RTOS Documentation

Kathleen Chung

kklchung@uwaterloo.ca

Connor Cimowsky

ccimowsky@uwaterloo.ca

Christian De Angelis

cdeangel@uwaterloo.ca

Jaclyne Ooi

jpooi@uwaterloo.ca

March 30, 2014

Abstract

This document describes a real-time operating system for the Keil MCB1700 Evaluation Board. It examines the design of the operating system, documents user-facing primitives, and provides an analysis of system performance.

Contents

List of Algorithms

List of Figures

Chapter 1

Design Description

1.1 Operating System Initialization

Once the operating system image has been copied to the LPC1768's on-chip ROM, execution can be initiated by pressing the `RESET` button on the MCB1700 board. The address of the `main` function is then loaded into the program counter and the CPU begins fetching and executing instructions. Subsequently, the board's hardware components (e.g., LEDs, timers, serial controllers) are initialized.

1.1.1 Memory Initialization

Now that the board's hardware is ready, it is necessary to initialize global data structures such as process control queues and the keyboard command registry. Most importantly, a portion of the on-chip SRAM is divided into fixed-size blocks (see Appendix A) and inserted into the global memory heap.

1.1.2 Process Initialization

Upon completion of memory initialization, this procedure iterates through the process initialization table and performs three tasks. First, it populates the process control block (PCB) for each process. Next, each PCB is placed in the ready queue (unless it is an i-process, since these processes are invoked by interrupt handlers). Finally, a stack frame is allocated for each process and the resulting stack pointer is stored in its PCB. At this point, the first process can be scheduled 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 (see Appendix A).

1.2.2 Requesting Memory Blocks

When a process invokes `request_memory_block`, the operating system first checks if the heap is empty. If so, it will block and then preempt the caller. Otherwise, the next available memory block is popped from the heap and returned.

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 points to a valid memory block. If so, the block is pushed onto the heap. If another process is blocked on memory, it will be unblocked and preemption will take place 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

Each process in the operating system is modelled by a process control block. This data structure contains the stack pointer, PID, priority, state, and message queue of a given process.

Depending on their state, processes can be placed in one of three process control queues. 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 in the `BLOCKED_ON_MEMORY` state. The third is the blocked-on-receive queue, which contains processes that are in the `BLOCKED_ON_RECEIVE` state.

1.3.2 Releasing the Processor

Since time slicing is not employed by the operating system, it is frequently necessary for a process to relinquish its usage of the processor. This mechanism is provided by the `release_processor` primitive.

When invoked, this procedure uses the scheduler to select the next process

for execution. If the priority of the selected process is greater than or equal to that of the currently executing process, a context switch will occur. Otherwise, the caller will resume execution.

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      ▷ Start at highest priority
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
```

1.3.3 Process Priority

As mentioned in the previous section, each process has a priority which is used to enforce correct precedence during scheduling, blocking, and preemption operations.

Processes may retrieve and modify the scheduling priority of themselves or other processes using two primitives: `get_process_priority` and

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
```

`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.

1.3.4 Interprocess Communication

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

Each message is modelled by two parts: a header used by the operating system, and an envelope used by the sender and recipient. The header contains the PID of the sender and recipient, as well as an expiry time for delayed messages. The envelope contains the message type and the data to be sent.

To send a message, a process must first invoke `request_memory_block` for an envelope. It then populates the envelope and invokes `send_message`, which populates the header and dispatches the message to the recipient. If necessary, the recipient will be unblocked and `release_processor` will

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
```

be invoked so that preemption may occur; otherwise, control will be returned to the caller.

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. When invoked, `delayed_send` places the specified message in the timer i-process's message queue. As described in Section 1.5.1, the timer i-process will dispatch the message at the correct time.

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
```

When a process invokes `receive_message`, the next message will be removed from its message queue and returned. If its message queue is empty, the process is blocked and then 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 the 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 executed. As such, the

null process simply invokes `release_processor` in an infinite loop.

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 interface to users of the operating system. Upon receipt of a command registration message, it uses a global registry to associate the specified command with the message sender. Each time a line of input is entered into the console, the KCD process determines if it is prefixed with a registered command. If so, the input is forwarded to the corresponding process.

1.4.3 CRT Process

The CRT process forwards text to the system console. In order to achieve this, it repeatedly forwards received messages to the UART i-process. It then 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.

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

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
```

- 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 a time counter and message queues (see appendices B.3.1 and B.3.2).

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 handles interrupts representing two scenarios:

- A character has been entered (see Figure 1.1)
- A character is ready for display (see Figure 1.2)

As such, the UART i-process acts as an interface between the user-facing console and processes in our operating system. To handle input and output, the UART i-process maintains its state between interrupts using global variables (see Appendix B.4).

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 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

Figure 1.1: Data Flow of Input Characters

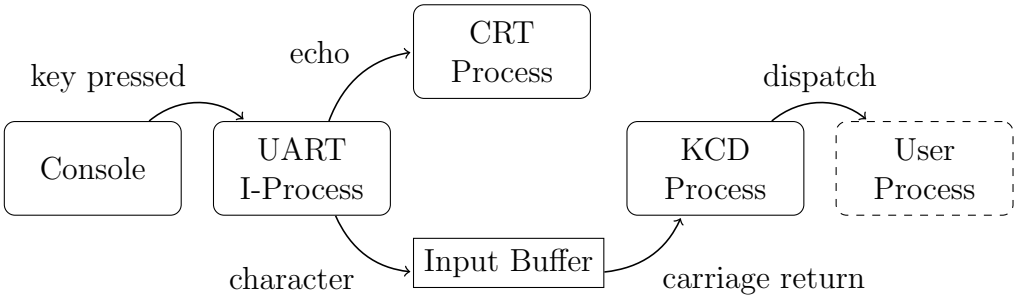
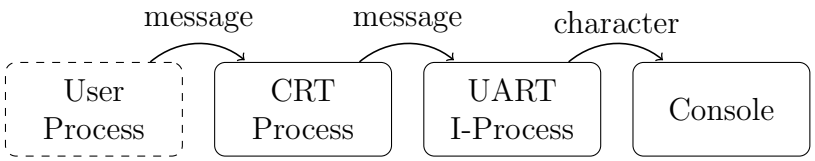


Figure 1.2: Data Flow of Output Characters



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.

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 operating system. 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 operating system exclusively in the debugger provided by the Keil μVision 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 procedure, and completed most of 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.

2.4 Linked Structures

Early on, we knew that we would need to store data in linked structures for different use cases. Since we wanted to avoid coupling these data structures with the type of data they were storing, we chose to make a generic node structure, `node_t`:

```
typedef struct node_t {
    struct node_t *mp_next;
    U32 m_val;
} node_t;
```

We then designed our linked structures to point to nodes of this type. For example, `queue_t`:

```
typedef struct queue_t {
    node_t *mp_first;
    node_t *mp_last;
} queue_t;
```

This way, we can declare new node types with additional members for different use cases. For example, here is a process control block:

```
typedef struct k_pcb_t {
    struct k_pcb_t *mp_next;
    U32 *mp_sp;
    U32 m_pid;
    PRIORITY_E m_priority;
    PROC_STATE_E m_state;
    queue_t m_msg_queue;
} k_pcb_t;
```

As long as the first member is still a pointer to the next node, our linked structures will accommodate these nodes. Creating these generic linked structures saved us a great deal of time in later deliverables.

2.5 Data Structure Allocation

Until the last deliverable, we allocated data structures in the `memory_init` routine. For example, here is the ready queue:

```
queue_t *gp_ready_queue[NUM_PRIORITIES];

for (i = 0; i < NUM_PRIORITIES; i++) {
    gp_ready_queue[i] = (queue_t *)p_end;
    queue_init(gp_ready_queue[i]);
    p_end += sizeof(queue_t);
}
```

This approach unnecessarily adds complexity to our code and also led to quite a few time-consuming bugs at the beginning of the project. To fix it, we began to statically allocate our data structures in the operating system image. Here is the same example using static allocation:

```
queue_t g_ready_queue[NUM_PRIORITIES];

for (i = 0; i < NUM_PRIORITIES; i++) {
    queue_init(&g_ready_queue[i]);
}
```

As a result, our data structures are less error-prone and our code is easier to read. This is the solution that we should have used at the beginning.

Appendix A

Configuration

A.1 Parameters

The following parameters can be adjusted to tune the operating system:

NUM_BLOCKS	Number of memory blocks to be placed in the heap.
BLOCK_SIZE	Size, in bytes, of each memory block in the heap.
USR_SZ_STACK	Size, in bytes, of the stack frame for each process.
NUM_KCD_REG	Maximum number of keyboard commands that can be registered using the KCD process.
KCD_REG_LENGTH	Maximum length of each command identifier.
MSG_LOG_SIZE	Number of recent messages to be logged.
MSG_LOG_LEN	Number of characters to be logged for each message.
NUM_TEST_PROCS	Number of user test processes.

A.2 Macros

The following macros can be defined to achieve additional behaviour:

DEBUG_0	Allow user test processes to print to the console.
DEBUG_1	Enable verbose output for debugging purposes.
DEBUG_LED	Monitor the memory heap using the on-board LEDs.
DEBUG_HOTKEYS	Enable the debug hotkeys (defined in the README).
DEBUG_TIMER	Scale the timer for debugging in the simulator.
DEBUG_DEMO	Leave the user test processes in a blocked state.

Appendix B

Global Variables

B.1 Memory

B.1.1 `g_heap`

During the memory initialization process, we allocate sections of SRAM into memory blocks. In order to efficiently serve the needs of processes, our operating system uses a linked list to manage these blocks.

B.1.2 `gp_heap_begin_addr`, `gp_heap_end_addr`

These variables are set during the memory initialization process and are used for bounds checking in `release_memory_block`.

B.1.3 `gp_stack`

A pointer to the top of the stack. Used during initialization to provide a stack frame for each process.

B.2 Processes

B.2.1 `g_pcbs`

An array of process control blocks for each process in the system. Indexed by PID for constant time access (e.g., from `send_message`).

B.2.2 `g_proc_table`

An array containing the information required (e.g., PID, priority, entry point) to initialize the process control block for each process in the system.

B.2.3 `gp_current_process`

A pointer to the PCB of the currently executing process. Used mainly for process control purposes (e.g., `release_processor`).

B.2.4 `g_ready_queue`

Implemented as an array of queues (one queue for each priority) containing pointers to process control blocks. Used by `release_processor` for scheduling processes.

B.2.5 `g_blocked_on_memory_queue`

An array of queues (one queue for each priority) containing pointers to process control blocks. Used by `request_memory_block` and `release_memory_block` for blocking and unblocking processes.

B.2.6 `g_blocked_on_receive_queue`

An array of queues (one queue for each priority) containing pointers to process control blocks. Used by `send_message` and `receive_message` for blocking and unblocking processes.

B.3 Timer I-Process

B.3.1 `g_timer_count`

The current system time. Used for determining when delayed messages have expired. Incremented each time a timer interrupt is received (1 ms intervals).

B.3.2 `g_timeout_queue`

A queue of messages that have not yet expired. Sorted by expiry time. Used for dispatching delayed messages at the correct time.

B.3.3 `g_timer_preemption_flag`

Used to indicate whether the timer interrupt handler should preempt the currently executing process on completion of the timer i-process.

B.4 UART I-Process

B.4.1 `g_input_buffer`

A buffer containing characters that have been entered by the user since the last carriage return.

B.4.2 `g_input_buffer_index`

The next available index of the input buffer. Incremented each time a character is appended to the input buffer.

B.4.3 `gp_cur_msg`

A pointer to the message currently being printed to the console.

B.4.4 `g_output_buffer_index`

The index of the next character of `gp_cur_msg` to be printed to the console.

B.4.5 `g_uart_preemption_flag`

Used to indicate whether the UART interrupt handler should preempt the currently executing process on completion of the UART i-process.

B.5 KCD Process

B.5.1 `g_kcd_reg`

An array containing keyboard commands that have been registered by other processes (e.g., the wall clock process).

B.6 Wall Clock Process

B.6.1 `g_wall_clock_start_time`

The system time at which the wall clock was started. Used for computing the elapsed time for display on the console.

B.6.2 `g_wall_clock_start_time_offset`

The start time specified by the '`%WS hh:mm:ss`' command. Used for computing the elapsed time for display on the console.

B.6.3 `g_wall_clock_running`

A flag used to indicate if the wall clock is currently running. Enables support for pausing and resuming the wall clock.