Joakim Svensson & Mohammad Hammoud

Table of content

1. Introduction

The initial part of the project included setting up the initialization for the kernel, including task management and configuring how it runs and terminates tasks. It also included how the tasks are queued in order of deadlines utilizing a linked list structure called Ready List. The second part is communication between tasks using mailboxes including them being blocked while waiting for a reply or not. The third part allows tasks to sleep for a certain duration in Timer List and then return to Ready List.

Operation of the implemented RMTK

Information about how it handles processes and scheduling.

Assisting Libraries

Information about the additional libraries implemented to perform needed tasks.

Integration Testing

Testing the created functions functionality

Functional Testing

Testing the functionality of the whole RMTK

2. Operation of Implemented RMTK

2.1 Operation

The RMTK functions by having linked lists that tell the kernel how to operate. The Ready List tells the kernel which task should be run, it also tells which tasks are next in line. It has pointers to the Head list object and the Tail list object.

The head list object is the currently running task (if the kernel is in running mode) or it is the shortest task. The tail pointer points to the list object containing the last task which is the idle task. The idle task is a dummy task in the case when all other tasks have been executed it just executes while (1) forever which makes the kernel stall until shut down. Each list struct contains pointers to the next and previous list objects in the list, they also contain the task that potentially could be run.

The task itself is of struct Thread Control Block, the TCB contains information about the stack, it has a stack pointer to the start of the task that should be ran, in the case of idle task it contains a pointer to the stack location where while (1) is located (*PC). It also includes more information about the stack, such as how large the stack segmentation is, it also contains registers R4 to R11 for when the kernel decides to switch context.

It also contains the deadline of said task, which decides where in the Ready List the list object should be located. The Ready List is ordered by the deadline, the shorter the deadline the earlier the task is located in the list, this is in order for the tasks that have less time left to be executed first.

2.2 Init Kernel

Init_Kernel initializes the kernel by setting the List Data Structures and creating the Idle Task with the maximum deadline possible. Once the kernel has been initialized and the Idle Task has been created, the tasks that actually provide value to the RMTK should be created and added to the Ready List, otherwise the RMTK is pointless as it executes only the idle task, which would is equivalent to sitting and doing nothing but still wasting power and most of all potential. The kernel also has a very necessary function in Terminate which looks at the currently running task and frees it from memory. Before doing that it sets the Global pointer NextTask to the pointer of the task of the list object which is the next task of the head of Ready List.

Once all the processes the RMTK should run have been set in Ready List the Kernel is ready for the run function which starts the execution of all the processes in the Ready List order with the smallest deadline first, once all that is done it will enter the idle task and need to be shut down.

In order for the kernel to function with some choices for scheduling more lists are needed. For this purpose it also has a Waiting List which allows for tasks to be blocked for certain amounts of time so that other tasks can be executed first.
The Waiting List is structured similarly to The Ready List, it also has a Head and a Tail pointer, it is also structured in the same way as Ready List with deadline being the way it is enqueued. The task with the lowest deadline will therefore be the first to exit. It also contains the stack pointer information and information about the task that is blocked in order for it to be able to be restored back to the Ready List later. The Waiting List is also made out of List Objects which have pointers to the next and the last. These list objects are what also contains the same things as Ready List. The List Objects also have additional fields that are msg and nCnt which is used to calculate how long a certain task has been blocked.

2.3 Mailbox Functions

Mailboxes are the communication of the tasks, in order for tasks to be blocked they will need to send a Message that has the fields. The mailbox is constructed differently compared to the lists, it is a first in first out (fifo) list where deadlines don't matter, only the order they were put into the mailbox matters. It also has an empty head pointer that only contains previous to itself and next to it's next list, be that tail or just one in the list. Tail contains nothing from the start, only a next pointer to itself and previous to head. When a message is inserted it is inserted into the tail, which then still points to itself with next. When there are two messages in the mailbox the inserting functions in, send wait receive wait and so on, they will insert the new message in the tail but then push the old tail back making it be head, message, tail that includes a message. Mailboxes contain a head pointer, a tail pointer, nDataSize which is the size that it's data type is such as unsigned int, it could be any type of struct as well. It has information about it's max allowed messages, it also has information about it's current amount of messages, and if there are any blocked messages.

2.4 Mailbox Creation

Mailboxes are created with the create mailbox function which allocates memory for the mailbox using calloc and it also performs a null check in case the memory is full. The function takes the arguments nMessages and unsigned int nDataSize and sets the according pointers to the correct value. It also initializes the head and tail pointers so that head points next to tail and reverse and both of them point to themselves with the pointer that is at the edge, that being next for tail and previous for head.

For removal of mailboxes the function remove_mailbox is used, it takes a mailbox pointer as an argument and checks if the messages are zero, if they are then it proceeds to free the mailbox and returns ok, if the mailbox is not empty then it returns the not empty exception.

2.5 Types of Messages

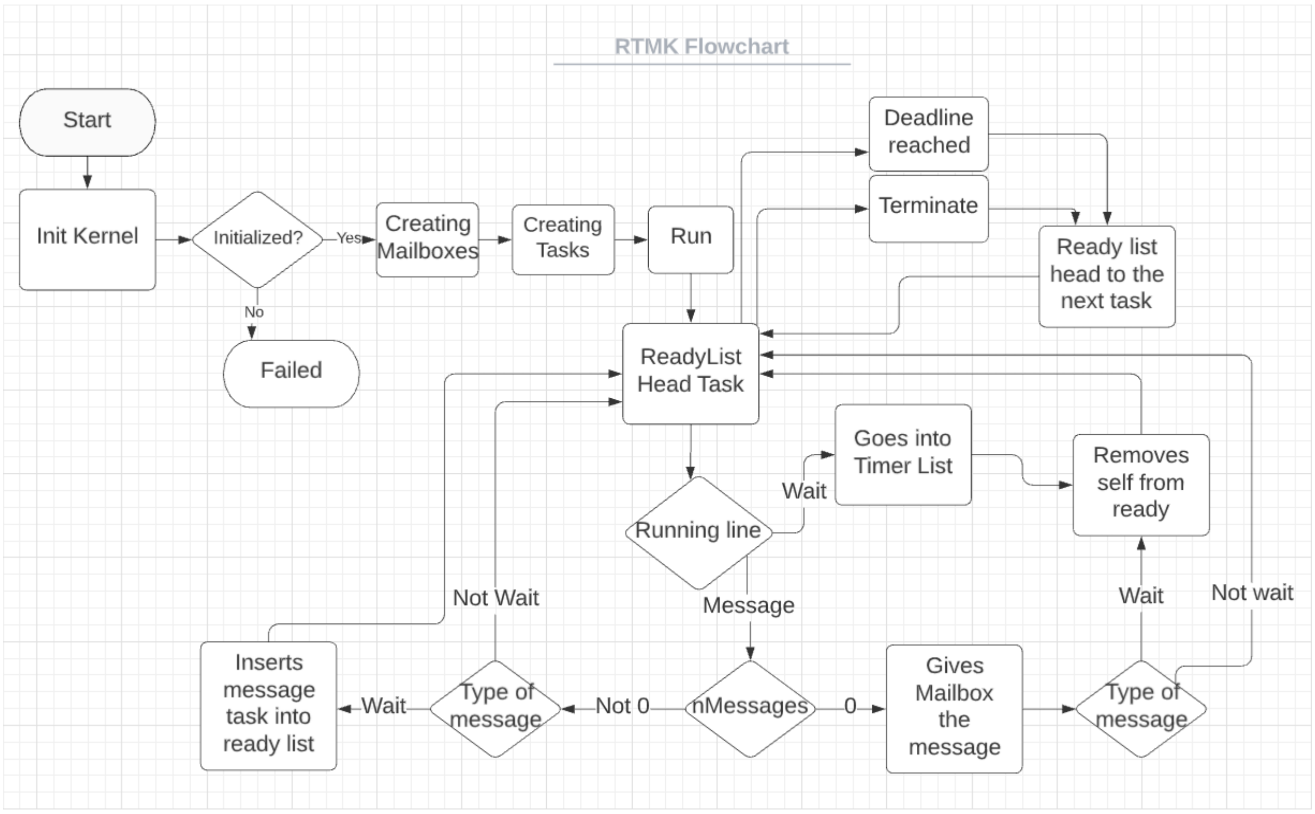
Tasks are moved to mailboxes by using functions such as Send Wait, Send no wait. Send wait makes the currently running task (the one that sent the command) go into blocking mode and go get removed from Ready List and get put into Waiting List, to do this it checks the mail boxes information such as how many messages it has. The amount of messages is calculated by receivers lowering the value by one and senders increasing the value by one. If the mailbox doesn't have any message in it at this time then it will allocate memory for a message and then insert the message into the mailbox. After that it will also remove the task from the ready list and switch context to the new ready list head task using switch context. If the mailbox has a receiving message then it will copy it's data to the receiving messages data pointer, then it removes the message from the mailbox.
Send no wait works differently, it doesn't block the task that sends it, meaning it is not stopped from running, so it sends the information in data without waiting for a reply. It also checks if there is a message waiting, if there is one it copies the data from the message to its own data pointer, then it proceeds to free the message and after that it inserts the receiving task into the ready list. If there was no message waiting it allocates memory for the necessary data, including a new message, the data for the message which is copied with memcpy. Then it performs a check to see if the mailbox is full, if it is not it inserts the message and doesn't block itself while also adding to the nMessages to mark that it is a sender.

Receive Wait also checks nMessages in the mailbox that it received as an argument, if the mailbox had any senders then it receives said message, if it was of type send wait then it unblocks the task. If there were no messages it places itself in the mailbox and blocks the running task by placing it in the waiting list. Receive no wait does mostly the same but doesn't block the running task.

2.6 Timing Functions

Timing functions include different ones that return data like ticks and deadline for current task, it has a function that allows the number of current ticks to be set to a chosen value, set deadline for the currently running task. The function TimerInt is called every system tick while interrupts are on and increments the ticks counter, it also checks if there are any sleeping or deadline expired tasks that should be put into the ready list.

This flowchart shows the general rule of how it works, not all the details.



2.7 Overview

The init kernel initializes Ready, Waiting and Timer list, then the system creates the tasks it was told to create, after that it creates the mailboxes that it was told to create. Then it runs when it is told to, it runs the ready list head task which is the one with the lowest deadline, after that the tasks and deadlines can do multiple things. If a message is sent the mailbox is checked and if it is of type wait the task is blocked unless there was a message in the mailbox already. If the task is terminated or if the deadline is reached, the ready list is set to the next and then that is the running task. If the wait function is called then the running task will be set to sleep and the next in ready list will be the new running task and the kernel will switch context. Timer int is called each tick when interrupts are on.

3. Assisting Libraries

An assisting library that was created is extra_functions, which was created for two functions, one that inserts the given task into the given list at a location checked in order where the smallest deadline is first. Another function by extra_functions is remove_from_list which removes the given list object from the list by setting it's next's list object previous pointer to its own previous and same thing but reversed for the previous task.

Another library that was used but just imported was stdlib which has useful functions such as calloc and malloc in order to allocate memory for all the different lists and list objects and tasks. It also has the function memcpy which allows you to copy the data of a certain pointer to another pointer.

4. Integration Testing

```
#include "system_sam3x.h"
#include "at91sam3x8.h"
#include "kernel_functions.h"

unsigned char kernel=1;
unsigned int low_deadline  = 100;
unsigned int high_deadline = 5000;

mailbox *charMbox;


//if it doesn't work it will stay on the same task as
//it will not be removed from the list
void task1(){

  TCB *oldOne = ReadyList->pHead->pTask;
  listobj *theNext = ReadyList->pHead->pNext;
  PreviousTask = ReadyList->pHead->pTask;
  insert_into_list(remove_from_list(ReadyList->pHead)->pTask, TimerList);
  ReadyList->pHead = theNext;
  NextTask = ReadyList->pHead->pTask;
  TCB *newOne = ReadyList->pHead->pTask;
  if (oldOne == newOne)
    while(1); //Removal didn't work
  SwitchContext();

  terminate();
}

void task2(){
  while (1); //Successful
}

void main()
{
  SystemInit();
  SysTick_Config(100000);
  SCB->SHP[((uint32_t)(SysTick_IRQn) & 0xF)-4] =  (0xE0);
  isr_off();

  exception returnValue = init_kernel();
  if (returnValue != OK) {
    kernel = FAIL;
    while(1);
  }

  if ( ReadyList->pHead != ReadyList->pTail ) {
    kernel = FAIL ;}
  if ( WaitingList->pHead != WaitingList->pTail ) {
    kernel = FAIL ;}
  if ( TimerList->pHead != TimerList->pTail ) {
    kernel = FAIL ;}

  if ( kernel != OK ) {
    while(1); //Kernel failed
  }
  charMbox = create_mailbox( 10 ,sizeof(char));

  create_task( task1, low_deadline );
  create_task( task2, 8*high_deadline );

  run();

  while(1){ /* something is wrong with run */}
}
```

5. Functional Testing

Testing of functions: this testing only checks basic functionality and memory issues

```
#include "system_sam3x.h"
#include "at91sam3x8.h"
#include "kernel_functions.h"

unsigned char kernel=1, firstmailfailed=0, mailcreationfailed=0, idle=1, basic=0,
//if any of the firstmailfailed or mailcreationfailed or taskcreationfailed variables are 1
//that means it ran into memory issues
taskcreationfailed=0;
unsigned int low_deadline  = 100;
unsigned int high_deadline = 5000;

int amountOfTasks = 5;
mailbox *charMbox;
mailbox *intMbox;
mailbox *unsignedintMbox;


void task1(){
  int           returnValue1;
  int           intTest1;

  returnValue1 = send_wait( charMbox, &intTest1 ); //basic test task
  if (returnValue1 != OK){
    firstmailfailed = 1; //Message failed to be created
  }

  basic = 1;
  terminate();
}

void task2(){
  int           returnValue2;
  int           intTest2;

  returnValue2 = receive_wait( charMbox, &intTest2 );//also basic test task
  terminate();
}
void task3(){//tests memory and message creation with it's functions and task 4's
  int           returnValue3;
  int           intTest3;
  for (int i = 0; i < 10; i++){
    returnValue3 = send_wait( intMbox, &intTest3 );
  }
  if ( returnValue3 != OK ){
    mailcreationfailed = 1; //failed
  }
  terminate();
}

void task4(){
  int           returnValue4;
  int           intTest4;
  for (int i = 0; i < 10; i++){
    returnValue4 = receive_wait( intMbox, &intTest4 );
  }
  if ( returnValue4 != OK ){
    mailcreationfailed = 1; //failed
  }
  terminate();
}

void main() //main
{
  SystemInit();
  SysTick_Config(100000);
  SCB->SHP[((uint32_t)(SysTick_IRQn) & 0xF)-4] =  (0xE0);
  isr_off();
  exception returnValue = init_kernel();
  if (returnValue != OK) {
    kernel = FAIL;
    while(1);
  }
  if ( ReadyList->pHead != ReadyList->pTail ) {
    kernel = FAIL ;}
  if ( WaitingList->pHead != WaitingList->pTail ) {
    kernel = FAIL ;}
  if ( TimerList->pHead != TimerList->pTail ) {
    kernel = FAIL ;}
  if ( kernel != OK ) {
    while(1); //Kernel failed
  }
  charMbox = create_mailbox( 10 ,sizeof(char));
  intMbox = create_mailbox( 10 ,sizeof(int));
  unsignedintMbox = create_mailbox( 10 ,sizeof(unsigned int));

  create_task( task1, low_deadline );
  create_task( task2, 8*high_deadline );

  for (int i = 0; i < amountOfTasks; i++){
    char result1 = OK;
    char result2 = OK;
    result1 = create_task( task3, (10+i)*high_deadline );
    result2 = create_task( task4, (5000+i)*high_deadline );
    if (result1 != OK || result2 != OK){
      taskcreationfailed = 1;
    }
  }

  run();

  while(1){ /* something is wrong with run */}
}
```