**RTX Part 3 Document**

**System.h**

**Macros:**

* **NULL\_PROC\_ID 0 –** The ID of the null process – set to 0.
* **KDC\_PROC\_ID 7 –** The ID of the keyboard decoder process – set to 7.
* **CRT\_PROC\_ID 8 –** The ID of the CRT process – set to 8.
* **UART\_IPROC\_ID 9 –** The ID of the UART iProcess – set to 9.
* **TIMER\_IPROC\_ID 10 –** The ID of the timer iProcess – set to 10.

**Structs:**

* **delayed\_send\_request**
  + **int exp –** Expiration time of the delayed send
  + **int process\_ID –** The ID of the process you’re sending to
  + **VOID \* envelope –** Pointer to the message
* **s\_char\_queue**
  + **struct s\_char\_queue\_item \* front –** Points to the head of the char item queue
  + **struct s\_char\_queue\_item \* back** – Points to the tail of the char item queue
  + **UINT8 num\_slots –** The number of items that can exist in the char item queue
* **s\_char\_queue\_item**
  + **struct s\_char\_queue\_item \* next** – Points to the next object in the char item queue
  + **char data –** Stores the actual char value of the object in queue
* **s\_message**
  + **int sender\_id –** The ID of the process to sending the message
  + **int dest\_id –** The ID of the recipient process
  + **int type –** Stores an integer representing the type of message being sent
  + **char \* msg\_text –** Pointer to the text to be sent by the message
* **s\_message\_queue**
  + **struct s\_message\_queue\_item \*front –** Points to the head of the message item queue
  + **struct s\_message\_queue\_item \* back –** Points to the tail of the message item queue.
  + **UINT8 num\_slots –** The number of messages that can be stored in the queue
* **struct s\_message\_queue\_item**
  + **struct s\_message\_queue\_item \* next –** Points to the next object in the message item queue
  + **struct s\_message \* data –** Pointer to the data of the message in the queue

**i\_proc.c**

**Global Variables:**

* **struct s\_pcb\_queue g\_iProc\_queue –** A separate PCB queue that only contains the iProcesses from the UART ISR and Timer ISR
* **struct s\_pcb\_queue\_item g\_iProc\_queue\_slots[10] –** An array storing the number of iProcesses that can be in the queue at a time.
* **struct s\_pcb \*g\_i\_interrupted\_process –** Pointer to the process control block that has been interrupted
* **struct delayed\_send\_request send\_reqs[10] –** An array storing the number of requests sent that are being delayed.

**Functions:**

**void \* uart()**

* **Purpose ­–** Interpret user input and handle it accordingly.
* **Pseudocode** – Initialize a char input buffer that holds the command to be sent to the keyboard decoder. This is a char array of size 100. An integer variable called inputBufferIndex is also initialized and set to 0. Then finally, a temporary pointer to a struct s\_message is initialized and set to 0.  
    
  The main body of the function is enclosed within an infinite while loop, in order to respond to interrupts. If rw is equal to 1 and the inputBufferIndex is less than zero, it means that we are reading from the UART and the input buffer is not full and it enters the if statement. Here, the uart function responds to specific inputs from the user.   
    
  If the user types ‘-‘, the current clock status is output to the debug terminal.   
    
  Else if the user types ‘\_’, it toggles the clock enable variable.   
    
  Else if the user types ‘+’, the value of the clock is displayed in seconds, minutes, and hours.   
    
  Else if the user types ‘=’, the OS checks for delayed messages. Using a for loop to iterate through the number of delayed slots, it first checks for a delayed message at that particular index. If a delayed message is found, the expiration time left is displayed, and then decremented by 5. It then checks if the expiration of the delay has hit 0 or below, in which case the message is ready to send to the appropriate slot. The message is sent by using the message\_push() function. If the receiving process is currently blocking on a message from the sender, unblock it and push it to the ready queue. Finally, it frees up the delayed send space by setting the envelope pointer to 0. One iteration of the for loop has now completed, and the next delayed message slot is checked.  
    
  Else if the user types ‘~’, the processes currently on the ready queue are displayed, along with their priorities.  
    
  Else if the user types ‘`’, the processes that are currently on the blocked memory queue are displayed along with their priorities**.**Else if the user types ‘!’, the processes that are currently blocking on receive are displayed along with their priorities.  
    
  Else if the user hits the tab key, an error is evoked and an rte is called**.**Else, add the typed character to the input buffer which will be sent to the keyboard decoder (kcd). Put the character in the output buffer by calling the buffer\_push() function. Increment the inputBufferIndex. If the user types enter to finish the command, enter the if statement and reset the input buffer to 0. Then, send the input buffer in a message to the keyboard decoder. The input buffer holds the user’s command. Then, put a newline character in the output queue to move to the next line. Exit the if statement. Transmit interrupts are then enabled so that the user’s input is echoed out.  
    
  If rw was 4 (0100) instead of 1 in the first place, the next character waiting in the output buffer is written. If the buffer is not empty, enable transmit interrupts so that the user’s input is echoed out.  
    
  Release\_processor() is called.

**void timer()**

* **Purpose –** Update the wall clock with correct values and increment it whenever called by the ISR.
* **Pseudocode –** Global variables that store values for hours, minutes, and seconds are initialized and set to 0. A variable named ticks\_since\_last\_run is initialized, which is meant to store “left over” milliseconds so that the clock can be updated while it runs in the background. A variable named temp\_counter is initialized and set to 0. This is to store the g\_clock\_counter value so that it can be instantly reset and not change during this process’ execution.  
    
  The main portion of this function exists within an infinite while loop. Temp\_counter is set to the value of g\_clock\_counter, and then g\_clock\_counter is reset to 0. Using a for loop to iterate through the number of delayed slots, any pending delay request message expiration counters are updated.  
    
  If wall clock is enabled, check if an update is necessary and perform all counter updates as necessary before sending the output message to the CRT for output. The value of ticks\_since\_last\_run is added to the temp\_counter to keep track of any time that has elapsed since the wall clock was last run. Then, the values of g\_seconds, g\_minutes, and g\_hours are updated as temp\_counter increments. The modulus function is used to find the correct string values that will be output the display. Then, a memory block is requested and a message is sent to the CRT to display the wall clock.  
    
  g\_timer\_is\_scheduled is set to 0. Release\_processor() is called.

**void crt()**

* **Purpose –** This function is for sending the text of messages sent to the CRT to the display. It parses through messages character by character and places them in the output buffer.
* **Pseudocode –** The main portion of this function exists within an infinite while loop. The message to be received is saved into a temporary variable by calling receive\_message with the sender’s ID as an argument. The type of the message is checked, and if its type equals 3 it enters the if branch. The function then starts pushing characters to the output buffer (up to 100 characters) by using a for loop. The for loop iterates through every character of the message, treating it as an array of characters. Each character is pushed to the output buffer, until a null character is found. When a null character is found, it breaks out of the loop. Transmit interrupts are then enabled to allow the message to be displayed. The memory block holding the message is then released at the end of the function.

**void c\_serial\_handler()**

* **Purpose –** This function acts as the ISR for the UART. It pushes the iProcess invoked by UART interrupts into the iProcess queue and pre-empts the current process if necessary.
* **Pseudocode –** At first, interrupts are disabled by moving 2700 into the status register. Then the function acknowledges the interrupt and determines if it is a read or write. If rw equals 1, then it is a read and enters the appropriate if branch. SERIAL1\_RD is sent to a local variable called charIn. The iProcess is then pushed to the proc table at the appropriate index. If the iProcess is not the current process and needs to be pre-empted, call release\_processor().  
    
  Else if rw equals 4, then it is a write. SERIAL1\_WD is set to the output character. The default value for this is null. Transmit interrupts are stopped. The iProcess is scheduled to see if more characters should be output. Then the function checks to see if the current process is the iProcess and if it is not, calls release\_processor().

**void c\_timer\_handler()**

* **Purpose –** This function is the ISR for the timer. It sets the timer\_is\_scheduled flag and pushes the iProcess of the timer to the appropriate index of the process table.
* **Pseudocode –** At first, interrupts are disabled by moving 2700 into the status register. Ticks\_since\_last\_run is incremented to track how many times the ISR has run since the last time the timer process has actually run. If the g\_timer\_is\_scheduled flag is not set, it is set to 1 and the iProcess is pushed to the process table. TIMER0\_TER is set to 2.