CPR E 458, Real-Time Systems
16 December 2021; Ames Iowa, United States

Real Time Scheduler on TI Robot

EDF & RMS Implementation on MSP432P401R REV D Micro Controller on TI-RSLK MAX Robot Kit

Nicholas Krabbenhoft
CPR E 458 Section 1
Iowa State University
College of Electrical and Computer Engineering
Ames Iowa, United States

***Abstract*— The ability to run multiple jobs on a single Micro Controller is vital to get the full utilization out of components. However having many jobs running on a single controller creates problems of when each job receives processor time and how each job can be scheduled to ensure every job meets it's deadline. RMS (*Rate-monotonic scheduling)* and EDF (*Earliest Deadline First)* scheduling is one method to insure this. This paper takes an in depth look at an implementation method for these two schedulers on the MSP432P401R REV D Micro Controller and considers different practice trade offs that were made in the design process. It will then work through the testing of the schedulers to insure that they are working properly and can handle high loads.**

Keywords—Earliest Deadline First (EDF), Rate-monotonic scheduling (RMS), Real Time, MSP432P401R

I. Introduction

Embedded systems are ubiquitous in the modern world and run some of the most important systems on the planet. These micro-controllers often have to do multiple tasks and they must be relied on to do those tasks. In order to know how many tasks a micro-controller is able to be assigned a schedule for the tasks is needed. This schedule must be able to guarantee that every task is completed on time. Two of the most common algorithms for scheduling are EDF (Earliest Deadline First) and RMS (Rate-monotonic scheduling).

In order to gain a deeper understanding of how these algorithms work in real world applications, this project will go through the process of implementing them and discuss the design designs and testing that is involved in the process. All of this will be done on the MSP432P401R REV D Micro Controller on TI-RSLK MAX Robot Kit from TI (Texas Instruments).

II. Objective

For this project, I will be attempting to schedule multiple arbitrary jobs on a single processor micro-controller. This will give it the ability to appear to do many jobs in parallel rather then being stuck doing a single job. It will also be able to assign processor time to each task in a deterministic way to insure that every task is completed before the given deadline.

A. System

This project uses the MSP432P401R REV D Micro Controller on TI-RSLK MAX Robot Kit from TI to run the scheduler on and a Linux desktop with the Arduino IDE to program the micro-controller. Then the PuTTY program is used to establish serial communications with the board for debugging purposes. All scheduling takes place on the board itself and ideally no general purpose computer is needed to interact with the micro-controller once it is fully programmed.

A couple important notes about the micro controller we are using from its data sheet in order to understand our hardware constrant. It has a "Arm® 32-bit Cortex®-M4F CPU with floating point unit and memory protection unit " and "frequency up to 48 MHz." It also has "up to 256KB of flash main memory," "16KB of flash information memory" "up to 64KB of SRAM," and "32KB of ROM with MSP432™ peripheral driver libraries." [1]

B. Real world applications

Real world applications for real time schedulers is massive. The number of micro-controllers in a single car is reaching the dozens and each of those micro-controllers have multiple tasks that must be scheduled to not interfere with one another. While this robot and it's relatively simple schedule is not on that scale it gives a glimpse at how those real world schedulers work and what considerations have to go into designing them.

C. Objectives and Scope

In order to constrain the project to a reasonable scope, let's first define the requirements that we are going to have for the project.

- Schedule an arbitrary number of jobs
- Each Job should have an arbitrary number of arguments (each job can have different number of arguments)
- The scheduler must implement the EDF and RMS scheduling algorithms
- Easily add more algorithms
- Scheduling failures should be non catastrophic and predictable

- Minimal complex code should be used as this is both a learning experience and should serve as a clear example to the readers

III. Development Steps

A. On computer Development

To give an easy development process and portability (much of the code was written on a laptop while traveling), this project was first designed on and for a standard Linux desktop environment. This allowed for significantly faster development and testing. All of the design iterations and testing discussed in the implementation section was done here and then the code was ported to run on the micro controller.

B. Transferring code to MSP432P401R Micro Controller

Once the code was fully functional, it still needed to be transferred to the micro-controller in order to run and that micro-controller needed to be tested to insure that it was working properly.

To insure that the micro-controller was working properly the functionality test hosted by TI was used [7]. In order to program the robot, I then used the Arduino IDE and the additional libraries provided by TI [8]. The next step to ensure that the micro-controller was working was running several of the demo projects given by the code library. Once this was completed, I used the dancing robots as a reference point to begin transferring over the code implemented on the Linux desktop environment[8].

Thankfully, there were only a few modifications that needed to take palace in the code to move it from the desktop environment to the micro-controller. First, the code was restructured so that the all of the initialization code took place in the setup function and then all of the repeating schedule code was moved to the loop function. Next the printf and puts calls were translated to sprintf and Serial.write calls. The Arduino IDE also broke how my newly created libraries were included, so that had to be re created. Other then that, everything worked fine.

IV. Implementation

A. Overall Program Design

The program is broken into 3 major parts: Setup, Adding Jobs, and Running Jobs. The program begins be initializing the Job queue with the proper comparator function, setting up serial communication with the controlling computer, and insuring all configuration registers for the GPIO pins are set properly. Finally, the program goes into an infinite loop waiting for a button to be pressed to start executing the main loop.

1) Main Loop

```
Main_Loop:
  int new_time = init_time
  int old_time
  while True :
    while queue_has_more:
        old_time = new_time
        new_time = time()
        add_new_jobs(old_time, new_time)
        job = queue_pop
        if job.cost + time() > job.deadline
          error – missed job
          continue
        call job.function(job.args)
```

The main loop consists of the 2 remaining parts and is conceptually very simple. The scheduler will first assign a variable old_time the value of the current time, which would be 0 on the first loop. Then it will iterate through every job seeing if it has arrived for processing between old_time and the new current time and adding them to the queue if true. This is done in the add_new_jobs function in the pseudo code.

Once all jobs are checked, the old_time variable is updated to the current time and the highest priority job is removed from the queue, checked if it is still valid. If it is still valid, it is then executed. This loop is then repeated forever adding and removing jobs to the queue.

2) Initialization Options

a) Defined Options

A large number of configuration options must be done before the code is compiled in the first part of the code where many constants are defined for later use in the code. The #define feature in the C compiler is used to do this to enable quick changes to the code and avoid magical numbers in the cade base itself. The current configuration options are the number of Jobs defined in the numOfJobs variable and what type of scheduling algorithm to use EDF or RMS.

b) Setup() function

The Arduino IDE sets up the program as follows rather then using the standard main function. First it executes the setup() function and then it runs the loop() function within an infinite while loop. This means that any initialization for the micro-controller must take place within the setup() function.

The setup() function in this project does some important tasks. It first setups up the serial output for debugging purposes. After that it initializes the GPIO pins and LED's to be used as outputs to provided indications if the micro-controller is working properly if the board isn't connected by serial console. It then waits for a button to be pressed before beginning execution to make sure it isn't accidentally started. Once started it assigns the initial call time of all the tasks and begins keeping track of time to add more jobs on the queue. Once all of that is done it drops into the main loop.

3) Including Jobs in the Program

The current implementation has 8 jobs included, however a theoretically unbounded number of jobs can be included in the program. Each job needs the actual code to run the job, a set of arguments that currently can not change, and a series of parameters to determine when to run the jobs and how often they need to be ran.

a) Function Arguments

Each job needs to take an char * array[] for the arguments. Due to the fact that every pointer can be cast to an char* in c this allows a theoretically infinite number of arguments. For example, to access an integer held pointed to by the first pointer you would use the following: (*(int *)args[0]). These arguments are held in an char ** array[] which is a global variable in the scheduler file. To initialize, each argument should be declared individually, then their pointers should be cast to a char * and placed in a char * jobXargs[], where X is the job number. Each of these jobXargs[] should then be placed into the overall job args array to be called later.

b) Job Function

Each possible job to be run must be included in the code as a function that can be referenced as a function pointer in the scheduler.c file. This function must take in only a char* [] as arguments as discussed above and return an int.

However, the most important thing for each function is that it has a known upper bound and will not cause a crash. The scheduler code has no way to recover from a faulty job or interrupt a job that runs over it's max time. This means that any faulty job can not be isolated and has a large probability of causing other jobs to fail or bringing down the entire controller.

The upside of this fragility is that each job can assume that it is the only job running on the system and can have complete access to all of the system resources. This removes any concerns with deadlocks or conflicts over shared resources. In future implementations it would be ideal to implement these protections while maintaining the advantages of a single job running at a time.

c) Job Meta Data

The final information needed for each job is the meta data concerning how long they take to run and when they needed to be run. It is assumed that all of the jobs need to be executed in a periodic fashion. Therefor, to know when every job needs to be run and how to schedule it the program stores several variables per job in an array called tasks.

The first variable it stores is the amount of time in seconds it takes to run the program. Second item in the array is the period it needs to be completed by. For example, a task with a period of 10 would need to be completed once every 10 seconds. The next variable is when the job needs to be called next. This is adjusted throughout the run time of the program, but setting it to a non zero number allows you to delay the starting of a program. For example, setting it to 100 would mean that the job wouldn't be added to the schedule until 100 seconds after initialization. Finally, the array stores the deadline of each task which is the number of seconds after its call time that it needs to be finished by.

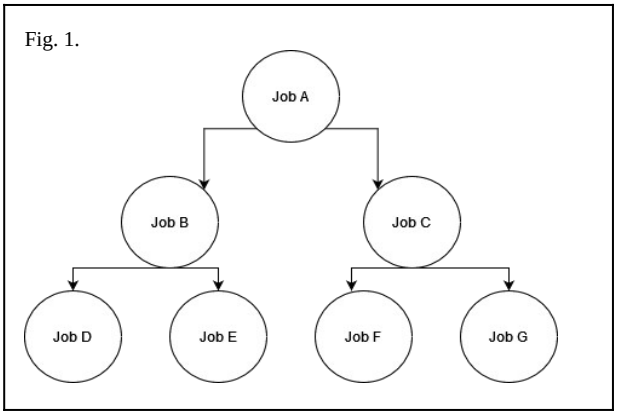
4) Jobs and the Queue

The queue of jobs is based on a heap which is implemented on top of an array. To understand this, let's work from the top down and start with an ordered heap then look at how the jobs are stored queue.

a) Job Heap

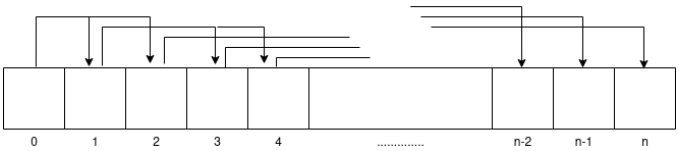
A max heap is a data structure where each node can have 2 children and the children must have a lower value then their parent node. This means that Job A must be the largest value in the heap, however Job G could have a greater value then Job B in the given example. In this implementation the space on the heap would be filled from top to bottom and left to right. So if there was one less job G's spot would not be filled.

Fig. 1.



This data structure was chosen because it has several extremely useful properties. First and foremost, it's ability to function as an ordered queue is the most important. We can always know that the highest priority task will be on the top of the heap and that's the one that should be executed next. It also allows for relatively easy additions and removals from the heap which is needed in order to add and remove jobs.

Fig. 2.



b) Heap on Array

The heap implemented in this project uses in based on an array. The array is initialized with the maximum number of jobs that will ever be on the heap. It then creates an heap where index 0 is the top of the heap and the left child of the top node is at index 1 and the right child is at index 2. Continuing this puts the left and right children of Job B at 3 and 4 respectively. Continuing this gives us the following equations for node at index n:

$$(1)\quad \text{left\_child} = 2 * n + 1$$

$$(2)\quad \text{right\_child} = 2 * n + 2$$

Now we that we have an understanding of how the queue is implementation on the heap, let's examine how to add and remove jobs from the queue.

c) Adding Jobs to the Queue

```
Insertion Algorithm(new_job):
  loc =  first_open_spot++
  array[loc] = new_job
  While array[loc].size >  array[parent(loc)].size:
    swap(array[loc], array[parent(loc)])
```

To add jobs to the Queue we begin by adding the new job to the smallest unfilled index in the array to maintain the top to bottom left to right method of filling the heap and allow us to keep track of the next available spot on the heap. Once the job is in the heap, we have to insure that it is in the correct position on the heap. Recall, that a child can not have a higher priority then the parent. We can also assume that only the node we added is in the incorrect location because the heap is correct before we added the new job.

This means that to repair the heap we only have to worry about the new job. This is done in the implementation by iteratively checking if the parent of the inserted node is less then it. If the parent is less then the inserted node, the two nodes are swapped and the process is repeated. Either the new node has the highest priority in the heap and it becomes the new top node or it stops before then because one of it's parents has a larger priority then it does.

d) Removing Jobs from the Queue

```
Removal Algorithm:
  toReturn = array[0]
  loc = --first_open_spot
  swap(array[0], array[loc])
  loc = 0

  While true:

    if array[loc].left < array[loc].right:
        left_less = True
    else:
        left_less = false

    if  array[loc] <  array[loc].left && !left_less
        swap(array[loc], array[loc].left)
    elif array[loc] < array[loc].right && left_less
        swap(array[loc], array[loc].left)
    else
        return  toReturn
```

Removing jobs from the queue is similar to adding jobs to the queue however the fact there is 2 children as apposed to 1 parent introduces additional checks. In order to remove a job from the queue in this implementation, the function first copies over the top job to the variable passed in. It then swaps the last and first element so that the first element can be considered to be empty.

Now the heap needs to be repaired because the current top node will probably not be the highest priority job on the heap. To do this, we will iteratively check to see which child is the largest and check if the largest child has a higher priority then then it's newly moved parent. If it is, the two are swapped and the process continues. Once neither child has a larger priority then the new parent we know that the heap has been repaired.

e) Job Structure

Each Job is stored in the queue inside of a C struct which is stored in the heap. Before execution, the information is copied into a local variable stored in the stack frame of the main loop. The job struct contains 2 types of information, what is needed to schedule the job and what is needed to run the job.

The job information contains a pointer to the jobs function and the char * array[] talked about before to be used as the functions arguments. In order to use this information, the scheduling book keeping information is the worst case computation time for the job, it's deadline, and, if RMS is being used, the period of the job. All of these variables are stored as int currently besides the function pointer and args.

B. Included Libraries

This project includes several external libraries that we will go through from most to least common.

- stdio.h – The final project sketch only uses one function from the stdio.h library, sprintf. This function is used to create one string out of a format string that can then be sent for the serial connection.
- string.h – This program uses several of string.h's functions to manipulate memory. In particular memset and memcpy are both used to move the data around within the heap.
- SimpleRSLK.h [8] – Probably the most important library used in this project is the SimpleRSLK.h library provided by TI. This library provides a relatively simple API to interface with the micro-controller and the robot it sits on top of. The current implement ion only uses it to set up the serial communication ports and access the time on the system, however any more complex jobs using this platform would have to make heavy use of the library or need to re implement the drivers needed to interact with the low level hardware of the micro-controller.

C. Pros and Cons of Implementation

There are several significant trade offs being made in this implementation of the job queue. One of the most significant ones is the requirement to copy bit by bit each job every time a job is swapped. This is due to the fact that the heap is based on an array and it is not possible to reorder objects in an array without copying them over. While an pointer based implementation of the heap would fix this problem it would introduce significant design complexity and require programming needed to keep the tree balanced to insure optimal adding and removing speed.

The next trade off to be considered when working on this project is the use of recursive or iterative method for solving problems. In particular the insertion and removal of jobs from the queue and the need to peculate a job up or down the heap is a perfect example of this. A recursive strategy would work perfectly in this case, but I decided to use the iterative method. Iterative methods are normally more cumbersome to work with for the programmer, however they normally result in more efficient code due to not needing to create large number of stack frames. This choice was due to the limited memory capacity of the micro-controller and the need to for high performance in the scheduler design.

V. Testing

A. Placeholder Jobs

9 different jobs were created for the testing of this program. Each job simply prints out a single character then sleeps for a second x number of times where x is the id of the job. This means that I can give a job the variables a and 5 and it will print the character a and then wait 1 second before repeating. It will repeat 5 times resulting in 5 a's to be printed in roughly 5 sounds using printf on the Linux machine and over the serial connection when ran on the micro controller.

B. Testing methodology

In order to test the scheduler, I used an known good output for several sets of jobs. I would create an RMS or EDF schedule from hand and then run the same set of jobs on the scheduler. The debugging output would give which job, if any, was worked on for every second. This was done for both of the scheduler types.

A couple of important notes about the test cases and what was included. The most obvious tests that I started with was a small number of tasks that only utilized part of the micro-controllers processing power. However, the system must be able to function under heavy loads and fail in a predictable manner. Therefor, the majority of the testing was focused on near failure states and overload conditions to insure that the system was able to continue functioning predictably in those conditions.

C. Results

Although several of the tests failed at the beginning of development, this was primarily due to an buggy heap implementation. Once the bugs with the heap were fixed tests completed correctly.

One other note is that very long term test will probably result in some errors due to the rounding errors on the actual time it takes to execute the jobs and the scheduling in between the jobs. The scheduler assumes that no overhead is taken up by the scheduler and the computation time allocated to the functions is only enough for the sleep call. Not the Serial communication and logic surrounding it. However, these errors are so small that they did not appear in the relatively short (1 hour max) testing I conducted.

VI. Continuing Work

A. Code separation and improvement

The code base for the project clearly shows that it was developed by a single person and needed to be reformatted multiple times to get it to function in multiple IDEs. This makes the readability of the code relatively poor and has almost certainly introduced fragility to the code base. Before future development, the jobs should be re separated out into a separate header file at least. It would also be beneficial to pass a 2nd variable to the jobs containing the number of arguments being passed to it similar to how the main function works in C programming. Finally, increasing the precision of the scheduling from 1 second down to micro or mili seconds would greatly increase real world applications of this scheduler.

B. Comparison of pointer based heap to array based heap

It is unclear if using a heap based on structs which point to their parent and 2 children rather then an array would be a better method. It would significantly reduce the number of memcpy calls required in the heap implementation. However, it would require constantly allocating and removing space on the program's memory heap. Implementing this and comparing it to the array based implementation would probably be worthwhile.

C. Interruption based scheduling

The MSP432P401R REV D micro controller does support timer based interrupts so allowing an interrupt driven scheduling would be a massive improvement to the scheduler. While basic timer based interrupts would not allow a task to be interrupted in the middle, it would protect the micro-controller from falling into an infinite loop. It should be possible to kill a task that takes longer then it's worst case scenario.

D. Additional Scheduling

Additional scheduling algorithms can be implemented relatively easily by creating an additional comparator methods and passed to the queue at initialization. A relatively easy example would be the least latency first (LLF) algorithm as the deadline and computation time is already within the job structure. Other algorithms may require additional book

keeping data to be held in the job structure, but the code should easily handle that.

VII. Conclusion

Overall this project was a great success and I was able to meet all of my objectives. I was able to create a fully functioning EDF & RMS scheduler on the MSP432P401R REV D micro-controller. I am particularly proud of the schedulers ability to take in an arbitrarily large number of functions and each of those functions having an arbitrarily large number of argument. The project has certainly solidified my understanding of EDF and RMS scheduling and how scheduling algorithms are created.

Project learning objectives	Status (Not/Partially/ Mostly/Fully Completed)	Pointers in the document
Self-contained description of the project goal, scope, and relevant requirements	Fully Completed	Section II Page 1
Self-contained description of the solutions (algorithms/protocol /applications/etc.)	Fully Completed	Section IV, page 2
Adequate description of the implementation details (data structures, pseudo code segments, libraries used, etc.)	Fully Completed	Section III, IV Page 2
Testing and evaluation – test cases, metrics, test results, any relevant performance results	Fully Completed	Section V. Page 5
Overall Project Success assessment	Fully Complete	

References

[1] Texas Instruments, “MSP432P401R Data sheet, Product Information and Support,” https://www.ti.com/product/MSP432P401R, Aug. 11, 2021. https://web.archive.org/web/20210811001055/https://www.ti.com/product/MSP432P401R (accessed Dec. 13, 2021).

[2] Texas Instruments, “MSP432P4xx SimpleLinkTM Microcontrollers Technical Reference Manual,” www.ti.com, 2020. https://web.archive.org/web/20200402132746/http://www.ti.com/lit/ds/symlink/msp432p401r.pdf.

[3] Texas Instruments, “MSP432P401R,” Ti.com, 2019. https://www.ti.com/product/MSP432P401R.

[4] Texas Instruments, “TI-RSLK MAX,” university.ti.com, 2019. https://university.ti.com/programs/RSLK/ (accessed Dec. 13, 2021).

[5] Texas Instruments, “Texas Instruments Robotics System Learning Kit User Guide,” www.ti.com, 2019. https://www.ti.com/lit/ml/sekp166/sekp166.pdf?ts=1635177457837 (accessed Dec. 13, 2021).

[6] Texas Instruments, “TI-RSLK Base Pin Map,” www.ti.com, 2019. https://www.ti.com/lit/ml/sekp171/sekp171.pdf?ts=1635177460390 (accessed Dec. 13, 2021).

[7] Texas Instruments, “TI RSLK MAX Debug,” dev.ti.com, 2019. https://dev.ti.com/gallery/view/1766484/RSLK_Debug/ver/2.0.1/ (accessed Dec. 13, 2021).

[8] Texas Instruments, “RSLK-Robot-Library,” *Amazonaws.com*, 2021. https://hacksterio.s3.amazonaws.com/uploads/attachments/1217431/RSLK-Robot-Library.zip (accessed Dec. 13, 2021).