# Computer systems engineering II

RTMK — Real-time Microkernel

Gr 20

Khaled Mhjazi & Hind Dakkeh

2023-03-05

## Abstract

The Real-time microkernel (RTMK) project involved the development of a microkernel using an Arduino Due, a J-Link debugger, and IAR software. The project was a culmination of three separate labs, with the report discussing the operation of the RTMK, the use of assisting libraries such as doubly linked lists and queues, testing procedures. The RTMK's importance as a mediator between hardware and the operating system was emphasized, and its ability to create, manage, and remove tasks was explored. The report also highlighted the crucial role of testing in validating the RTMK's functionality. The project provided valuable experience in microkernel development and offered insights into the challenges and opportunities in this field.

## Introduction

The Real-time Microkernel (RTMK) project is a critical component of the Computer Systems Engineering II course, utilizing an Arduino Due with an Atmel SAM3X8E ARM Cortex-M3 CPU. The project aims to develop a functional RTMK, which forms the basis of an operating system by managing tasks, facilitating inter-process communication, and handling timing functions. The kernel is an essential part of any operating system, responsible for tasks such as memory management and task management, serving as the intermediary between the user interface and hardware in the system.

The course comprises three labs, each focusing on a particular aspect of the RTMK. Task Administration lab involves creating the kernel that allows for the creation, execution, and termination of tasks. This consists of implementing a doubly linked list to manage tasks using predefined structures. Inter-Process Communication lab develops a way to facilitate inter-process communication by implementing mailboxes. The mailboxes serve to send data between tasks either synchronously or asynchronously. Queues are used to store and manage the mail in the mailboxes. The timing functions lab involves utilizing timing functions to enable the switching of contexts when tasks are blocked or put to sleep. This ensures that tasks are scheduled correctly and that the RTMK runs smoothly.

## Operation of the RTMK

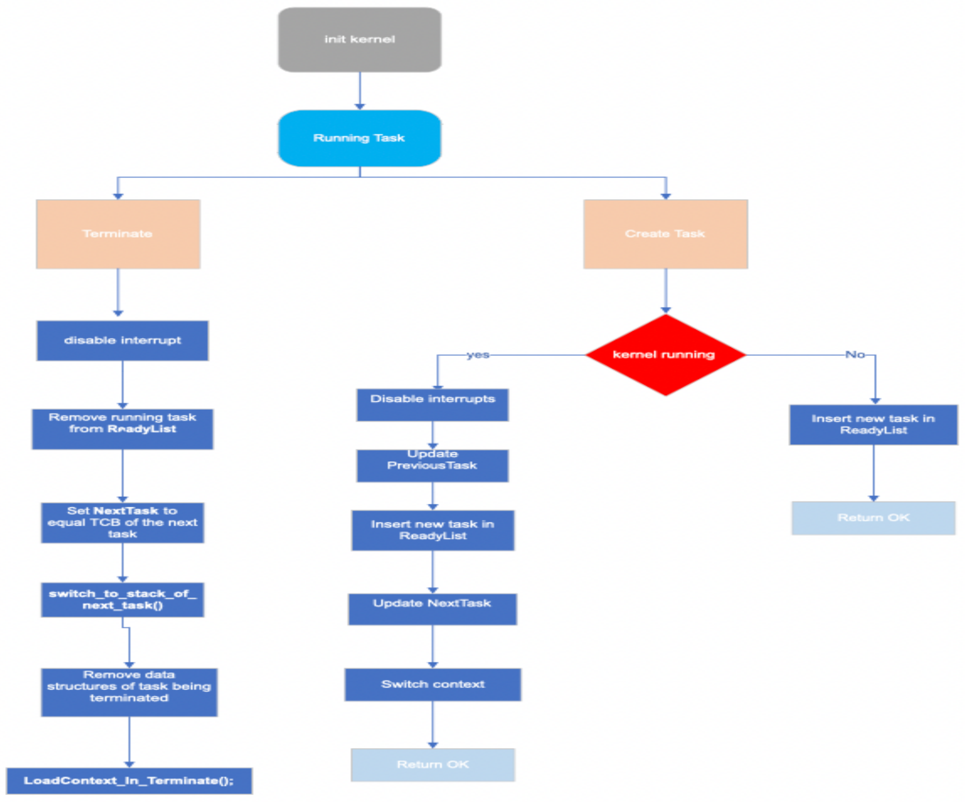
The RTMK has a structure with three lists that hold tasks waiting to be executed: a ReadyList for tasks ready to run, a TimerList for tasks sleeping for a specific time, and a WaitingList for tasks blocked while waiting for a message to be received. Each task has its own set of registers and a stack to hold its information while running or in a blocking/sleeping state. This information is stored in the TCB (Task Control Block), which includes details about the task's deadline, registers, a program counter (PC), and stack pointer (SP). The TCB is essential for continuing the execution of a sleeping or waiting task. Additionally, each task has pointers to its neighboring tasks and a "pMessage" pointer used for communication between tasks. A "nTCnt" (Timer Counter) is also used when a task is sleeping to determine how long it should wait in the TimerList. To manage tasks, the RTMK has a task administration structure in place.

## Task Administration

At the beginning of RTMK, initializes the kernel by allocating memory for the ready, timer, and waiting lists. An "idle task" is created with a maximum deadline to be available in case no other task is running in the ReadyList. The kernel mode is set to "INIT," and the "create task" function creates a new job for RTMK. During task creation, memory is allocated for the task, and the TCB is set with the necessary values. Once the task is created, further details are stored in a node within the system.

When a task is created while the kernel is running, interrupts are disabled, and the Previous task pointer is updated. The task is inserted into the ReadyList, and the Next task pointer is updated before a context switch occurs. However, if the kernel is in the "INIT" state, the task is simply inserted into the ReadyList without any context switching.

The ReadyList is where tasks wait to run, and they are sorted based on their deadline, with the shortest deadline being first. The task at the head of the ReadyList is the one that runs. When its command terminate is called, the head of the ReadyList is replaced with the next task in the list, and the first task is removed. The global variable "Next task" is currently being worked on, and it moves one step further, while "Previous task" is the one just worked on [2].

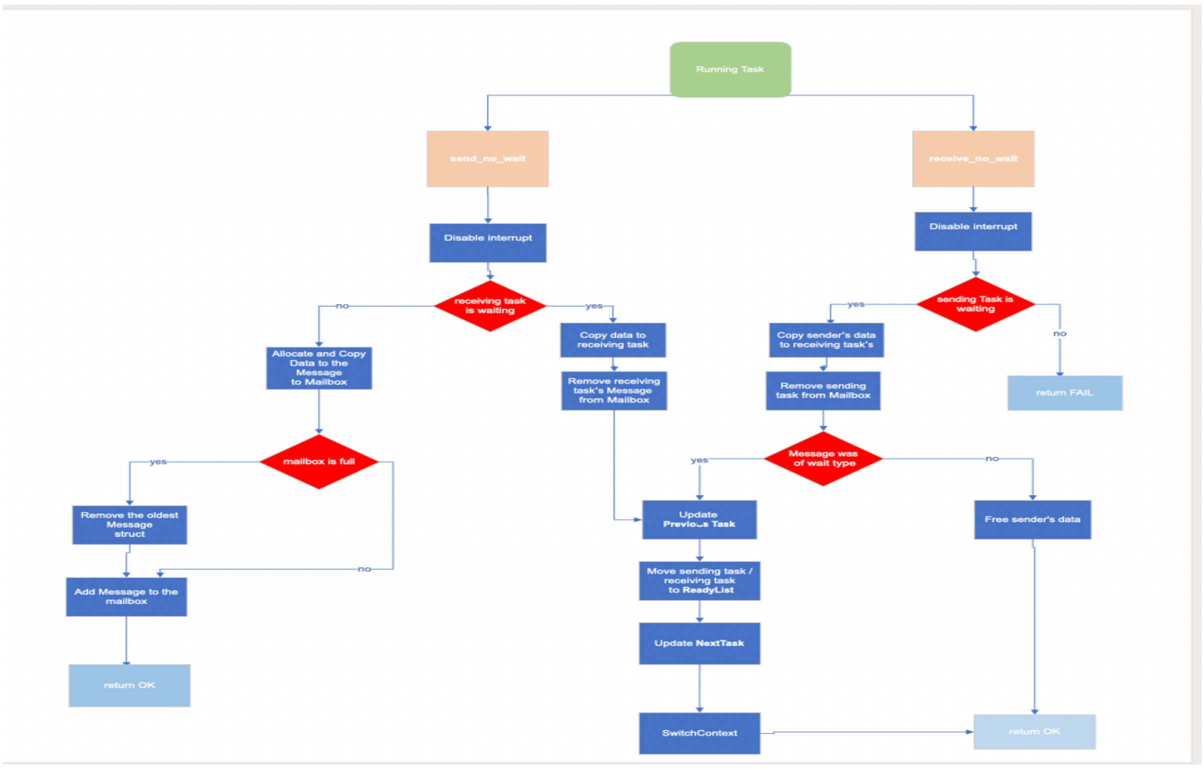

init kernel
Running Task
Terminate
Create Task
disable interrupt
Remove running task from ReadyList
Set NextTask to equal TCB of the next task
switch_to_stack_of_next_task()
Remove data structures of task being terminated
LoadContext_In_Terminate();
kernel running
yes
No
Disable interrupts
Update PreviousTask
Insert new task in ReadyList
Update NextTask
Switch context
Return OK
Insert new task in ReadyList
Return OK

## Inter-process communication

The project involves multiple doubly linked lists. The main objective is to move tasks between lists depending on their states, such as the Ready List for tasks ready for execution, the Waiting List for tasks waiting, and the Timer List for tasks that are in sleeping mode. To facilitate communication between these lists, "Mailboxes" are used, which are created using the "create_mailbox" method that involves allocating memory and creating a structure. The mailboxes are removed using the "remove_mailbox" method, which erases them from memory. There are four methods for communicating through these mailboxes: "send_wait," "receive_wait," "send_no_wait," and "receive_no_wait."

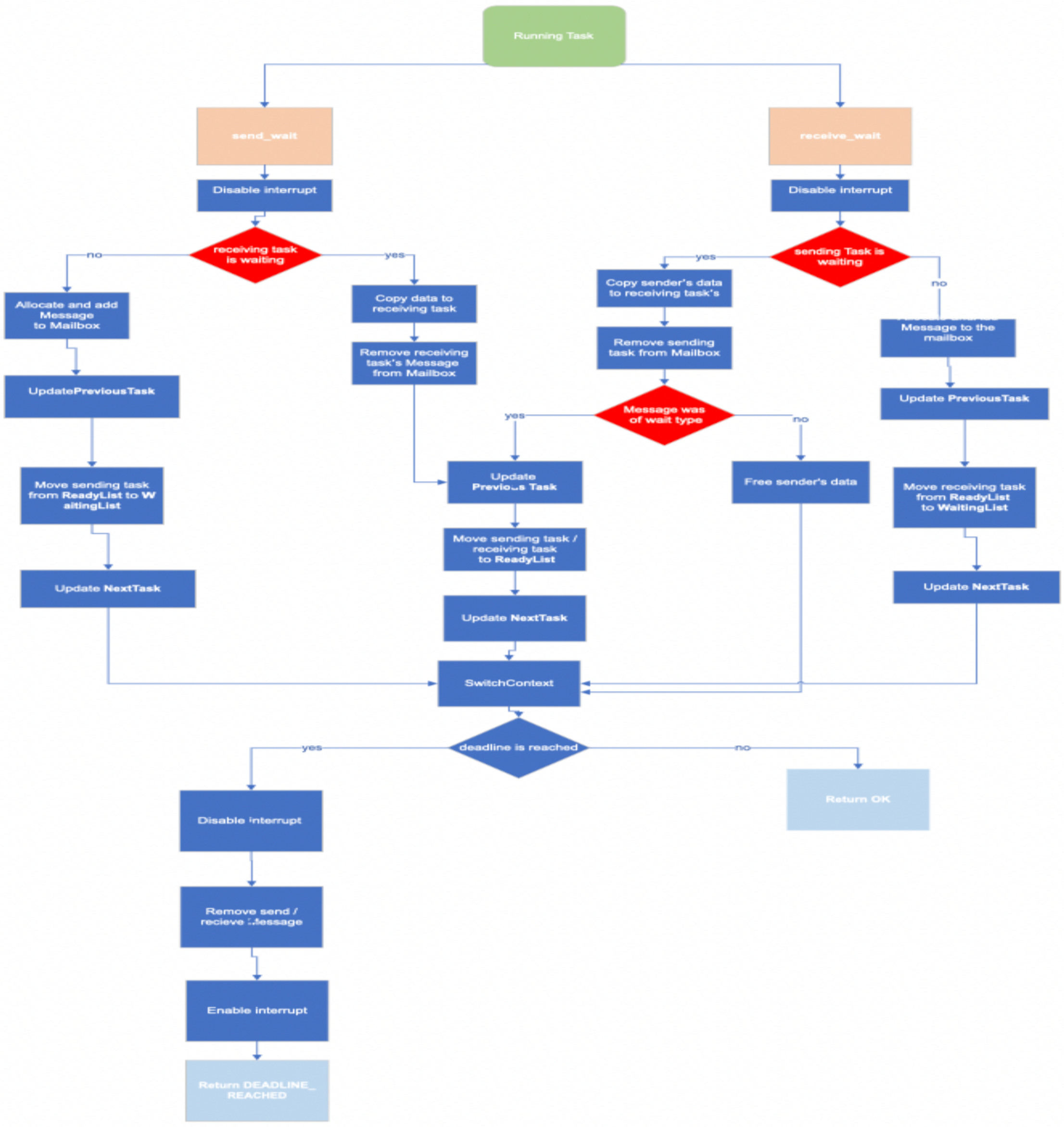
## Asynchronous communication

Asynchronous communication is a form of communication where the sending and receiving tasks are not synchronized. This means that the sending task does not need to wait for the receiving task to receive the message before executing other tasks. The "send_no_wait" method is an example of asynchronous communication. When a task is sent using this method, the sending task can continue executing immediately after sending the information to the mailbox without waiting for a response from the receiving task. Similarly, "receive_no_wait" is a method that retrieves a message from the mailbox only if one message is already waiting and inserts it in the ReadyList. If no message is waiting, the method will return a failure. Overall, asynchronous communication provides greater flexibility and efficiency in executing tasks by allowing them to continue running without waiting for each other to complete them [3].


Running Task
send_no_wait
receive_no_wait
Disable interrupt
Disable interrupt
receiving task is waiting
no
yes
sending Task is waiting
yes
no
Allocate and Copy Data to the Message to Mailbox
Copy data to receiving task
Remove receiving task's Message from Mailbox
Copy sender's data to receiving task's
Remove sending task from Mailbox
return FAIL
mailbox is full
yes
no
Message was of wait type
yes
no
Remove the oldest Message struct
Add Message to the mailbox
return OK
Update Previo..s Task
Move sending task / receiving task to ReadyList
Update NextTask
SwitchContext
Free sender's data
return OK

## Synchronous communication

"send_wait" and "receive_wait" are two methods of synchronous communication. Synchronous communication is when the sending and receiving tasks are synchronized, meaning they are both waiting for each other to complete a particular task before continuing. In the case of "send_wait," when a task calls this method, it will be blocked until the corresponding "receive_wait" or "receive_no_wait" method is called by the receiving task. This is because the sending task is waiting for a response from the receiving task and cannot continue executing until it receives that response. Once the synchronous communication has occurred, and the message has been received, the task is moved from the "Waiting List" to the "Ready List" and becomes executable. This movement between the lists allows the task to be executed once it has been received and processed. Once the synchronous communication has taken place and the task is moved to the "ReadyList," the deadline check is performed. If the deadline has been reached, the message is dequeued, and the task returns a "DEADLINE_REACHED" message or returns an "OK" message [3].

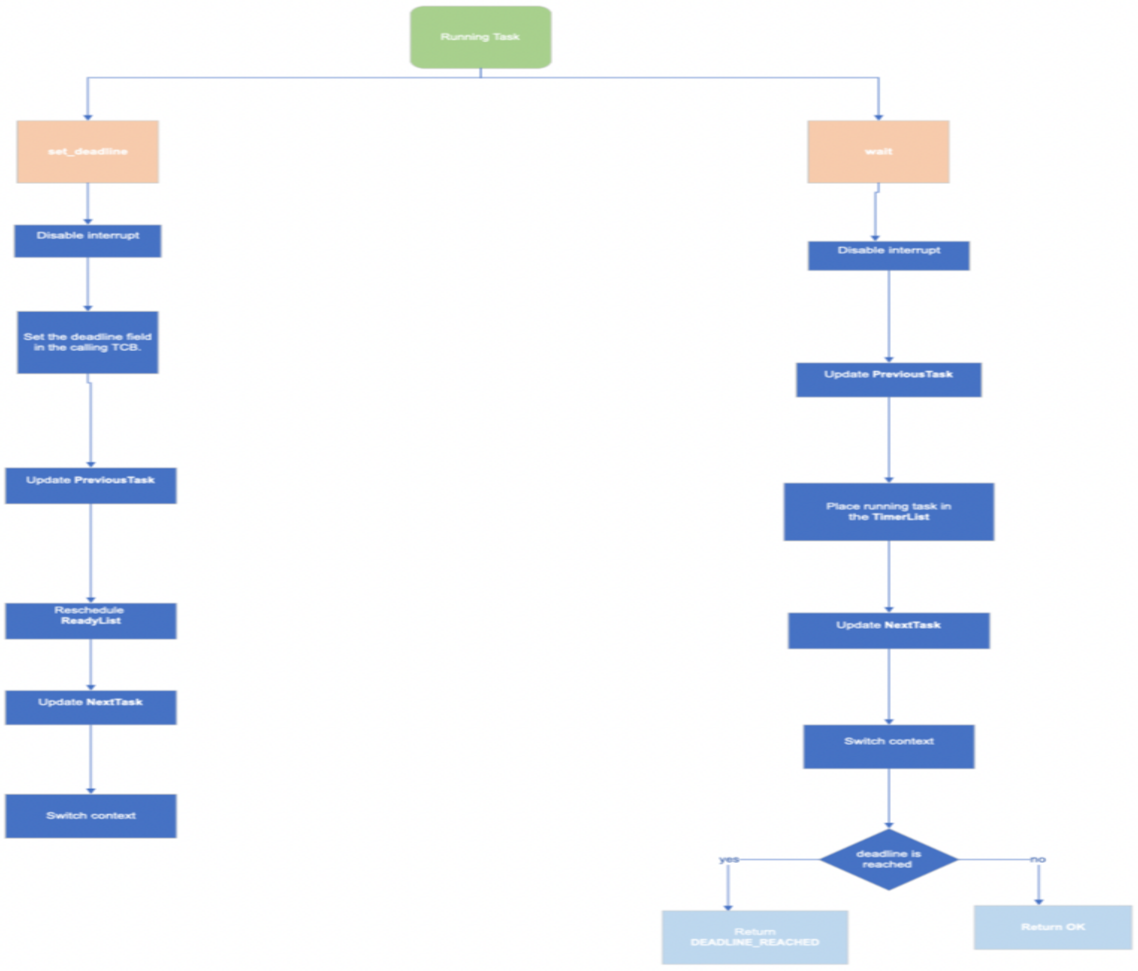


## Timing Functions

"set_deadline" setting a deadline for a task involves updating the TCB with the new deadline value, reinserting the task into the ReadyList based on its new deadline value, updating the pointers to the previous and next tasks in the list, and switching the context to the next task with the earliest deadline.

The “wait” method is a way to put a task to sleep for a set amount of time. This can be useful when a task needs to wait for some event to occur, such as data being available for processing or for a certain amount of time to pass before continuing with its work. To put the task in sleeping mode, the wait method first updates the Previous pointer to point to the currently running task. This is important because it allows the scheduler to know which task was running before the current task was put to sleep. Next, the currently running task is removed from the ReadyList and added to the TimerList . The TimerList is a list of tasks currently sleeping or waiting for some event to occur. The sleeping time is stored in the nTCnt parameter of the TCB for the sleeping task [4].

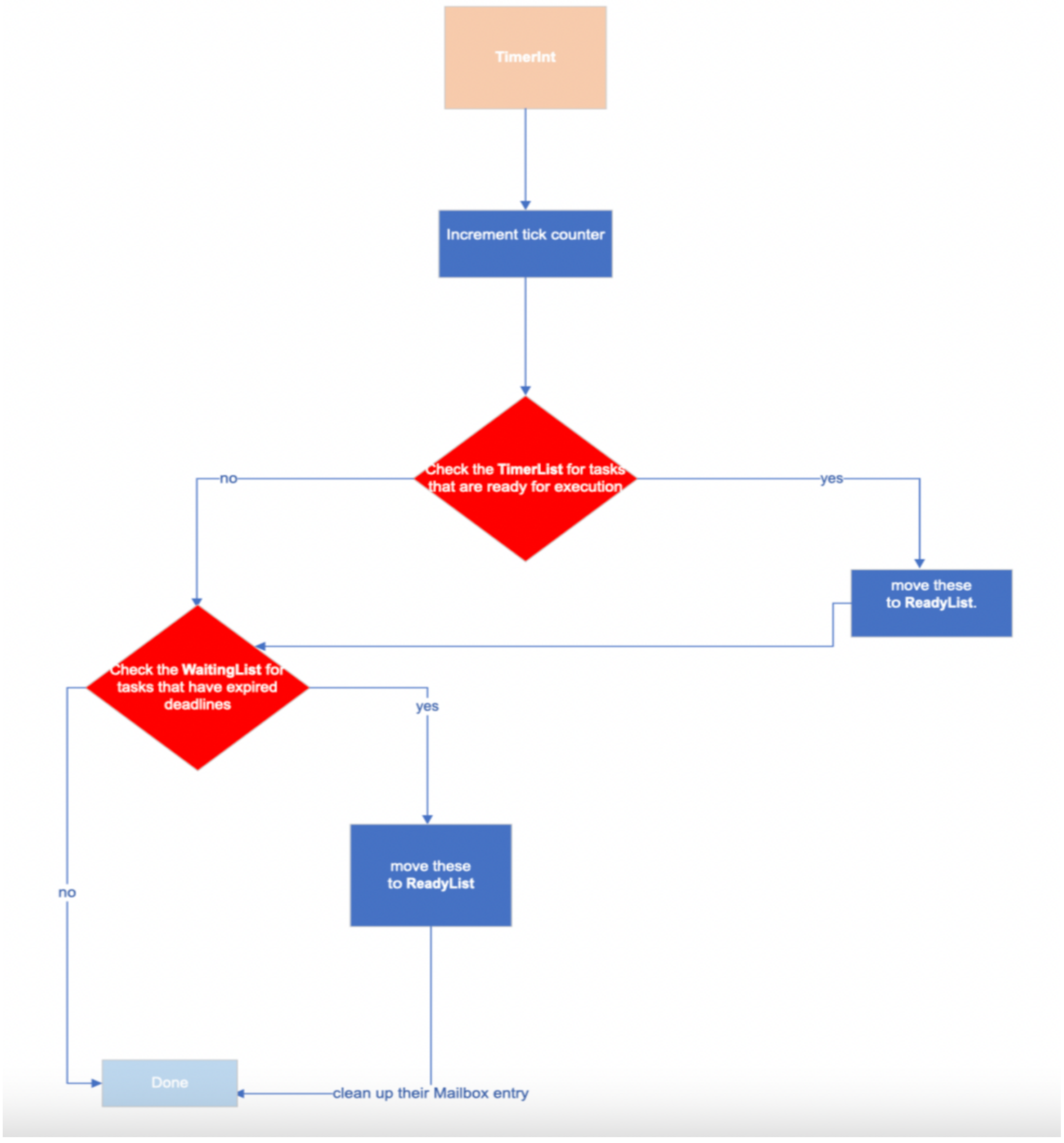
The "set_ticks" method is used to set the value of the tick counter. The "ticks" method simply returns the current value of the tick counter. Finally, the "deadline" method returns the deadline value for the currently running task.



## Timerint

The scheduling process is responsible for managing the execution of tasks based on their deadlines. In this system, the scheduling process is called by the systick interrupt, which occurs at regular intervals. When the systick interrupt occurs, the TimerInt method is called. The TimerInt method first increments the global variable Ticks by one. This variable is used to track the passage of time and to determine when tasks should be moved from the TimerList or WaitingList to the ReadyList.

The TimerInt method checks each task in the TimerList to see if it has finished sleeping or if its deadline has been reached. If the task has finished sleeping or the deadline is reached, it will move to the ReadyList. TimerInt method also checks each task in the WaitingList to see if its deadline has been reached. If the deadline has been reached, the task is moved to the ReadyList.


TimerInt
Increment tick counter
Check the TimerList for tasks that are ready for execution
no
yes
move these to ReadyList.
Check the WaitingList for tasks that have expired deadlines
yes
no
move these to ReadyList
Done
clean up their Mailbox entry

## Assisting libraries

Real-time microkernels are designed to handle a wide range of tasks with high levels of efficiency and responsiveness and often use specialized data structures to manage and manipulate data in a way optimized for real-time performance. These data structures are often implemented in C, which is a popular programming language for real-time microkernel development due to its efficiency and low-level access to system resources. C provides access to features like pointers and memory management, which are essential for implementing efficient data structures in real-time microkernels.

### Doubly Linked List

In a real-time kernel, tasks are often managed using lists to keep track of their state. For example, a task can be in the ready state, waiting state, or timer state. Each of these states is represented by a separate list.

To implement these lists, a data structure called a doubly-linked list is often used. This type of list is similar to a linked list, but it has an additional pointer in each node that points to the previous node in the list. A doubly-linked list allows for efficient insertion and deletion of nodes, as well as easy traversal of the list in both directions. This makes it a useful data structure for managing tasks in a real-time kernel.

To implement a doubly-linked list, a struct is typically used to represent each node in the list. This struct contains a data part and two pointers: one that points to the next node in the list and one that points to the previous node in the list. The list has two global variables pointing at the beginning of the list head and at the end tail [1].

### Insertion Sort

The insertion sort algorithm is a simple sorting algorithm that works by iteratively inserting new elements into a sorted sublist until all elements are sorted. In the context of a real-time kernel, the insertion sort algorithm can be used to sort tasks based on their deadlines, which is a common requirement in real-time systems. When a new task is added to the ready list, it is inserted into the list using the insertion sort algorithm. Specifically, the new task is compared to the other tasks in the list, starting from the beginning of the list. The comparison is based on the deadline of each task, which is a value that represents the time by which the task must be completed. If the new task has a less deadline than the current task being examined, it is inserted into the list before the current task. If the new task has a higher deadline than all the other tasks in the list, it is inserted at the end of the list.

### Queue

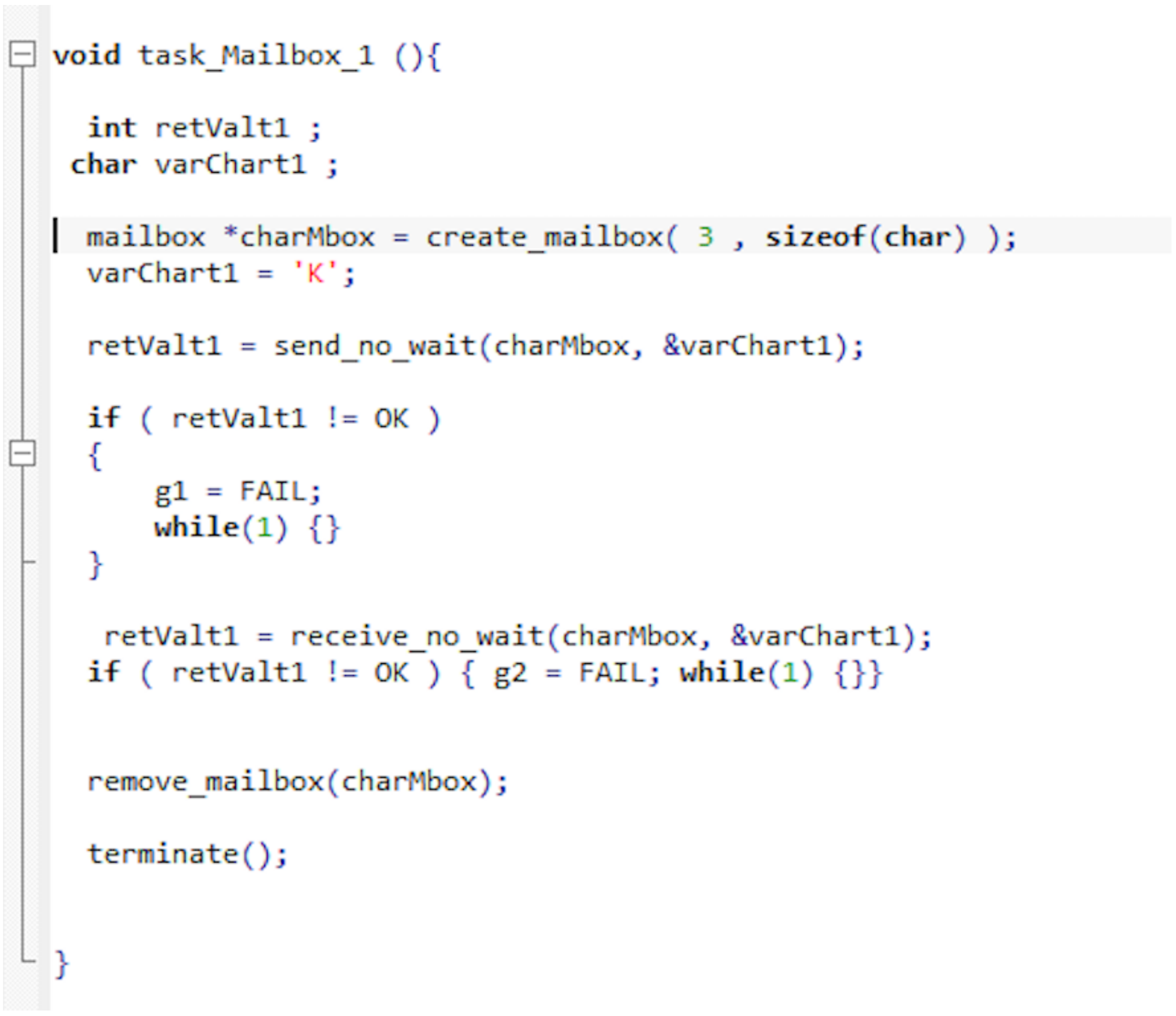
The mailbox is implemented as a queue, which means that messages are enqueued at the back of the queue and dequeued from the front of the queue in a First In First Out (FIFO) order. The enqueue operation increases the number of messages in the mailbox, while the dequeue operation decreases it. This is a simple and efficient way to manage messages in the mailbox.

# Testing

## *Unit tests*

### Remove Mailbox:

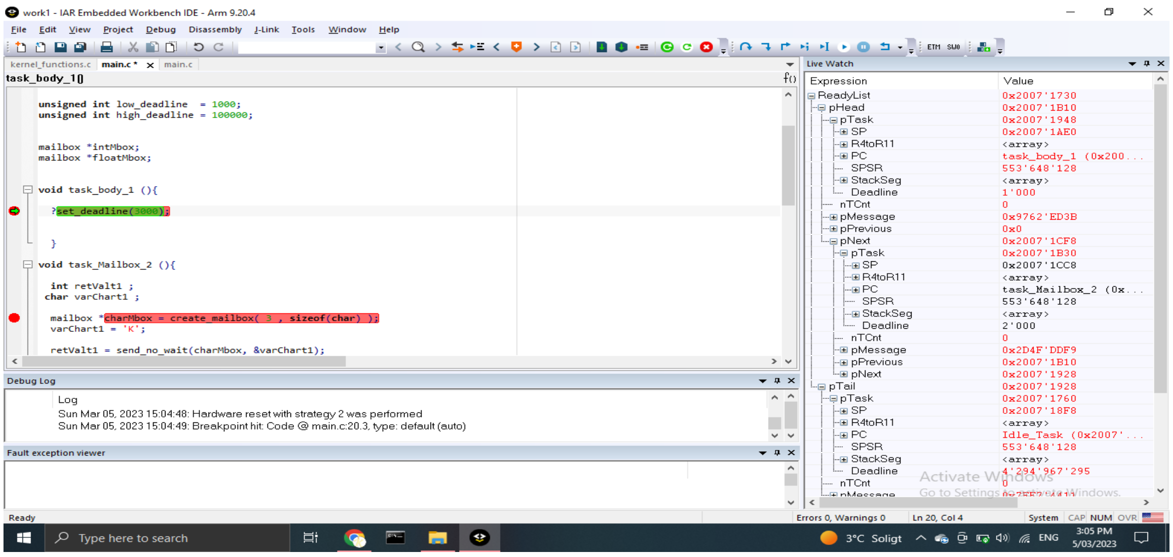
This test creates a mailbox, sends a character to the mailbox, receives a character from the mailbox, and then removes the mailbox and terminates the task. If any errors occur during the send or receive operations, the task enters an infinite loop and sets a global variable to indicate failure.



```
void task_Mailbox_1 (){

   int retValt1 ;
  char varChart1 ;

   mailbox *charMbox = create_mailbox( 3 , sizeof(char) );
   varChart1 = 'K';

   retValt1 = send_no_wait(charMbox, &varChart1);

   if ( retValt1 != OK )
   {
       g1 = FAIL;
       while(1) {}
   }

    retValt1 = receive_no_wait(charMbox, &varChart1);
   if ( retValt1 != OK ) { g2 = FAIL; while(1) {}}


   remove_mailbox(charMbox);

   terminate();


}
```

## Set deadline:

We create two tasks, first one called task_body1 with Deadline 1000 and the other called Mailbox2 with Deadline 2000. Inside the first one, we change the Deadline of the first task from 1000 to 3000. Thus, the function set deadline will reschedule the task in Readylist by placing it second after the task that has a Deadline of 2000.


work1 - IAR Embedded Workbench IDE - Arm 9.20.4

task_body_1()

```
unsigned int low_deadline  = 1000;
unsigned int high_deadline = 100000;


mailbox *intMbox;
mailbox *floatMbox;

void task_body_1 (){

  ?set_deadline(3000);


  }
void task_Mailbox_2 (){

  int retValt1 ;
 char varChart1 ;

  mailbox *charMbox = create_mailbox( 3 , sizeof(char) );
  varChart1 = 'K';

  retValt1 = send_no_wait(charMbox, &varChart1);
```

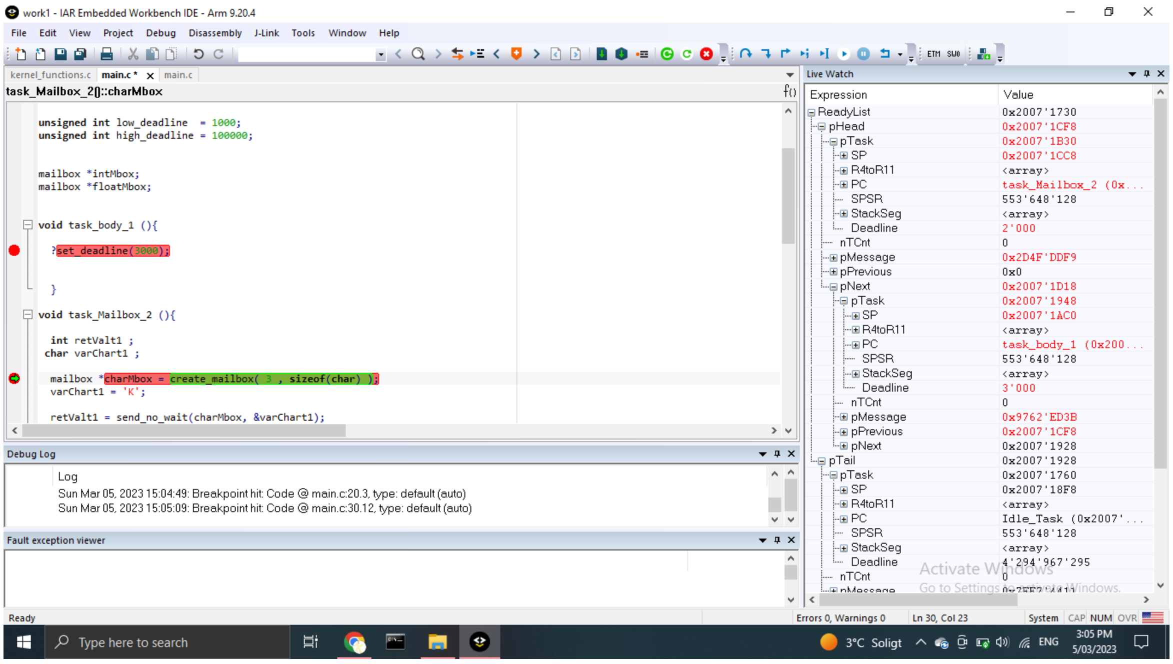
Debug Log

```
Log
Sun Mar 05, 2023 15:04:48: Hardware reset with strategy 2 was performed
Sun Mar 05, 2023 15:04:49: Breakpoint hit: Code @ main.c:20.3, type: default (auto)
```

Live Watch

```
Expression              Value
⊟ ReadyList             0x2007'1730
  ⊟ pHead               0x2007'1B10
    ⊟ pTask             0x2007'1948
      ⊞ SP              0x2007'1AE0
      ⊞ R4toR11         <array>
      ⊞ PC              task_body_1 (0x200...
        SPSR            553'648'128
      ⊞ StackSeg        <array>
        Deadline        1'000
      nTCnt             0
    ⊞ pMessage          0x9762'ED3B
    ⊞ pPrevious         0x0
    ⊟ pNext             0x2007'1CF8
      ⊟ pTask           0x2007'1B30
        ⊞ SP            0x2007'1CC8
        ⊞ R4toR11       <array>
        ⊞ PC            task_Mailbox_2 (0x...
          SPSR          553'648'128
        ⊞ StackSeg      <array>
          Deadline      2'000
        nTCnt           0
      ⊞ pMessage        0x2D4F'DDF9
      ⊞ pPrevious       0x2007'1B10
      ⊞ pNext           0x2007'1928
  ⊟ pTail               0x2007'1928
    ⊟ pTask             0x2007'1760
      ⊞ SP              0x2007'18F8
      ⊞ R4toR11         <array>
      ⊞ PC              Idle_Task (0x2007'...
        SPSR            553'648'128
      ⊞ StackSeg        <array>
        Deadline        4'294'967'295
      nTCnt             0
```

Ready | Errors 0, Warnings 0 | Ln 20, Col 4


work1 - IAR Embedded Workbench IDE - Arm 9.20.4

task_Mailbox_2()::charMbox

```
unsigned int low_deadline  = 1000;
unsigned int high_deadline = 100000;


mailbox *intMbox;
mailbox *floatMbox;

void task_body_1 (){

  ?set_deadline(3000);


  }
void task_Mailbox_2 (){

  int retValt1 ;
 char varChart1 ;

  mailbox *charMbox = create_mailbox( 3 , sizeof(char) );
  varChart1 = 'K';

  retValt1 = send_no_wait(charMbox, &varChart1);
```

Debug Log

```
Log
Sun Mar 05, 2023 15:04:49: Breakpoint hit: Code @ main.c:20.3, type: default (auto)
Sun Mar 05, 2023 15:05:09: Breakpoint hit: Code @ main.c:30.12, type: default (auto)
```

Live Watch

```
Expression              Value
⊟ ReadyList             0x2007'1730
  ⊟ pHead               0x2007'1CF8
    ⊟ pTask             0x2007'1B30
      ⊞ SP              0x2007'1CC8
      ⊞ R4toR11         <array>
      ⊞ PC              task_Mailbox_2 (0x...
        SPSR            553'648'128
      ⊞ StackSeg        <array>
        Deadline        2'000
      nTCnt             0
    ⊞ pMessage          0x2D4F'DDF9
    ⊞ pPrevious         0x0
    ⊟ pNext             0x2007'1D18
      ⊟ pTask           0x2007'1948
        ⊞ SP            0x2007'1AC0
        ⊞ R4toR11       <array>
        ⊞ PC            task_body_1 (0x200...
          SPSR          553'648'128
        ⊞ StackSeg      <array>
          Deadline      3'000
        nTCnt           0
      ⊞ pMessage        0x9762'ED3B
      ⊞ pPrevious       0x2007'1CF8
      ⊞ pNext           0x2007'1928
  ⊟ pTail               0x2007'1928
    ⊟ pTask             0x2007'1760
      ⊞ SP              0x2007'18F8
      ⊞ R4toR11         <array>
      ⊞ PC              Idle_Task (0x2007'...
        SPSR            553'648'128
      ⊞ StackSeg        <array>
        Deadline        4'294'967'295
      nTCnt             0
```

Ready | Errors 0, Warnings 0 | Ln 30, Col 23

## Integration tests

### Phase 1: Task Administration

In the first phase of the project, the focus was on task administration, which involves handling tasks in both their initial state (init state) and while they are running (running state). The Phase include a multi-step process of configuring a program's environment and kernel structure. an "idle task" was created and inserted into a list called the "Readylist". The Readylist is a list of tasks that are ready to run the tasks. Other tasks were also added to this same list, and then the tasks were executed to test the system's functionality [2].

### Phase 2: Inter-Process Communication

The integrated test for mailboxes involves testing both synchronous and asynchronous sending and receiving of messages through the mailboxes. Synchronous sending and receiving mean that a task will wait for a response before proceeding, while asynchronous sending and receiving means that a task will continue to execute while waiting for a response. In addition to testing the sending and receiving of messages, the test also includes checking that the mailbox is empty at the beginning and after sending and receiving messages. This ensures that the mailbox is properly managing messages and not allowing messages to accumulate unnecessarily. The test involves creating a mailbox and ensuring that it can successfully accept input. This is important to confirm that the mailbox is properly configured and can be used by tasks to communicate with each other [3].

### Phase 3: Timing Functions

In the last phase of the testing process, the focus is on developing and testing the scheduler, which is responsible for managing and executing tasks in the system. The final lab involves testing the functionality of the scheduler to ensure that it is properly scheduling tasks and running them at the correct times. There is a test for detecting when a task has exceeded its deadline, which is important for ensuring that tasks are completed in a timely manner [4].

## Conclusion

The project's primary goals were to establish a functional real-time microkernel that included a timer, waiting and ready list, and mailboxes to enable task communication. The kernel acts as a mediator between hardware and the operating system, enabling the creation, management, and deletion of tasks with varying states depending on the application's needs. To accomplish this, we employed supporting libraries such as doubly linked lists and queues with insertion sort to integrate the different components of the kernel. The project and labs provided us with a comprehensive understanding of the concepts involved in creating and managing a real-time microkernel. We gained valuable experience in identifying and addressing potential faults, managing memory, and integrating various components.

## References

[1] https://www.softwaretestinghelp.com/doubly-linked-list-2/

[2] H Ali, M Fazeli, and O Andersson. Lab 1 - Task Administration, 2023.

[3] H Ali, M Fazeli, and O Andersson. Lab 2 - Inter Process Communication, 2023.

[4] H Ali, M Fazeli, and O Andersson. Lab 3 - Timing Functions, 2023.