UART I-PROCESS

The UART I-Process handles two types of interrupts: the keyboard input interrupt and the transmission (output) interrupt.

Keyboard input characters are handled on a character-by-character basis. The input character is first compared against pre-defined hotkeys, then the corresponding hotkeys function is called should the input matches the pre-defined string. If none of the hotkey is matched, then this character is placed into two distinct message envelopes to be sent to KCD, and CRT for further processing.

Transmissions are carried out also once per character. For each character in the output string, a message envelope containing this character is expected from CRT. When this arrives, the UART I-Process prints it to the console. But before any transmission takes place, UART I-Process has to ensure that the transmission port is ready to accept data, also special handlings are required and implemented for certain characters(e.g. enter).

Uart\_i\_process

If (data is waiting for input){

Get charIn from port

Put charIn in message envelopes m1, and m2

send m1 to KCD

send m2 to CRT

}else if (output port is ready){

message = receive\_message(&sender\_id)

if (sender\_id == timer || sender\_id == CRT)

string = get message string from message

for (each character of the string)

while (output port is not ready)

stay in the loop

print next character

destroy message

}

Timer I Process and Delayed\_send

The timer I process forms the foundation for any time based primitives and processes. It keeps track of the current time and updates this value every 10 milliseconds. The timer I process also maintains the timeout queue used for delayed\_send.

The timeout queue is implemented with the expiry\_time as priority. This implementation enables easy update when an element needs to be removed when its expiry\_time is reached.

timer\_i\_proc

disable all other interruts

backup the current pcb

current\_time ++;//Increment the time counter

message = receive message,non-blocking

while (message != NULL){

delay\_time = \*(message+offset)

expiry\_time = delay\_time\*unit conversion factor+current time

put the message into the timeout queue, in ascending order of expiry\_time.

message = receive message, non-blocking

}

expire\_found= check if the first element in the timeout queue has expired

while( expire\_found){

remove it from timeout queue.

send its message

expire\_found = check the first one in the timeout queue(the one after the element just got removed), see if it timeouts

}

restore pcb.

The delayed\_send() sends message containing necessary information to timer\_interrupt\_i\_process, and then timer interrupt takes over the remaining tasks.

delayed\_send()

fill the message header with the sender id, receiver id, and delayed time.

send message to timer interrupt I process

Wall Clock

There are two ways to change the Wall Clock value: by Wall Clock’s own update or by user command. The wall clock receives messages and then act accordingly based on whether it is sent from user or itself.

If the message is from user, the clock will validate and parse the command, and finally set or unset its time. If the clock is set, it will delay\_send\_message once every second to update values.

If the message is sent from itself, this means one second has elapsed and the wall clock should update its value, readjust if any time overflow happens, and call CRT to display the new time.

wall\_clock()

loop forever{

Incoming\_message = blocking\_receive\_message(&sender\_id)

If (sender\_id == kcd\_id)

If (the first character from the incoming\_message == ‘S’) {

check if the content in incoming\_message is valid

update hour, minute, second based on the information from incoming\_message

clock\_on = true

new\_msg = request\_memory\_block

fill the hour, minute, second to be displayed in new\_msg

send new\_msg to CRT for display

my\_msg = request\_memory\_block

delayed\_send my\_msg to wall\_clock itself with one second

}

else if (the first character from the message == ‘W’) {

clock\_on = false

}

Destroy incoming\_msg

}

Else if (sender\_id == wall\_clock\_id) {

If (clock\_on == true) {

Delayed\_send incoming\_message to itself again with one second

Increment second

If overflow happens:

Adjust minute/second

New\_msg = request\_memory\_block

Fill new\_msg with hour, minute and second for display

Send new\_msg to CRT for display

}

Else {

Destroy incoming\_msg

}

}

}

KCD

Wall Clock and Set Priority process are able to use KCD to register commands. When a command is registered, a message containing process id and the exact command name is sent to KCD. KCD maintains an internal command table of sufficient size to keep track of the command names and their corresponding process IDs. KCD receives two types of messages, the command message or the registration message, which are designed to have different message types. When any message is receives, the KCD either registers them as a new command in its internal command table, or looks up the command table and carry out the command (i.e. sending messages with relevant contents to the matching processes) based on the message types.

KCD

While (true)

Message = Receive message(&sender\_id)

If(message type is command)

Extract character from message to internal buffer

Check if the character is carriage return

If yes

Check internal buff to determine if it contains valid command

Find the corresponding process according to content in internal buffer

Pack the rest of the buffer in a message block

send this block to the corresponding process

Send message to the corresponding processes

If no

Clear buffer

restart

else if( message type is registration )

Update the internal command table with message content

else

Error states

Message Envelope Structure

System assigned memory blocks are used to accommodate message envelopes. The first 64 Bytes of the envelope contains message header information including the message type, sender ID and receiver ID. The rest of memory block contains the message content as following: the 64-67 Bytes contains an integer indicating the length of the content, n. and the proceeding n Bytes contains the contents.

Message Type

Sender ID

Receiver ID

Delay Time

Content Length

….rest of header

Message content…..

Header

Body