Real-Time Scheduling: Project Report

Sampreet Sarkar*

1 Introduction

The goal of this project is to simulate the behaviour of a Preemptive Fixed-Priority Scheduler *(Rate Monotonic)*. We shall move along with the assumption that all the tasks are periodic, with the task deadline equal to the task period. The design of the software will be detailed in the following sections, along with the accompanying source code, which can be found in the appendix.

A scheduler is an extremely important part of any Operating System, as it is in charge of breaking up tasks into jobs and executing them in a particular order to maintain the illusion of pseudo-parallelism that we commonly associate with the term `"multitasking"`. This behaviour becomes doubly important in the case of a Real-Time Operating System, because in addition to executing tasks correctly, the scheduler has to also keep in mind to return results on time, otherwise the result of the computation is invalid, and cannot be used for any good purpose.

Before we begin to describe the common functionalities and the design of the software, it is imperative that we take a look into the programming paradigm being followed in the development of the scheduler. The core of the scheduler has been implemented in the C++ language, which although is a close second choice of language for developping embedded applications, provides a fast enough speed and above all, embodies the use of STL classes and Object-Oriented Programming benefits. The STL template classes *(especially vector and algorithm)* provide us with many helper functions, which make our life easier. Another choice to justify C++ as the optimal language of choice for this particular project is due to the innumerable open-sourced libraries that are available for the language, which lets the user accomplish a lot of complicated programming in a matter of minutes. The final executables are generated by the CMake[3] tool, which allows us to automate the build process and give our software a little more polish. The version control is done using Git, and the online repository for the project can be found here.

In the upcoming sections, we will discuss four main aspects of a scheduler—*viz.*, data acquisiton, schedulability analysis, simulation of the schedule, and evaluation of the metrics. The `data acquisition` section will deal with the general idea of how to represent the contents of a simulated task to our scheduler, the `schedulability analysis` will deal with running several schedulability tests on the set of tasks in order to determine whether a given taskset is indeed schedulable or not, the `simulation` of the schedule would ideally include some representation of how the scheduling algorithm places jobs at each time instant, and the `metrics` we would want to evaluate will give us a benchmark to compare the performance of our algorithm against other solutions. Some of the metrics that define the efficiency of a scheduler is the number of context switches, the maximum idle time for the processor, and the average response time for each task.

*M2 Control and Robotics: Embedded Real-Time Systems, *École Centrale de Nantes*

2 Data Acquisition

In this section, we are interested in obtaining the data from a simulated task, and generating our taskset τ from this data. If we think in terms of the *(simulated)*scheduler, the only information we need about a task are its identifier, the computation time, the period, and the deadline of the task. Since it has already been established that in our case, the period is equal to the deadline, we can make do with either information. We also understand that we would need some kind of data structure to contain this information in an easily-retrievable fashion. Thus, we can utilize a data structure as shown in Fig. 1 below:

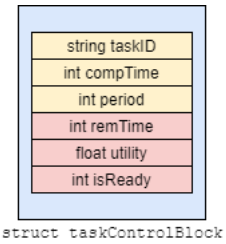


Figure 1: Data structure of a single task control block

We also notice that we have some extra fields in our data structure, namely `remTime`, `utility`, and `isReady`. These are not assigned by the user, but are really important to carry out the scheduling of the taskset. Here, the variable remTime stores the remaining execution time of the task, utility stores the utility of the task, and isReady is a flag that denotes the state of the task.

With all this information at hand, we can go ahead and create our data structure as denoted in the code listing below. In order to increase readability, we have distributed the entirety of the code into specialised header and source files, each of which provide some special functionality to the entire software. As a result of this distributed approach, one would find it extremely easy to navigate through specific section of code, to better understand the functionality or to modify it according to the development need.

```
typedef struct {
  std::string taskID;          // Stores the task identifier
  int compTime;                // Stores the computation time for the task
  int period;                  // Stores the period(=deadline) of the task
  int remTime;                 // Stores the remaining execution time of the task
  float utility;               // Stores the utility of the task
  int isReady;                 // Stores the current state of the task
} taskControlBlock;

extern std::vector<taskControlBlock> taskSet;
```

You will also observe that we have created a vector of these data structures, so that when a new task is created, the equiavalent taskControlBlock can be pushed into this vector. This will allow us to perform various vector based operations on the taskset*(sorting, pushing at the tail, getting the size)* which have already been defined in the C++ STL `<vector>`.

Moving forward, we would like to facilitate the user's process of creating a task. For this purpose, we propose the function `createTask()`, which will allow us to create a new taskControlBlock element, and push it into the list of tasks. The implementation of the function can be seen in the following code listing[1]:

[1]The main inspiration behind the nomenclature of the functions has been Trampoline, as it is a very elegantly designed RTOS, and since we already had some prior exposure to the functionalities.

```
void createTask(std::string tName, int c, int p) {
  taskControlBlock temp;

  temp.taskID   = tName;
  temp.compTime = c;
  temp.period   = p;
  temp.remTime  = c;
  temp.utility  = (float)c/(float)p;
  temp.isReady  = 1;

  taskSet.push_back(temp);
}
```

We have thus been able to devise a way for the user to define a task to be scheduled. Note that unlike other Fixed-Priority schedulers, we do not ask for a priority level from the user, we get around this by simply taking the size of the taskset as the maximum priority level, and ordering our taskControlBlock elements within the vector in ascending order of period. This saves us from having to validate user input, and lets us abstract the concept of priority assignment from the user.

In the upcoming section, we will focus on performing rigorous schedulability tests on a populated taskset, and let the user know if the schedule is indeed feasible or not.

3 Schedulability Analysis

Our goal in this section is to provide an insight to the feasability of a taskset. In order to do this, we must understand the conditions for schedulability of a taskset, laid out by Liu and Layland[1]. There are two conditions, the sufficient condition and the necessary condition.

Necessary Scheduling Condition

A taskset τ is schedulable by a Rate-Monotonic scheduler if and only if:

$$U \leq 1 \tag{1}$$

Where U is the total processor utilisation.

Sufficient Scheduling Condition

A taskset τ is schedulable by a Rate-Monotonic scheduler if and only if:

$$U \leq n(2^{\frac{1}{n}} - 1) \tag{2}$$

Where U is the total processor utilisation, and n is the number of tasks in the taskset τ.

From the equation 2, we observe that the as the number of tasks increase, the value of $n(2^{\frac{1}{n}} - 1)$ converges to $ln(2) = 0.693$. Thus, we can make an inference that the rate monotonic scheduler is not the best algorithm for maximising the processor usage. It is however, extremely deterministic in its functioning, which is why it is a widely used scheduler for mission-critical systems.

Now, let us take an example taskset and try to perform these checks to determine whether the taskset will be feasible or not. This is done in order to "think in terms of the scheduler", and will eventually help us design the algorithm. Consider the following taskset τ[2]:

[2] τ can also be represented in a more compact fashion as follows: $\tau = [(T_1, 1, 8), (T_2, 2, 5), (T_3, 2, 10)]$

Task	Computation Time	Period
T1	1	8
T2	2	5
T3	2	10

Assuming that the deadline for each task is equal to the period, we can proceed to checking whether this taskset τ is indeed feasible or not. We begin by computing the total processor utilisation U, which is calculated as:

$$\begin{aligned} U &= \sum_{i=1}^{3} \frac{C_i}{T_i} \\ &= \frac{1}{8} + \frac{2}{5} + \frac{2}{10} \\ &= \frac{29}{40} \\ &= 0.725 \end{aligned} \tag{3}$$

As we see that the value of U is less than 1, we can say that the necessary condition has been verified. This lets us know that the taskset might be feasible. We need to further verify this claim by checking the sufficient condition check, calculated as follows:

$$\begin{aligned} n(2^{\frac{1}{n}} - 1) &= 3(2^{\frac{1}{3}} - 1) \\ &= 3(1.259 - 1) \\ &= 3(0.259) \\ &= 0.777 \end{aligned} \tag{4}$$

As we see that the total processor utilisation *(from eqn.3)* is lesser than 0.777, we can say that the taskset τ is schedulable. Now if we want to figure out the schedule, we need to focus on the algorithm, presented as follows:

Rate-Monotonic Scheduling Algorithm

The task with the lowest period is scheduled first, and then the task with the next lowest period is scheduled. Preemption occurs when a low priority task is scheduling and a higher priority task is released.

If we follow this algorithm, we can obtain the schedule as follows:

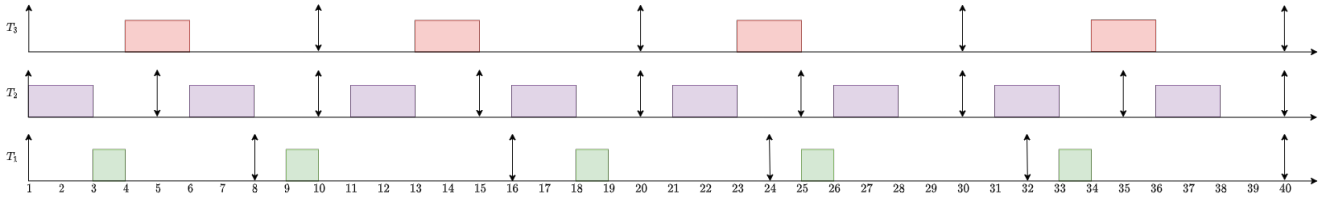


Figure 2: Feasible schedule of the taskset τ

These conditions can be very easy to be implement in code, as we can harvest all of the important information from the task control block or the taskset itself. During the writing of the software, we found it was more convenient to include an additional field for storing the remaining execution time and the utility of

the task itself. This would allow for lesser computations overall, hence making our program more efficient.

In the next section, we will see the scheduler simulate three different task sets, and we will see a few limitations of the software and the scheduling algorithm itself.

4 Simulation Results

The scheduled task set will be simulated from time zero to the first hyperperiod of the task set. We simulate only up till the first hyperperiod because the behaviour of the system is cyclic with respect to the hyperperiod, *i.e.*, the same schedule will be repeated each hyperperiod. We will simulate three use cases to understand the design and capabilities of the software. We will have our first taskset which can be scheduled without any conflict by the scheduler, then we will see a second taskset which the scheduler will not be able to schedule and for the final simulation, we will carefully choose a taskset which we know is schedulable, but the scheduler will say it is not, thus showing the limitations of such a scheduler.

4.1 Case I: Taskset scheduled without any conflict

In this section, we will simulate a simple taskset that will be schedulable by the scheduler after having passed both the schedulability tests. Consider the following taskset τ:

Task	Computation Time	Period
T1	2	8
T2	4	16
T3	1	4

We can observe the output of the program and the schedule in the following figures 3 and 4 respectively.

```
sampreets3@DESKTOP-QKMPQMV:/mnt/c/Users/Sampreet/Documents/GitHub/scheduler-RM/build$ ./task
        ======== TASK STATS: ========

Task ID     Computation Time    Period  Utility State
   T1           2               8       0.25    READY
   T2           4               16      0.25    READY
   T3           1               4       0.25    READY
                ====== x ======


-------- SCHEDULABILITY ANALYSIS -------
[INFO] Number of tasks          : 3
[INFO] Hyperperiod is           : 16 time units
[INFO] Tota Processor Utilisation : 0.75
[INFO] Feasibility Metric is    : 0.779763
[INFO] Running Sufficient Test  : PASSED
[INFO] Running Necessary Test   : PASSED
-----------------------------------

        Schedule:
Task    |    T3  T1  T1  T2  T3  T2  T2  T2  T3  T1  T1 IDLE T3 IDLEIDLEIDLE

Time    |    0   1   2   3   4   5   6   7   8   9   10   11   12   13   14   15
------------------------------
[INFO] Maximum Processor Idle     : 4 time units

sampreets3@DESKTOP-QKMPQMV:/mnt/c/Users/Sampreet/Documents/GitHub/scheduler-RM/build$
```

Figure 3: The program output on taskset τ

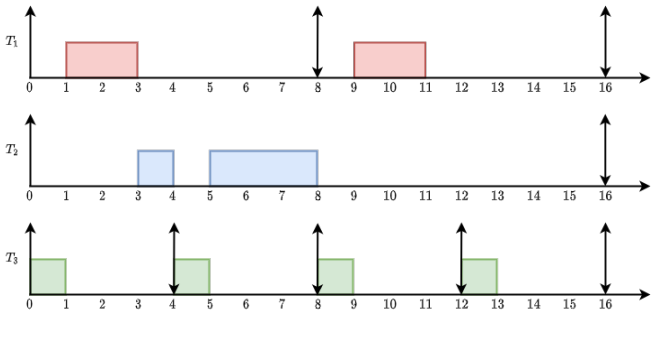


Figure 4: The produced schedule

4.2 Case II: Taskset Not Schedulable

Our aim in this case is to provide the scheduler with an unschedulable taskset, and see if it can effectively reproduce the results, *i.e.*, let us know that the taskset is not schedulable. We will take a simple example so that it is easy to study by hand as well. Consider the following taskset τ:

Task	Computation Time	Period
T1	3	10
T2	2	4
T3	3	6

We can deduce by looking at the taskset that it is not going to be schedulable, as the total processor utilisation U is 1.3, which is greater than the upper bound of 1. We will now observe the output of the program to verify our results.

```
sampreets3@DESKTOP-QKMPQMV:/mnt/c/Users/Sampreet/Documents/GitHub/scheduler-RM/build$ ./task
        ======== TASK STATS: ========

Task ID     Computation Time    Period  Utility State
   T1           3               10      0.3     READY
   T2           2               4       0.5     READY
   T3           3               6       0.5     READY
                ====== * ======

-------- SCHEDULABILITY ANALYSIS -------
[INFO] Number of tasks            : 3
[INFO] Hyperperiod is             : 60 time units
[INFO] Tota Processor Utilisation : 1.3
[INFO] Feasibility Metric is      : 0.779763
[INFO] Running Sufficient Test    : FAILED
[INFO] Running Necessary Test     : FAILED
-------------------------------------

        Schedule:
sampreets3@DESKTOP-QKMPQMV:/mnt/c/Users/Sampreet/Documents/GitHub/scheduler-RM/build$
```

Figure 5: Simulation of the taskset τ, which is not schedulable.

4.3 Case III: Taskset scheduled with conflict

This is a particularly interesting case, where we can observe that although the taskset does not pass any of the schedulability tests, the taskset is indeed scheduled by our scheduler. This highlights the integgrity and the flexibility of the software we have written.

Consider the following taskset τ:

Task	Computation Time	Period
T1	1	2
T2	1	4
T3	2	8

We can observe from the taskset that the total processor utilisation here will be 1. This will cause the necessary and sufficient conditions to fail. However, when we run the program, we will see that it indeed produces a schedule which is feasible. This goes on to show that our scheduler tries to fit in the taskset to the best of its abilities. Note that this schedule is incomplete of information, we will need to know if there are any shared resources, and how they interact with each other to know the blocking time. This will let us know whether the taskset is indeed schedulable or not.

```
        ======== TASK STATS: ========

Task ID     Computation Time    Period  Utility State
   T1           1               2       0.5     READY
   T2           1               4       0.25    READY
   T3           2               8       0.25    READY
                ====== * ======


-------- SCHEDULABILITY ANALYSIS -------
[INFO] Number of tasks            : 3
[INFO] Hyperperiod is             : 8 time units
[INFO] Tota Processor Utilisation : 1
[INFO] Feasibility Metric is      : 0.779763
[INFO] Running Sufficient Test    : FAILED
[INFO] Running Necessary Test     : FAILED
-------------------------------------

        Schedule:
Task    |    T1  T2  T1  T3  T1  T2  T1  T3

Time    |     0    1    2    3    4    5    6    7
------------------------------
[INFO] Maximum Processor Idle     : 0 time units
```

Figure 6: The program output on taskset τ

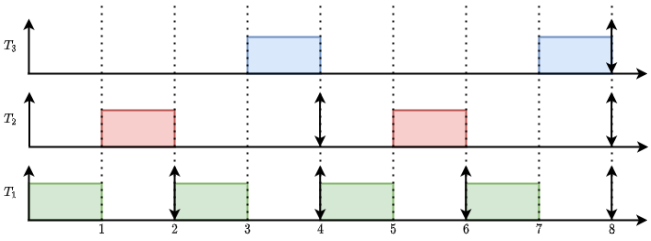


Figure 7: The produced schedule

5 Evaluation of Main Metrics

In this section we will evaluate the main metrics for the scheduler. This will give us more information about the performance of each task during the scheduling.

We will see in the code the following evaluation of metrics:

- $$\text{Number of Releases, } N_i = \frac{Hyperperiod(\tau)}{Period(\tau_i)} \forall\ \tau_i \in\ \tau \tag{5}$$
- $$\text{Total Idle Time, } T_{idle} = Hyperperiod(\tau) - (U \cdot Hyperperiod(\tau)) \tag{6}$$
- $$\text{Release Time, } R_i = j \cdot \text{Period}(i), \quad \forall\ i \in \text{ Number of tasks in } \tau \tag{7}$$
- $$\text{Waiting Time, } W_i = \text{Start Time}(\tau_i) - \text{Release Time}(\tau_i) \tag{8}$$

These metrics have been evaluated using a simple function, called `evaluateMetrics`, a part of which is shown in the listing below:

```
1  void evaluateMetrics() {
2
3    //calculate number of releases:
4    std::vector<double> noOfReleases;
5    double hp = (double)computeHyperPeriod(taskSet);
6    std::cout << std::endl;
7
8    std::cout << std::endl;
9    std::cout << "------EVALUATION OF METRICS----------" << std::endl;
10   std::cout << "[INFO] Total number of Releases(Task-wise): " << std::endl;
11
12   for(auto elem: taskSet){
13     std::cout << "\tTask\t" << elem.taskID << "->" << (int)hp/elem.period << std::endl;
14   }
```

```
15
16   std::cout << "[INFO] Maximum Processor Idle      : " << idleTime << " time units" <<std
       ::endl;
17
18   std::cout << std::endl;
19 }
```

6 Conclusion

The aim of this project was to gain an understanding about how a rate-monotonic scheduler functions. We have been able to explore the core principles of a Fixed-Priority Rate Monotonic Scheduler in some detail, where we saw how we can conceptualise a simple task control block, how to store this information in a structured format, and how to facilitate creation, addition, deletion, and sorting of the tasks internally with the list of tasks using C++ STL classes.

In the next section, we gained insight on how we would acquire data for our scheduler, highlighting the branching out of source code into multiple files, enabling file input/output, and in general giving it a more application feel. Once we had acquired the data, we wanted to check if the taskset was indeed feasible or not before trying to compute a schedule for it. For this purpose, we made use of the necessary and sufficient schedulability checks, and learnt how to compute them by solving an example by ourselves.

Finally, when the schedule had been computed, we had to display the main metrics and evaluate our scheduler against the rest of the schedulers out there. We saw a very simple method of comparision, and how we can achieve that in code.

As far as proposed updates are concerned, the popular plotting library matplotlib will be included in order to directly provide the GANTT diagrams. Work is also underway with extrapolation of this software to incorporate multi-processor based scheduling.

Overall, it was an extremely enriching experience, which has helped us understand not only the basics of rate monotonic scheduling, but also about software development practices in general.

References

[1] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM*, 20(1):46–61, 1973.

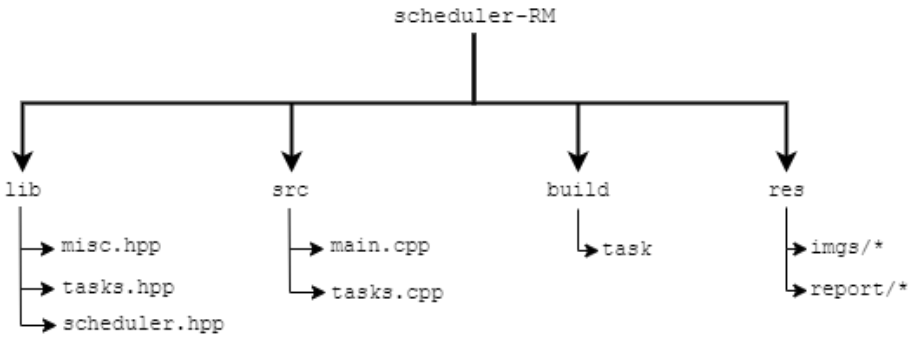
[2] Steve Oualline. *Practical C++ programming.* " O'Reilly Media, Inc.", 2003.

[3] Ken Martin and Bill Hoffman. *Mastering CMake: a cross-platform build system.* Kitware, 2010.

Appendix

Code listing for the software

The entire source code is split up into multiple files, which are orgainised in the following manner:



main.cpp:

```
#include "tasks.hpp"
#include "misc.hpp"
#include "scheduler.hpp"

int main(void){

  // Create three tasks
  createTask("T1", 1, 8);
  createTask("T2", 2, 5);
  createTask("T3", 2, 10);

  // Show the set of tasks
  showTaskSet();

  // Run the scheduler
  //std::vector <std::string> s = scheduleRM();
  printSchedule(scheduleRM());
  return 0;
}
```

tasks.cpp:

```
/* function createTask : creates a task by initialising
 * a taskControlBlock element with the given values and
 * pushing it to the vector of taskControlBlock elements.
 */
void createTask(std::string tName, int c, int p) {
  taskControlBlock temp;

  temp.taskID   = tName;
  temp.compTime = c;
  temp.period   = p;
  temp.remTime  = c;
  temp.utility  = (float)c/(float)p;
  temp.isReady  = 1;

  taskSet.push_back(temp);
}

/* function printTaskState : Small helper function to print the state of the task
```

```
 */
std::string printTaskState(taskControlBlock t) {
  if(t.isReady)
    return "READY";
  else
    return "WAITING";
}

/* function showTaskSet : Iterates through the set of
 * tasks and print them.
 */
void showTaskSet() {

  std::cout << "\t======== TASK STATS: ========\n" << std::endl;
  std::cout << "Task ID" << "     Computation Time" << "\tPeriod" << "\tUtility" << "\
    tState" << std::endl;
  for(auto elem: taskSet) {
    std::cout << "    " << elem.taskID << "\t\t" << elem.compTime << "\t\t" << elem.
    period << "\t" << elem.utility << "\t" << printTaskState(elem) << std::endl;
  }
  std::cout << "\t\t====== * ======" << std::endl;
}
```

tasks.hpp:

```
#ifndef TASKS
#define TASKS
#include <iostream>
#include <vector>

typedef struct {
  std::string taskID;         // Stores the task identifier
  int compTime;               // Stores the computation time for the task
  int period;                 // Stores the period(=deadline) of the task
  int remTime;                // Stores the remaining execution time of the task
  float utility;              // Stores the utility of the task
  int isReady;                // Stores the current state of the task
} taskControlBlock;

extern std::vector<taskControlBlock> taskSet;

void createTask(std::string tName, int c, int p);
void showTaskSet();
std::string printTaskState(taskControlBlock t);

#endif
```

misc.hpp:

```
#ifndef MISC
#define MISC

#include "tasks.hpp"

/* function comparePeriod : takes in two taskControlBlock
 * elements, and returns true if the first element has a
 * smaller task period than the second element.
 */
bool comparePeriod(taskControlBlock x, taskControlBlock y) {
  if(x.period < y.period)
    return true;
  else
```

```
    return false;
}

/* function gcd : Computes the Greatest Common Denominator
 * between two given numbers using recursion
 */
int gcd(int a, int b) {
  if(b == 0) {
    return a;
  }
  else {
    return gcd(b, a%b);
  }
}

/* function computeHyperPeriod : computes the hyperperiod
 * of a given vector of taskControlBlock elements.
 */
int computeHyperPeriod(std::vector <taskControlBlock> vec) {
  int n = vec.size();               // Calculate the size of the vector

  int ans = vec.begin()->period;   // Choose the first element as the default answer

  for (auto elem : vec) {
    ans = (int)(elem.period * ans) / (gcd(elem.period, ans));
  }
  return ans;
}

#endif
```

scheduler.hpp:

```
#ifndef SCHEDULER
#define SCHEDULER

#include "misc.hpp"
#include "matplotlibcpp.h"
#include <algorithm>
#include <cmath>

bool necTest = false, sufTest = false, isSched = false;
float totalUtil = 0.00, feasible = 0.0;
int idleTime = 0;

namespace plt = matplotlibcpp;                  // Defining the namespace for matplotlib

/* function testSchedulability : Runs important schedulability
 * tests on the given set of task
 */
bool testSchedulability() {

  std::cout << "\n\n" << "-------- SCHEDULABILITY ANALYSIS ------- " << std::endl;

  for(auto elem: taskSet)
    totalUtil += elem.utility;            // Iterate through the task set and compute
   the total utilisation

  feasible = taskSet.size() * (std::pow(2, 1/(double)taskSet.size()) - 1);
  std::cout << "[INFO] Number of tasks            : " << taskSet.size() << std::endl;
  std::cout << "[INFO] Hyperperiod is             : " << computeHyperPeriod(taskSet) <<
    " time units" << std::endl;
```

```
28    std::cout << "[INFO] Tota Processor Utilisation : " << totalUtil << std::endl;
29    std::cout << "[INFO] Feasibility Metric is      : " << feasible << std::endl;
30    std::cout << "[INFO] Running Sufficient Test    : ";
31    if(totalUtil < feasible) {              // Sufficient schedulability test
32      std::cout << "PASSED" << std::endl;
33      sufTest = true;
34    }
35    else {
36      std::cout << "FAILED" << std::endl;
37      sufTest = false;
38    }
39
40    std::cout << "[INFO] Running Necessary Test     : ";
41    if(totalUtil < 1.0) {                   // Necessary schedulability test
42      std::cout << "PASSED" << std::endl;
43      necTest = true;
44    }
45    else {
46      std::cout << "FAILED" << std::endl;
47      necTest = false;
48    }
49
50    std::cout << "-------------------------------------" << std::endl;
51
52    if(necTest & sufTest)
53      isSched = true;                     // Returns true if both tests pass
54
55    return true;
56  }
57
58  /* function scheduleRM : Implements the Fixed Priority Rate Monotonic
59   * scheduling algorithm on the task set
60   */
61  std::vector<std::string> scheduleRM(){
62
63    std::vector<std::string> sched;
64    int hp = computeHyperPeriod(taskSet);
65    int n = taskSet.size();
66    sort(taskSet.begin(), taskSet.end(), comparePeriod);      // Sort the task set in
      ascending order of period
67
68    if(testSchedulability()){                                 // If the task list is
      schedulable, then go on
69      for(int i=0 ; i< hp ; i++){
70        if(i>=taskSet[0].period)
71          for(int j=0 ; j<n ; j++){
72            if(!(i%taskSet[j].period))                      // If the task is restarted
73                taskSet[j].isReady=1;                       // Set the state to ready
74          }
75          for(int j=0 ; j<n ; j++){
76            if(taskSet[j].isReady && (taskSet[j].remTime)){  //If task is ready and has
      remaining execution time
77              if(!(--(taskSet[j].remTime))){                //If task has executed fully
78                  taskSet[j].isReady=0;                     //Set task status to waiting
79                  taskSet[j].remTime=taskSet[j].compTime;   //Reset the remaining
      execution time of the task
80              }
81              sched.push_back(" " + taskSet[j].taskID + " ");
82              break;
83            }
84            if(j == n-1) {
```

```
85              sched.push_back("IDLE");
86              idleTime++;
87            }
88          }
89      }
90    }
91    return sched;
92  }
93
94  void printSchedule(std::vector<std::string> s) {
95    std::vector<int> timeStamp;
96
97    std::cout << "\n\tSchedule:\t\t" << std::endl;
98    std::cout << "Task\t|    ";
99
100   for(auto elem: s)
101     std::cout << elem ;
102
103   std::cout << "\n\nTime\t|    ";
104   for(int t = 0; t < computeHyperPeriod(taskSet); t++) {
105     std::cout <<"  " << t << "  ";
106     timeStamp.push_back(t);
107   }
108   std::cout << "\n------------------------------" <<std::endl;
109   std::cout << "[INFO] Maximum Processor Idle     : " << idleTime << " time units" <<std
        ::endl;
110   std::cout << std::endl;
111
112   // plot the results
113   plt::plot(timeStamp);
114   plt::show();
115 }
116
117 #endif
```