Real Time Micro Kernel
” RTMK”

Computer Engineer

Computer System Engineering II

Group 16

Rahaf Darouich
Sara Alterkawi

Contents

Abstract

To be defined, The RTMK project is a real-time software kernel whose main function is the masterful management of the collateral operations and resources that mainly consists of lists, such as the ReadyList, the WaitingList, and the TimerList.
Of course, every one of these lists is allotted a certain function. For example, the ReadyList consists of a list of operations prepared to run when needed. The WaitingList consists of operations that look for finding some new resources. The TimerList consists of operations that are set to calculate the specific required while before the run of these lists. Moreover, RTMK is a kind of software that coincidentally deals with various kinds of operations that arise the efficiency of system working, and that also give it a way to do many tasks at the same time. Therefore, the system in RTMK works to guarantee the right order of the operations' running and the ideal usage of the resources.
In a nutshell, the RTMK project is a real-time software kernel that can guarantee the ideal working of the system functions and the parallel operations of the tasks, and it also aims to masterfully manage collateral operation and resources by benefiting from the ReadyList, the WaitingList and the TimerList.

1. Introduction

This project begins with configuring the tasks management and their working processes as a first step in the kernel's preparation set-up. In addition, this project also uses a linked list formula, the ReadyList, that shows the arrangement of tasks according to deadlines. Then it comes the mailboxes which offer a way to the system to communicate with the tasks to enable the blocking of the mailboxes while waiting for a reply or not. The ReadyList is the final section that enables the tasks to sleep for a specified period of time in the TimerList before getting back to the ReadyList.

The goal of the Real-time Micro-Kernel project is to be explained in detail in this document, along with its functionality, specifications, and implemented concepts. The documentation is divided into four sections.

In the first section, the overall scope of the project is discussed, along with the necessary steps for proper implementation of the developed kernel.

The second section provides specific explanations of the implemented code. Each assignment required for the project will be explained to give a comprehensive understanding of the entire code. Flow Chart Diagrams are also available in the report to aid in understanding. The assignments were completed in the following order: Task Administration, Inter-Process Communication, and Timing Functions. The third section contains a description of the libraries used in the project. The fourth and final section describes the testing functions, the reasons for their implementation, and the different types of tests used for the project.

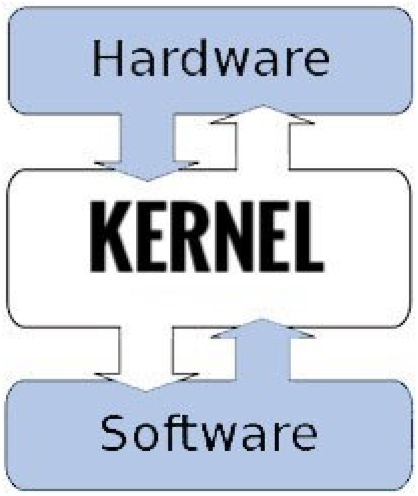

Figure 1 Kernel connects software to the hardware of a computer.

2. RTMK

2.1. Operation

The linked lists, which gives information about the operation process of the kernel, are what the RMTK begins its tasks with. As for the ReadyList function, which contains pointers to the Head List object and the Tail List object, it guides the kernel for running the suitable task, and informs the user about the following ordered task.

In case of being in operation, the instant (the shortest) task that operates the kernel is the Head List object. The Tail pointer works to indicate to the list object that includes the idle task. The idle task is the final task that works to carry out while {1} once and for all after the execution of all the other tasks, so this mock task leads to the closure of the kernel stall unit. Every list struct includes pointers for the Previous and Next list objects included in the list. They also include the possible running task.

The task contains struct Thread Control Block which in its turn works to give information about the stack. TCB has a stack pointer that indicates the running initiation of the needed task. Concerning the idle task, it includes a pointer(*PC) that indicates to the place of the stack in which while {1} can be found. Furthermore, other feedback about the stack can be found in the idle task. For example, it contains information about the stack fragmentation magnitude as well as providing the kernel with registers R4 to R11 if it needs to switch context.

Moreover, TCB includes the said task deadline that sets the suitable placement of the object list in the ReadyList. However, the ReadyList is arranged according to the duration of the deadline because it depends on the priority of executing the shortest task, i.e. the shortest deadline is located at the beginning of the list to be primarily implemented.

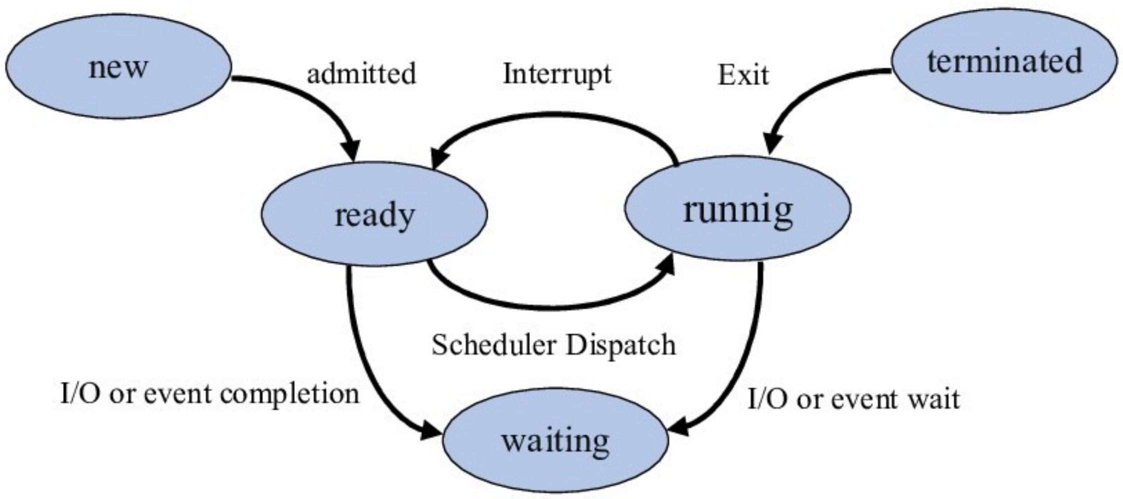

Figure 2 Task States and Scheduling

2.2. *Init Kernel*

By configuring the List Data Structure and forming the idle Task along with its highest potential deadline, the kernel is run by the Init_Kernel. So, the tasks, which really give value to the RTMK, should be formed, and put in the ReadyList for guaranteeing the right execution of the RTMK and ensuring its good working. Anyway, the Kernel's task in Terminate is considered to be of great value because it sets the Global Pointer NextTask to the task pointer of the list object which is the Head of ReadyList NextTask. Then it works on the on-going task monitoring and its memory freeing up.

The Kernel now can do the run function after putting all the processes, which were supposed to be run, in the ReadyList. Therefore, the run function works to perform all those processes in the ReadyList that is arranged according to the principle of the priority of executing the shortest deadline task, before going into the idle task and closing it. However, for doing this process and for setting the most needed lists, the Kernel contains a WaitingList that temporarily blocks the other tasks to be able to perform the shortest task first.

The WaitingList and the ReadyList are so comparable, and they greatly resemble each other. Like the ReadyList, the WaitingList contains both a Head and Tail Pointer, and it is also created according to the principle of the priority of closing the shortest deadline. In addition, the WaitingList also has the stack pointer (*PC) information, and it gives feedback about the blocked task, so that the ReadyList can retrieve it afterwards. Like the ReadyList, the WaitingList is also created with List Objects that contains Next and Previous pointers, and which contains msg and nCnt whose function is to know the task's blocking duration.

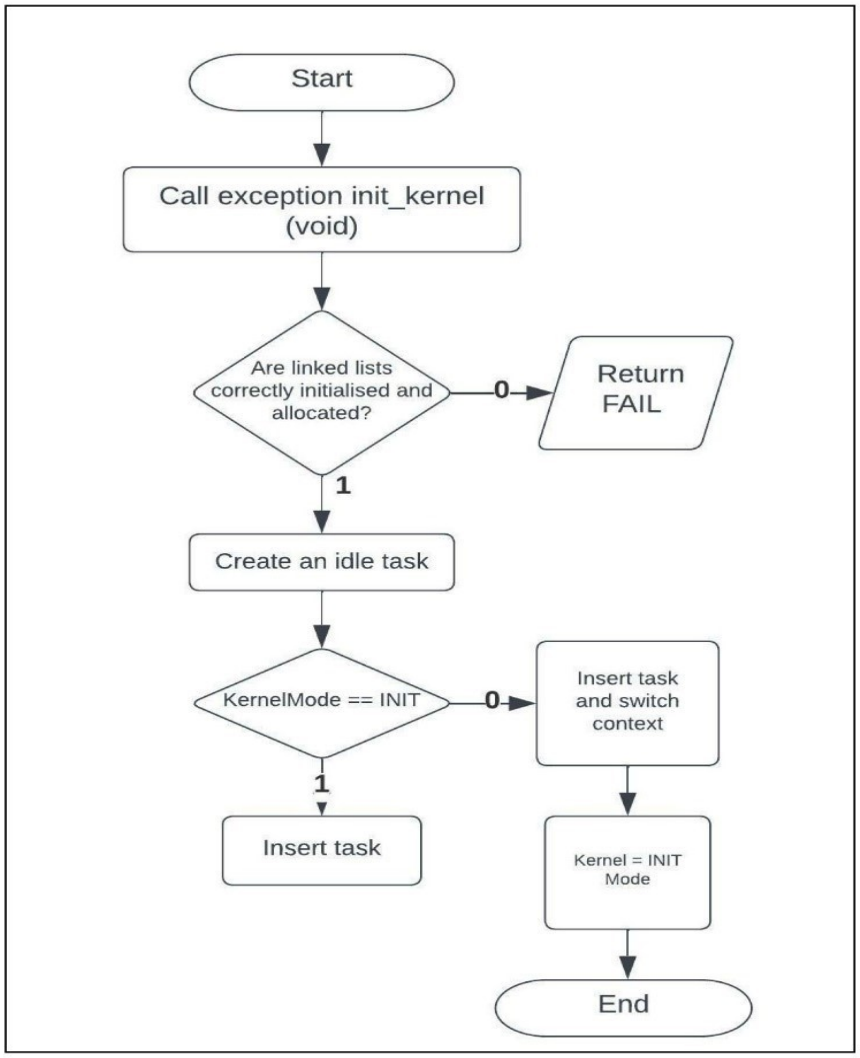
Start
Call exception init_kernel (void)
Are linked lists correctly initialised and allocated?
0
Return FAIL
1
Create an idle task
KernelMode == INIT
0
Insert task and switch context
1
Insert task
Kernel = INIT Mode
End

Figure 3 Task administration

2.3. Mailbox Functions

The tasks communicate with each other using mailboxes that provide the fields with messages to block the tasks. Concerning the construction of the Mailbox, it is not structured the same as the lists. Rather, it depends on the (FIFO), first in first out, way. In the Mailboxes, there is no consideration for the deadline matter. Instead, they are simply arranged according to their natural order in the Mailbox. In addition, the Mailbox contains an empty Head pointer which has Previous to itself and Next to its Next list, whether it is that Tail or any Tail from the list. Tail just has a Next pointer to itself or Previous to Head. In the Tail, the message is put with repointing to itself using Next. If the Mailbox gets two messages, the entry system activates the 'send_wait receive_wait' mode that puts the new messages in the Tail and leaves the old Tail in the back to become Head, message, Tail which has a message. The components of Mailboxes are a Head pointer, a Tail pointer and nDataSize. nDataSize represents the data type size, like the unsigned int or any other struct. NDataSize also talk about the highest permitted number of messages, the number of the existing messages and the blocked messages in the Mailbox.

2.4. *Mailbox creation*

The Mailbox is the result of the Mailbox function that uses Calloc to allocate the Mailbox the memory, and in case of burning the memory, the Mailbox function makes a null check procedure. In addition, this function takes the arguments nMessages and unsigned int nDataSize to give the right value the parallel pointers. Moreover, the Mailbox function works to activate both the Head and the Tail poniters in a way that the Head points next to Tail and vice-versa. So that, each the Head and the Tail pointer, being at the edge, indicate itself, Next points to the Tail and Pervious points to the Head.
At last, the function remove_mailbox is benefited from when there is a need for deleting the Mailboxes. It works by using the (mailbox argument); if there are no messages, it directly empties the Mailbox and returns OK, but in case that the Mailbox is full, then it returns the NOT-EMPTY exception.

2.5. *Types of Messages*

send_wait and send_no_wait functions are used by the tasks that are transmitted to Mailboxes. The send_wait is the function that works on inserting the command sender task, the on-going running task, into the blocking mode to be deleted from the ReadyList and inserted in the WaitingList. For doing this process, the send_wait function works by looking for some feedback about the Mailboxes, like the number of messages that it received. Messages number is known when receivers decrease the value by one, while senders increase the value by one. In case that there is not any message in the Mailbox, it works on allocating memory for putting a message in the Mailbox. Then it deletes the task from the ReadyList and uses the SwitchContext() to switch context to the emerging ReadyList Head task. However, in case that the Mailbox received a message, it works to duplicate its data to move and paste it into the receiving messages data pointer before it finally empties the Mailbox from any messages.

On the other hand, the send_no_wait function works in a completely different way. This function sends the data, not waiting for a response because it keeps the sent task running, and it doesn't block it. Moreover, the send_no_wait function works on duplicating the message's data on its special data pointer in case of finding any waiting message, then it immediately empties the Mailbox from any messages to put the receiving task in the ReadyList. But in case of finding no waiting messages, then the necessary data, the copied message data with memcpy, is allocated memory. After that, the send_no_wait function examines the Mailbox in order to put a message in case it was empty, and in order not to block itself and to signify that it is a sender while the messages are added.

The following flowchart [Figure 4] display the information about send_wait & send_no_wait

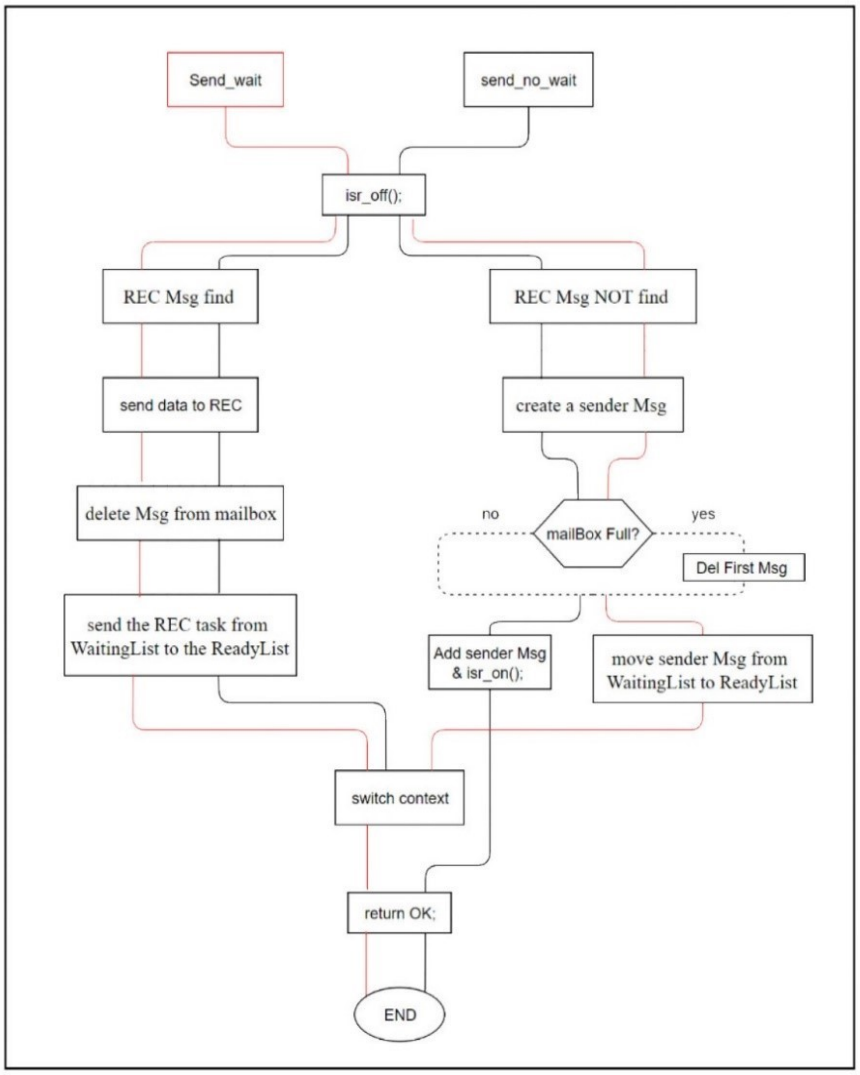

Figure 4 send_wait & send_no_wait

Finally, the receive_wait function works to look for any received nMessages in the (mailbox) to consider them like an argument. This argument works to see if there are any senders in the Mailbox to receive said messages, and then to unblock the task if it was from the send_wait function kind. But in case the Mailbox does not have any messages, the receive_wait function gets into the Mailbox and blocks the running task moving the WaitingList to its new place. To be noted, the receive_no_wait function perform a very similar task, except for the running task which is not blocked in this function.
The following flowchart [Figure 5] display the information about receive _wait & [Figure 6] display the information about receive _no_wait

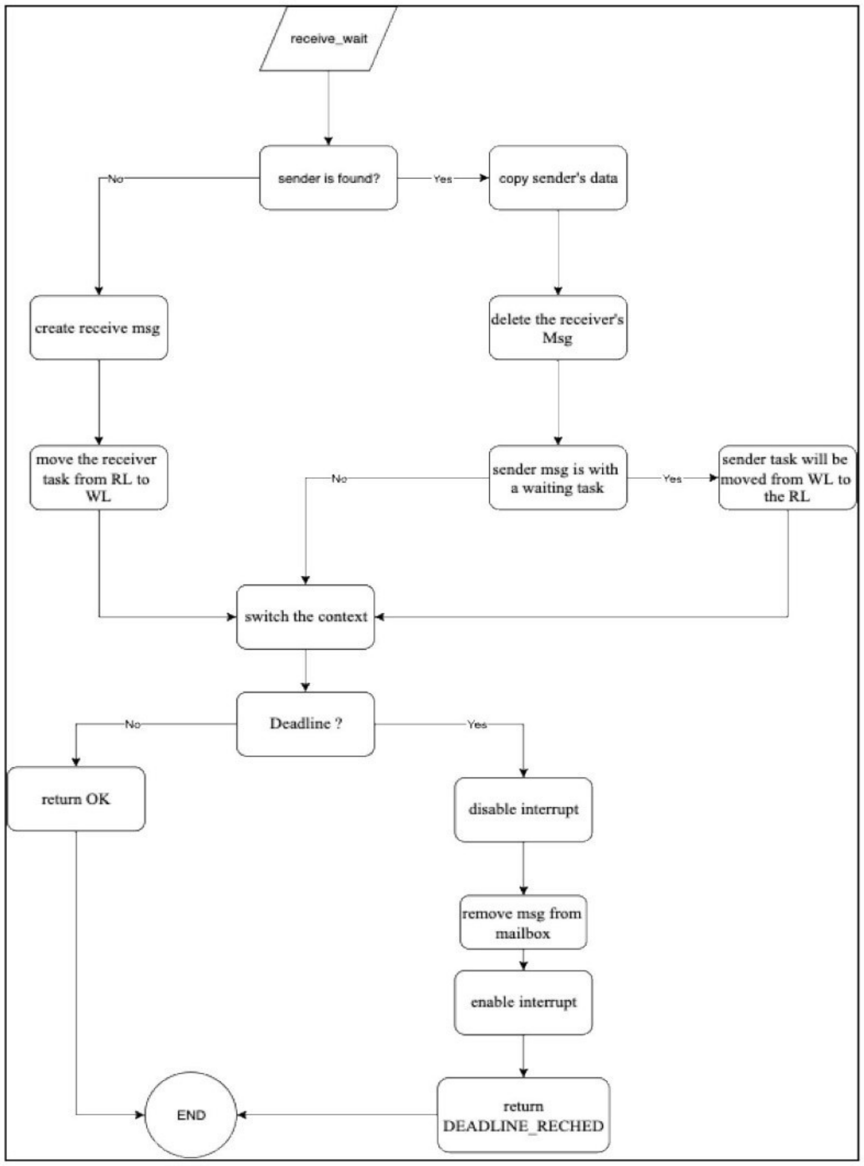

Figure 5 receive_wait

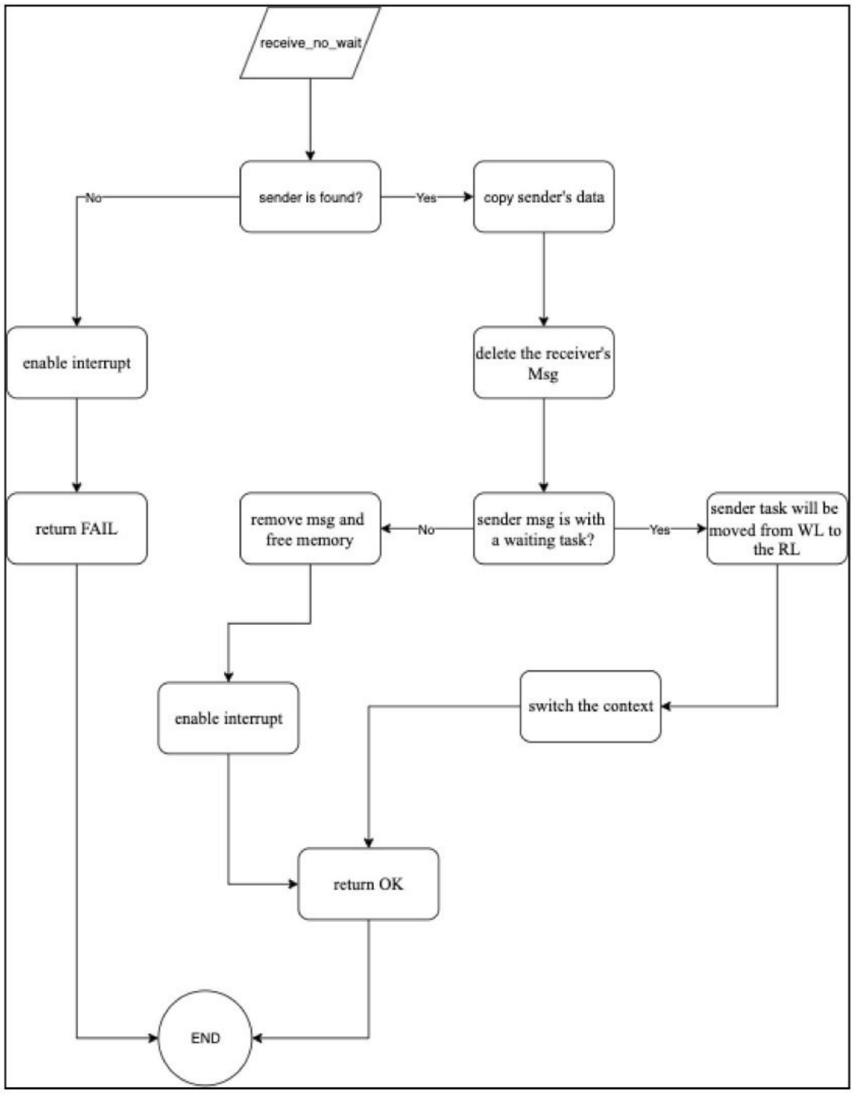

Figure 6 receive_no_wait

2.6. Timing Functions

Various tasks and roles are done by the timing functions. For instance, they work on retrieving the data of the on-going task, such as the Ticks and the deadlines. They also work on choosing a value for the present ticks' number and configuring the present running task deadline. The TimerInt function is called each tick of the system while interrupts continue running and increase the timer of the Ticks. Moreover, the timing functions look for finding any tasks that are sleeping or with a run out deadline to insert them in the ReadyList.

The following flowchart representation [Figure 7] doesn't specifically display all the information, it only gives information about the basic principle of the working process.

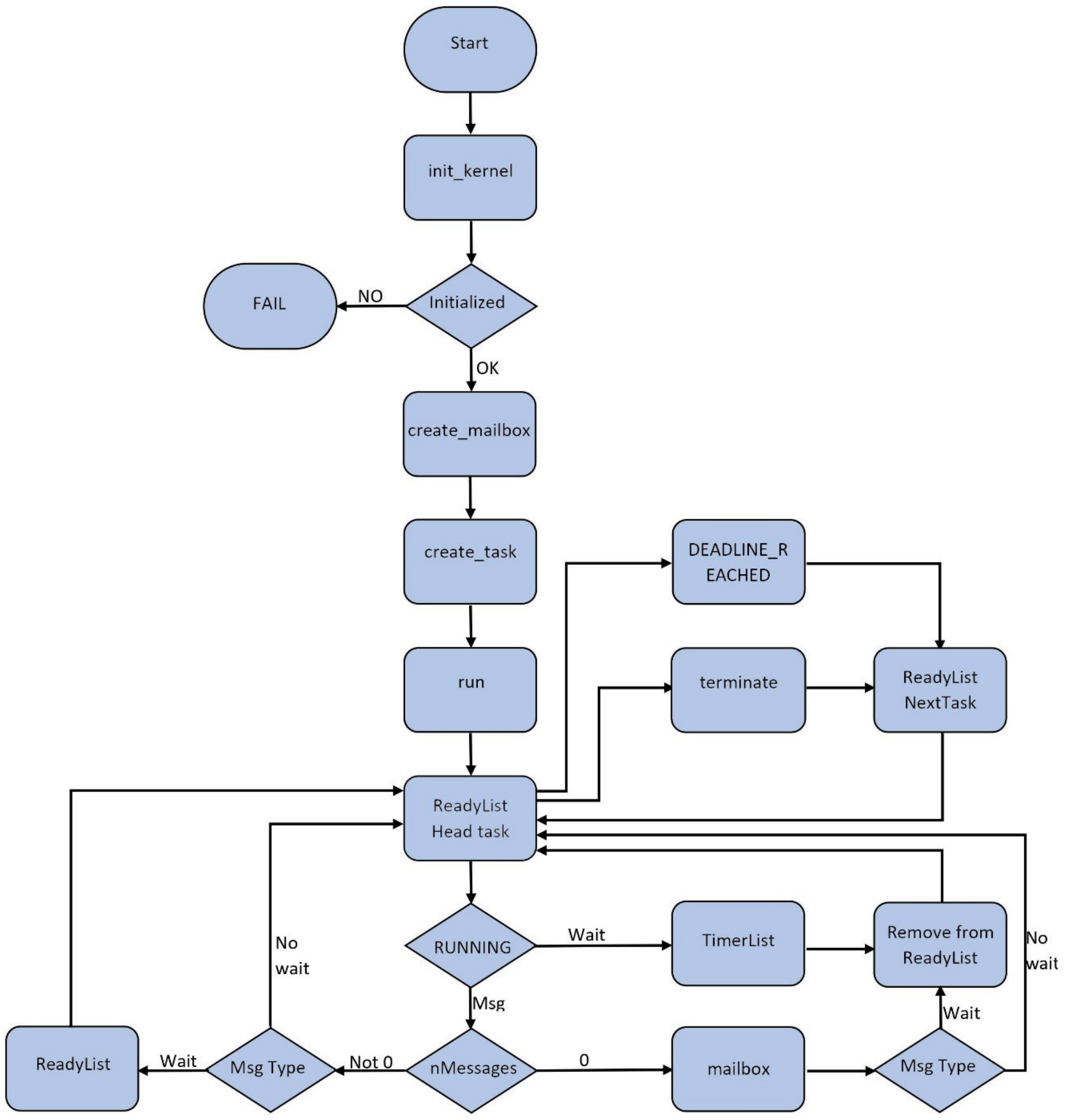

Figure 7 The Flowchart representation

2.7. Overview
After starting the Ready, Waiting and Timer list in the init_kernel, the systems start to work automatically as ordered. It starts by producing the tasks followed by the Mailboxes. Next, it initiates the ReadyList Head task operation that depends on the priority of running the shortest deadline, so activating the multi-functionality of the tasks and the deadlines. In case that there is not any message in the Mailbox, check out is done for the Mailbox to look for any sent messages to block the task if it was of type wait. However, the ReadyList is configured to the Next to go to the running task in case of the termination of the task or the expiration of the deadline. In case of needing the wait function, the running task will be put in the sleeping mode to initiate after that the emerging running task in the ReadyList, and so the kernel proceeds to switch context. If the interrupts are running, the TimerInt is called every tick.

3. Assisting libraries
This section is dedicated to functions created to meet all needs set in this project.
Description and obligations of the functions created in the " kernel_functions.c" and "mailbox_ functions.c " files dealt with below.

list *emptyList();

Initializes an empty list and returns it.

exception initList();

Initializes the three main lists (ReadyList, WaitingList, and TimerList), and returns an exception value indicating whether initialization was successful.

listobj *createNode(TCB *task);

Creates a new list node containing a pointer to a TCB (task control block) struct, a pointer to a message, and other information.

int getSize(list *List);

Returns the size of a list.

listobj *extract(list **List);

Removes returns the head node of a list and removes it from the list.

exception nullLista(list **List);

Set the next and previous pointers of the head and tail nodes of the list to NULL, indicating that the list is empty (Refere if last node is removed).

exception sortedInsertion(list **List, listobj *Node);

Inserts a new node with a task in a sorted position in the list., based on the deadline of the task.

listobj* getNode (list **List,listobj *Node);

Removes and returns the node given by a pointer.

msg *createMsg(void *pData);

Creates a message with given data and initializes all other fields of the message structure to their default values. Returns a pointer to the newly created message.

exception insertMail(mailbox **mBox,msg **message);

FIFO Inserts a message into the mailbox. If there is enough space in the mailbox, the message is added to the end of the mailbox's message list. Otherwise, the function returns a NO_SPACE exception. This function uses a first-in-first-out (FIFO) approach.
msg* getMsg(mailbox **mBox);

Extracts the top message from the mailbox. If there is at least one message in the mailbox, this function returns the first message and removes it from the mailbox. Otherwise, the function returns a FAIL exception.

4. Testing

4.1. Unit tests:

Task_body_4 tests the wait function. It first calls wait() with a duration of zero and expects an immediate return value of OK. It then receives an integer from intMbox and checks if its value is correct. Next, it calls wait() with a duration shorter than low_deadline and expects a return value of OK. Finally, it calls wait with a duration longer than high_deadline and expects a return value of DEADLINE_REACHED. If the wait() calls return the expected values, the task sets g5 to OK, indicating that the wait function passed the unit test.

```
void task_body_4(){
          int retVal_t4;
          int varInt_t4 = 0;
/*The first wait() call waits for zero ticks and should  return
immediately.*/
          retVal_t4 = wait(0);
          if ( retVal_t4 !=  OK ){
                    g5 = FAIL;
                    while(1) {/* no use going further */}
          }
          retVal_t4 = receive_wait( intMbox, varInt_t4);
          if ( varInt_t4 != 255){
                    g5 = FAIL;
                    while(1) {}
          }
/*The second wait() call waits for a period shorter than the
deadline set by low_deadline, which should return OK. */
```

```
          retVal_t4 = wait (500);
          if ( retVal_t4 !=  OK ){
                   g5 = FAIL;
                   while(1) {/* no use going further */}
          }
/* The third wait() call waits for a period longer than the
deadline set by high_deadline, which should cause the function
to return DEADLINE_REACHED. */
          retVal_t4 = wait (2*high_deadline);
          if ( retVal_t4 !=  DEADLINE_REACHED ){
                   g5 = FAIL;
                   while(1) {/* no use going further */}
          }
/* The test will pass if g5 is set to OK at the end of the task,
otherwise, it will fail.*/
          g5 = OK;
          terminate();
}
```

4.2. Integration tests

Each task in this system should be checked out by the programmer to ensure the correct functioning of the operating system. In this system, the check over task can inform the user if there is any fault, and so it is allotted a great importance. In the following, there are some highlights on the kernel's main file:

For checking out the kernel, six unsigned int variables exist with a certain checking function for each variable. The g0, to be mentioned, can test if the kernel is running, while g1 can test the working status of the mailbox, etc… To test the kernel, however, four task bodies do exist and a specific deadline is allotted for each task. Body-task-1, for instance, contains a low-deadline, while task-body-2 has 8* high-deadline, and so on.

The various mailboxes are initialized by the main function, and a different size is assigned for each mailbox. Most importantly, in this main function, the kernel's functioning is tested along with checking if there is anything in the linked lists. Then finally, the various tasks are initialized with a specific deadline for each one of them.

To begin with, this function works to check the status of the mailboxes. In case they have any messages, then the mailboxes' initializing is incorrect since they were originally empty.

Next, three char messages are sent to the mailbox by the send-no-wait (). This process is checked out to test the kernel's ability of sending messages. However, if the send-no-wait has any fault, then a FAIL exception will be given back, which causes the fixing of the task in the while loop.

task_body_1():

The first task (task_body_1) tests an empty mailbox. It tries to receive messages from the mailbox without waiting and checks that it returns the expected value, which should be FAIL. It then sends three messages to the mailbox and receives them one by one, checking that they are the correct messages. Finally, it sends an int message to intMbox using send_wait and a float message to floatMbox using send_wait. The float message should cause a DEADLINE_REACHED error since the task waits indefinitely until it receives a message from floatMbox, but the message is never sent. The task terminates if all communication calls worked correctly.We receive all of these messages using the receive_no_wait() function once all messages have been delivered using the send no wait() function. The two if statements that follow the reading check to see if any messages have been blocked and if the number of messages is zero, indicating that all three messages have been read.

task_body_2 ():

The second task (task_body_2) receives two int messages from intMbox and checks that they are the correct messages. It then waits until task_body_1 sends a message to floatMbox. Once it receives the message, it exits the while loop and terminates.

task_body_3 ():

The third task (task_body_3) tests the mailbox functionality with blocking messages. It sends and receives messages using blocking calls and checks that the messages block as expected. If the messages block and unblock correctly, it returns OK. The message was sent successfully.
task_body_4():

The fourth task (task_body_4) tests the kernel's task scheduling behavior, it waits for 900 system ticks, receives an int message from intMbox, checks that it is the correct message, waits for 4000 system ticks, and then terminates.

task_body_4 contains a deadline, and if that deadline is not met, the function will enter an infinite loop. **task_body_5():**

The fifth task (task_body_5) is test for the mailbox functionality, sends an int message to intMbox using send_no_wait, and waits for low_deadline ticks. If the wait function returns OK, it terminates the task.

Overall, this code is testing the core features of the kernel, including basic functionality of mailboxes, timing functions, task scheduling, and performance including blocking and nonblocking send and receive messages, and deadline handling. However, it could be improved by adding more assertions and checks to ensure that the expected behavior is occurring. Additionally, task_body_5 should be completed to fully test the behavior of send_no_wait messages.

main():

Several Mailboxes of various sizes are generated in the main function. The main method also handles the LinkedList's testing and Kernel startup. Five unique assignments are ultimately established, each with a different deadline—either a short deadline or a long date.

5. Conclusion

In conclusion, the Operating System is a critical component of digital devices as it controls all commands and information exchanges between different components. The RTMK project provides an efficient and reliable real-time software kernel that manages collateral operations and resources in the system. It uses linked lists to prioritize tasks according to their deadlines and ensure the optimal usage of resources. Mailboxes are used for interprocess communication, and tasks are managed through the ReadyList, WaitingList, and TimerList. Overall, the RTMK project offers a useful tool for developers and programmers to create a functional kernel for their real-time systems. Through this project, we have gained knowledge on task management and inter-process communication, which are essential skills in software development.

6. References

1. Lab 1: Task Administration. Högskolan i Halmstad.
2. Lab 2: Inter Process Communication. Högskolan i Halmstad.
3. Lab 3: Timing functions. Högskolan i Halmstad.