memory_block()
2  register with Command Decoder as handler of \%A commands (use p)
3  loop forever
4    p <- receive_message(NULL)
5    if the message contains the \%A command then
6      release_memory_block(p)
7      exit the loop
8    else
9      release_memory_block(p)
10   endif
11  endloop
12
13  num <- 0
14  loop forever
15    p <- request_memory_block()
16    // set msg_type field of p to "count_report"
17    // set actual message data of p to num
18    send p to process B
19    num <- num + 1
20    release_processor()
21  endloop

```

**Process B1**

```

1  loop forever
2    receive a message
3    send the message to process C1
4  endloop

```

**Process B2**

```

1  loop forever
2    receive a message
3    send the message to process C2
4  endloop

```

**Process C1**

```

1  Perform any needed initialization and create a local message queue
2  loop forever
3    if (local message queue is empty) then
4      p <- receive_message(NULL)
5    else
6      p <- dequeue the first message from the local message queue
7    endif
8
9    if msg_type of p is "count_report" then
10     num <- extract from actual message from p
11     if num is evenly divisible by 20 then
12       q <- request_memory_block()
13       send q to CRT process to display "Process C"

```

```

14     //hibernate for 10 sec
15     send a delayed send (10 sec, msg_type<-wakeup10)    to itself using p
16     loop forever
17     //block and let other processes execute
18     p <- receive a message
19     if (message_type <- wakeup10) then
20         exit this loop
21     else
22         put message on the local message    queue
23     endif
24     endloop
25     endif
26     endif
27     release_memory_block(p)
28     release_processor()
29
30     endloop
31
32 \subsubsection{Process C2}
33 \begin{lstlisting}
34 q <- request_memory_block()
35 Perform any needed initialization and create a local message queue
36 loop forever
37     if (local message queue is empty) then
38         p <- receive_message(NULL)
39     else
40         p <- dequeue the first message from the local message    queue
41     endif
42     free <- true
43     if msg_type of p is "count_report" then
44         num <- extract from actual message from p
45         if num is evenly divisible by 20 then
46             send p to CRT process to display "Process C"
47             free <- false
48             //hibernate for 10 sec
49             send a delayed send (10 sec, msg_type<-wakeup10) to itself using q
50             loop forever
51                 //block and let other processes execute
52                 q <- receive_message(NULL)
53                 if (msg_type of q is wakeup10) then
54                     exit this loop
55                 else
56                     put q on the local message queue
57                 endif
58             endloop
59         endif
60     endif
61     if (free) then
62         release_memory_block(p)
63     endif
64     release_processor()
65 endloop

```

## 5.3 Initialization

Initialization of an OS is heavily dependant on its architecture. We must understand the memory mapping of the coldfire board to link object codes efficiently. The first component to initialize will be interrupts. This will be done in assembly code because we do not want an unspecified state of the OS at boot time. Once the memory regions are defined precisely in the kernel, the core of the kernel will be able to start.

## 5.4 Process Initialization Table

To start processes, the operating system has to know how initialize and how to start them. That is why we have the **process initialisation table**. The structure of this table is shown in Table 5.4.

| Name             | Type of datum  |
|------------------|--|
| process_id       | Integer : unique id of the process                                   |
| process_priority | Enum : priority of the process                                       |
| init_SP          | Pointer to memory : address of the first byte of the stack           |
| init_PC          | Pointer to memory : address of the first byte of the code to execute |

Table 5.1: Description of the Process Initialization Table

## 5.5 Implementation/Test Plan

### 5.5.1 Storage of the code

The code will be stored using a subversion repository on the unix server **eceunix**.

### 5.5.2 Compilation of the code

The code will be cross-compiled on unix. Most of the code will be compiled with the gcc<sup>1</sup>.

### 5.5.3 Tools

All tools for compilation and coding will be open source. The gnu binutils will be compiled for the motorolla architecture to be able to assemble assembly code. All the compilation works will be handled by makefiles to win a lot of time.

### Portability

The os will be coded as portable as possible : all the arch-specific code will be stored in a specific file, to allow portability and to abstract the architecture stuffs.

### Emulator

The emulator was successfully compiled and tested, so it will be used for testing purposes for each module of the os.

<sup>1</sup>gcc : the Gnu C Compiler

#### **5.5.4 Testing on the board**

Some things can not be run on the emulator, like running the timer interrupt at the good speed and so on. When we will want to test precise hardware stuffs, we will test in on the real MFC5307 boards.

#### **5.5.5 Using an oscilloscope**

An oscilloscope may be needed to test some very rapid interrupts or delays. Putting a value in a PIO register is very fast, and we can measure it efficiently with an oscilloscope.

## 5.6 Task Division

We are going to use the google agenda tool to share documents.

| Name                 | Tasks   |
|----------------------|---|
| Johan MATHE          | Setting up the svn repository<br>Implementing data types<br>Coding context switch and timer interrupt handler |
| Pooyan NAJAFI        | Software interrupts<br>Hardware interrupts<br>Queuing   |
| Chung Hong CHEUNG    | System processes<br>Messaging   |
| Gohulan BALACHANDRAN | Memory structure<br>Implementation of primitives  |

Table 5.2: Description of tasks division

| StakeHolder        | Action                             | Due Date |
|--------------------|------------------------------------|----------|
| Gohulan and Johan  | Global Information and DataStructs | June 20  |
| Gohulan and Johan  | Primitives                         | June 25  |
| Pooyan and Patrik  | Timer IRQ                          | July 1   |
| Pooyan and Patrik  | Software Interrupt                 | July 3   |
| Patrik             | Rest of HW interrupts              | July 5   |
| Pooyan and Gohulan | System and user Processes          | July 7   |
| Whole Group        | Compile and testing                | Mid-July |

Table 5.3: Description of milestone