Department of Electrical and Computer Engineering
University of Waterloo

MTE 241 Project Description

1 Project Overview

A small real time executive (RTX) is to be designed and implemented in C++. The executive will provide a basic multi-programming facility, with 4 priority levels, storage management, interprocess communication, basic timing service, system console I/O and debugging support. The intended application is real time control in an embedded microcomputer system. To support the RTX, the microcomputer must provide real-time clock interrupts and serial I/O interface. A cooperative, non-malicious software environment is assumed.

Embedded software is often developed on a platform different from the one on which it will ultimately execute (e.g. a work-station). The testing and debugging of such software is carried out in two steps. In the first step, the software is tested and debugged in the development environment. To carry this out, additional software may have to be developed to simulate the embedded hardware interfaces. In the second step, the software is ported to the embedded computer and debugged/tested there. If the first step was done right, the second step is relatively straightforward.

Because of time constraints, you will not be required to proceed with step two. Instead, you will demonstrate the correct operation of the RTX in the Linux environment. For the demonstration, you will develop three RTX application processes. In this project, the development environment consists of Linux workstations. The RTX will execute as a single Linux process. Two other Linux processes developed by your group will simulate the serial I/O of the embedded microcomputer and Linux system services will be used to generate real-time clock ticks.

2 Summary of RTX Requirements

2.1 Scheduling Strategy

Four (4) priority levels, no pre-emption and no time slicing.

2.2 Primitives and Services

See the section on primitives and services.

2.3 Processor Loading

A reasonably 'lean' implementation is expected.

2.4 Error Detection and Recovery

At the minimum, the RTX kernel must detect two types of simple errors: 1) an attempt to send a message to or, 2) to set the priority of a non-existent process or if an illegal priority level is specified. The respective primitive will return an error code. No kernel-level error recovery is required. It may be assumed that the application processes can deal with the error codes returned.

3 Deliverables

You will work in groups with a maximum of 4 members per group. The project has four deliverables:

1) RTX SDD (Software Design Description): This document will contain a complete design description of the RTX. It must show all classes and their relationships. Each individual function and major internal procedure must be defined as to input/output parameters, any return value and, where appropriate, a description of its functionality and sufficiently complete pseudocode. The detail of the design and pseudo code must be at a level that permits easy coding in C++. The design must show the inter-process and internal primitive interaction diagrams. The design will follow an object oriented approach, applying the object and, where appropriate, the dynamic and functional modelling techniques. The design description will include a brief section outlining the planned division of implementation and testing responsibilities among the group members. It is important to keep this document reasonably small, usually 30 pages or less. A design

description should be shorter than the actual implementation. Both a hard and soft copy is to be submitted.

2) RTX Implementation: The design described in your RTX SDD will be implemented in C++. It is possible that you will discover gaps in your original design during the implementation phase. You are allowed to adjust your implementation to reflect such discoveries. However, it is expected that you will document any major changes in the Final Project Report (see below).

3) Demonstration of RTX: The operation of the RTX must be demonstrated to a TA or to the instructor on a Linux workstation. The RTX application processes that will be used in the demonstration are described later. All group members must attend the demonstration.

4) Final Project Report: The final report will include:

- A brief summary of major design changes that occurred during implementation, if any.
- A 'lessons learned' summary (a summary of what you felt you did do well, both technically and organizationally, and what you would do differently if you were to do the project again). This part of the summary should be brief (1-2 pages).

3.1 Due Dates and Weights

Deliverable	*Weight*	*Due Date*
Design Description[1]	40%	October 21 (11:59 pm)
Demonstration	50%	November 30 – December 2
Source Code[2]	(included in demo)	November 28 (11:59 pm)
Final Report[1]	10%	Within 48 hrs of project demo

Notes:

1. Document submission: submit both a soft copy and hard copy. The acceptable formats for the soft copy are OpenOffice.org 'odt' files, Microsoft Word “doc” files or PDF files. The soft copies are submitted to\the course ACE as per instructions on the site. The 'hard' copies are submitted as indicated in lectures.
2. Code electronic submission: Put all source, header files, makefile in directories such that execution of your makefile will build your system. Do not include any object or executable files. Include a README file with group identification, description of directory contents. Your system will be built at time of demo using the makefile. Compress these directories (i.e.using ZIP) and submit to the course ACE as per instructions on the site. More details will be given near the due date.

4 Development and Demonstration Environment

The development and demonstration environment for the project is the ECE Department's ecelinux. Nexus and personal computers (using ssh) can also be used for remote access to the ecelinux host. The C++ compiler is g++, and the debugger is gdb or ddd. Read the man pages for more information on each of these. gcc follows the ANSI standard for C. Some code fragments will be provided to illustrate various Linux system calls. For additional project related information, see the course ACE.

Although the above facilities are available, you may prefer to do the initial development on a platform which is more convenient for you (e.g. home PC, personal Linux computer, Nexus etc.). However, the project must execute and be demonstrated on the ecelinux host.

5 Description of RTX Primitives & Services

This section lists the RTX primitives and services. This defines the API of the RTX. We assume that there is a global class “RTX” defined and also the class “*MsgEnv*” which defines the message envelopes. The following primitives are methods within the public methods of the “RTX” class and are callable by user level processes. [see Appendix C] *The primitives listed below return a value, either a pointer to some object or an integer return code. In the latter case, the return code value of 0 indicates success, a non-zero value (some error code) indicates a failure where applicable.* Do

not change the API signature for any specified primitive or CCI command.

5.1 Interprocess Communication

int RTX:: send_message(int dest_process_id, MsgEnv * msg_envelope)

Inserts the invoking process's process_id and the destination process's_id into the appropriate fields of the message envelope. Delivers to the destination process a message carried in the message envelope, may possibly change the state of the destination process to ready to execute if needed/allowed. The sending process does not block.

MsgEnv * RTX:: receive_message( )

If successful, it returns a pointer to a message envelope. This is a potentially blocking receive. If there is no message already waiting to be received by the requesting process, then the process becomes blocked [waiting for message] and another process is selected for execution. Otherwise, the message is delivered and the process continues.

5.2 Storage Management

The RTX supports a simple memory management scheme. The memory is divided into message envelopes of fixed size (e.g., multiples of 256 bytes). The size and the number of these message envelopes is to be a configuration parameter defined at compile time. The message envelopes can be used by the requesting processes for storing local data or as envelopes for messages sent to other processes. A message envelope which is no longer needed must be returned to the RTX. For simplicity, we assume that the RTX does not track ownership of the message envelopes once they have been allocated to a process. Two primitives are to be provided:

MsgEnv * RTX:: request_msg_env( )

Returns to the calling process a pointer to a standard size message envelope object. If no message envelope is available, the process is blocked until an envelope becomes available.

int RTX:: release_msg_env(MsgEnv * msg_env_ptr)

Returns a no longer needed message envelope to the RTX. If there is a process blocked on this resource, the message envelope is given to it and the process is made ready. If several processes are blocked waiting for a message envelope, the process with the highest priority will be served first. The caller does not block.

5.3 Processor Management

int RTX:: release_processor( )

Control is transferred to the RTX (i.e., the invoking process voluntarily requests a process switch). The invoking process remains ready to execute and another ready process may possibly be selected for execution.

int RTX:: request_process_status(MsgEnv * msg_env_ptr)

The RTX will return the current status and priority of all processes. The status (i.e. ready, blocked_on_envelope_request, blocked_on_receive, etc.) and priority is returned in the message envelope provided by the invoking process. The following format can be used: an array of tuples [proc_id, status, priority], where the array is preceded by an integer representing the number of tuples to follow. The invoking process does not block.

int RTX:: terminate( )

This requests the RTX to terminate its execution. The action is to terminate all processes and stop execution of the RTX in an orderly manner. Its effect is immediate. [In the development environment, this will terminate the RTX program, its associated child processes, release any Linux resources and return control to Linux.].

int RTX:: change_priority(int new_priority, int target_process_id)

For a valid input argument list, the priority of the target process is changed to new_priority. The change is immediate, and if applicable, may affect the target's position within any ready/resource queue.

5.4 Timing Services

The RTX requires at least 100 msec time resolution. The 'resolution' is often dependent on the eceLinux load at the time of execution. So, smaller time values are preferable if they can be reliably obtained. The following primitive is to be provided:

int RTX:: request_delay(int time_delay, int wakeup_code, MsgEnv * message_envelope)

The time_delay represents the number of 100ms. intervals for the delay. The invoking process does not block. A message (in the message envelope provided on invocation) will be sent to it by the timing service after the expiration of the delay (time-out). The wakeup message will have the message_type field set to *wakeup_code*.

5.5 System Console I/O

The RTX is intended for embedded microcomputer applications in which the embedded microcomputer can communicate with the operator (or software developer) using a system console. The console consists of an input device (keyboard) and an output device (character display). The communication is serial, through an advanced UART (universal asynchronous receiver/transmitter). A description of the UART can be found in Appendix A. Appendix B shows how the UART can be simulated in the Linux environment.

int RTX:: send_console_chars(MsgEnv * message_envelope)

The message envelope contains the character string to be sent to the system console display. The string to be printed is in the usual C/C++ string format terminated by a null character. This service will send the original envelope back to the requesting process as an acknowledgement after the characters have been transmitted from the UART. The returned message_type will be display_ack. The invoking process does not block.

int RTX:: get_console_chars(MsgEnv * message_envelope)

The invoking process provides a pointer to a message envelope (previously allocated) and continues executing. A message is sent to the invoking process using the envelope provided when a full line (terminating with an end-of-line), is received from the console keyboard or when the Receiver buffer in the UART fills up. Its return message_type is console_input. The message contains the characters received. The end of the keyboard string is indicated by a null character.

Note: successive invocations of either primitive are processed serially (e.g. an invocation of get_console_chars is kept until a preceding invocation has been satisfied; it does not override it).

5.6 Interprocess Message Trace

The executive will maintain a trace of the 16 most recent successful invocations of both the send and of the receive primitive. For example, if process A sends a message to process B then two events will be recorded. The first event is associated with the original send and the second event is when process B receives the message. This is often very useful in debugging large systems when some messages appear to become lost or if some processes never seem to receive any messages.

Each entry in the trace buffer will include the following items:

- destination_process_id
- sender_process_id
- message_type
- time stamp of the transaction

The time stamps refer to the value of the RTX clock (not the wall clock) when the message was initially sent and to the value of the clock when the destination process actually received the message due to a successful receive invocation. The message trace service will take a snapshot of the buffers at the instant of the request.

int RTX:: get_trace_buffers(MsgEnv * message_envelope)

The invoking process supplies a pointer to a message envelope (which it has previously obtained). The executive will copy the current contents of the two 16-element traces into the message envelope. The contents of the envelope will be the send trace followed by the receive trace. The invoking process does not block.

Obviously, this may take some time, which might impact the performance in real-time. However, the primitive would be very infrequently invoked (usually during debugging) and should have negligible impact.

6 Initialization

When the RTX is started, the following actions are carried out:

1. create instances and initialize all RTX objects.
2. set up the timing services.
3. set up application processes defined in the process initialization table with entries:
   - process_id
   - priority
   - size of stack
   - process start address
1. initialize all iprocesses, shared memory, the signalling system and fork the helper processes.
2. transfer control to the highest priority ready process.

7 Console Command Interpreter

The Console Command Interpreter (CCI) is essentially a high priority RTX user application process. It provides an interface to the RTX allowing the system console user to monitor or affect the execution of the RTX and its processes. Through the appropriate RTX service (get_console_chars), the CCI receives the console keyboard input during the RTX execution. The CCI parses the keyboard input and performs the requested action. It outputs the result to the system console display using the appropriate RTX service (send_console_chars). Unknown commands and commands with syntax errors are to be detected and the user prompted. The CCI can employ other user level processes to perform any of the operations.

The CCI will also maintain and display a 24-hour clock (hh:mm:ss). The display is to be refreshed every second. (Note: In the demonstration environment, groups may use the appropriate Linux utility to position the time display in the upper right location of the display, so that the time will not scroll off the screen).

The following specification applies to the CCI:

The user is displayed the string, CCI:, as a prompt. All complete command strings appear on a single line and are terminated by a carriage-return, <cr>. All character strings are separated by white space (e.g. tab, space, etc.). A blank command (i.e. white space followed by a <cr>) is to be ignored and the CCI: prompt sent to the console. The command strings are case independent. Your system should be able to handle any 'random' input from the keyboard and deal appropriately with the input.

The following commands are to be supported (**<cr>** is the return key, i.e. Carriage returen):

CCI: s <cr>

This causes the CCI to send a message envelope to User_Process_A (see section 8). The message envelope will not be returned to CCI. This causes the user demonstration processes to begin execution since User_Process_A is blocked waiting to receive a message at system start-up.

CCI: ps <cr>

Display the process status of all processes. For each known process, display the process_id, priority and process_status (e.g. ready, executing, blocked_on_..) in a tabular form with appropriate column headings and ordered according to process_id. [See request_process_status.] Try to format it in a human-readable display.

CCI: c hh:mm:ss <cr>

This sets the CCI wall clock to the hours, minutes, seconds specified using a 24 hour format (hh:mm:ss). The effect is immediate. Note that this has no effect on the RTX internal time stamp clock. The CCI uses this wall

clock to print the time on the console screen. Make certain the time specified is a legal time.

CCI: cd <cr>

This command allows the display of the wall clock on the console. When the system first starts, the wall clock is disabled and does not appear on the console although the wall clock is maintained and updated internally.

CCI: ct <cr>

This command stops the display of the wall clock on the system console. During the time that the wall clock is not displayed, the wall clock's time is maintained internally and may be redisplayed by the 'cd' command. The display of the wall clock can be turned on and off by alternating between the 'cd' and 'ct' commands repetitively.

CCI: b <cr>

Display the current contents of the send and receive trace buffers. The trace will be printed on the system console display (in a suitable human readable format) upon command from the keyboard. The RTX and the message tracing will operate normally while the trace is being printed on the screen. [See get_trace_buffers.]

CCI: t <cr>

Terminate all processes and stop execution of the RTX. (In the demonstration environment, terminate the RTX process and the UART processes, release all resources acquired from Linux and return to Linux console control.) [See terminate.]

CCI: n new_priority process_id <cr>

Where process_id and new_priority are integers. This command changes the priority of the specified process to new_priority. The change in priority level is immediate. It could also affect the target process's position on a ready/resource queue. The arguments must be verified to ensure a valid process_id and priority level is given. Illegal arguments can be ignored with an error message printed on the console window.
Note: It must not be allowed to alter the priority of the 'null process'. The priority of all other processes (other than any iprocesses) must be allowed to be changed to any legal priority level by this command.

You are also allowed to implement additional commands to aid in debug.

8 RTX Application Processes for Demonstration

The user application level processes are informally described as:

```
Process A:
        receive a message   // sent by CCI in response to a "s" command from the keyboard
        deallocate the received message envelope
        num ← 0
        loop forever
                get a message envelope
                set the message_type field to "count_report"
                set the msg_data[1] field to num
                send the envelope to process B
                num ← num + 1
                release_processor
        endloop

Process B:
        loop forever
                receive a message
                send the message to process C
        endloop

Process C:
        perform any needed initialization
        loop forever
                if (local message queue is empty)
                     then receive a message
```

```
                    else dequeue the first message from the local message queue
                if msg_type == “count_report”
                then if msg_data[1] is evenly divisible by 20
                       then send "Process C" to display using msg envelope
                              wait for display_ack
                              go_passive for 10 seconds
                       endif
                endif
                deallocate message envelope
                release_processor
        endloop
The line "go_passive for 10 seconds" may be further expanded as:
        request a 10 second delay with wakeup_code set to “wakeup10”
        loop
          receive a message          //block and let other processes execute
          if (message_type ==  “wakeup10”)
          then exit this loop
          else put message on the local message queue for later processing
        endloop
```

Process C maintains a local queue onto which it enqueues (in FIFO order) messages which arrive while it waits for the 10 sec wakeup message. It processes these messages later.

Select an appropriate set of priority levels for these three application level processes such that one would get the expected output from the system. During the demo procedure, you will be asked to vary the priorities of these processes and observe the resulting behaviour.

APPENDICES

A. System Console UART

The embedded microcomputer communicates with the system console using an advanced UART. The UART is connected to the microprocessor address, data and control buses on one side, and to the incoming (keyboard) and outgoing (character display) serial data link. The UART consists of two parts, the Receiver and the Transmitter.

The Receiver contains a 128-byte long buffer to store the characters received and a Rcount register which keeps the count of stored characters. The receiver accepts the characters coming in over the serial link and stores them in the buffer until a character with the code matching the pattern stored in an Rstop_pattern register is received, or until the buffer is filled. The Receiver then generates an interrupt towards the microprocessor. Subsequently, the Receiver ignores any characters arriving over the serial link until the microprocessor sets Rcount to zero, whereupon it resumes normal operation.

The Transmitter contains a 128-byte long buffer that stores the characters to be transmitted. It also contains the register Xcount with the count of characters to be transmitted. The microprocessor writes the characters to be sent out into the Transmit buffer. Then the micro writes the number of characters into Xcount. The nonzero content of Xcount will cause the Transmitter to start sending the characters from the buffer over the serial link, decrementing Xcount. When the last character is transmitted out (Xcount becomes zero), the Transmitter will generate an interrupt to the microprocessor.

B. Linux Embedding

The RTX is intended for use in embedded microcomputers. To support the RTX services, the embedded microcomputer has a UART for communication to/from the system console and a real-time clock module that generate interrupts at 10 msec intervals.

In the development and demonstration environment, in which the RTX will execute as a Linux process, these hardware facilities must be simulated. The basic idea is to use Linux signals instead of hardware interrupts and Linux shared memory in place of I/O device registers and buffers. The recommended approach is given below:

1. Clock tick interrupt: Use the Linux ALARM signal, enabled by the ualarm(...) Linux system call;
2. UART: UART functionality is simulated in software. Linux shared memory is used for UART buffers/registers, and Linux signals for UART-generated interrupts. A Linux process, developed by you, will act as the Receiver part of the UART (accept characters from the workstation keyboard, put them in a shared memory and generate a Linux signal towards RTX at the appropriate time). Another Linux process, written again by you, will act as the Transmitter part of the UART (take characters deposited in its shared memory by the RTX, cause them to be displayed in a workstation window and generate a Linux signal towards RTX).

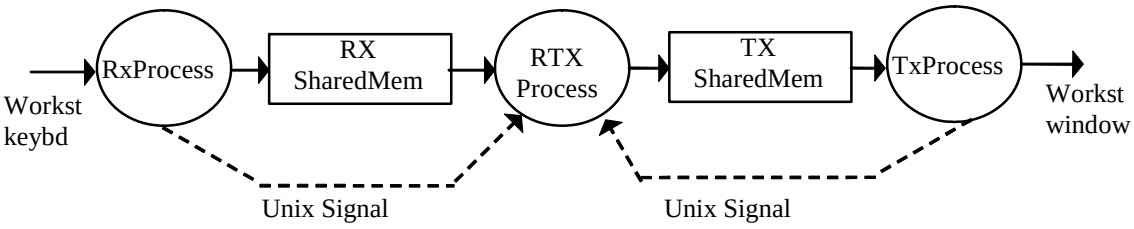

Figure 1: UART Simulation in Unix Environment

C. RTX class

```
class MsgEnv;

class RTX {

  private:
           // any required private methods not user visible
  public:   // only a subset of the public methods
   int  send_message( int dest_process_id, MsgEnv * msg_envelope );
   MsgEnv * receive_message( );
   MsgEnv * request_msg_env( );
   int  release_ msg_env ( MsgEnv * memory_block );
   int  release_processor(  );
   int  request_process_status( MsgEnv * memory_block );
   int  terminate(  );
   int  change_priority(int new_priority, int target_process_id);
   int  request_delay( int time_delay, int wakeup_code, MsgEnv * message_envelope );
   int  send_console_chars(MsgEnv * message_envelope );
   int  get_console_chars(MsgEnv * message_envelope );
   int  get_trace_buffers( MsgEnv * message_envelope);
};
```

User level processes will invoke only the public methods defined in the class RTX. Thus, the public methods in class RTX define the user API for your real-time kernel. The MsgEnv class will also have a set of public interface methods (your design) that can be called by user processes.