# RTOS Documentation

Kathleen Chung

kklchung@uwaterloo.ca

Connor Cimowsky

ccimowsky@uwaterloo.ca

Christian De Angelis

cdeangel@uwaterloo.ca

Jaclyne Ooi

jpooi@uwaterloo.ca

March 27, 2014

# Contents

# List of Algorithms

# Chapter 1

# Design Description

## 1.1 Operating System Initialization

### 1.1.1 Memory Initialization

After initializing the hardware components of the MCB1700 board, such as timers and UART controllers, it is necessary to dedicate a portion of the onboard SRAM to various components of our operating system. Starting from the end address of our operating system's image, we reserve memory for structures such as process control blocks, process control queues, and the keyboard command registry. Most importantly, we divide a portion of memory into fixed-size blocks and insert them into our global memory heap.

### 1.1.2 Process Initialization

Once our process control structures have been initialized by our memory initialization routine, we iterate through our process initialization table and perform three tasks. First, we populate the process control block (PCB) for each process with information such as process identifier (PID) and priority. Next, we enqueue each PCB in the ready queue (unless it is an i-process, since these processes are invoked using interrupt handlers). Finally, we allocate an exception stack frame for each process and store the resulting stack pointer in the appropriate PCB. At this point, our operating system is ready to begin executing the first process by invoking `release_processor`.

## 1.2 Memory Management

### 1.2.1 Heap Data Structure

Our memory heap structure is implemented using a generic linked list. Each node in the list represents a single memory block that can be requested by invoking the `request_memory_block` primitive and released by invoking the `release_memory_block` primitive. Nodes in the memory heap are spaced apart using a predefined block size.

### 1.2.2 Requesting Memory Blocks

When a process invokes `request_memory_block`, the operating system first checks if any blocks are available in the heap. If the heap is empty, the kernel enqueues the caller in the blocked-on-memory queue and invokes `release_processor` so that preemption may occur. Otherwise, a pointer to the next available memory block is popped from the heap. This pointer is then incremented by a predefined offset to compensate for message envelope headers and then returned to the caller.

**Algorithm 1** Requesting Memory Blocks

```
1: procedure K_REQUEST_MEMORY_BLOCK()
2:     while mem_heap is empty do
3:         K_ENQUEUE_BLOCKED_ON_MEMORY_PROCESS(cur_proc)
4:         K_RELEASE_PROCESSOR()
5:     end while
6:     mem_blk ← POP(mem_heap)
7:     return mem_blk
8: end procedure
```

### 1.2.3 Releasing Memory Blocks

When a process invokes `release_memory_block`, the operating system first ensures that the provided address corresponds to a valid memory block. If valid, the block is then pushed onto the heap. Next, if the blocked-on-memory queue is non-empty, the highest-priority process that is blocked on

memory is moved to the ready queue. Finally, `release_processor` is invoked so that the caller can be preempted if necessary.

**Algorithm 2** Releasing Memory Blocks

```
 1: procedure K_RELEASE_MEMORY_BLOCK(mem_blk)
 2:     if mem_blk is invalid then
 3:         return RTOS_ERR
 4:     end if
 5:     PUSH(mem_blk, mem_heap)
 6:     if blocked_on_memory_queue is not empty then
 7:         blocked_proc ← K_DEQUEUE_BLOCKED_ON_MEMORY_PROCESS()
 8:         blocked_proc.state ← READY
 9:         K_ENQUEUE_READY_PROCESS(blocked_proc)
10:         K_RELEASE_PROCESSOR()
11:     end if
12:     return RTOS_OK
13: end procedure
```

## 1.3 Process Management

### 1.3.1 Process Control Structures

In order to fairly manage resource usage, each process in the operating system is modelled by a process control block. Each PCB contains the stack pointer, PID, priority, state, and message queue of the process it represents. Using PIDs as indices, a global array stores a pointer to each PCB for constant time access by process control primitives.

Our operating system maintains three queues in which PCBs are stored according to their scheduling priority. The first is the ready queue, which contains processes in the `NEW` and `READY` states. The second is the blocked-on-memory queue, which contains processes that are waiting on a memory block and are thus in the `BLOCKED_ON_MEMORY` state. The third is the blocked-on-receive queue, which contains processes that are waiting to receive a message and are thus in the `BLOCKED_ON_RECEIVE` state.

### 1.3.2 Releasing the Processor

Since our operating system does not employ time slicing, it is often necessary for a process to relinquish usage of the processor. This mechanism is provided by our operating system and can be employed by invoking the `release_processor` primitive.

When invoked, this primitive uses the scheduler to determine which process should be executed next. To do this, the scheduler iterates through the ready queue in order of decreasing priority, exercising a first-in first-out (round-robin) scheduling policy for processes of the same priority.

Once a process has been selected by the scheduler, a context switch will only occur if its priority is greater than or equal to that of the currently executing process. Otherwise, the caller will resume execution.

In order to perform a context switch, the context of the current process (i.e., the current stack pointer) must be saved to its process control block. Next, the context of the selected process is restored (i.e., its stack pointer is restored). When the saved registers are popped from the stack pointer of the selected process, execution will resume at the point where it left off.

### 1.3.3 Process Priority

Each process has a priority which is stored in its PCB. This priority is used to enforce correct precedence during scheduling, blocking, and preemption operations.

Our operating system allows processes to retrieve and modify the scheduling priority of themselves or other processes by providing two primitives: `get_process_priority` and `set_process_priority`. If a process's priority is changed while it resides in a process control queue, it is removed and then reinserted into the queue corresponding to its new priority. The `release_processor` primitive is then invoked to ensure that preemption will occur if necessary.

**Algorithm 3** Releasing the Processor

```
 1: procedure K_RELEASE_PROCESSOR()
 2:     if ready_queue is empty then
 3:         return RTOS_OK          ▷ Do nothing if the ready queue is empty
 4:     end if
 5:     next_proc ← K_DEQUEUE_READY_PROCESS()  ▷ Invoke the scheduler
 6:     if cur_proc.state ≠ BLOCKED then
 7:         if next_proc.priority > cur_proc.priority then
 8:             return RTOS_OK          ▷ Do nothing if the priority is lower
 9:         end if
10:     end if
11:     K_CONTEXT_SWITCH(cur_proc, next_proc)
12:     return RTOS_OK
13: end procedure

14: procedure K_DEQUEUE_READY_PROCESS()
15:     for i ← 0 to NUM_PRIORITIES do
16:         if ready_queue[i] is not empty then
17:             return DEQUEUE(ready_queue[i])
18:         end if
19:     end for
20:     return NULL
21: end procedure
```

**Algorithm 4** Context Switching

```
1:  procedure K_CONTEXT_SWITCH(prev_proc, next_proc)
2:      next_state ← next_proc.state
3:      if next_state ≠ NEW and next_state ≠ READY then
4:          return              ▷ Do nothing if next_proc is unable to execute
5:      end if
6:      if prev_proc.state = EXECUTING then
7:          prev_proc.state ← READY
8:          K_ENQUEUE_READY_PROCESS(prev_proc)
9:      end if
10:     prev_proc.sp ← __GET_MSP()
11:     next_proc.state ← EXECUTING
12:     __SET_MSP(next_proc.sp)
13:     if next_state = NEW then
14:         __RTE()          ▷ For new processes, pop the exception stack frame
15:     end if
16: end procedure
```

### 1.3.4 Interprocess Communication

Messages are sent between processes using message envelopes. Each PCB has a 'mailbox', implemented as a queue of message envelopes.

There are two structures used to represent messages: a kernel-facing message header containing the information required for communication, and a user-facing message envelope which contains only the data that is relevant to the sender and recipient. The header includes the PID of the sender, the PID of the recipient, and an expiry time. The user-facing envelope only contains the message type and the message data itself.

When a process wishes to send a message, it will first invoke `request_memory_block` for the envelope. It then populates the message envelope fields and invokes `send_message`. This primitive populates the message header fields and enqueues the message in the recipient's message queue. If necessary, the recipient will be unblocked and `release_processor` will be invoked so that preemption may occur; otherwise, control will be returned to the caller.

**Algorithm 5** Process Priority

```
 1: procedure K_GET_PROCESS_PRIORITY(proc)
 2:     if proc.pid is invalid then
 3:         return RTOS_ERR
 4:     end if
 5:     return proc.priority
 6: end procedure

 7: procedure K_SET_PROCESS_PRIORITY(proc, priority)
 8:     if proc.pid is invalid or priority is invalid then
 9:         return RTOS_ERR
10:     end if
11:     if proc.priority = priority then
12:         return RTOS_OK          ▷ Do nothing if the priority is unchanged
13:     end if
14:     if proc.state = NEW or proc.state = READY then
15:         REMOVE_FROM_QUEUE(proc, ready_queue)
16:         proc.priority ← priority
17:         K_ENQUEUE_READY_PROCESS(proc)
18:     else if proc.state = BLOCKED_ON_MEMORY then
19:         REMOVE_FROM_QUEUE(proc, blocked_on_memory_queue)
20:         proc.priority ← priority
21:         K_ENQUEUE_BLOCKED_ON_MEMORY_PROCESS(proc)
22:     else if proc.state = BLOCKED_ON_RECEIVE then
23:         REMOVE_FROM_QUEUE(proc, blocked_on_receive_queue)
24:         proc.priority ← priority
25:         K_ENQUEUE_BLOCKED_ON_RECEIVE_PROCESS(proc)
26:     else
27:         proc.priority ← priority
28:     end if
29:     K_RELEASE_PROCESSOR()
30:     return RTOS_OK
31: end procedure
```

**Algorithm 6** Sending Messages

```
 1: procedure K_SEND_MESSAGE(recipient, msg)
 2:     if recipient.pid is invalid then
 3:         return RTOS_ERR
 4:     end if
 5:     if K_SEND_MESSAGE_HELPER(cur_proc, recipient, msg) = 1 then
 6:         if recipient.priority ≤ cur_proc.priority then
 7:             return K_RELEASE_PROCESSOR()
 8:         end if
 9:     end if
10:     return RTOS_OK
11: end procedure

12: procedure K_SEND_MESSAGE_HELPER(sender, recipient, msg)
13:     msg.sender ← sender
14:     msg.recipient ← recipient
15:     ENQUEUE(msg, recipient.msg_queue)
16:     if recipient.state = BLOCKED_ON_RECEIVE then
17:         REMOVE_FROM_QUEUE(recipient, blocked_on_receive_queue)
18:         recipient.state ← READY
19:         K_ENQUEUE_READY_PROCESS(recipient)
20:         return 1            ▷ 1 indicates that the recipient was unblocked
21:     else
22:         return 0
23:     end if
24: end procedure
```

The procedure for delayed message sending is similar, with the added requirement that an expiration time is included in the header. Instead of directly enqueueing the message in the recipient's message queue, `delayed_send` enqueues the message in the timer i-process's message queue. As described in Section 1.5.1, the timer i-process will then take care of dispatching the message at the correct time.

When a process invokes `receive_message`, the next message will be dequeued from its message queue and returned, as long as there is at least

**Algorithm 7** Sending Delayed Messages

```
 1: procedure K_DELAYED_SEND(recipient, msg, delay)
 2:     if recipient.pid is invalid then
 3:         return RTOS_ERR
 4:     end if
 5:     msg.expiry ← cur_time + delay
 6:     msg.sender ← cur_proc
 7:     msg.recipient ← recipient
 8:     ENQUEUE(msg, timer_i_proc.msg_queue)
 9:     return RTOS_OK
10: end procedure
```

one pending message in the queue. Otherwise, the process is placed in the blocked-on-receive queue and preempted using `release_processor`.

**Algorithm 8** Receiving Messages

```
1: procedure K_RECEIVE_MESSAGE(sender)
2:     while cur_proc.msg_queue is empty do
3:         K_ENQUEUE_BLOCKED_ON_RECEIVE_PROCESS(cur_proc)
4:         K_RELEASE_PROCESSOR()
5:     end while
6:     msg ← DEQUEUE(cur_proc.msg_queue)
7:     sender ← msg.sender
8:     return msg
9: end procedure
```

## 1.4 System Processes

### 1.4.1 Null Process

The null process has the lowest priority of any process in our operating system. Since the processor must always be busy, the null process acts as a fail-safe for situations when no other process can be scheduled for execution. As such, the null process simply invokes `release_processor` in an in-

finite loop, allowing other processes to be scheduled as soon as they are ready.

**Algorithm 9** Null Process

```
1: procedure NULL_PROC()
2:     while true do
3:         RELEASE_PROCESSOR()
4:     end while
5: end procedure
```

### 1.4.2 KCD Process

The Keyboard Command Decoder (KCD) process provides a console-like mechanism to users of our operating system. Upon receipt of a command registration message from another process (retrieved using `receive_message`), it uses a global registry to associate the specified command with the message sender. This registry is implemented as a linked list of structures which contain a command identifier string and the PID of the registered process. Each time a line of input is entered through the console, the KCD process uses the registry to determine if it is prefixed with a registered command; if so, the input is forwarded to the process specified in the registry entry using `send_message` so that it may act upon the command.

### 1.4.3 CRT Process

The purpose of the CRT process is to print text to the system console. In order to achieve this, it repeatedly invokes `receive_message` and forwards received messages to the UART i-process using `send_message`. After it does this, it triggers a UART output interrupt so that the UART i-process may execute. The mechanism by which the UART i-process achieves interrupt-driven output is outlined in Section 1.5.2.

**Algorithm 10** KCD Process

```
1: procedure KCD_PROC()
2:     while true do
3:         msg ← RECEIVE_MESSAGE(sender)
4:         if msg.type = MSG_TYPE_KCD_REG then
5:             reg ← next unused kcd_reg entry
6:             reg.id ← msg.data
7:             reg.pid ← sender
8:         else if msg.type = MSG_TYPE_DEFAULT then
9:             id ← first token of msg.data
10:            if kcd_reg contains id then
11:                dispatch_msg ← REQUEST_MEMORY_BLOCK()
12:                dispatch_msg.type ← MSG_TYPE_KCD_DISPATCH
13:                dispatch_msg.data ← msg.data
14:                SEND_MESSAGE(kcd_reg[id].pid, dispatch_msg)
15:            end if
16:        end if
17:        RELEASE_MEMORY_BLOCK(msg)
18:    end while
19: end procedure
```

**Algorithm 11** CRT Process

```
1: procedure CRT_PROC()
2:     while true do
3:         msg ← RECEIVE_MESSAGE()
4:         if msg.type = MSG_TYPE_CRT_DISP then
5:             SEND_MESSAGE(uart_i_proc, msg)
6:         else
7:             RELEASE_MEMORY_BLOCK(msg)
8:         end if
9:     end while
10: end procedure
```

## 1.5 Interrupt Processes

Both the timer and UART i-processes are scheduled exclusively by their respective interrupt handlers; they are never placed in process control structures (e.g., the ready queue). When an interrupt is received, the following operations are performed:

- The context of the currently executing process is pushed onto the stack
- The appropriate i-process is invoked
- If necessary, the currently executing process is preempted
- The previously saved context is popped off of the stack

### 1.5.1 Timer I-Process

In order to provide timing services to our operating system, we programmed one of the LPC1768's on-chip timers to signal an interrupt once every millisecond. As described above, this means that the timer i-process will be scheduled at the same interval. The timer i-process is responsible for dispatching delayed messages (sent using `delayed_send`) at the correct time. This is achieved through the use of message queues and a global counter, used for keeping track of the system time.

**Algorithm 12** Timer I-Process

```
1:  procedure TIMER_I_PROC()
2:      msg ← K_NON_BLOCKING_RECEIVE_MESSAGE()
3:      SORTED_ENQUEUE(msg, timeout_queue)
4:      while timeout_queue contains expired messages do
5:          expired_msg ← DEQUEUE(timeout_queue)
6:          K_SEND_MESSAGE_HELPER(expired_msg)        ▷ Non-preemptive
7:          if expired_msg.recipient.priority ≤ cur_proc.priority then
8:              ▷ preempt cur_proc on completion
9:          end if
10:     end while
11: end procedure
```

### 1.5.2 UART I-Process

The UART i-process acts as an interface between the system console and other processes in our operating system. It is triggered by two types of interrupts, each corresponding to one of the two tasks performed by this process.

The first responsibility of the UART i-process is handling user input. To facilitate this, a global input buffer is used. When the user presses the return key, this buffer is copied into a message which is then sent to the KCD process for interpretation using a non-preemptive version of `send_message`.

When an output interrupt occurs, the UART i-process retrieves the next message in its queue using a non-blocking version of `receive_message`. It then uses repeated interrupts to sequentially print characters of the message until it reaches the end.

## 1.6 User Processes

### 1.6.1 Wall Clock Process

The wall clock process uses `delayed_send` to display a digital clock that updates every second. The process sends itself messages with a delay of one second, triggering "tick" updates. Each time the clock ticks, a message is sent to the CRT process using `send_message` to display the current wall clock time. Upon startup, the wall clock process registers three keyboard commands (`%WR`, `%WS`, `%WT`) with the KCD process which allow the time to be set, reset, and stopped.

### 1.6.2 Set Priority Command Process

Upon startup, this process registers a keyboard command (`%C`) with the KCD process. This command is handled by invoking `set_process_priority` using the specified process identifier and priority.

**Algorithm 13** UART I-Process

```
1: procedure UART_I_PROC()
2:     if input_char is available then
3:         if mem_heap is not empty then                    ▷ Avoid blocking
4:             msg ← K_REQUEST_MEMORY_BLOCK()
5:             msg.type ← MSG_TYPE_CRT_DISP
6:             msg.data ← input_char
7:             K_SEND_MESSAGE_HELPER(uart_i_proc, crt_proc, msg)
8:             ▷ preempt cur_proc on completion
9:         end if
10:        if input_char ≠ carriage return then
11:            add input_char to input_buffer
12:        else
13:            msg ← K_REQUEST_MEMORY_BLOCK()
14:            msg.type ← MSG_TYPE_DEFAULT
15:            msg.data ← input_buffer
16:            K_SEND_MESSAGE_HELPER(uart_i_proc, kcd_proc, msg)
17:            ▷ preempt cur_proc on completion
18:        end if
19:    else if output_msg is available then
20:        UART0 ← output_msg.data[output_msg_index]
21:        output_msg_index ← output_msg_index + 1
22:    end if
23: end procedure
```

### 1.6.3 Test Processes

In order to ensure the correct behaviour of our operating system, we use six user-level test processes to invoke kernel primitives and check for any incorrect results. These test processes use global variables to coordinate with each other and ensure that our operating system works as expected. Some of the key mechanisms that are tested include preemption, modification of process priority, memory block assignment, interprocess communication, and system processes such as the KCD and CRT processes.

# Chapter 2

# Lessons Learned

## 2.1 Version Control

At the very beginning of our project, we decided to use Git to manage the source code of our kernel. Paired with GitHub, it provided a robust system for ensuring code quality and minimizing bugs. Throughout the project, we obeyed a strict code review policy; all work was done in individual branches, submitted as a pull request, and then exhaustively reviewed by every member of the team before being merged into the `master` branch. This policy allowed us to catch several bugs that would have been incredibly difficult to catch in testing.

## 2.2 Simulation

For the majority of our first deliverable, we tested our kernel exclusively in the debugger provided by the Keil $\mu$Vision IDE. Before submitting the deliverable, we tested our code on the MCB1700 board; everything worked correctly. This substantiated our assumption that the debugger provided a close approximation to the hardware. In reality, as we discovered during our work on the second deliverable, there is a major difference: the SRAM on the MCB1700 board is not initialized the way it is in the debugger. This led to a number of serious bugs that only surfaced when we tested our code on the board. One of the most common bugs involved dealing with strings. Since we relied on the null character to delimit strings, there were portions of code which would work perfectly in the simulator and not on the board. These

types of bugs only appeared in the second deliverable, since message passing and console I/O both rely heavily on strings. After dealing with these bugs, we learned to write code more defensively and test our code primarily on the board instead of using the debugger.

## 2.3 Documentation

Beginning with the first deliverable, we decided to write a portion of our documentation alongside each submission. In the time between the first and third deliverables, we compiled a structural description of each part of our project, wrote pseudocode for each major kernel procedure, and completed the Lessons Learned section. This work greatly reduced the amount of time it took to complete our final report, as our documentation was nearly complete by the time we submitted the third deliverable.