ECE 350 Lab 3 Report

Group number: 1
Group members:
Fuhai Gao(f27gao)
Yuan Xie (y226xie)
Bailan Yuan(b8yuan)
Zheng Pan(z69pan)

Data Structure

Circular queue (ring buffer)

We used the circular queue (ring buffer) to implement the mailbox of tasks in order to get the better performance. The circular queue will act as a one-dimensional array, but the front of array “connects” to the head of the array.

We have 4 important variables: head, tail, start_of_mbx, end_of_mbx pointers. Head will always point to the first message in the circular queue, and tail will always point to the end of the last message (position for the next incoming message). start_of_mbx will always point to the front of mailbox, and the end_of_box will always point to the start_of_mbx + mailbox’s size, which is the end of mailbox.

The head and tail will both be NULL until we have inserted something into the circular queue.

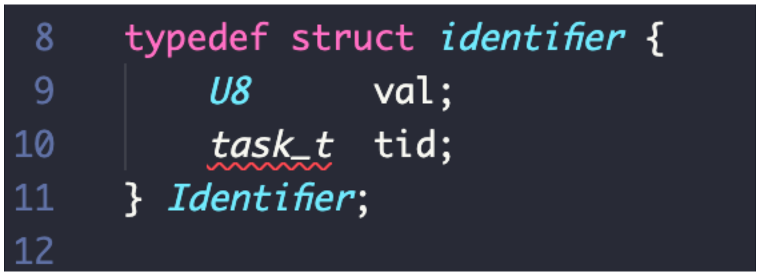
Before an insertion happen, we will check head and tail current position:

1. If they both equal to NULL, which implies, the queue is empty. At this case, we can simply insert the message to the front of mail box, and update head = start_of_mbx, tail = head + sizeof(first_message).

2. If tail == end_of_mbx or nearly equal, we will try to see if the front has any space to fit in the new message (since it is circular). If the front cannot even hold a new message, it means the queue is full.
3. If the position of tail is the same as the head or its position is near to the front of the head, at this case, it also means that our queue is either full or nearly full. So, we cannot insert any message
4. Otherwise, we can simply append the new item to the tail of the queue and will update tail towards the end_of_mbx by the size of the message.

Identifier

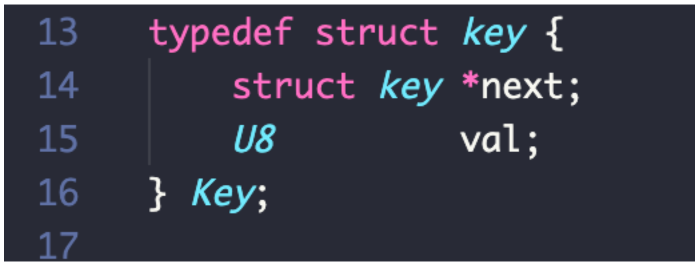
We also defined a data structure called Identifier used to store the registration of a task with a specific identifier. It contains a val which stores identifier (i.e., 'a', 'b'), and the tid stores the corresponding task_id. Since all the valid identifers must be in this range: [aA – zZ] && [0 – 9], there are 62 distinct identifiers in total. So, we use a identifiers array which has a size of 62 to store identifier registration information.



```
typedef struct identifier {
    U8      val;
    task_t  tid;
} Identifier;
```

Key

We defined a data structure called Key to store user's input from keyboard. A key will be a node of our linked list. Once, a `return` key has been detected, this queue will be sent to the corresponding task id.

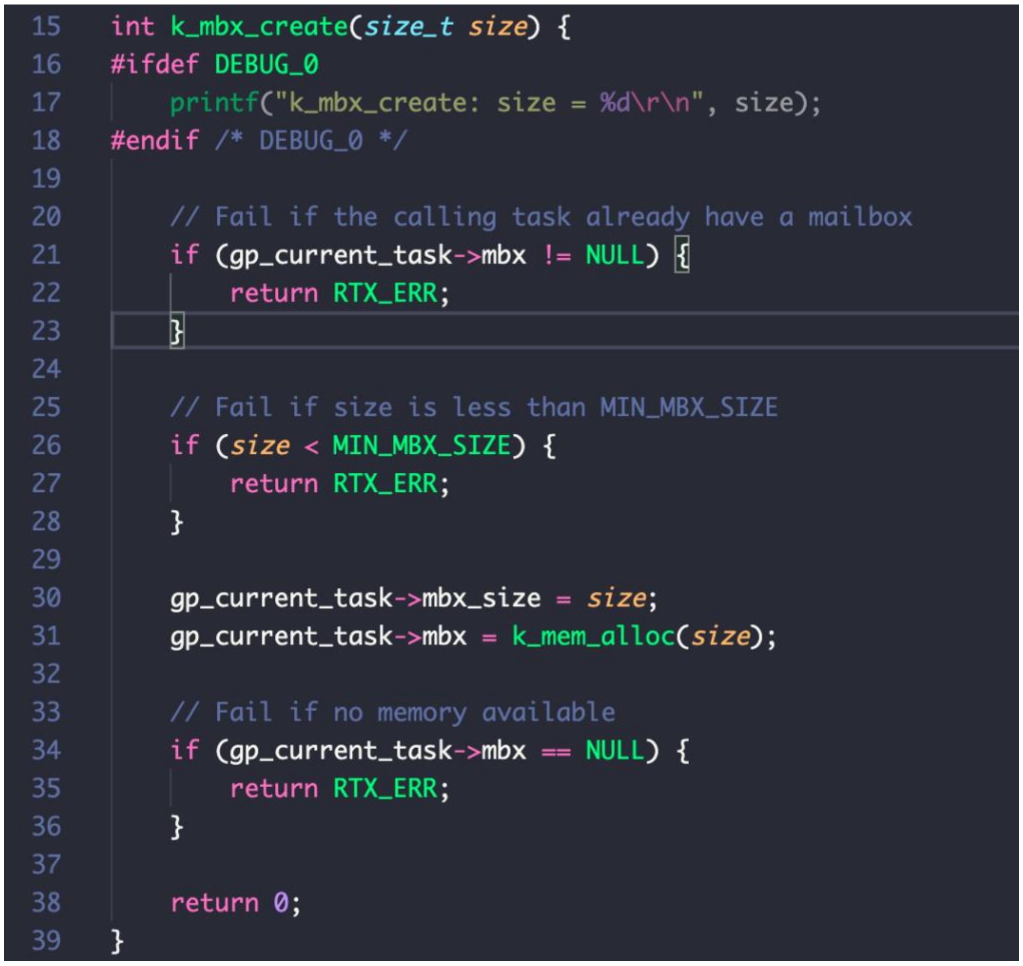


```
typedef struct key {
    struct key *next;
    U8          val;
} Key;
```

k_task fucntions

k_mbx_create

Create a mailbox for calling task based on the parameter size. If the size of mailbox is less than the MIN_MBX_SIZE, or current calling task already has a mailbox, or if there is not enough memory available for allocating a mailbox, the function will return -1. Otherwise, it returns 0.



```
int k_mbx_create(size_t size) {
#ifdef DEBUG_0
    printf("k_mbx_create: size = %d\r\n", size);
#endif /* DEBUG_0 */

    // Fail if the calling task already have a mailbox
    if (gp_current_task->mbx != NULL) {
        return RTX_ERR;
    }

    // Fail if size is less than MIN_MBX_SIZE
    if (size < MIN_MBX_SIZE) {
        return RTX_ERR;
    }

    gp_current_task->mbx_size = size;
    gp_current_task->mbx = k_mem_alloc(size);

    // Fail if no memory available
    if (gp_current_task->mbx == NULL) {
        return RTX_ERR;
    }

    return 0;
}
```

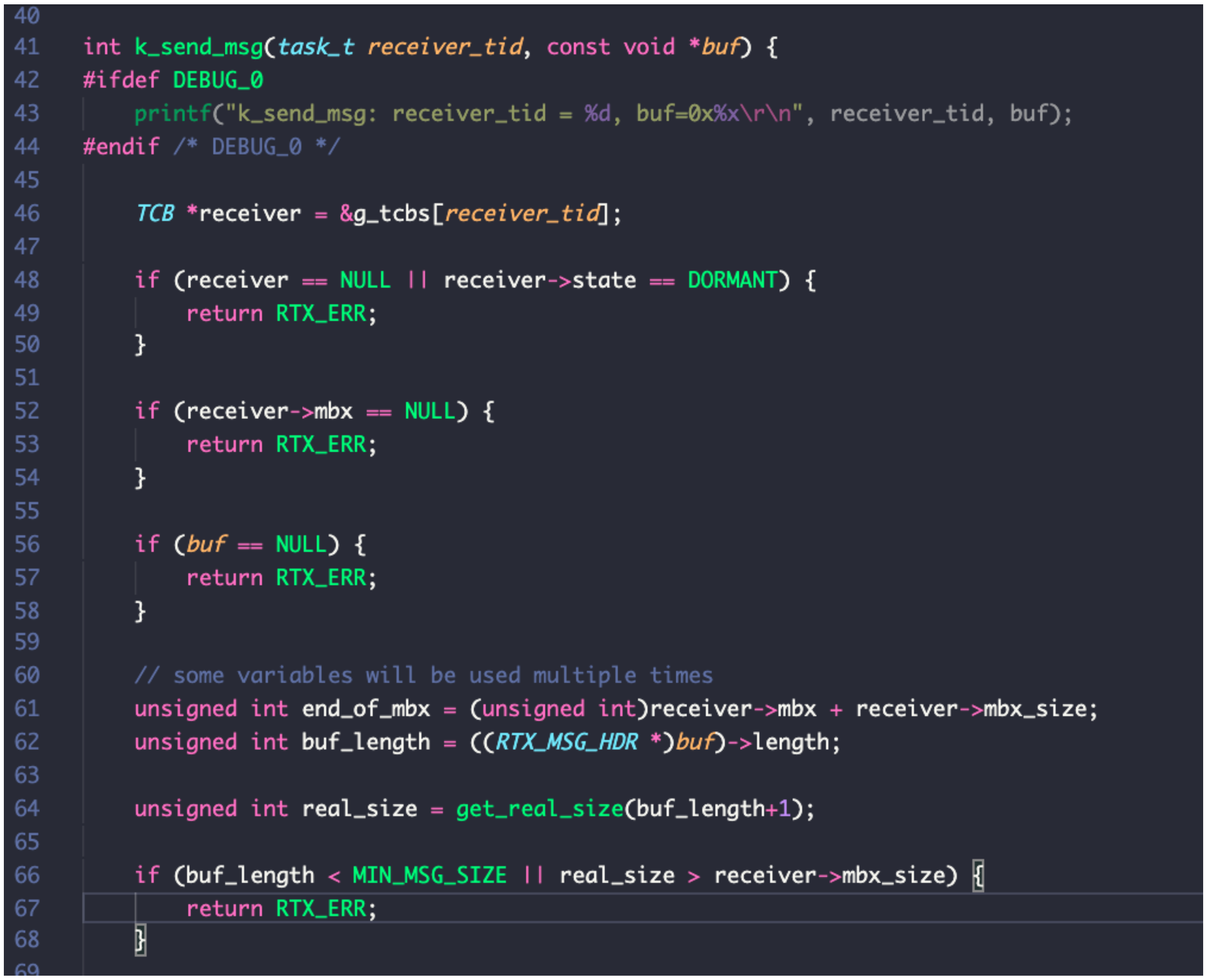
k_send_msg

This function delivers the message to the mailbox of the task identified by the receiver_tid. We have some error checks at the very beginning of the function:

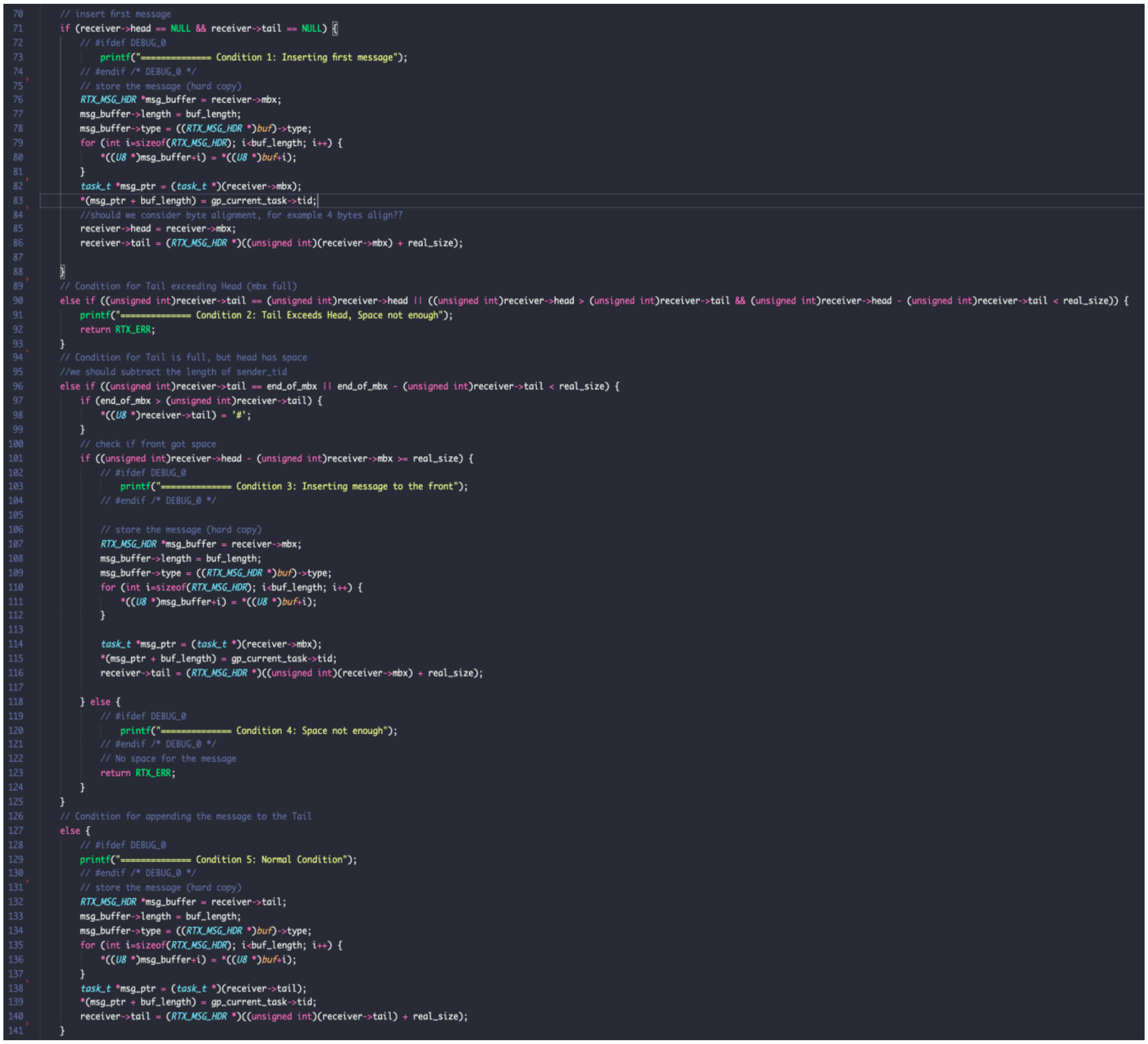
1. If the receiver task's does not exist or it becomes dormant, it returns -1.
2. If the receiver does not have a mailbox, it returns -1
3. If the message is not a valid pointer, or if the message size is less than minimum required message size, or the mailbox does not have enough space to hold a new message, it returns -1

There are basically 4 scenarios for inserting a message into the mailbox:

1. If the mailbox is initially at empty state, then we want to insert our first message. It was been implemented by setting the head pointer points to the position of the start_of_mbx, and the tail pointer points to the end of the first message.
2. We check the position of the tail and compare it with the head. If the tail curtly points the same position as the head, or the tail is at front of the head, it means that our queue is either full or nearly full. In other words, it cannot fit in any new message, function returns -1
3. If the position of tail points the same address as the end_of_mbx, or the end_of_mbx minus the position of tail is less than message size, we will check if the front has any available space (since it is a circular queue). If the front cannot even store a new message, then it also means that our queue is full, so function returns -1. If the front can fit a new message, then we set the receiver tail points to the position of the start_of_mbx + size_of_msg
4. The queue is not full, which means we can simply insert the new message into the mailbox. We update the position of the tail pointer toward the end of the mailbox by the size of the message.



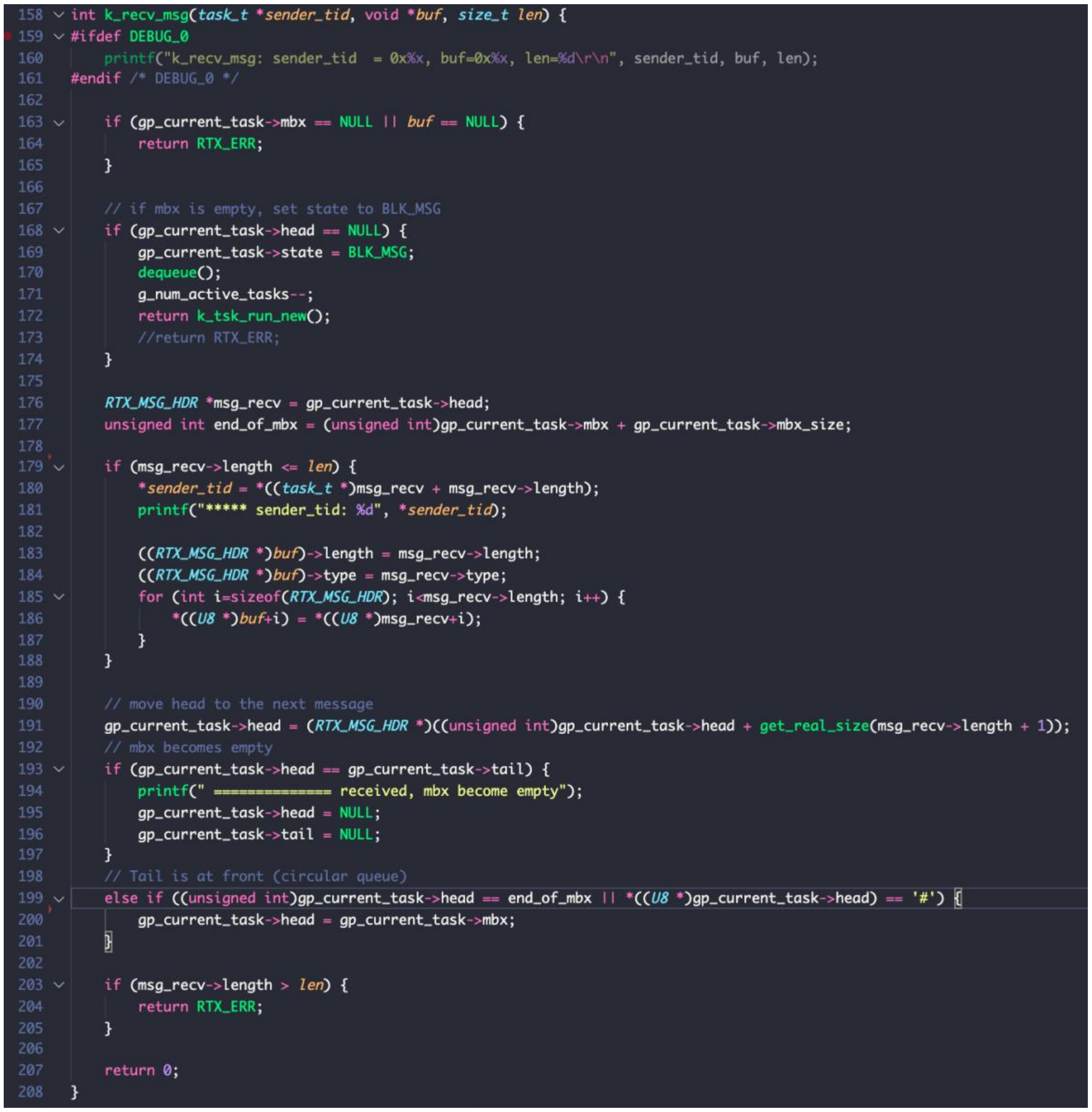
```c

int k_send_msg(task_t receiver_tid, const void *buf) {
#ifdef DEBUG_0
    printf("k_send_msg: receiver_tid = %d, buf=0x%x\r\n", receiver_tid, buf);
#endif /* DEBUG_0 */

    TCB *receiver = &g_tcbs[receiver_tid];

    if (receiver == NULL || receiver->state == DORMANT) {
        return RTX_ERR;
    }

    if (receiver->mbx == NULL) {
        return RTX_ERR;
    }

    if (buf == NULL) {
        return RTX_ERR;
    }

    // some variables will be used multiple times
    unsigned int end_of_mbx = (unsigned int)receiver->mbx + receiver->mbx_size;
    unsigned int buf_length = ((RTX_MSG_HDR *)buf)->length;

    unsigned int real_size = get_real_size(buf_length+1);

    if (buf_length < MIN_MSG_SIZE || real_size > receiver->mbx_size) {
        return RTX_ERR;
    }

```



```c
    // insert first message
    if (receiver->head == NULL && receiver->tail == NULL) {
        // #ifdef DEBUG_0
            printf("============= Condition 1: Inserting first message");
        // #endif /* DEBUG_0 */
        // store the message (hard copy)
        RTX_MSG_HDR *msg_buffer = receiver->mbx;
        msg_buffer->length = buf_length;
        msg_buffer->type = ((RTX_MSG_HDR *)buf)->type;
        for (int i=sizeof(RTX_MSG_HDR); i<buf_length; i++) {
            *((U8 *)msg_buffer+i) = *((U8 *)buf+i);
        }
        task_t *msg_ptr = (task_t *)(receiver->mbx);
        *(msg_ptr + buf_length) = gp_current_task->tid;
        //should we consider byte alignment, for example 4 bytes align??
        receiver->head = receiver->mbx;
        receiver->tail = (RTX_MSG_HDR *)((unsigned int)(receiver->mbx) + real_size);

    }
    // Condition for Tail exceeding Head (mbx full)
    else if ((unsigned int)receiver->tail == (unsigned int)receiver->head || ((unsigned int)receiver->head > (unsigned int)receiver->tail && (unsigned int)receiver->head - (unsigned int)receiver->tail < real_size)) {
        printf("============= Condition 2: Tail Exceeds Head, Space not enough");
        return RTX_ERR;
    }
    // Condition for Tail is full, but head has space
    //we should subtract the length of sender_tid
    else if ((unsigned int)receiver->tail == end_of_mbx || end_of_mbx - (unsigned int)receiver->tail < real_size) {
        if (end_of_mbx > (unsigned int)receiver->tail) {
            *((U8 *)receiver->tail) = '#';
        }
        // check if front got space
        if ((unsigned int)receiver->head - (unsigned int)receiver->mbx >= real_size) {
            // #ifdef DEBUG_0
                printf("============= Condition 3: Inserting message to the front");
            // #endif /* DEBUG_0 */

            // store the message (hard copy)
            RTX_MSG_HDR *msg_buffer = receiver->mbx;
            msg_buffer->length = buf_length;
            msg_buffer->type = ((RTX_MSG_HDR *)buf)->type;
            for (int i=sizeof(RTX_MSG_HDR); i<buf_length; i++) {
                *((U8 *)msg_buffer+i) = *((U8 *)buf+i);
            }

            task_t *msg_ptr = (task_t *)(receiver->mbx);
            *(msg_ptr + buf_length) = gp_current_task->tid;
            receiver->tail = (RTX_MSG_HDR *)((unsigned int)(receiver->mbx) + real_size);

        } else {
            // #ifdef DEBUG_0
                printf("============= Condition 4: Space not enough");
            // #endif /* DEBUG_0 */
            // No space for the message
            return RTX_ERR;
        }
    }
    // Condition for appending the message to the Tail
    else {
        // #ifdef DEBUG_0
        printf("============= Condition 5: Normal Condition");
        // #endif /* DEBUG_0 */
        // store the message (hard copy)
        RTX_MSG_HDR *msg_buffer = receiver->tail;
        msg_buffer->length = buf_length;
        msg_buffer->type = ((RTX_MSG_HDR *)buf)->type;
        for (int i=sizeof(RTX_MSG_HDR); i<buf_length; i++) {
            *((U8 *)msg_buffer+i) = *((U8 *)buf+i);
        }
        task_t *msg_ptr = (task_t *)(receiver->tail);
        *(msg_ptr + buf_length) = gp_current_task->tid;
        receiver->tail = (RTX_MSG_HDR *)((unsigned int)(receiver->tail) + real_size);
    }
```

k_recv_msg

This function basically takes message out of mailbox if there is any. If the mailbox is empty, then we will set the current task to be blocked and run the scheduler to run a new task. It will always try to grab the first message from the mailbox. The function first checks if the buffer has enough memory to hold the first message, if not, the message will be discarded, and will update the head pointer to the next message in the queue. It will also detect that after removing messages out of the mailbox, if the mailbox becomes empty or not. If the mailbox becomes empty, we will set the head and tail pointers to be NULL. Also, if after removing the messages, the tail is at the front of the head, we will set the head points to the position of the start_of_mbx.



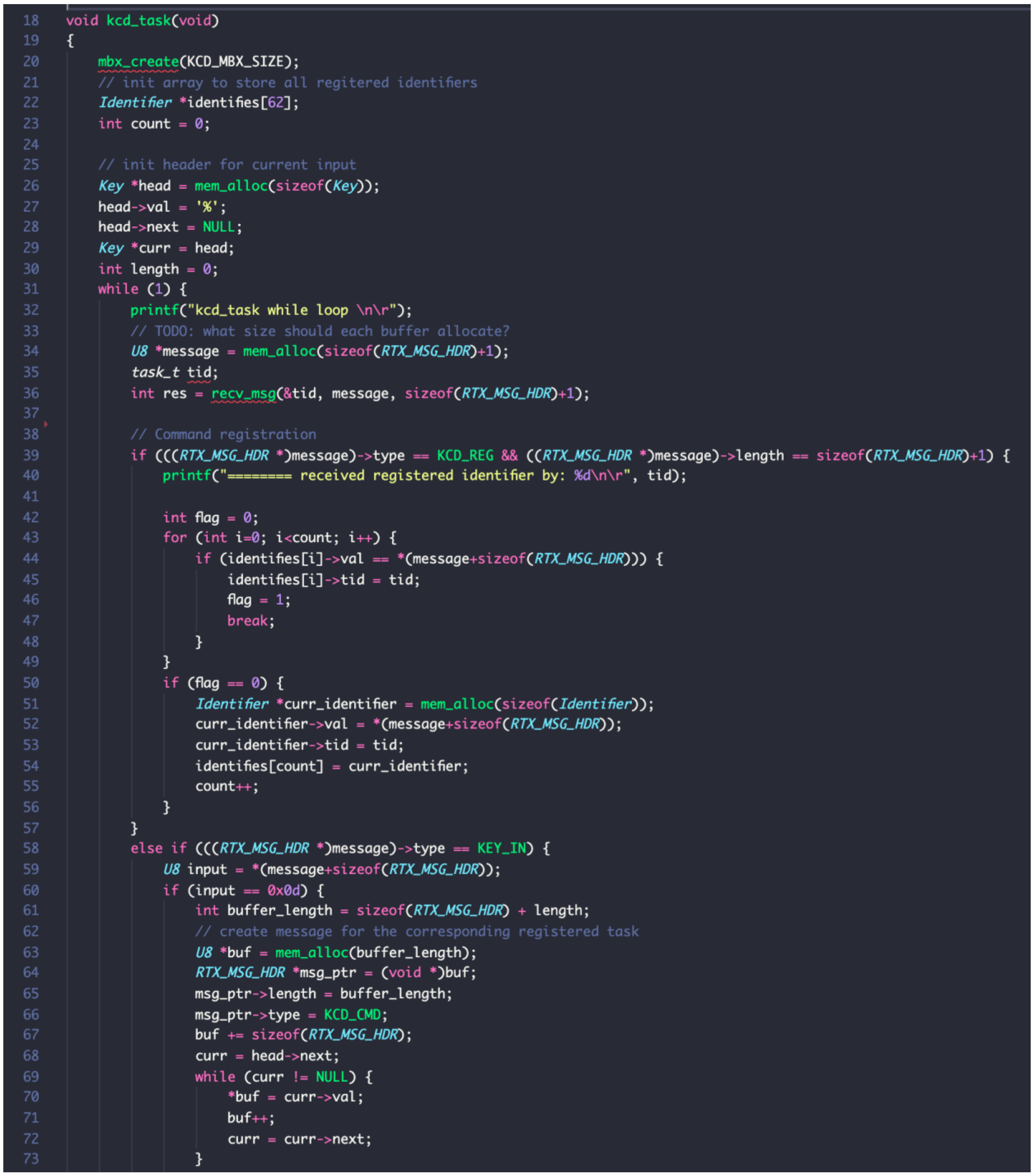
```c
int k_recv_msg(task_t *sender_tid, void *buf, size_t len) {
#ifdef DEBUG_0
    printf("k_recv_msg: sender_tid  = 0x%x, buf=0x%x, len=%d\r\n", sender_tid, buf, len);
#endif /* DEBUG_0 */

    if (gp_current_task->mbx == NULL || buf == NULL) {
        return RTX_ERR;
    }

    // if mbx is empty, set state to BLK_MSG
    if (gp_current_task->head == NULL) {
        gp_current_task->state = BLK_MSG;
        dequeue();
        g_num_active_tasks--;
        return k_tsk_run_new();
        //return RTX_ERR;
    }

    RTX_MSG_HDR *msg_recv = gp_current_task->head;
    unsigned int end_of_mbx = (unsigned int)gp_current_task->mbx + gp_current_task->mbx_size;

    if (msg_recv->length <= len) {
        *sender_tid = *((task_t *)msg_recv + msg_recv->length);
        printf("***** sender_tid: %d", *sender_tid);

        ((RTX_MSG_HDR *)buf)->length = msg_recv->length;
        ((RTX_MSG_HDR *)buf)->type = msg_recv->type;
        for (int i=sizeof(RTX_MSG_HDR); i<msg_recv->length; i++) {
            *((U8 *)buf+i) = *((U8 *)msg_recv+i);
        }
    }

    // move head to the next message
    gp_current_task->head = (RTX_MSG_HDR *)((unsigned int)gp_current_task->head + get_real_size(msg_recv->length + 1));
    // mbx becomes empty
    if (gp_current_task->head == gp_current_task->tail) {
        printf(" ============== received, mbx become empty");
        gp_current_task->head = NULL;
        gp_current_task->tail = NULL;
    }
    // Tail is at front (circular queue)
    else if ((unsigned int)gp_current_task->head == end_of_mbx || *((U8 *)gp_current_task->head) == '#') {
        gp_current_task->head = gp_current_task->mbx;
    }

    if (msg_recv->length > len) {
        return RTX_ERR;
    }

    return 0;
}
```

KCD_task

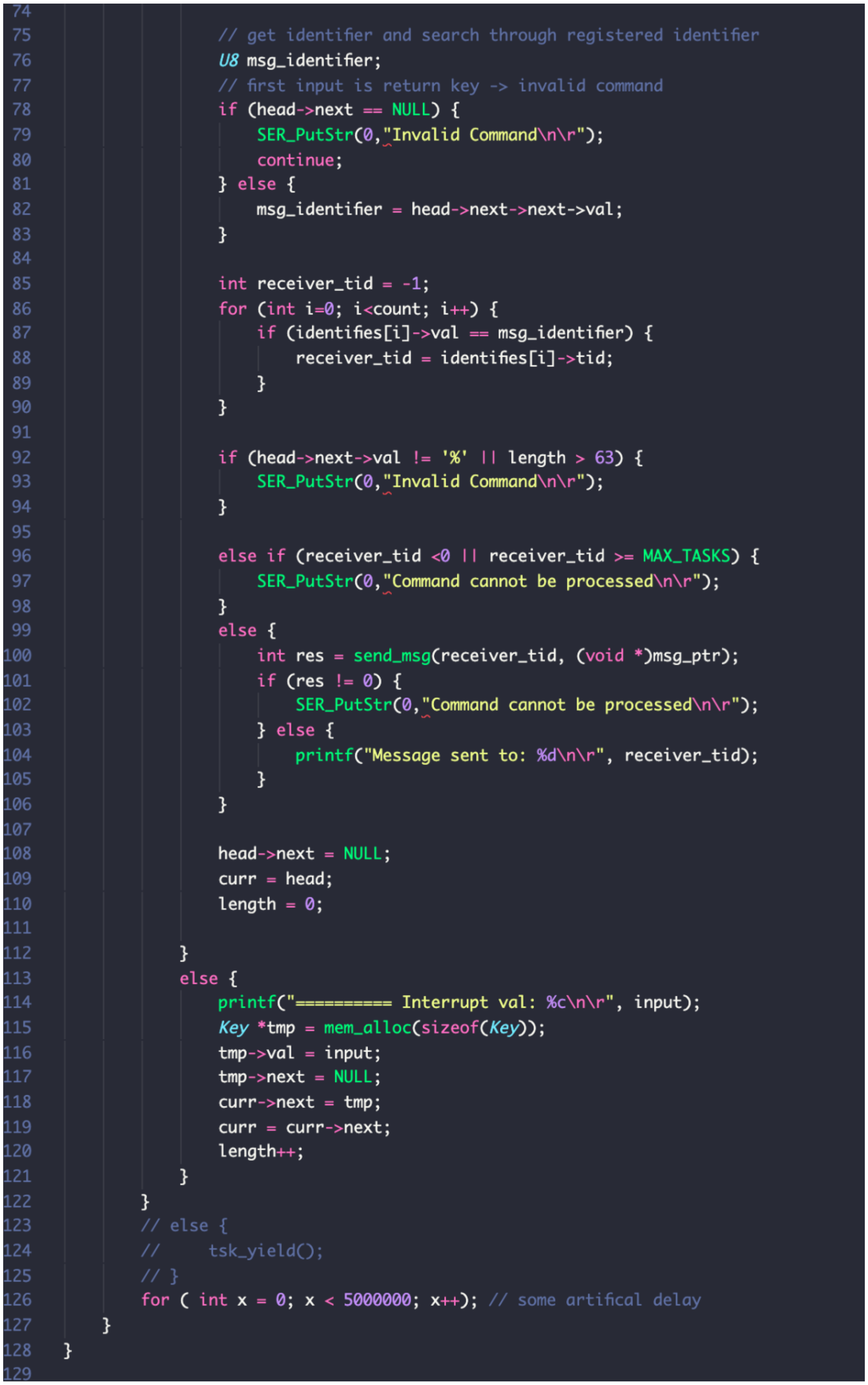
This function basically runs an infinite loop, it keeps calling recv_msg function to receive messages from its mailbox, and it only responds two types of messages: command registration and terminal keyboard input.

It will initiate a mailbox at the very beginning. Inside the infinite loop, we will first grab the current message's type.

1. If the type of the message is KCD_REG (command registration) and the length of the message is one character. At this case, KCD will map the message to the corresponding task's tid. In other words, this message will be used as the identifier for the task. We also allow the task to re-registered with a different identifier. However, we will only forward a command to the mailbox of the latest task that has registered the command's identifier
2. If the type of the message is terminal keyboard input (KEY_IN). Then we will use a linked list to store the message, until a return key has been detected. If a return key has been detected, then the message, we will search for the identifier and try to send the message.
3. If a message failed to be sent or the command's identifier is not registered, then we will output "command cannot be processed"
4. If the key_in type message not started with a % or the length of the command (include % and return key) greater than 64 Bytes, function will output "invalid command"



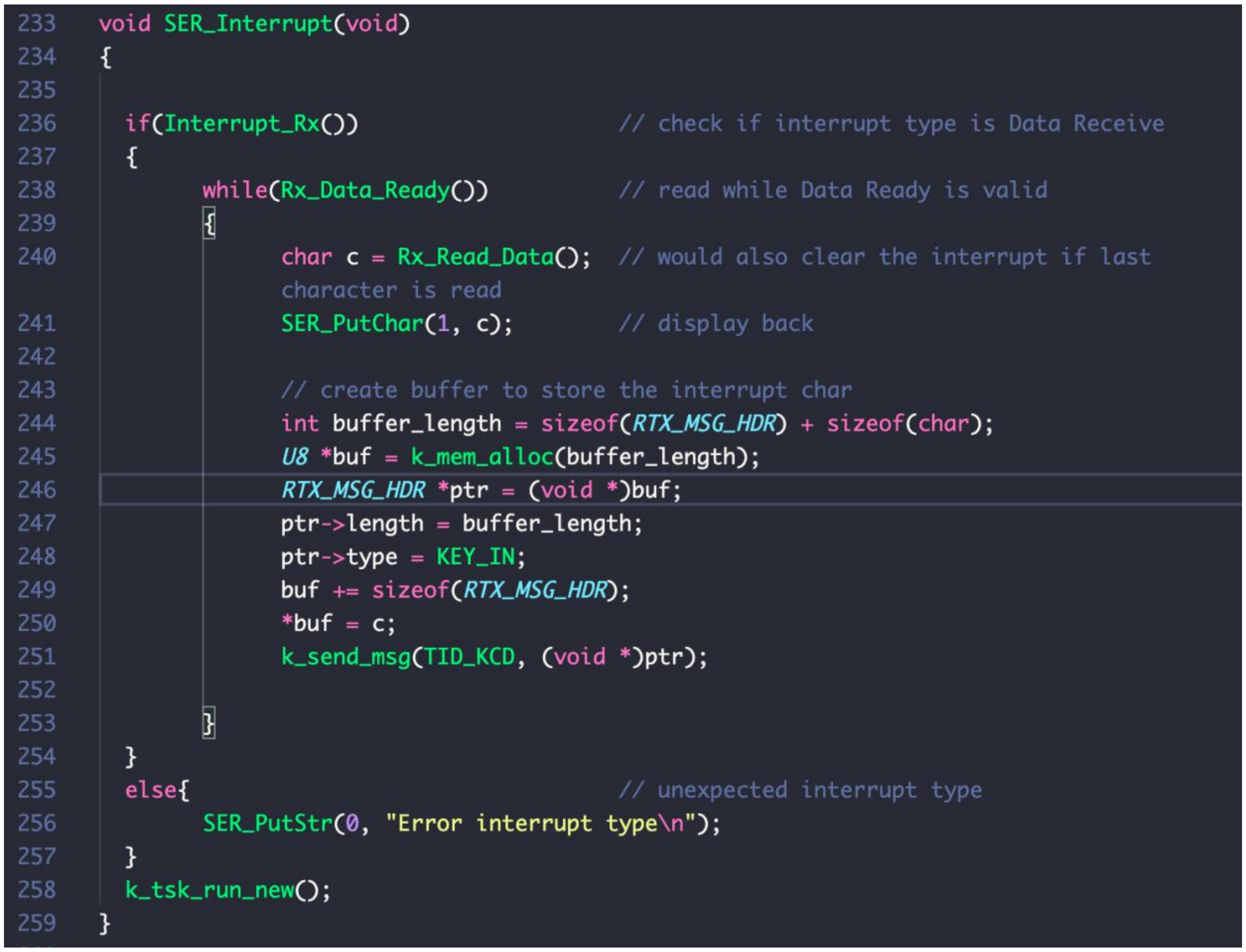
```
void kcd_task(void)
{
    mbx_create(KCD_MBX_SIZE);
    // init array to store all regitered identifiers
    Identifier *identifies[62];
    int count = 0;

    // init header for current input
    Key *head = mem_alloc(sizeof(Key));
    head->val = '%';
    head->next = NULL;
    Key *curr = head;
    int length = 0;
    while (1) {
        printf("kcd_task while loop \n\r");
        // TODO: what size should each buffer allocate?
        U8 *message = mem_alloc(sizeof(RTX_MSG_HDR)+1);
        task_t tid;
        int res = recv_msg(&tid, message, sizeof(RTX_MSG_HDR)+1);

        // Command registration
        if (((RTX_MSG_HDR *)message)->type == KCD_REG && ((RTX_MSG_HDR *)message)->length == sizeof(RTX_MSG_HDR)+1) {
            printf("======== received registered identifier by: %d\n\r", tid);

            int flag = 0;
            for (int i=0; i<count; i++) {
                if (identifies[i]->val == *(message+sizeof(RTX_MSG_HDR))) {
                    identifies[i]->tid = tid;
                    flag = 1;
                    break;
                }
            }
            if (flag == 0) {
                Identifier *curr_identifier = mem_alloc(sizeof(Identifier));
                curr_identifier->val = *(message+sizeof(RTX_MSG_HDR));
                curr_identifier->tid = tid;
                identifies[count] = curr_identifier;
                count++;
            }
        }
        else if (((RTX_MSG_HDR *)message)->type == KEY_IN) {
            U8 input = *(message+sizeof(RTX_MSG_HDR));
            if (input == 0x0d) {
                int buffer_length = sizeof(RTX_MSG_HDR) + length;
                // create message for the corresponding registered task
                U8 *buf = mem_alloc(buffer_length);
                RTX_MSG_HDR *msg_ptr = (void *)buf;
                msg_ptr->length = buffer_length;
                msg_ptr->type = KCD_CMD;
                buf += sizeof(RTX_MSG_HDR);
                curr = head->next;
                while (curr != NULL) {
                    *buf = curr->val;
                    buf++;
                    curr = curr->next;
                }
```



```c

                    // get identifier and search through registered identifier
                    U8 msg_identifier;
                    // first input is return key -> invalid command
                    if (head->next == NULL) {
                        SER_PutStr(0,"Invalid Command\n\r");
                        continue;
                    } else {
                        msg_identifier = head->next->next->val;
                    }

                    int receiver_tid = -1;
                    for (int i=0; i<count; i++) {
                        if (identifies[i]->val == msg_identifier) {
                            receiver_tid = identifies[i]->tid;
                        }
                    }

                    if (head->next->val != '%' || length > 63) {
                        SER_PutStr(0,"Invalid Command\n\r");
                    }

                    else if (receiver_tid <0 || receiver_tid >= MAX_TASKS) {
                        SER_PutStr(0,"Command cannot be processed\n\r");
                    }
                    else {
                        int res = send_msg(receiver_tid, (void *)msg_ptr);
                        if (res != 0) {
                            SER_PutStr(0,"Command cannot be processed\n\r");
                        } else {
                            printf("Message sent to: %d\n\r", receiver_tid);
                        }
                    }

                    head->next = NULL;
                    curr = head;
                    length = 0;

                }
                else {
                    printf("========== Interrupt val: %c\n\r", input);
                    Key *tmp = mem_alloc(sizeof(Key));
                    tmp->val = input;
                    tmp->next = NULL;
                    curr->next = tmp;
                    curr = curr->next;
                    length++;
                }
            }
            // else {
            //     tsk_yield();
            // }
            for ( int x = 0; x < 5000000; x++); // some artifical delay
        }
    }

```

SER_Interrupt

This function basically receives a key from the keyboard, and it stores the key into the buffer, then sends it to the KCD task in a message, with a message type KEY_IN



```
void SER_Interrupt(void)
{

  if(Interrupt_Rx())                    // check if interrupt type is Data Receive
  {
        while(Rx_Data_Ready())          // read while Data Ready is valid
        {
              char c = Rx_Read_Data();  // would also clear the interrupt if last
              character is read
              SER_PutChar(1, c);        // display back

              // create buffer to store the interrupt char
              int buffer_length = sizeof(RTX_MSG_HDR) + sizeof(char);
              U8 *buf = k_mem_alloc(buffer_length);
              RTX_MSG_HDR *ptr = (void *)buf;
              ptr->length = buffer_length;
              ptr->type = KEY_IN;
              buf += sizeof(RTX_MSG_HDR);
              *buf = c;
              k_send_msg(TID_KCD, (void *)ptr);

        }
  }
  else{                                 // unexpected interrupt type
        SER_PutStr(0, "Error interrupt type\n");
  }
  k_tsk_run_new();
}
```

<u>Test Cases:</u>

Test case 1: In this test case, we have verified our implementation of circular queue by continually sending and receiving messages to test the cyclic behavior.

Ex.

Assume mbx can only hold 5 messages.

Send ‘a’ (insert 1st) -> Send ‘b’ (append to tail) -> Send ‘c’ (append to tail) -> Send ‘d’ (append to tail) -> Send ‘e’ (append to tail) -> Send ‘f’ (fail due to full) -> Receive (remove ‘a’) -> Send ‘g’ (update tail to front) -> Receive (remove ‘b’) -> Send ‘h’ (append to tail) -> Send ‘I’ (fail, due to tail exceeds head, full)

Test case 2: In this test case, we have tested the end-to-end of creating a mailbox, and sending and receiving messages by:

1. Create task1, and init mailbox, then yield to task2
2. Task2 send message to task1, then yield to task1
3. Task1 receive the message and check the data
4. Task1 successfully received the message with correct data

Test case 3: In this test case, we have tested the command registration and forwarding with the KCD task, as well as the command registration replacement with KCD task.