RTMK - Real-Time Micro kernel

Abd Alrahman Atieh (abdati21@student.hh.se)
Abdulfattah Morad (abdmor21@student.hh.se)

September 21, 2024

Abstract

This project will explore the details of building a real-time microcontroller, RTMK. RTMK is a small operating system. It executes tasks according to its deadlines. RTMK is used in everyday systems such as smart houses, hospitals, and cars. It has (5) basic features, creating tasks, terminating tasks, sync and async communication, and scheduling. The primary data structure for the RTMK is the double-linked list. This project is a part of the course Computer Systems Engineering II (DT4013), the third trimester in 2023, at Halmstad university.

Contents

1 Introduction

1.1 Role of operating systems in general? and Why do we need them in embedded systems?

The operating system is a program that the processor executes, controlling the hardware parts and coordinating their use among the applications. Operating systems hide the hardware details of the programmer, giving a convenient interface for using the system to the programmer. Operating systems manage the resources like Input/ Output, memory such as primary and secondary, and execution time of the processor. The kernel is a part of the operating system located in the main memory, and it contains functions frequently used in the OS. The OS manages application execution, allowing the user to switch among multiple applications so that all will appear to be progressing[1][2].

1.2 What type of OS you are implementing in this project work? and what features it offers.

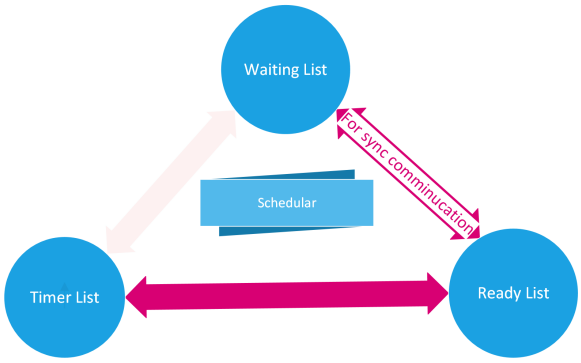

Figure 1: The type of the OS, an RTMK

This operating system is an RTMK, a **R**eal-**T**ime **M**icro **K**eral. These operating systems are used in devices and systems that require to do tasks with deadlines. Throughput is optional here as long as the tasks are done before their deadlines. It is essential to mention that the task is running based on the EDF algorithm, Earliest Deadlines First. The operations of this project's RTMK are task administration, time scheduling, and sync and async communication.

1.3 The organisation of the report

This report has six main sections. The first section is an introduction section that gives a general view of operating systems. Operation of the RMTK section goes through our OS in detail with flow charts to explain some parts. Another section is an assisting libraries section that describes the libraries used in this project. The fourth section is a section that explains the different unit tests that are done. The fifth section is a conclusion section. And last, a references section.

2 Operations of the RTMK

2.1 Task Adminstration

```
1       exception init_kernel();
2       exception create_task(void (*taskBody)(), unsigned int deadline);
3       void run();
4       void terminate();
5
6       // Essential global variables
7       extern int Ticks;       /* global sysTick counter */
8       extern int KernelMode;
9       extern TCB *PreviousTask, *NextTask;
10      extern list *ReadyList, *WaitingList, *TimerList;
```

In task administration, the essential functions are initializing and starting the kernel, creating and terminating tasks. In the next sections, are implementations of these functions.

2.1.1 Initialize the kernel

Clear the counter (*Ticks*). Create the *ReadyList*, *WaitingList*, and *TimerList*. These lists are double-linked lists; see section 3.1.1. Create and insert the idle task in the *ReadyList*, see section 2.1.7. Set the kernel in *INIT* mode. Then return the status message.

2.1.2 Creating a task

Each task has a function body, an address of where the instructions are located in memory. It also has a deadline, thus the name, real-time microkernel. First, allocate some memory for the TCB. See section 2.1.6. Second, save the status of the task in the TCB. Create a list object for this TCB. See section 3.1.2. Insert the newly created list object in ReadyList, in descending order, where the value compared is the deadline, see section 3.1.4. The steps before are in case the kernel is in the INIT mode, but if the kernel is running, there is another procedure.

When a kernel is in RUNNING mode, we must disable all the interrupts. Update the PreviousTask by assigning it the current task, i.e the NextTask. Insert the newly created list object in ReadyList in descending order. Update the NextTask to be head of the ReadyList, the newly running task, in case the newly added task has a smaller deadline. see section 2.1.8.

2.1.3 Running the kernel

Clear the counter (Ticks). Set the kernel mode to RUNNING. Load the NextTask, which is the head of the ReadyList. Load the context, see section 2.1.5.

2.1.4 Terminating a task

Disable the interrupts. Pop the head of the ReadyList. Update the NextTask. Switch the context. Free the memory that was occupied by the removed task. Then load the context.

2.1.5 Context Switching and Loading

```
1       extern void     isr_off(void);
2       extern void     isr_on(void);
3       extern void     SwitchContext( void );
4       extern void     LoadContext_In_Run( void );
5       extern void     LoadContext_In_Terminate( void );
```

- **isr_off()**: Disable all interrupts.
- **isr_on()**: Enable all interrupts.
- **SwitchContext()**: Loads the context of NextTask.
- **LoadContext_In_Run()**: Loads the context of NextTask. Only used in function run().

- **LoadContext_In_Terminate()**: Loads the context of NextTask. Only used in function terminate().

2.1.6 Task Control Body, TCB

Task Control Body, TCB. It is a struct that holds the address of the stack pointer (SP). It also holds the registers from R4 to R11. The program counter (PC). The stack pointer status register (SPSR). A stack segment with a fixed size of 100 bytes. And last but not least, the deadline.

2.1.7 The idle task

The idle task is called when there are no other tasks running. If the idle task is running, the kernel is in a free state and waiting for new tasks to arrive. The idle task should have the largest deadline and always be at the end of ReadyList, at the tail.

2.1.8 PreviousTask and NextTask

PreviousTask saves the TCB for the previously running task. *NextTask* saves the TCB for the currently running task. These two variables are used when a context switch is done.

2.2 Communication

```
mailbox        *create_mailbox( uint nMessages, uint nDataSize );
exception       remove_mailbox(mailbox *mBox);
exception       send_wait( mailbox* mBox, void* pData );
msg            *create_msg(exception status, listobj *block);
exception       receive_wait( mailbox* mBox, void* pData );
exception       send_no_wait( mailbox* mBox, void* pData );
int             receive_no_wait( mailbox* mBox, void* pData );
```

2.2.1 Send And wait (sync)

```
exception       send_wait( mailbox* mBox, void* pData );
```

This enables tasks to send data into a specific mailbox. This function will move the current task to *Waitinglist*. It is a synchronous function. See figure 2.

First, disable all interrupts. Then, check if the receiver has come to the mailbox before the sender. Then, copy the sender's data *pData* to the receiver's message structure. The copying is done using *memcopy(address of destination, address of origin, the size of copied data)*. After copying, dequeue the message structure from the mailbox. The receiver does not need to wait anymore. Thus it is moved from *WaitingList* to *ReadyList*. In case the receiver task has not yet come. Then, the sender will allocate memory space for the message structure. Inside the memory structure, there is *pData* which needs memory allocated for it. After the allocation is done, copy the sender's data to the message structure. Then add the message to the mailbox using *enqueue()*. At last, move the sender to the *WaitingList*. After moving from or to the *ReadyList*, the PreviousTask and NextTask are updated. A possible SwitchContext could occur. On leaving the function, a check for the deadline is done. If the deadline is reached, the process is started, then a *DEADLINE_REACHED* is returned, see section 2.2.5 for details on that process. If the deadline is not reached, *OK* is returned.

2.2.2 Receive And wait, sync

```
exception       receive_wait( mailbox* mBox, void* pData );
```

This function is synchronous. The calling task will be put in the *WaitingList*. It enables tasks to receive messages from a specific mailbox. See figure 3.

First, it disables all interrupts. Then checks if the sender has sent the message. Then, copy the sender's data to the receiver's data area, and dequeue the message from the mailbox. After removing the message from the mailbox, it checks if it has been sent by a synchronous task, i.e., if the sender

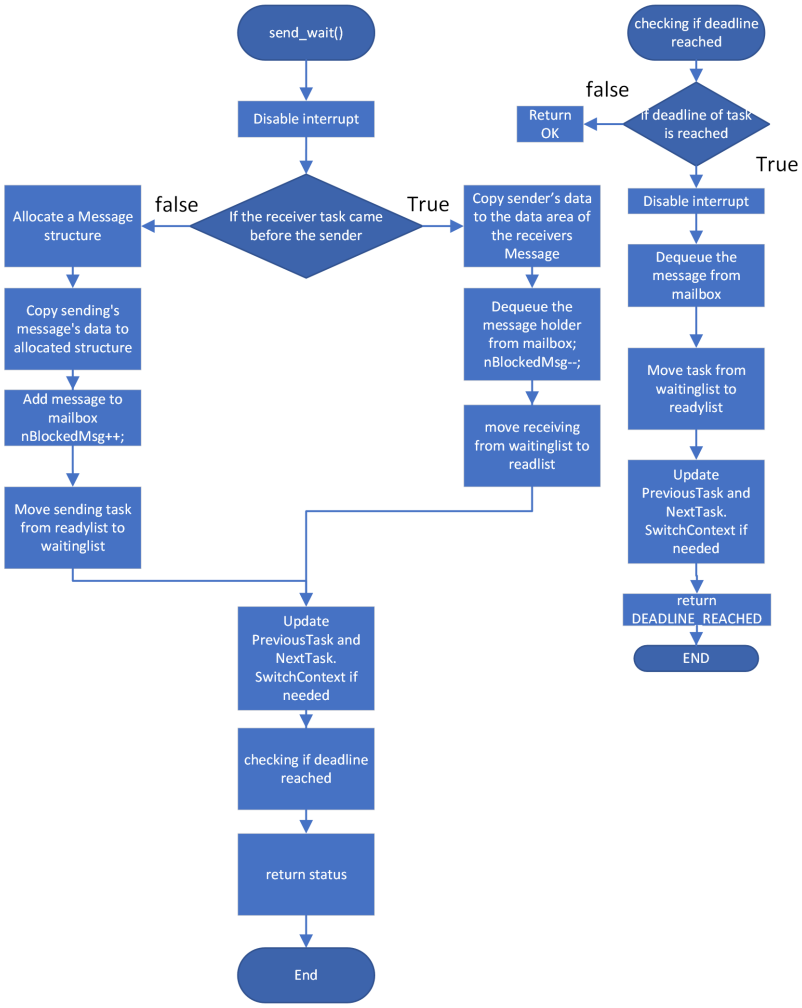

Figure 2: Sync send message and deadline check funcitons

is waiting in *WaitingList*. Then, move the sender to *ReadyList* and update the PreviousTask and NextTask. Else, free the sender's message *pData* area.

If the sender has yet to arrive at the mailbox, the receiving task must allocate a memory place for the message structure and data area. Then, the receiver will go to the *WaitingList*. The PreviousTask and NextTask are updated. Lastly, a SwitchContext could occur. On leaving the function, a deadline check is done. See the section 2.2.5 for details. The status will be returned. Either *OK* or *DEADLINE_REACHED*.

2.2.3 Receive And do not wait (async)

```
1       exception       receive_no_wait( mailbox* mBox, void* pData );
```

This function is asynchronous. The receiving task will check the mailbox. If the message is there, it will take it. Otherwise, it will return *FAIL*, see figure 4. It starts by disabling all interrupts. Then, it checks if the sender has sent the message. If not, return *FAIL*. Copy the sender's data and paste it into the receiver's data area. Dequeue the message from the mailbox. If the sender is not waiting, free the sender's data area. Else, move the sender to the *ReadyList*. Update the PreviousTask and NextTask, then do a SwitchContext if needed. Lastly, return *OK*.

2.2.4 Send And do not wait (async)

```
1       exception       send_no_wait( mailbox* mBox, void* pData );
```

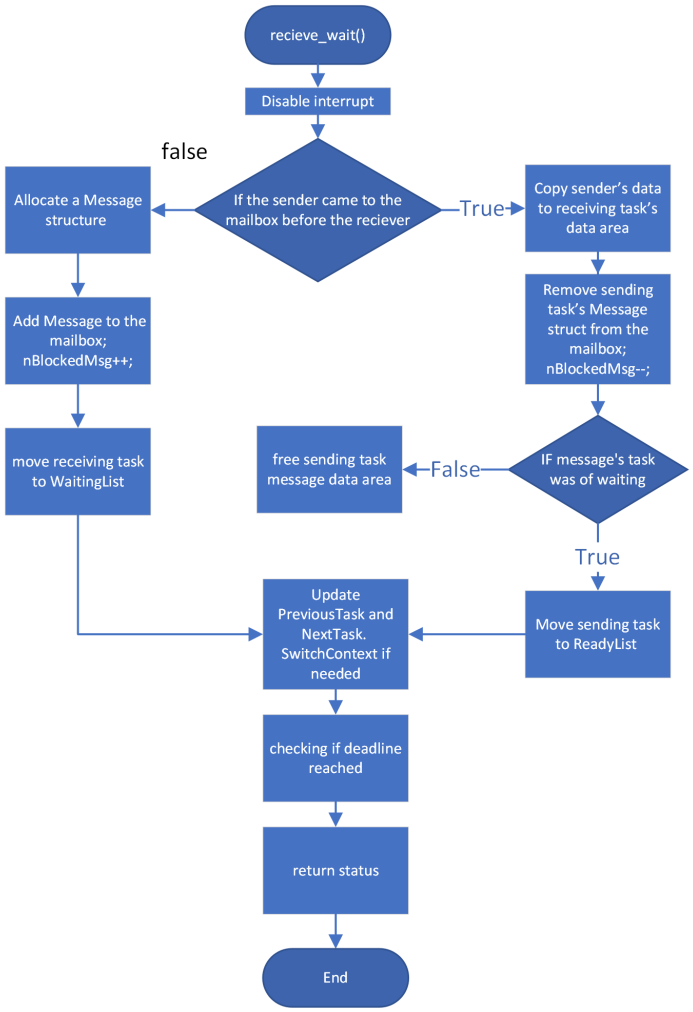

Figure 3: Sync receive message funciton

This function is asynchronous. The sender will check the mailbox and see if a receiver is waiting. If the receiver is waiting, the sender will copy its data to the receiver's data area and then move the receiver to the *ReadyList*. The PreviousTask and NextTask will be updated, meaning SwitchContext could occur. In case of the sender has not found the receiver waiting, it will allocate memory for the message structure. Then, it will paste its data in the message's data area, but it will not go to the *WaitingList*. The message will be added to the mailbox. See figure 4.

2.2.5 is deadline reached?

This process, shown in figure 2, checks if the deadline of the head message in the mailbox has been reached. This is done by comparing it with *Ticks* counter. If the deadline has been reached, the message owner will be moved to the *ReadyList*. The PreviousTask and NextTask will be updated. A possible SwitchContext could occur.

2.3 Timing

```
1       exception        wait( uint nTicks );
2       void            set_ticks( uint nTicks );
3       uint            ticks( void );
4       uint            deadline( void );
5       void            set_deadline( uint deadline );
6       extern int      Ticks;      /* global sysTick counter */
```

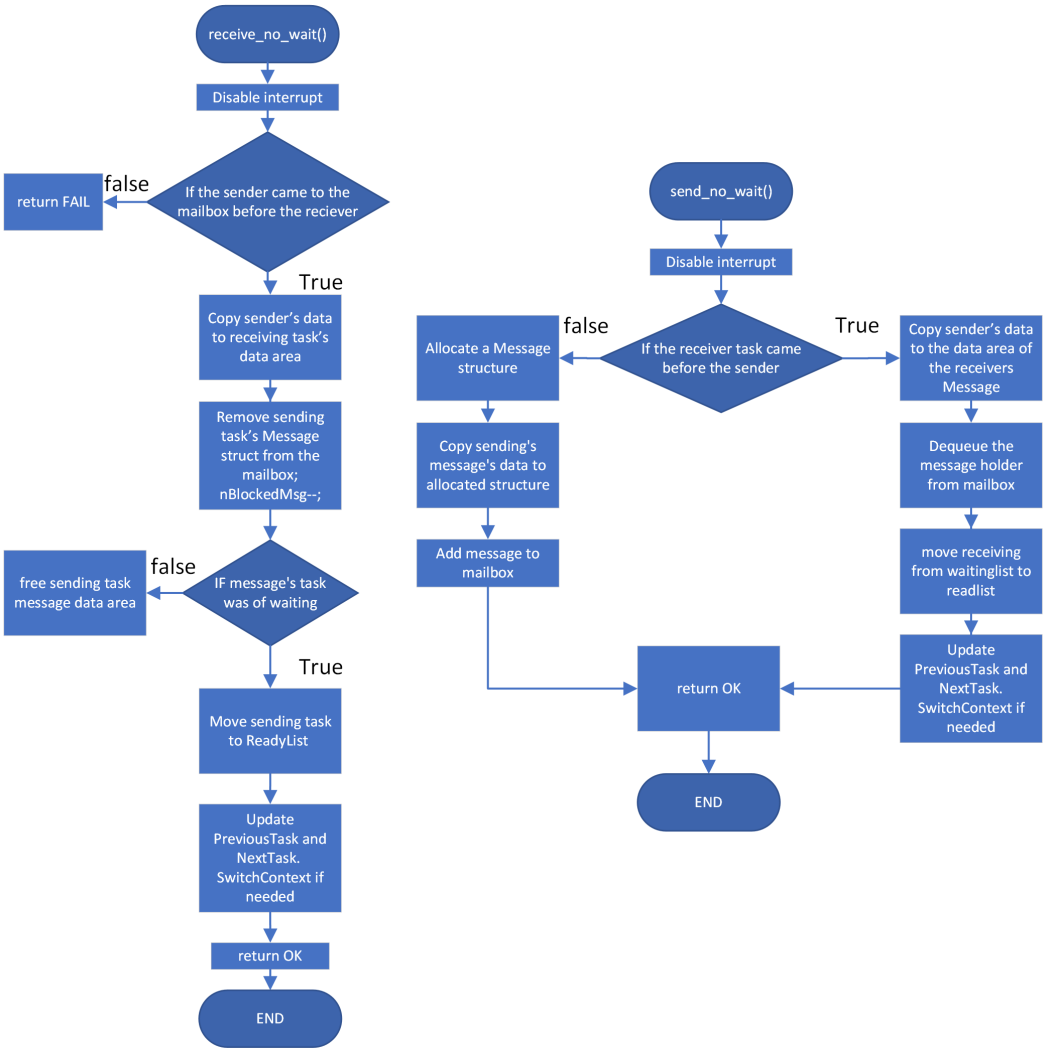

Figure 4: Async send/receive message funcitons

```
7       void            TimerInt()
```

2.3.1 Move from ReadyList to TimerList

```
1       exception       wait( uint nTicks );
```

This function will send the running task, *NextTask*, to the *TimerList* to sleep for a duration of time *nTicks*. Notice that the timer is implemented as a down counter, i.e., when it reaches zero, the counting is finished.

It starts by disabling all interrupts and updates the previous task to be *NextTask*. Remove the running task from *ReadyList*, which is the head of the list. Set the time for the removed task, the *nTCnt*. Then, insert that task into *TimerList*. Update the NextTask. A possible SwitchContext could occur. While the task is inside *TimerList*, the deadline might be reached. If so, return *DEADLINE_REACHED*. Else, return *OK*. See figure 5.

2.3.2 Updating deadlines (Rescheduling)

```
1       uint            deadline( void );
2       void            set_deadline( uint deadline );
```

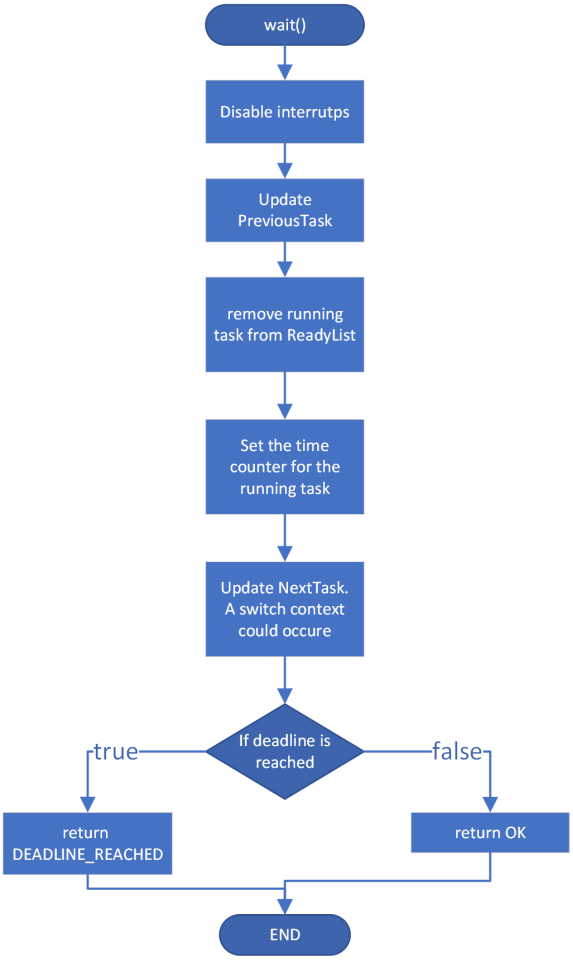

Figure 5: wait(): sends tasks to timer list for a specific time

To reschedule a task, all it is needed is to edit its deadline, and this can be done by *set_deadline()*, see figure 6. It takes the new deadline as an argument. First, disable all interrupts. Then, edit the deadline for the running task, *NextTask*. The rescheduling is very simple, remove the running task from the *ReadyList*, and insert it back using the function mentioned in section 3.1.4. While doing that, it is crucial to update *PreviousTask* and *NextTask*, which means a context switch could occur.

2.3.3 Periodic timer interrupts (TimerInt)

```
1       void TimerInt();
2       void check_deadlines(list *lis);
3       void decrement_TC_check_deadline();
```

The Timer Interrupt is called periodically by *sysTick*. *TimerInt* is our scheduler. It has a *Ticks* counter. It is always checking in *TimerList* if the sleep duration is over and the deadline is reached. When the schedule is going though the *TimerList*, it also updates the counters, *nTCnt*, for the sleeping tasks, see figures 7 and 8. The schedular is responsible for checking the deadlines for the tasks in *WaitingList*. It goes through the list and checks if a task's deadline is lower than or equal to *Ticks*, see figure 9. For both *TimerList* and *WaitingList*, if any task has a deadline reached or a sleep time is over, it will be moved back to the *ReadyList*. That may cause a context switch.

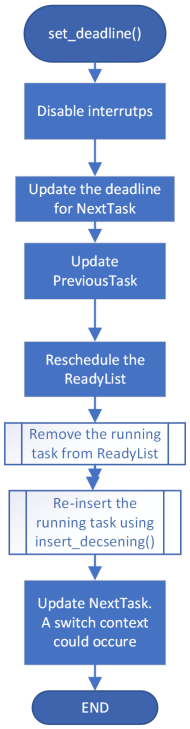

Figure 6: This call will set the deadline for the calling task

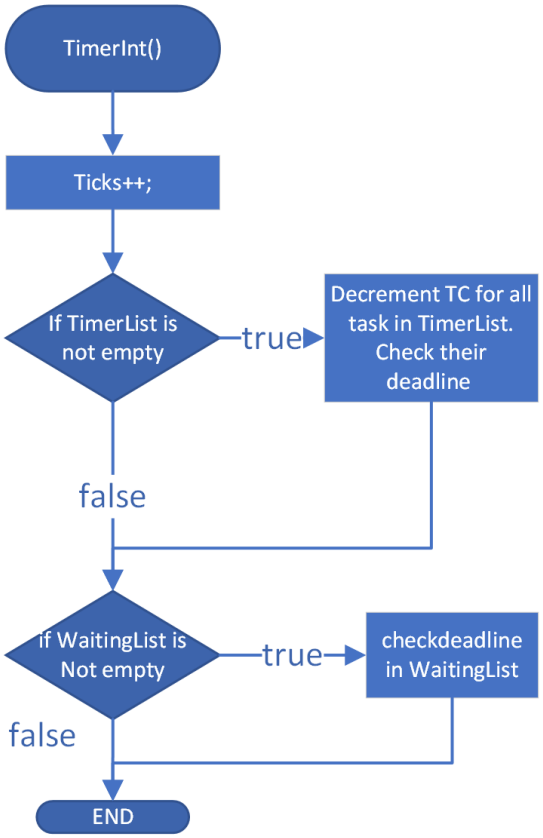

Figure 7: TimerInt()

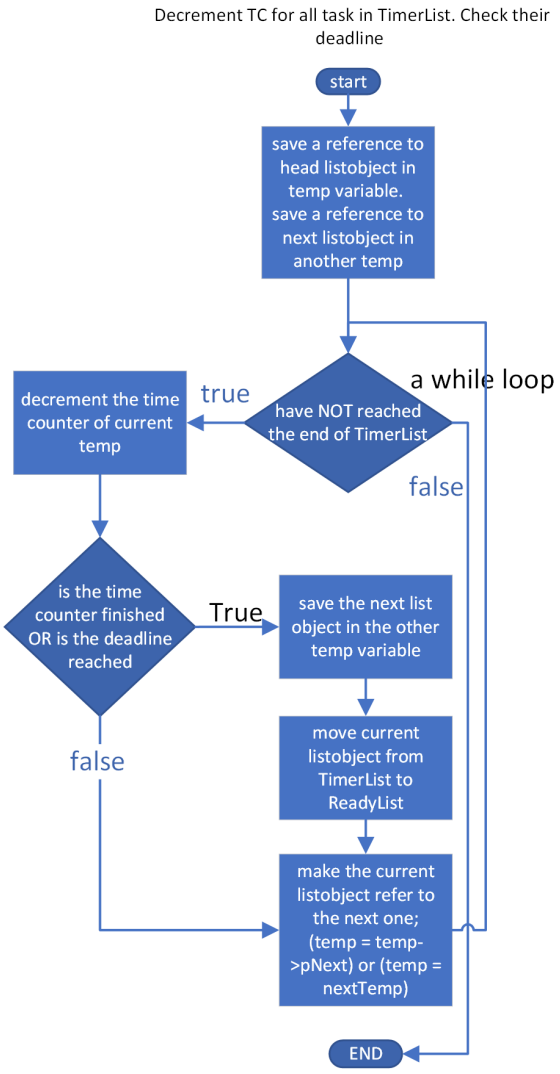

Figure 8: decrement_TC_check_deadline()

3 Assisting libraries

3.1 Double-linked Lists

A double-linked list consists of head and tail pointers. They are interpreted as the first and the last objects of a list. A double-linked list means every node object has a pointer to the next node, which comes after it. It also has a pointer to the previous node of the next, which comes before it.

3.1.1 Creating a double-linked list

In this implementation, the list has head and tail pointers and a size counter. First, a memory space is allocated for the list. The size counter helps with getting the status of the list if it is empty or not.

3.1.2 Creating A list object (a node):

```
1       listobj *create_listobj(TCB *task, uint tcnt, msg *m)
```

First, a memory space is allocated for the node. After that, a safe check is done to ensure the memory allocation succeeded. If there is insufficient memory, the function will return a null value. The user can assign a TCB, Task Control Block, a time counter, and a message directly when creating the list

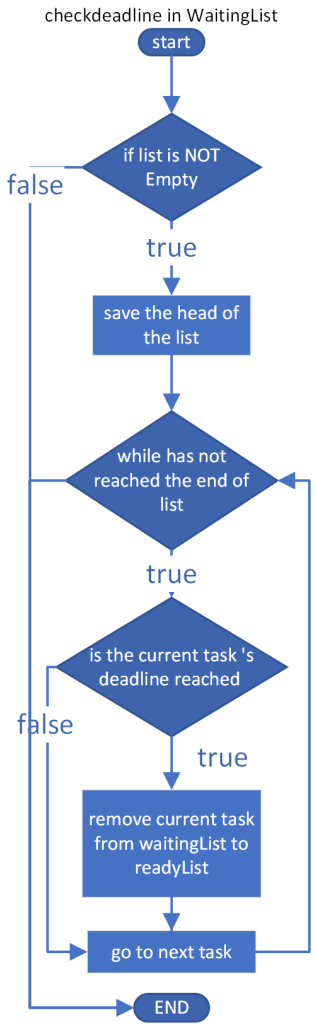

Figure 9: check_deadlines(list)

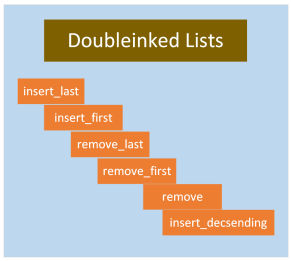

Figure 10: Double-linked list structure

object, the node. The list object is also a node containing a pointer to TCB, a time counter, a pointer to a message object, and a pointer to the next and previous list-objects.

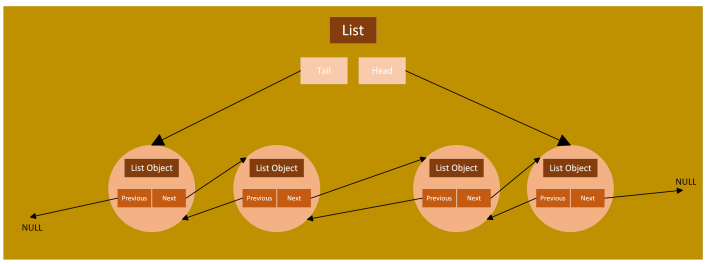

Figure 11: Double-linked list exemple

3.1.3 Inserting in a Double linked list

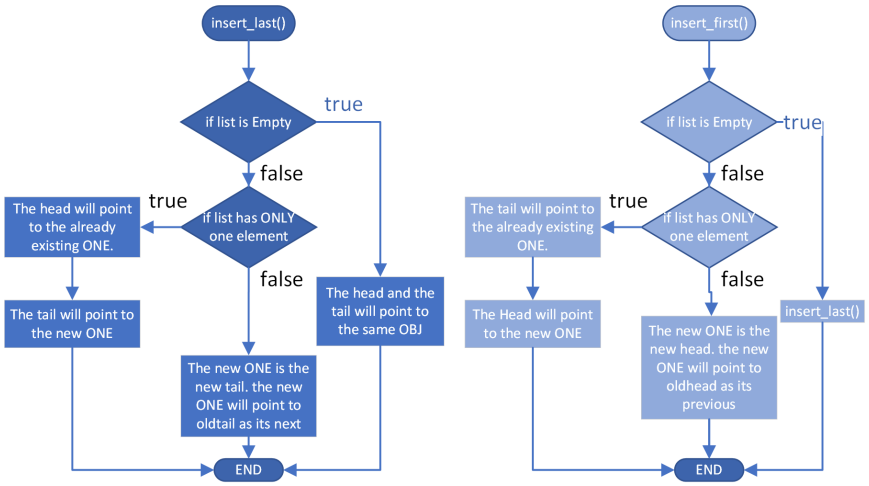

Figure 12: insert_first() and insert_last()

```
1       void insert_last(list **l, listobj **obj);
2       void insert_first(list **l, listobj **obj);
3       void insert_descending(list **l, listobj **obj);
```

The user can insert in three different ways: at the tail, at the head, and in descending order, where the head is the smallest object. This way, the double-linked lists can be implemented using FIFO (First In, First Out) or LIFO (Last in, First Out). See figure 12. In other words, it can be implemented as stacks or queues. To achieve this, remove first and remove last are implemented in a later section. For insert first and insert last, there are three different cases.

case 1: When the double-linked list is empty, the head and tail will point to the same node upon insertion. The node will point at null values because there is neither a previous nor next node in the list.

case 2: When the list double-linked list has one existing node.

1. Suppose inserting first, at the head. The tail will stay pointing at the existing node, but the head will point to the new node. The old node will point to the new node as its next. The new node will point to the old node as its previous node.
2. Suppose inserting last, at the tail. The head will stay pointing at the existing node, but the tail will point to the new node. The old node will point to the new node as its previous. The new node will point to the old node as its next node.

case 3: In general, the method is the same as in case 2, but it saves the head/tail in a temporary node first. Then it follows the same precoder as in case 2.

3.1.4 Inserting in a descending order

Inserting in descending order, this is achieved linearly. It starts at the head and keeps comparing until it reaches a node with a smaller or equal value. Then it is inserted before it. Here, there are three insert cases:

case 1: The list is empty, then insert using functions insert_(first/last).

case 2: The list has one node. If the node has a bigger value, insert it at the head. Otherwise, insert it at the tail.

case 3: A general case. It starts at the head. It keeps comparing until it reaches a node with a smaller or equal value. Then it is inserted before it.

see figure 13

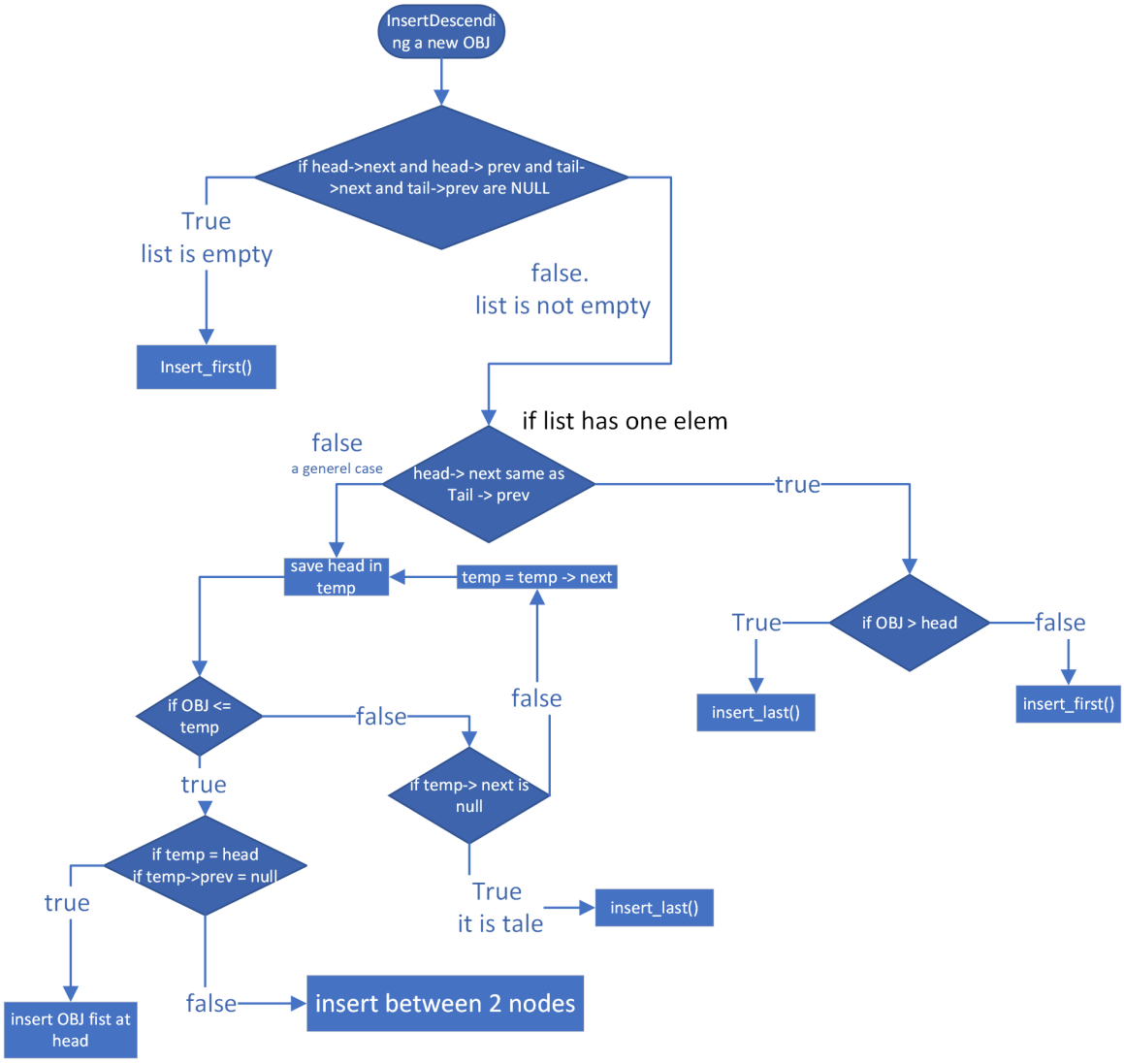

Figure 13: Double-linked inserting in a descending order

3.1.5 Removing a node from a double linked list

Three methods were implemented: removing the head, the tail, and a specific node. For removing from the head or tail, save the head/tail in a temporal variable. Make the head/tail point to the

next/previous node in the list if the list has one node left. Then the head and tail will be pointing to the same node. Notice that before returning the removed node, make the node point to null values. This way, you do not return the whole list but the node itself.

To remove a specific node, first, search for it. The search is linear, starting at the head and searching until the specific node is found or reaches the tail. When the node is found, save pointers to the previous and next node of the specific node. Before returning the specific node, the next and previous nodes will point at each other directly, and the specific node will point at null values.

```
1       listobj * remove(list **l, listobj *obj);
2       listobj * remove_last(list **l);
3       listobj * remove_first(list **l);
```

3.1.6 Help functions for the double-linked lists

There are functions to get the size of a specific list and a function that tells if a specific is empty.

```
1       uint get_size(list *l);
2       int isEmpty(list *l);
```

3.2 Mailbox (as an assisting library)

FIFO-type data structure. This means the message which came in first will leave first.

3.2.1 Creating an empty mailbox

```
1  mailbox *create_mailbox(uint nMessages, uint nDataSize);
2
3  typedef struct {
4          msg             *pHead;
5          msg             *pTail;
6          int             nDataSize;
7          int             nMaxMessages;
8          int             nMessages;
9          int             nBlockedMsg;
10 } mailbox;
```

A mailbox is a struct data type. It is similar to a double-linked list data structure but with some modifications. The mailbox has *nDataSize*, which specifies the size of the data each message will have. There should be a different mailbox for different data types.*nMaxMessages* is the capacity of the mailbox. *nMessages* is the size/length of the mailbox. On creating an empty mailbox, memory space is allocated.

3.2.2 Removing a mailbox

```
1              exception remove_mailbox(mailbox *mBox)
```

When a mailbox is empty and will not be used anymore, the user needs to free the allocated memory. This is done with this function.

3.2.3 Adding a message to the mailbox

```
1       void enqueue(mailbox *mb, msg *m)
```

Enqueuing a message, adding a message to the mailbox. There are 4 cases when adding a message to the mailbox. See figure 14. In the first case, when the mailbox is complete, the oldest message will be removed, and the new message will be inserted instead. The second case is when the mailbox is empty. The head and the tail of the mailbox will become the same message, i.e., point to the same message. The third case is when the mailbox has two messages. The head will stay pointing to the same message, but the tail will point to the new message. The last case is a general case. The tail will point to the new message. Then the new message will point to the old message, and the old message will point to the new message. The counters are incremented.

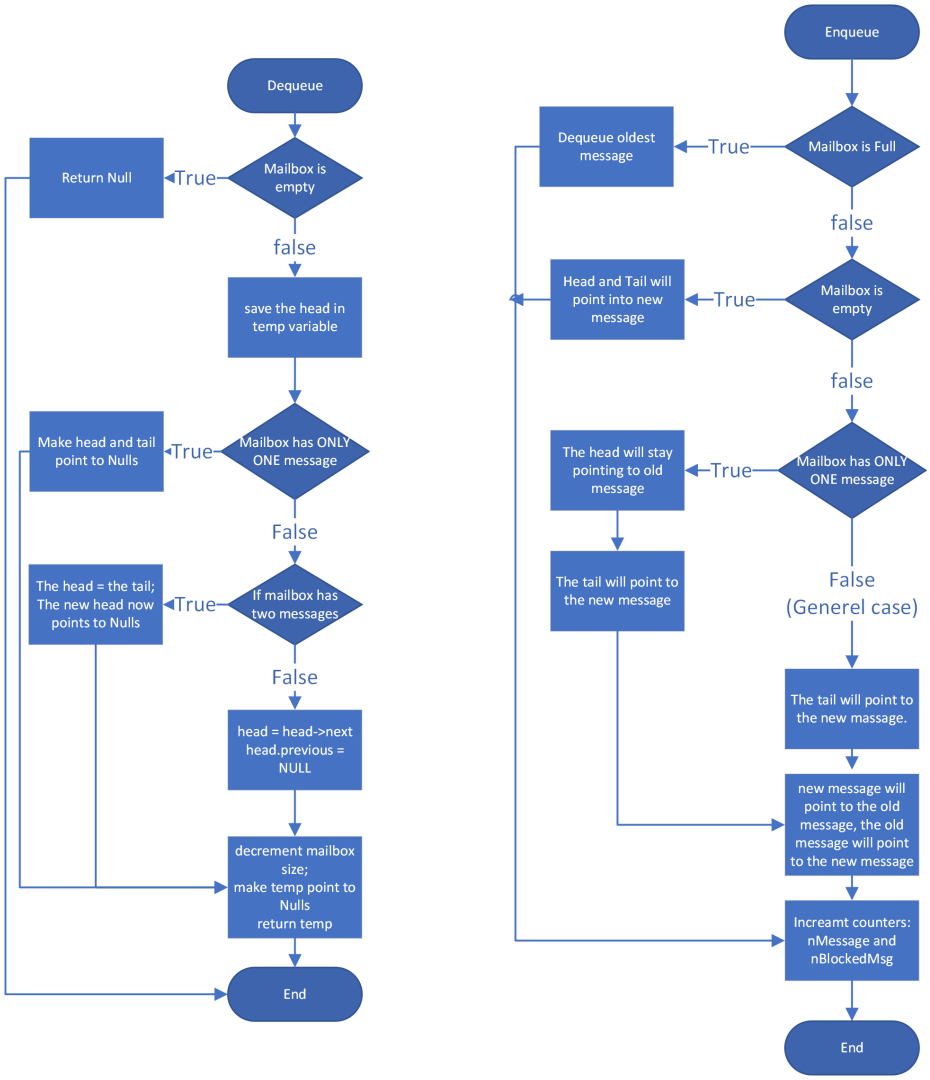

Figure 14: Mailbox Enqueuing and Dequeueing functions

3.2.4 Removing a message to the mailbox

```
msg * dequeue(mailbox *mb)
```

Denqueuing a message, removing a message from the mailbox. There are 3 cases when removing a message from the mailbox. See figure 14. The first case is when the mailbox is empty. A null value is returned. The second case is when the mailbox has one message. The head and the tail will be null. The third case is when the mailbox has two messages. The head will be the same as the tail. In a general case, The head will point to the next message in the mailbox. Then the message will be returned. The counters are decremented.

3.2.5 Other functions for the mailbox

```
exception isMailboxEmpty(mailbox *mb);
exception isFull(mailbox *mb);
int no_messages(mailbox *mBox);
exception remove_msg_from_mailbox(mailbox **mBox, msg *m);
```

isMailboxEmpty will return 0 if the size of the mailbox is 0, otherwise, 1. *isFull* will return one if the size and capacity of the mailbox are equal, otherwise 0. *no_messages* will return the number of the messages. *remove_msg_from_mailbox* will remove a specific message from the mailbox.

4 Testing

4.1 Unit tests For each function implemented in the assisting libraries or the RTMK

4.1.1 Double-linked List

```
// these test cases are in  RTMK/lab1.2/main.c
void test_insert_descending()
void test_create_listobj()
void test_create_empty_list()
void test_insert_last()
void test_insert_first()
void test_remove_last()
void test_remove_first()
// These test cases are in file RTMK/lab2/main.c
void test_remove1()
```

- *test_insert_descending()*: A list and a few list objects with different deadline values were created. The list objects were inserted randomly. In order to see if the functions were working, watch windows were opened. There we can see the order, which was correct.
- *test_create_listobj()* and *test_create_empty_list()*: Simple functions to test if the data structures were created correctly.
- *test_insert_last()* and *test_insert_last()*: Different list objects were inserted. These tests included insertion when the list was empty. The list has one node. The result was seen in the live watch window.
- *test_remove_last()* and *test_remove_first()*: These tests included removing from either head or tail, removing when the list is empty, and removing when the list has 1 or 2 list-objects.
- *test_remove1()*: When removing a specific object from a list.

4.1.2 Tests for the mailbox

```
void test_create_msg();
void test_remove_mailbox();
void test_enqueue_dequeue();
```

- *test_create_msg()*: Creating messages with different parameters to see its behavior.
- *test_remove_mailbox()*: Removing a mailbox when it is not empty and when it is empty.
- *test_enqueue_dequeue()*: Creating messages and then inserting them to test the different cases mentioned in section 3.2.3. If inserting is done, the dequeuing function is tested for the different cases mentioned in section 3.2.4.

4.2 Integration tests of different functions, short discriptions.

4.2.1 Phase 1: Task Administration

- **Initializing components**: The function *init_kernel()*, explaind in section 2.1.1, is tested. Initializing of *ReadyList*, *WaitingList*, and *TimerList* is also tested.
- **Creating tasks when kernel is in *INIT* mode**: Creates multiple task with different deadlines. This also checks if the implementation of *insert_descending()*, which is explained in section 3.1.4, is done correctly. The TCB is also tested in this stage. If it did save the actual information wanted.
- **Creating tasks when the kernel is in *RUN* mode**: The recursion is tested here when you use recursion, what happens, and how does the kernel handle them.

4.2.2 Phase 2: Inter-Process Communication

- **Initializing components**: Tests if the implementation of *create_mailbox()* is correct. Also, the removal of the mailbox is tested.
- **tests on an empty mailbox**: are done using the function *receive_no_wait()*. The mailbox was then empty.
- **Test the async functionalety**: By using *send_no_wait()* multiple times with different data. This tests if the mailbox sorts them correctly. After that *receive_no_wait()* is used for the same times. This tests the FIFO principle.
- **Test the sync functionalety**: By using *send_wait()* multiple times with different data. This tests if the *WaitingList* is working correctly. FIFO is also tested here.

4.2.3 Phase 3: Timing Functions

This mainly tests the *TimerList* and *WaitingList*. In the first one, we test if the task when calling *wait()* stays in *TimerList* for that given time. Tests if the scheduler knows when DEADLINES are reached while on either list and if it knows what to do.

5 Conclusion

In Conclusion, the result is RTMK which has (5) basic features, creating tasks, terminating tasks, sync and async communication, and scheduling. This was achieved on a SAM3X8E processor using the IAR workbench for embedded development [3]. This project demonstrated that working with operating systems requires developers to have a broad understanding of many fields, such as data structures, search and sort algorithms, memory management, and unit testing. The primary data structure for the RTMK is the double-linked list. Some difficulties were found during the development process. The first one was debugging. Debugging required a physical hardware kit that was not available at all times for all the developers working on the project. That made the process a little bit slower. Another difficulty or problem was volatile memory. This problem was encountered in the first lab, where variables were vanishing. That problem was solved by adding an extra if statement, which checks if the variables exist. A beneficial tool was git, a version control system [4]. It helped us track the changes, go back in history, and make branches, i.e., project organization.

References

[1] M. Fazelii, "Lecture 2: Introduction to operating system," 2023, last accessed 12 March 2023.

[2] M. Fazeli, "Lecture 3: Processes," 2023, last accessed 12 March 2023.

[3] I. S. Group, "Iar embedded workbench for arm," 2023, last accessed 12 March 2023. [Online]. Available: https://www.iar.com/products/architectures/arm/iar-embedded-workbench-for-arm/

[4] git scm, "Git–fast-version-control," 2023, last accessed 12 March 2023. [Online]. Available: https://git-scm.com/about

List of Figures