Department of Electrical and Computer Engineering
University of Waterloo

SE 350 Project Description

1 Introduction

In this project, you will design a small real-time executive (RTX) and implement it on an LPC1768 based microcontroller board. The executive will provide a basic multiprogramming environment, with 5 priority levels, preemption, simple memory management, message-based inter-process communication, a basic timing service, system console I/O and debugging support.

Such an RTX is suitable for embedded computers which operate in real time. A cooperative, non-malicious software environment is assumed. The design of the RTX should allow its placement in ROM. Applications and non-kernel RTX processes must execute in the *unprivileged* level of LPC1768. The RTX kernel will execute in the *privileged* level. We do not require application processes to use PSP (Process Stack Pointer). You can use the MSP (Main Stack Pointer) both for your kernel and non-kernel code. However we reserve the right to apply bonus mark to any non-trivial not-required implementations.

The description of the microcomputer and other project related information is available on Learn (http://learn.uwaterloo.ca). It has 32K of RAM for use by the RTX and application processes. It contains four timers, four UARTs and several other peripheral interface devices. The board has two RS-232 interfaces, from which UART0 is used for your RTX system console.

2 Summary of RTX Requirements

2.1 Scheduling Strategy

Four user priority levels plus an additional “hidden” priority level for the Null process, preemption, no time slicing, FIFO (First In, First Out) discipline at each priority level.

2.2 RTX Primitives and Services

Refer to the section RTX Primitives and Services.

2.3 RTX Footprint and Processor Loading

A reasonably 'lean' implementation is expected. No standard C library function call is allowed in the kernel code.

2.4 Error Detection and Recovery

At minimum, the RTX kernel must detect one type of error: an attempt to *send_message* to or *set_process_priority* of a non-existent process_id. The primitive will return an error code (a non-zero integer value). No error recovery is required. It may be assumed that the application processes can deal with this situation.

3 Organization and Deliverables

3.1 Project Groups

The project is done in groups of four. A group of less than four members is not recommended. There is no reduction in project deliverables regardless the size of the project group.

Group sign-up is done in the Course Book System (see link on Learn website). The deadline for group signup is January 19th, 2012. A 5% late sign-up penalty will be applied to students who do not sign up for a project group in the Course Book System by the deadline.

Everyone in the group normally gets the same mark. If you notice workload imbalance, try to solve it as soon as possible within your group or split-up the group as the last resort.

Group split-up is only allowed once. A written notice given at least one week before a deliverable due date needs to be submitted to the lab instructor. There is a 1% group break-up penalty. We highly recommend everyone to stay with your group members as much as possible, for the ability to do team work will be an important skill in your future career.

3.2 Deliverables

The project has four deliverables, where the first three deliverables are evaluated by in lab demonstrations and ALL project group members are required to be presented during the demonstrations. The deliverables are as follows:

1. *RTX Project P1*: This is the source code and documentation of an OS which provides memory management and processor management services. You need to implement APIs listed in Sections 4.1, 4.2 and 4.5 together with processes in 5.1.1 and 5.3.4 and the corresponding part in Sections 6 and 7. Note you need to write your own testing processes to demonstrate that your implementation meets the requirements.
2. *RTX Project P2*: This is the source code and documentation of a simplified version of the final RTX. On top of P1, you will add APIs listed in Sections 4.3 and 4.4. You need to finish implementing all processes as described in Sections 5.1 and 5.2 and 5.3.2 and the corresponding part in Sections 6 and 7.
3. *RTX Project P3:*. This is the final RTX source code to implement the specifications in sections 4, 5 6, and 7 based on the P1 and P2 implementations done previously. It is possible that you will discover gaps or flaws in your original design during the implementation phase. You are allowed to adjust your implementation to reflect such discoveries. *It is expected that you will document any major changes in the Final Project Report.*
4. *Final Project Report:* This report will include
   a) RTX Project Software Design Description*:* This document describes the RTX design. It should include:
      - a structural description of the design (procedures and their interconnect; data structures; processes);
      - a functional description of all procedures (pseudo code; show all input/output parameters and globals);
      - implementation, testing and measurement plan (include responsibilities of individual team members).

      This part document should be kept reasonably small (no more than 30 pages, not including appendices). A design description should be shorter than the actual implementation.
   b) An overview of major design changes made during implementation, if any.
   c) The measured times for each of the *send_message*, *receive_message* and *request_memory_block* primitives, and a check of the reasonableness of the values measured
   d) A 'lessons learned' summary (what you did do well, both technically and organizationally, and what you would do differently if you were to do it again). This summary should be brief (1-2 pages).

3.3 Due Dates and Mark Distribution

Deliverable	*Weight*	*Due Date*	*File Name*
RTX Project P1 Implementation [0,3]	25%	TBA[2]	p1_Gid.ext
RTX Project P1 Demonstration[4]		TBA[2]	
RTX Project P2 Implementation[0,3]	30%	TBA[2]	p2_Gid.ext
RTX Project P2 Demonstration[4]		TBA[2]	
RTX Project P3 Implementation[0,3]	20%	TBA[2]	p3_Gid.ext
RTX Project P3 Demonstration[4]		TBA[2]	
Final Project Report[0,1]	25%	TBA[2]	frpt_Gid.ext

Notes:

[0] The name for RTX project P1 implementation, RTX project P2 implementation, RTX project P3 implementation, and final project report should be ***p1_Gid.ext, p2_Gid.ext, p3_Gid.ext,* and *frpt_Gid.ext*** respectively. Replace the 'id' with your group id number (for example G099) and "ext" with proper file extension. The ECE Course Book System is to be used for softcopy submission.

[1] Electronic submission in Microsoft Word format. Both soft and hard copies of submissions are required.

[2] To be confirmed with the class. The dates will be posted on the Learn web site.

[3] Electronic submission. Put all source, header files, and binaries in a separate (submit) directory. Include a README file with group identification, description of directory contents and compilation procedure (and a makefile). Compress the directory contents into a single file. For archiving, you must choose *zip*.

[4] Demonstration may be waived when TAs can mark the deliverable off-line.

3.4 Development and Demonstration Environment

Software development support includes MDK-ARM from Keil on Nexus workstations. The course laboratories are located in E2-2363. They contain Nexus workstations and LPC1768 based MCB1700 microcontroller boards. The demonstration of RTX operation (cf. Deliverables) will be done in the course laboratories. Project TAs and lab instructor will be available in during scheduled lab hours and their office hours to assist you with your implementation problems. Note that all the scheduled lab sessions for SE 350 are shown in Course Book System. .

You may prefer to do the initial development on a platform which is more convenient to access for you (e.g. a home computer). With some preparation, you could also test and debug most of your code on such a platform (you would need to use a simulator within the MDK-ARM). However, as stated earlier, the project must be demonstrated on the MCB1700 boards in the course laboratories.

4 Description of RTX Primitives & Services

This section lists the RTX primitive and services. You must implement theses as described and may not modify the prototypes in any way. The primitives listed below will always return a value, either a pointer or an `int` return code. In the latter case, the return code value of 0 indicates success; non-zero value indicates a failure where applicable.

4.1 Memory Management

The RTX supports a simple memory management scheme. The memory is divided into blocks of fixed size (128 bytes *minimum*). The size and the number of these blocks is a configuration parameter. The blocks can be used by the requesting processes for storing local variables or as envelopes for messages sent to other processes. A block which is no longer needed must be returned to the RTX. Two primitives are to be provided:

`void * 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 until a memory block becomes available. If several processes are waiting for a memory block and a block becomes available, the highest priority waiting process will get it.

`int release_memory_block (void * MemoryBlock)`
This primitive returns the 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.

4.2 Processor Management

`int release_processor ()`
Control is transferred to the RTX (the calling process voluntarily releases the processor). The invoking process remains ready to execute. Another process may possibly be selected for execution.

4.3 Interprocess Communication

The RTX will support a message-based IPC discussed in lectures. Messages are carried in envelopes (memory blocks, see below) with a header which is less than 64 bytes. Two IPC primitives will be implemented. The two primitives are:

`int send_message (int process_ID, void * MessageEnvelope)`
Delivers to the destination process a message carried in the message envelope (a memory block). Changes the state of destination process to ready_to_execute 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 fields *sender_process_id*, *destination_process_id* and *message_type* are all of type `int`. The sender fills in the *message_type* field of the message envelope before invoking the primitive.

`void * receive_message (int * sender_ID)`
This is a blocking receive. If there is a message waiting, a pointer to the message envelope containing it will be returned to the caller. If there is no such message, the calling process blocks and another process is selected for execution. The sender of the message is identified through sender_ID, unless it is NULL. Note the sender_ID is an output parameter and is not meant to filter which message to receive.

4.4 Timing Services

`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* (timeout, given in msec units).

4.5 Process Priority

Process priorities have an integer *priority* value (0, 1, 2, 3, 4) where 0 is the highest priority level.

`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 (see also section 5.1). The caller of this primitive never blocks, but could be preempted. This preemption may affect the currently executing process.

`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".

5 Required Processes

This section describes the processes which you must implement for the project.

5.1 System Processes

System processes are those processes needed by the system to perform basic services (scheduling and I/O).

5.1.1 Null Process

This process runs as the lowest priority process (level 4) in the RTX.. The Null process is the only process assigned to level 4. Level 4 is basically a "hidden" priority level reserved for the Null process. This preserves the 4 levels of user priorities (levels 0, .. , 3). Process id 0 is reserved for the null process. Initially, the following pseudo code can be used to design your null process:

```
loop forever
      release_processor( )
end loop
```

Once you have preemption working, then the "release_processor( )" line could be removed from the infinite loop.

5.1.2 System Console I/O Processes

The system console is used for communication with the RTX and application processes. It consists of two devices: keyboard and CRT display. These two devices communicate serially with the microcomputer; using the receive and transmit lines of one of the two RS-232 ports.

The RTX will include two system processes, the Keyboard Command Decoder (KCD) process and the CRT Display process. These processes work in cooperation with the UART interrupt handler i-process.

5.1.2.1 Keyboard Command Decoder process

A keyboard command starts with the prompt character %, followed by a single (or multiple) letter command identifier and possibly additional command data. For example, %WS12:45:00 could be a command to the wall clock process, telling it to start the wall clock, setting the current time to 12:45:00 (where the command format is %WS hh:mm:ss).

The command decoder process responds to two types of messages: console keyboard input and command registration. The latter contains the command identifier and the process id of the process to which such commands are to be delivered when entered on the console keyboard. The processing of messages received depends on their type:

1. **Command Registration**

The command identifier is associated with the registrant's process id.

2. **Keyboard Input**
   The string input is sent to CRT display for output. If the string begins with a registered command identifier, it is also sent to the registered requester.

5.1.2.2 CRT Display Process

This process responds to only one message type: a CRT display request. The message body contains the character string to be displayed. The string may contain control characters (e.g. newline). The process causes the string to be output to the console CRT. In printing to the console display, the process must use the UART i-process. Any message received is freed using the `release_memory_block` primitive.

5.2 Interrupt Processes (I-Processes)

Two interrupt handling processes are required:

5.2.1 Timer I-Process

The timer i-process is executed each time a hardware timer interrupt occurs. The timer i-process should handle the delivery of delayed send messages after the required time has expired.

5.2.2 UART I-Process

The UART i-process uses interrupts for both the transmission and receiving of characters from the serial port. No polling or busy waiting strategies may be implemented. The UART i-process forwards characters (or commands) received to the KCD, and also responds to messages received from the CRT display process to transmit characters to the serial port.

5.2.2.1 Hot Keys

As well, the UART i-process is used to provide debugging services which will be used during the demonstration. Upon receiving specific characters (*hot keys* - your choice, e.g., **!**) as input, the UART i-process will print the following to the RTX system console:

1. The processes currently on the ready queue(s) and their priority.
2. The processes currently on the blocked on memory queue(s) and their priorities.
3. The processes currently on the blocked on receive queue(s) and their priorities.

As well, you are free to implement other hot keys to help in debugging. For example, a hot key which lists the processes, their priorities, their states; or another which prints out the number of memory blocks available. Like all other debug prints, the hot key implementation should be wrapped in

```
#ifdef _DEBUG_HOTKEYS
...
#endif
```

preprocessor statements and should be turned off during automated testing. If the automated test processes fail, you may be asked to turn the hot keys on again in determining why the test processes are failing.

Another hotkey debug printout may be used to display recent interprocess message passing. A (circular) log buffer keeps track of the 10 most recent `send_message` and `receive_message` invocations made by the processes; upon receiving a specific hotkey, these most recent10 sent and 10 received messages are printed to the debug console. The number 10 is used only as an example. The information printed could contain information such as:

1. Sender process id
2. Destination process id
3. Message type
4. First 16 bytes of the message text

5. The time stamp of the transaction (using the RTX clock)

5.3 User Processes

These processes will be used to demonstrate the operation of your system.

5.3.1 Set Priority Command Process

This process registers itself with the Keyboard Command Decoder process as the handler for the %C command. The %C command has two parameters:

% C *process_id* *new_priority*

where *process_id* and *new_priority* are integers. This command changes the priority of the specified process, *process_id*, to *new_priority*. The change in priority level is immediate. It could also affect the target process's position on a ready queue or a blocked resource queue. The parameters must be verified to ensure a valid *process_id* and priority level is given. A %C command with illegal parameters will be ignored with an error message printed on the console.

5.3.2 24 Hour Wall Clock Display Process

This process registers itself with the Keyboard Command Decoder process as the handler for the %W command. The %WS hh:mm:ss command sets the current wall clock time to hh:mm:ss, starts the clock running and causes display of the current wall clock time on the console CRT. The display will be updated every second. The %WT command will cause the wall clock display to be terminated.

5.3.3 Demonstration of User-Level Processes

An important category of software tests are the stress tests. These tests seek to verify the behavior of the system under heavy stress scenarios. One such scenario is depletion (or near depletion) of system resources. For the demonstration of this project, you will implement three processes whose behavior is described below. The stress scenario being tested is depletion of memory blocks.

Process A:

```
p <- request_memory_block
register with Command Decoder as handler of %Z commands
loop forever
    p <- receive a message
    if the message(p) contains the %Z command then
       release_memory_block(p)
       exit the loop
    else
       release_memory_block(p)
    endif
endloop
num = 0
loop forever
    p <- request memory block to be used as a message envelope
    set message_type field of p to "count_report"
    set msg_data[0] field of p to num
    send the message(p) to process B
    num = num + 1
    release_processor()
endloop
// note that Process A does not deallocate
// any received envelopes in the second loop
```

Process B:

```
loop forever
  receive a message
  send the message to process C
```

```
    endloop
```

Process C:

```
    perform any needed initialization and create a local message queue
    loop forever
       if (local message queue is empty) then
          p <- receive a message
       else
          p <- dequeue the first message from the local message queue
       endif
      if msg_type of p == "count_report" then
          if msg_data[0] of p is evenly divisible by 20 then
             send "Process C" to CRT display using msg envelope p
             hibernate for 10 sec
          endif
      endif
      deallocate message envelope p
      release_processor()
    endloop
```

The line "**`hibernate for 10 sec`**" is further expanded as:

```
    q <- request_memory_block()
    request a delayed_send for 10 sec delay with msg_type=wakeup10 using q
    loop forever
        p <- receive a message      //block and let other processes execute
        if (message_type of p ==  wakeup10) then
           exit this loop
        else
           put message (p) on the local message queue for later processing
        endif
    endloop
```

Notes:

- Process C has a local message queue (distinct from the incoming message queue maintained by the RTX) onto which it enqueues (in FIFO order) messages which arrive while it hibernates. It processes these messages later.
- For your own testing, set the priority levels for processes A, B and C to values which are most likely to cause memory block depletion in the RTX. During project demo, you may be asked to re-initialize your RTX with TA/instructor specified priorities for A, B, and C and vary the total number of message envelopes available.
- <u>It is recommended that test processes A, B and C have process ids 7, 8, and 9 respectively</u>. If you choose not to do this, you should have this information ready before the demonstration begins.

5.3.4 User-Level Test Process

Write up to six user-level test processes to test your own OS. These test processes should run in *unprivileged* level and do not assume any kernel level data structures. These test processes only call the RTX APIs. The test processes should provide at least two and at most 6 test cases and finish testing within 30 seconds. The process id 1, 2, 3, 4, 5, and 6 are reserved for these processes. Please refer to section 7 for more information on how to format your testing results and output it to the Hyper-Terminal.

6 RTX Initialization

To make the RTX more generally applicable, the RTX will be configured at initialization as specified in the RTX Configuration Table. This table has three sections:

1. Memory configuration section: memory block size, number of memory blocks created;
2. System process section;

3. Application process section.

The process sections list the processes to be created. Each entry contains the following data: process id, priority, stack size, start address, and for system processes, whether the process is an i-process. *All initializations must take place after the RTX execution starts*.

7 Testing

The demonstration of your working project will consist of two phases. In the first phase, initial manual testing will be performed using the specified keyboard commands without the testing processes you wrote. There should be exactly 30 memory blocks available to the user processes for this part of the demonstration.

In the second phase, you will be required to include the test processes (written by yourself, see section 5.3.4) in your RTX and execute the test sequence.

One requirement is that the six testing processes have reserved process ids 1, 2, 3, 4, 5 and 6. An overview of the expected mapping of process ids to processes is given in the table below:

Process_id	Owner Process	Process_id	Owner Process
0	Null process	6	Test_process_6
1	Test_process_1	7	Process A
2	Test_process_2	8	Process B
3	Test_process_3	9	Process C
4	Test_process_4	10…	Other processes start from pid 10
5	Test_process_5	…	

Since the test processes have no knowledge of your detailed internal design, they only invoke the functions specified by the RTX API. The test processes can use the timer that is not used by the RTX for timing testing. We require the testing results to follow the following format and you output the results to the HyperTerminal (i.e. UART0):

```
Gid_test: START
Gid_test: test n OK
Gid_test: test m FAIL
Gid_test: x/N tests OK
Gid_test: y/N tests FAIL
Gid_test: END
```

For example, if you are group G099 and you have 3 testing cases in total. Two of the testing cases pass and one of the testing cases does not pass. The final testing results should be output to HyperTerminal as follows:

```
G099_test: START
G099_test: total 3 tests
G099_test: test 1 OK
G099_test: test 2 OK
G099_test: test 3 FAIL
G099_test: 2/3 tests OK
G099_test: 1/3 tests FAIL
G099_test: END
```