University of Colorado Boulder

ECEN 5623 – Real-Time Embedded Systems (Summer-2019)

Final Project Report

Project Details	Description
Created By:	**Rushi James Macwan**
Project Title:	**Time-Lapse**
Platform Used:	**Raspberry Pi 3B+**
Project Type:	***INDIVIDUAL***
Report Submission Date:	***Aug 17th, 2019***
Attachments Link	***https://drive.google.com/open?id=1TUxTlb0y1AdN21ut_03J0r9C6MTAnWM5***

Credits: *All codes utilized for completing this course project have been reused and modified accordingly. The codes are originally authored by* ***Dr. Siewert*** *and is* ***NOT*** *entirely my original work. Appropriate code explanations and citations have been made wherever required with comments.*

Please Note:

The images included in this report may not be very clear and as desired. Please, refer to the attached zipped file (the link above) which contains all the images and other files provided/explained/referred to in this report. ***The zipped folder submitted for Exercise-1 contains Time-Lapse videos and other frames for reference. The zipped folder for this report contains only the basic 3-min test frames.***

I have used the terms "Soft Real-Time (SRT) Software System" and "Real-Time (RT) Software System" interchangeably in this project report submission and I am hereby treating the information covered in this report within the context of SRT Software Systems which is at the core of the proof-of-concept of the Time-Lapse Project that I have built for the ECEN-5623 final project.

This is the second and final version of the Project Report. Some content in this report have been taken exactly as it were from the previous version AND is by default, my own original work unless specified otherwise.

Table of Contents

Introduction

In this project, I have built a Soft Real-Time (SRT) software system that serves as a Time-Lapse acquisition device. With the use of Raspberry Pi 3B+, and the core principles of RT-Embedded-Systems learnt in the class, I have implemented a SRT software system that can associate with a camera device and record the environment with appropriate features meeting the specifications of the camera device. For this project, I have used the Logitech C270 HD webcam which I have interfaced with the R-Pi 3B+ board. The specified Dev board establishes a reliable connection with the specified camera device and initializes it for usage under a SRT environment. Once the connection is established, the board exploits its underlying Linux-based operating system. With the help of the Linux RT patch (CONFIG_PREEMPT_RT) and the Native POSIX Thread Library (NPTL), the board operates in a SRT environment and begins the communication with the camera device with the help of an I/O decoupled and Non-Blocking SRT environment. The board reads the image frames from the camera device at a user-specified frequency and stores it in the available NAND Flash Memory on the board in a user-specified file format which can be a colour (RGB) or a grayscale image frame. This version of the project handles a much resource-constrained SRT environment with limited error tolerance margin and with higher reliability on the user's end because of its operation at a higher frequency rate. This version also introduces the concept of stabilizing the camera for stabilized frame acquisition before it is actually stored to the NAND Flash Memory. In this entire process, there are 3 SRT services running on one CPU core and 1 BE (best effort) service running on a separate CPU core.

Functional (Capability) Requirements

The proposed SRT software system for the Time-Lapse project consolidates and embodies a set of system functional (capability) requirements that forms the basis of the execution and the working for the RT system features requirements as explained in the next section. The requirements are specified as below:

Requirements:

- The proposed SRT software system that is built over the design philosophy of a Time-Lapse capturing device, it is designed to use the R-Pi 3B+ Dev board and a Logitech C270 HD webcam that partake in the SRT software system execution.
- The system uses the V4L2 API as support by a Linux OS. This API is utilized by the board for establishing a reliable and robust communication bridge between the camera and the board. The V4L2 API is used for exploiting the camera device and its properties (i.e. changing the acquisition from capturing images to videos, modifying other image frame properties like colour intensity, brightness, etc. and more). In general, the API allows the user to develop a software system that can use the camera device in a RT environment for various applications.
- For the Time-Lapse project, this API is used to acquire frames at a user-specified resolution and aspect ratio – which is 640 x 480 (VGA) – (4:3) aspect ratio by default and the images are stored as a grayscale image (.png file format). The design also allows for using up to 4 other resolutions (see Appendix) with a common aspect ratio (i.e. 4:3). However, these designs have not been tested due to the limited availability of time.
- The new development allows for a reliable Time-Lapse capture of over 1800 frames (i.e. over 30 minutes of capturing time @ 1 Hz operation frequency) with no observable error (at a resolution of 1 second). While these images are acquired using the SRT environment, they are stored as well on the local NAND Flash memory available on the board using the same SRT environment.
- The new development also allows for a reliable Time-Lapse capture of over 6000 frames (i.e. over 10 minutes of capturing time @ 30 Hz operation frequency) with approximately a 50-75 milli-seconds average error rate for every 610 frames that are captured. The error can be easily attributed to the presence of a bug in the code since the error rate tends to remain the same and accumulates linearly as the code execution progresses. Although, the I/O is decoupled in this project, along with the camera operating in the Non-Blocking mode, I believe that this is a "offset by 1" error which can be solely attributed to a bug in my code that I have not been able to resolve in the time that was available to me. Based on the above observations, *it can be said that the system is quite deterministic with a deterministic error-rate which can be dealt with an offset in real-world scenarios where the offset is known to be stable and without any drift.*
- In addition to the SRT services, 1 BE service provides a system status to the user as some form of acknowledgement of where the system execution has reached. Although, this service can be helpful, for testing purposes, the printf statement in this service was commented out as it would eventually add unexpected latencies to the system as it would deal with the system kernels which may throw off the scheduling (SCHED_FIFO followed by RMA) policy that has been implemented.
- The SRT system also provides time-stamps for every critical system software execution point.

Real-Time Requirements

The SRT software system discussed in the previous section incorporates the use of POSIX threads and other synchronization primitives to exploit the processing capabilities of the board (R-Pi 3B+) in a SRT application environment. With the use of Inter-Process Communication (IPC) mechanisms (e.g. circular ring buffers), the blocking of SRT system services can be prevented while the synchronization primitives prevent concurrency bugs and critical events like deadlocks, unbounded priority inversion and chained blocking. The implementation of I/O-decoupling and Non-Blocking frame acquisition system interface with the camera enables the system to operate as a SRT system.

The SRT software system uses 3 SRT services on one CPU core (core 3) and 1 BE service on another (core 2). The below segment explains it more in detail:

a. **Sequencer:** This is the POSIX thread that sequences and schedules the other two service threads for a SRT system application. This thread implements the use of a Rate-Monotonic Algorithm (RMA) and the Linux SCHED_FIFO policy that guarantees that the threads are executed in a specific order and pre-empted (as per service priorities associated and the assigned CPU affinity) when a thread is waiting for resources and the CPU resource is not being utilized. The sequencer is operated at a 30 Hz rate (at all times) and provides the synchronization primitives to other threads (e.g. semaphores) to both schedule the execution of the other threads and prevent concurrency bugs.

b. **Service-1 (Frame Acquisition Thread):** This is the first POSIX SRT thread that read frames from the camera device that is already initialized before the SRT system environment comes alive. This thread discards the first few (e.g. 200) image frames for allowing some time to stabilize the camera device before the image frames are actually passed for processing to the other thread. Based on the user-specified information, this thread will accumulate image frames in grayscale (.ppm or .pgm) file formats at a specified frame rate (i.e. 1 Hz or 10 Hz) and with the image resolution of 640 x 480.

c. **Service-2 (Frame Storage Thread):** This is the second POSIX SRT thread that stores the image frames as the first service thread feeds it; ran at a similar or lower frequency than the first service thread. With the use of a simple circular ring buffer mechanism with sufficient storage space, the memory latency associated with the process of erasing memory sectors can be prevented from affecting the SRT system response. This is so because this thread has enough margin even at a 10 Hz operation rate which allows for storing multiple frames within operation cycle that may not have been saved previously due to a memory latency. This service stores the image frames into the /root/… folder as per the naming: "test%08d.xyz" (where .xyz is a file format – could be .ppm or .pgm).

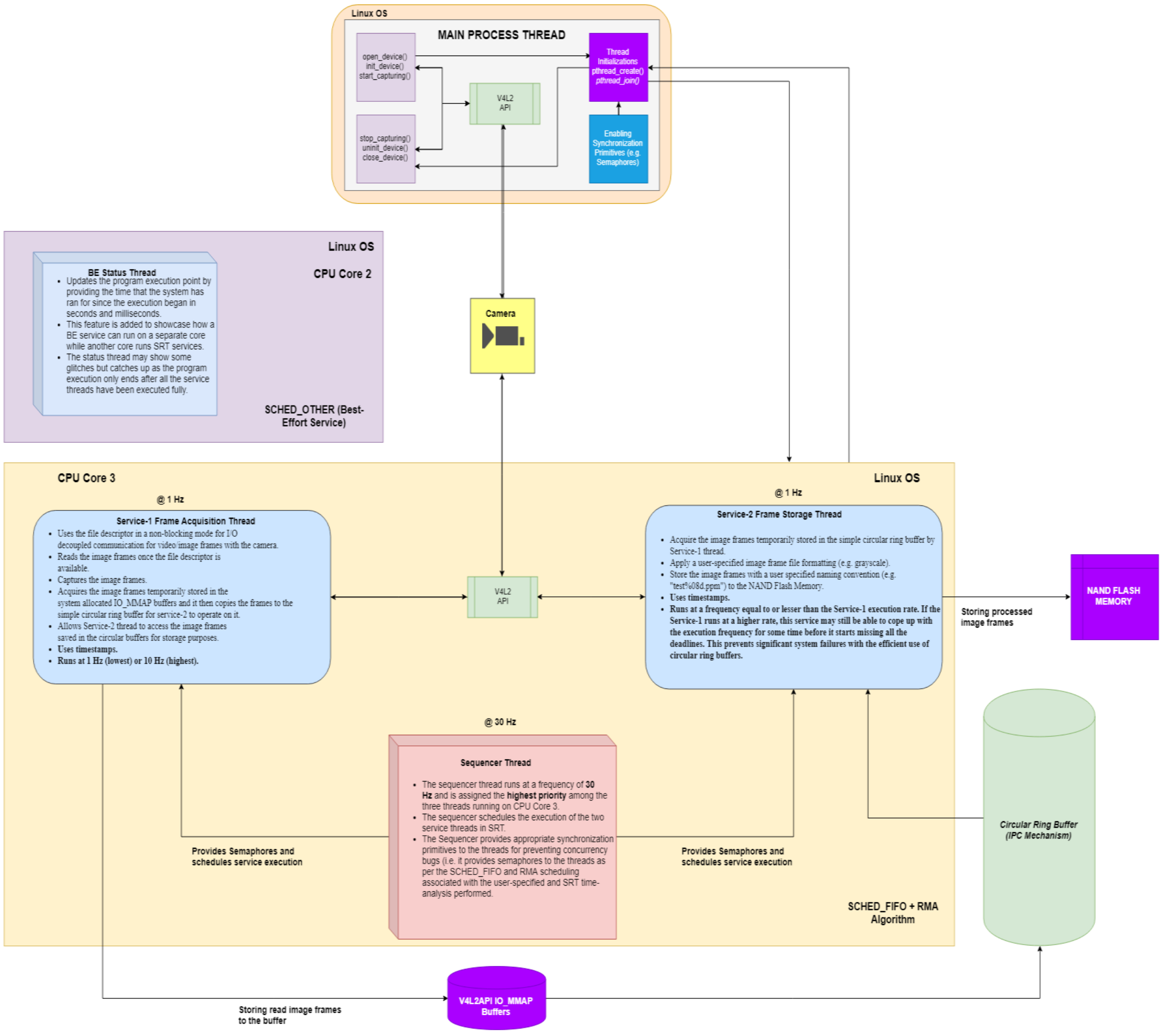
d. **Service-3 (BE Status Thread):** This is the third POSIX thread that operates as a BE service. It simply provides a system execution point status to the user so that the user can approximate the system execution time. Since this thread contains printf statements that affect the execution of the SRT services, for testing purposes, the printf statements were commented out during the 10 Hz testing. For the 1 Hz testing however, it tends to work extremely well because of the immense error tolerance margin.

The proposed SRT software system observes a well-tested set of timing values as discussed in the section on RT Analysis and Associated Design. Additionally, time-stamping allows for accurate reporting of service execution timings in SRT. The timestamps are available in the /var/log/syslog text file that is available in the board.

Functional Design Overview and Diagrams

The functional design block diagram is as under which showcases the basic SRT software system building blocks in addition to the state machine architecture which is briefly as portrayed below:

Basic block diagram


Linux OS
MAIN PROCESS THREAD
open_device()
init_device()
start_capturing()
stop_capturing()
uninit_device()
close_device()
V4L2 API
Thread Initializations pthread_create() pthread_join()
Enabling Synchronization Primitives (e.g. Semaphores)
Linux OS
CPU Core 2
BE Status Thread
Updates the program execution point by providing the time that the system has ran for since the execution began in seconds and milliseconds.
This feature is added to showcase how a BE service can run on a separate core while another core runs SRT services.
The status thread may show some glitches but catches up as the program execution only ends after all the service threads have been executed fully.
SCHED_OTHER (Best-Effort Service)
Camera
CPU Core 3
Linux OS
@ 1 Hz
Service-1 Frame Acquisition Thread
Uses the file descriptor in a non-blocking mode for I/O decoupled communication for video/image frames with the camera.
Reads the image frames once the file descriptor is available.
Captures the image frames.
Acquires the image frames temporarily stored in the system allocated IO_MMAP buffers and it then copies the frames to the simple circular ring buffer for service-2 to operate on it.
Allows Service-2 thread to access the image frames saved in the circular buffers for storage purposes.
Uses timestamps.
Runs at 1 Hz (lowest) or 10 Hz (highest).
V4L2 API
@ 1 Hz
Service-2 Frame Storage Thread
Acquire the image frames temporarily stored in the simple circular ring buffer by Service-1 thread.
Apply a user-specified image frame file formatting (e.g. grayscale).
Store the image frames with a user specified naming convention (e.g. "test%08d.ppm") to the NAND Flash Memory.
Uses timestamps.
Runs at a frequency equal to or lesser than the Service-1 execution rate. If the Service-1 runs at a higher rate, this service may still be able to cope up with the execution frequency for some time before it starts missing all the deadlines. This prevents significant system failures with the efficient use of circular ring buffers.
Storing processed image frames
NAND FLASH MEMORY
@ 30 Hz
Sequencer Thread
The sequencer thread runs at a frequency of 30 Hz and is assigned the highest priority among the three threads running on CPU Core 3.
The sequencer schedules the execution of the two service threads in SRT.
The Sequencer provides appropriate synchronization primitives to the threads for preventing concurrency bugs (i.e. it provides semaphores to the threads as per the SCHED_FIFO and RMA scheduling associated with the user-specified and SRT time-analysis performed.
Provides Semaphores and schedules service execution
Provides Semaphores and schedules service execution
Circular Ring Buffer (IPC Mechanism)
SCHED_FIFO + RMA Algorithm
Storing read image frames to the buffer
V4L2API IO_MMAP Buffers

The block diagram identifies the following list of functional design elements which are either associated with the *Main Process Thread* or the *CPU Core* running the *3 SRT services* as previously mentioned:

1. A Logitech C270 HD camera
2. The V4L2 API embedded into the Linux environment for communication with the camera device (e.g. device opening/closing, setup & initialization, etc.)

3. The thread initialization block
4. The synchronization primitives initializer block
5. The SRT CPU Core block running 3 SRT services (CPU Core 3)
6. The BE CPU Core block running 1 BE service (CPU Core 2)
7. V4L2 IO_MMAP allocated buffers
8. The circular buffer storing multiple acquired frames from the camera shared between the SRT services
9. NAND Flash Memory storage block

Brief overview of the functional design

The functional design incorporates the main process thread and the CPU core running the 3 SRT services at a lower priority than the main process thread. The main process thread establishes a communication bridge with the camera by the help of the following functions after taking in the device parameters required for the I/O decoupling and I/O select for a non-blocking mode communication.

`open_device()`

`init_device()`

`start_capturing()`

The above functions will set the camera device for allowing the SRT thread to request for image frames and thereby read it. The main process thread then initializes the 3 SRT POSIX threads to run using the SCHED_FIFO scheduling policy followed by the applied RMA algorithm in the RMA. The main process thread will also initialize the only BE thread that will act as a status beacon. The threads once initialized when the main process thread passes in the appropriate parameters, are created and executed using the below functions:

`pthread_create()`

`pthread_join()`

Once the above instructions are executed by the main process thread, the pthread_join() function is executed, the services are run as per their scheduling policies. The Sequencer will have the highest priority and will run at 30 Hz while other SRT threads will be run at either 1 Hz or 10 Hz depending on the need of the user. Once the threads are exited, the execution returns back to the main process where the program execution ends with or without an error as per the user-specified assertions.

Below section will briefly provide an overview of the tasks that the CPU will perform for the Core 3 execution:

CPU State Machine for the POSIX threads

The SRT POSIX threads will be all run on the CPU Core 3 as set and defined by the affinity in the program.

The following brief events will take place in case of each of the POSIX threads:

Sequencer thread:

- The sequencer thread runs at a frequency of **30 Hz** and is assigned the **highest priority** among the three threads running on CPU Core 3.

- The sequencer schedules the execution of the two service threads in SRT.
- The Sequencer provides appropriate synchronization primitives to the threads for preventing concurrency bugs (i.e. it provides semaphores to the threads as per the SCHED_FIFO and RMA scheduling associated with the user-specified and SRT time-analysis performed.

<u>Service-1 thread (for frame acquisition):</u>

- Uses the file descriptor in a non-blocking mode for I/O decoupled communication for video/image frames with the camera.
- Reads the image frames once the file descriptor is available.
- Captures the image frames.
- Acquires the image frames temporarily stored in the system allocated IO_MMAP buffers and it then copies the frames to the simple circular ring buffer for service-2 to operate on it.
- Allows Service-2 thread to access the image frames saved in the circular buffers for storage purposes.
- **Uses timestamps.**
- **Runs at 1 Hz (lowest) or 10 Hz (highest).**

Functions used:

```
mainloop()       - For capturing frames continuously using IO_MMAP
read_frames()    - For storing frames to the buffer
```

<u>Service-2 thread (for frame storing):</u>

- Acquire the image frames temporarily stored in the simple circular ring buffer by Service-1 thread.
- Apply a user-specified image frame file formatting (e.g. grayscale).
- Store the image frames with a user specified naming convention (e.g. "test%08d.ppm") to the NAND Flash Memory.
- **Uses timestamps.**
- **Runs at a frequency equal to or lesser than the Service-1 execution rate. If the Service-1 runs at a higher rate, this service may still be able to cope up with the execution frequency for some time before it starts missing all the deadlines. This prevents significant system failures with the efficient use of circular ring buffers.**

Functions used:

```
process_image() - For processing the acquired frames to grayscale
dump_pgm()      - For storing grayscale files in the .pgm file format
```

<u>Service-3 BE thread (for status updates):</u>

- The sequencer thread runs at a frequency of **30 Hz** and is assigned the **highest priority** among the three threads running on CPU Core 3.
- The sequencer schedules the execution of the two service threads in SRT.
- The Sequencer provides appropriate synchronization primitives to the threads for preventing concurrency bugs (i.e. it provides semaphores to the threads as per the SCHED_FIFO and RMA scheduling associated with the user-specified and SRT time-analysis performed.

RT-Timing Analysis & Associated System Design Review

For this project, I carried out the RT timing analysis for the service (S_i) properties like – periods (P_i), deadlines (D_i) and execution timings (C_i and WCET). Based on the methods to perform the analysis, I carried out the following RT timing analysis tests (in the order as their listed) on the basis of which I have designed my system:

1. Jitter analysis using timestamps
2. Hand-drawn SRT service scheduling using RMA
3. Cheddar analysis for the SRT service scheduling using RMA
4. Scheduling Point and Completion Test analysis

The following sections will briefly explain how the different RT timing analysis that I performed.

Jitter analysis using timestamps

Jitter analysis served as the primary testing and evaluation element for the SRT software system that I have designed. With the efficient use of the logger and the syslog logging mechanism, I time-stamped my service thread executions for an iteration of 100 execution cycles. Over these 100 cycles, I was able to establish a clear range of the execution timings for each service thread. The screenshots provided in the Appendix section provide a clear understanding of how service threads 1 and 2 play a role in the sharing of the CPU resources (as they share one core along with the sequencer thread fulfilling the 2+ services/core requirement).

From the screenshot explaining the execution timing of Service-1 thread, it shows that on an average, the service will consume about 40-43 milliseconds of execution time. This is primarily because the Service-1 thread will open a file descriptor and will wait until a timeout (in the I/O decoupled non-blocking mode) for an I/O operation. Once the file descriptor is available for operation, the Service-1 thread will read the frames of images and will control the I/O and update the circular ring buffer with the image frames stored by the camera device in the system allocated IO_MMAP memory buffers. This being a lengthy process involving some kind of I/O latencies although non-blocking because of I/O decoupled operation as defined by the timeout, takes about 40-43 milliseconds on an average as previously mentioned.

The Service-2 thread will then process the image frames stored in the circular ring buffer memory and will dump the image file into a specified file format using the specified naming convention (e.g. test00001800.ppm) in the root space in the NAND Flash Memory. Service-2 thread will consume about 5-6 milliseconds of execution time on an average. As observed in the screenshots provided in the Appendix section, the following tables show the timing analysis for both the service threads.

Service-1 Frame Acquisition Thread Timing Analysis	
S_i	1
C_i	40-43 ms
WCET	60 ms (margin between C_i and WCET: 15-17 ms)
Ti	1000 ms (@ 1 Hz) / 100 ms (@ 10 Hz) (margin between WCET and T_i – 940 ms (@ 1 Hz) / 40 ms (@ 10 Hz))
Di	1000 ms (@ 1 Hz) / 100 ms (@ 10 Hz) (margin between WCET and T_i – 940 ms (@ 1 Hz) / 40 ms (@ 10 Hz))

Service-2 Frame Storage Thread Timing Analysis	
S_i	2
C_i	5-6 ms
WCET	15 ms (margin between C_i and WCET: 9-10 ms)
Ti	1000 ms (@ 1 Hz) / 100 ms (@ 10 Hz) (margin between WCET and T_i – 985 ms (@ 1 Hz) / 85 ms (@ 10 Hz))
Di	1000 ms (@ 1 Hz) / 100 ms (@ 10 Hz) (margin between WCET and T_i – 985 ms (@ 1 Hz) / 85 ms (@ 10 Hz))

These service thread timing analysis as observed is built around the philosophy that a margin of 10-15 milliseconds for both the services will be suitable as for the frame rate with the least error tolerant system (i.e. 10 Hz), the service deadlines will be at 100 milliseconds while the WCET would be approximately 60 and 15 milliseconds leaving about 25 milliseconds or in other words 25% of CPU utilization margin "free". This means that in case if the services override their decided WCETs, there will still be a 25 milliseconds or 25% CPU safe resource margin before which the deadlines may be missed. This can be considered a good and ideal design as according to RM LUB theory, the system services CPU utilization must be at or below 77.97% for three services (inc. of the sequencer thread which consumes about 1 milliseconds of WCET for every 1000 milliseconds of its execution time – as observed from the timestamps). Since, the RM LUB resource margin is met, it is evident the system services are schedulable while the worst CPU resource utilization can reach up to 77% (60% because of Service-1 thread, 15% because of Service-2 thread and approximately a maximum of 1-2% as a result of the Sequencer) which is still lower than the RM LUB. Therefore, the SRT system design is schedulable since no deadlines will be missed.

Please note, the system had a huge outlier in the timing test for the first frame that was read by the Service-1 thread and I realized that this was because my camera device is not capable of stabilizing so fast. To prevent that and the disruption of the first frame acquisition, I have implemented a mechanism where the Service-1 thread calibrates the camera by reading dummy frames for about 200 iterations in the beginning after which the system actually starts Service-2 execution for processing those good frames that are acquired after the first 100 iterations of Service-1 thread.

Hand-drawn SRT service scheduling using RMA

The hand-drawn SRT service scheduling for the two operating frequencies (1 Hz and 10 Hz) showcases that the system is dependable and schedulable. This is because in the case of the 1 Hz operation rate for service threads 1 and 2, the total WCET for both the threads including the Sequencer would be around 76 milliseconds (i.e. 60 milliseconds for Service-1 thread, 15 milliseconds for Service-2 thread and around 1 millisecond for the Sequencer for a 1000 millisecond time period). This leaves about 924 milliseconds of CPU time as ideal and provides a very huge error tolerance margin.

For the second case where the service threads are executed at 10 Hz, the system will have a resource margin of about 25 milliseconds (i.e. 60 milliseconds for Service-1 thread, 15 milliseconds for Service-2 thread and around 0.1 millisecond for the Sequencer for a 100 millisecond time period). This is again dependable and schedulable as the system will still meet the RM LUB pessimistic evaluation and that the service threads will not meet the deadlines with the assumed WCETs.

Please, refer to the Appendix section for the screenshot which shows the hand-drawn timing analysis in both the cases (i.e. 1 Hz and 10 Hz). Please note that the analysis does not involve the Sequencer as a third thread. This is because of two reasons: first because it consumes a very negligible portion of the CPU resource, second it has a higher priority than all other threads and will never be pre-empted (as it is the one giving out the semaphores to the service threads and scheduling them) and third, it will not affect or decide the schedulability of the services as it requires only about 0.1% of the total CPU resource utilization factor at 1 Hz and 10 Hz.

Cheddar analysis for the SRT service

Cheddar analysis for the SRT service shows how the same service parameters as discussed in the previous sections are schedulable as all deadlines will be met. The RM Cheddar analysis results are consistent with the above system design parameters at both 1 Hz and 10 Hz. Please, refer to the Appendix section for the screenshot which shows the RM Cheddar simulation and feasibility at both 1 Hz and 10 Hz.

Scheduling Point and Completion Test analysis

The Scheduling Point and Completion Test analysis using the appropriate algorithms was performed for both the cases. The algorithms test for the necessary & sufficient (N&S) feasibility test for the RM analysis. The screenshots provided in the Appendix section showcase that the algorithm portrays that the service parameters as discussed in the previous sections are feasible which is consistent with both the hand-drawn RM scheduling and the Cheddar analysis.

Proof-of-Concept

The Proof-of-Concept for the Time-Lapse project can be realized and is evident based on the image frames and MPEG video frames for both a natural transition in the environment and the transition of the clock with a resolution of 1 seconds at 1 Hz frame rate and a resolution of 1 millisecond at 10 Hz frame rate.

Implementation of RT scheduling policies

The system accounts for a reliable and error-free operation at 1 Hz. However, the system does face consistent and deterministic error issues when it was tested at the 10 Hz operating frequency.

As previously mentioned, there were errors of about 50-75 ms for every 610 minutes and these errors were accumulated over time. This implies that the system is deterministic with a deterministic error and a known offset that does not showcase any drift in SRT can be used to correct the system response in the real-world applications (which are NOT mission-critical, e.g. video streaming services).

Apart from the above observations, I have not been able to identify other faults in the system and the system reports for a good SRT system which may be used in non-RT applications. To focus here, the sub-sections in this section explains the underlying glitches present in the stopwatch that was used to perform the test. It undermines the sole responsibility for system incorrectness of my design and rather highlights other factors that could potentially lead to glitches in my system. The timestamps associated with image header frames will throw more light on the same.

Syslog timestamps tracing

The syslog timestamp tracing for both the 1 Hz and 10 Hz has been provided as a PDF file in the attached zipped folder. The files showcase the timestamps taken for an acquisition time of 3 minutes and 1 second at both 1 Hz and 10 Hz rate operation rates. The timestamp tracing shows that there are slight deterministic errors as reported above for the 10 Hz operation rate at certain deterministic intervals which can resolved either by debugging the code for errors or by adding a known offset to the system so that it calculates time reliably and efficiently in RT. The other aspect of this analysis would be the fact that no clock in reality is clearly accurate of time but has a limited resolution like the stopwatch that can read up to a resolution of 1/100th of a second or a 1/1000th of a second. At some level, these clocks would account for rounding off the errors which would on an average not impact the total timing. In a similar fashion, my project showcases that there is a consistent error which is however adding up and that proves that it is an unaddressed and unidentified bug in the code (possibly due to a common error like "offset by 1").

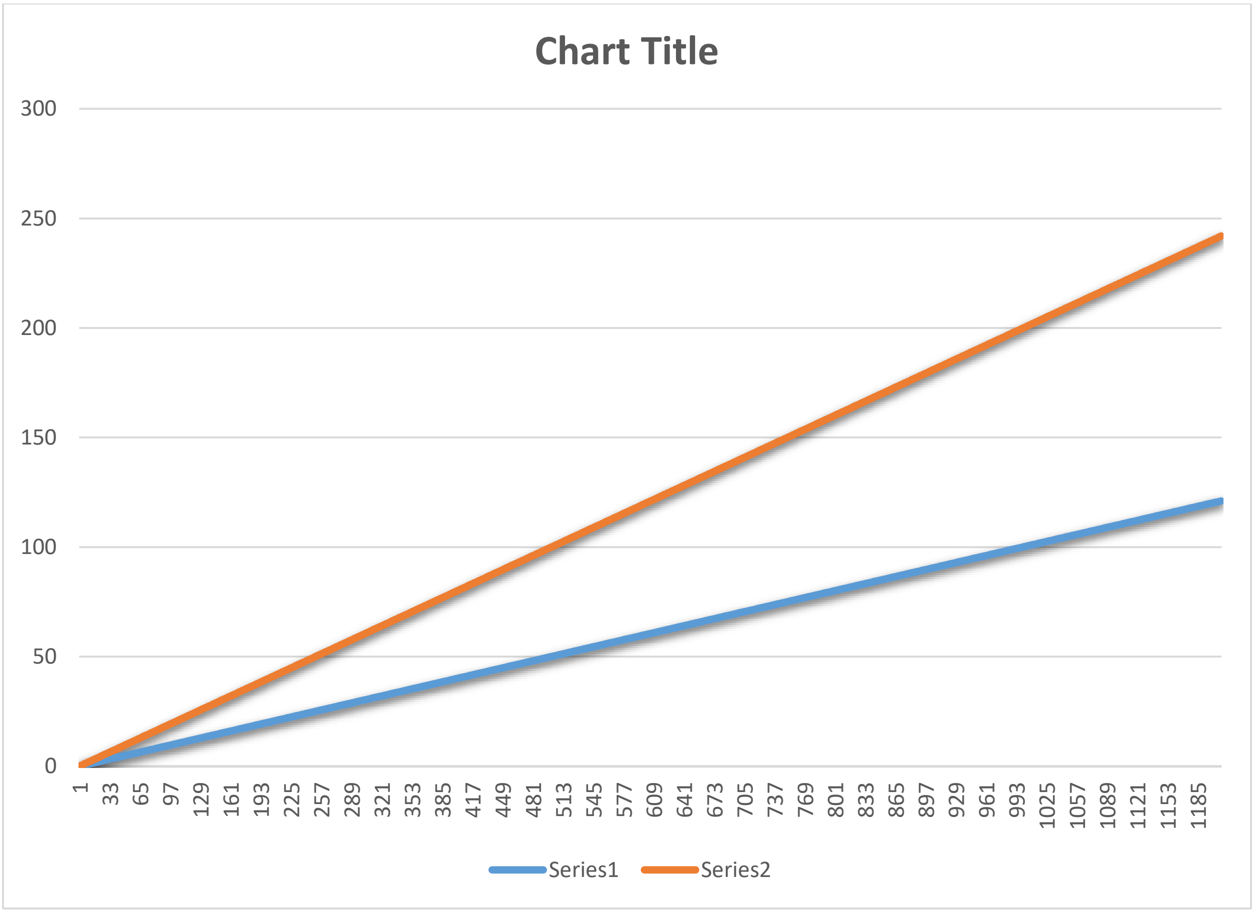
Clock Jitter Analysis

From the Jitter Analysis presented in this segment, it clearly appears that the error with the 10 Hz system testing is deterministic, gradually accumulating and is as a direct result of a bug in the code (possibly due to an "offset by one" error. The analysis shows that at certain intervals, once after about every 30 seconds since the system execution has begun, there appears to be a sudden jump in the frame timing differences from 0 milliseconds to approximately a few tens of milliseconds. Compared to the timing error from the beginning of the execution to the end of it at the 61st second (i.e. frame – 1 through 601 for the 10 Hz execution), it is seen that there is a sudden rise of approximately 75 milliseconds on an average in the timing differences. Ideally, the timing differences should fall way close to 1 minutes and 0 seconds. But given that the system is tested with a resolution of 1/10th of a second, an error of 75 milliseconds, is really significant. Although, this error is inevitable given the current code and the bug in it, a known offset could be implanted in the system that can provide accurate results in SRT (for non-critical missions).

Furthermore, from the fair comparison between the timing analysis of the system internal time differences using the image header file syslog stamps and the clock image frames, it often appears that the clock shows a considerable error of approximately twice or thrice more than the real error. For an example, at an instance in the attached file representing these timing differences, it shows that the timing difference between the 1st frame and the 601th frame (that is a period of 1 minute and 1 second) is supposed to show a time difference of 1 minute theoretically while the image header time stamps show a difference of 75 milliseconds. On the contrary, the clock represented in the image frames shows an error approximately of 200 milliseconds which is nearly 2.7 times the error represented by the timestamps.

The above observation establishes a simple fact that although the timing analysis for the 10 Hz test shows a significant error (100-200 milliseconds) as evident from the image frames, the timestamps speak differently and imply that the image frames are pessimistic about the level of error. Moreover, this highlights the very fact that the stopwatch used on the smart phone (that was ideally run using the internet connection) did not effectively provide accurate timing. More to this, the loading on the smartphone processor could directly affect the performance of that clock/stopwatch since it would not be ideally an application specific device focussed on HRT responses of that clock.

Given the above constraints, it emerges with inevitable integrity that the clock does in fact showcase more pronounced errors (which may then not appear to be deterministic and gradually accumulating). This helps in proving that the clock is pessimistic about the system response while the timestamps showcase the real limitations of the system with a deterministic, gradual and repeatable increase in the error of the system response over time. This pronounces the very principle that an ideal clock can theoretically never show any error like the straight line (x = y axis) as shown in the graph. However, in the real world case, the clock is guaranteed to have some error (either intermittently or deterministically) and these errors can be seen by the tightly coupled graph that is strongly correlated with the theoretical curve. However, the deviations represented by the clock show a more pessimistic picture and this highlights the fact that the system response is indeed as poor as is represented by the image frames but is obviously far from the theoretically anticipated observations.



Here, the y axis on the left 0-300 represents the theoretical time in milli-seconds while the Y-axis shows the syslog timestamps are shown on the X-axis. The red graph shows the timestamps values which deviate from the blue value which is the ideal value.

Conclusion

This Time-Lapse project has exposed to me some of the most commonly occurring challenges and design issues associated with the design and development of a SRT system. Along with that, I have realized and learnt some of the most commonly used SRT architectural features that prevent concurrency bugs, blocking, unbounded priority inheritance protocol and chained blocking. Moreover, I learnt about the inheritance of service thread priorities that can help prevent both chained blocking and unbounded priority inversion.

Some of the most commonly occurring design principles that I discovered and learnt while working on this project are as under:

- I/O latencies and de-coupling issues.
- Memory latencies and its impact on the SRT software system response.
- The presence of IPC mechanisms (e.g. message queues or circular ring buffers) and its impact on the responsiveness and reliability of POSIX thread implementations.
- The use of RMA in real-world scenarios where Linux RT patch (i.e. SCHED_FIFO pre-emptive scheduling) with the help of RMA can be put together to build SRT systems.
- The use of timestamps in real-world RT systems and the study of timing analysis associated with such RT projects.
- The RT timing designs that require an assumption and testing of the WCET for a service period/deadline that is user-specified and/or judged by the system parameters.
- Commonly occurring RT software bugs (e.g. not using synchronization primitives like semaphores that keeps the shared memory unprotected to unauthorized accesses and results into either a faulty service execution or a complete crash of the program) which can affect both the availability and reliability of the system. One that I am facing in this project was the "offset by 1 error" which was based on the fact that I would deterministically accumulate on an average an error of 50-75 ms for every minute of capturing time at 10 Hz. From the previous section, it was seen how an accumulation of 75 ms between the 1st and 601th would result into an error of 150 ms between the 601th and 1200th frame and so and so forth. For a 3-minutes 1-second testing time, this would often result into a 100-200 ms error with a complete success rate of 40-60% where ***no tracing statements (e.g. syslog timestamps, printf statements) are executed and only the most important clocking instructions are executed (execution of instructions like clock_gettime() is eliminated) – reducing the timing load on the system due to these tracing instructions.***
- The drifts associated with the SRT systems and how it impacts the overall system reliability and responsiveness. In my case, there was a 100-200 ms deterministic drift for a testing of grayscale image frames at a 10 Hz operation rate for a total of 1810 frames. This exposed the potential "offset by 1" bug in my code which I was unable to find out in the time that I had. Although, it was an error nonetheless, but since it was very deterministic, a known and reliable offset (with no drift in SRT) would work as a quick fix for systems that serve non-critical applications (e.g. video streaming). More to it, the error was seen for almost 60% of the time and its occurrence was however, non-deterministic. The use of MobaXterm was really helpful in increasing the odds of obtaining a reliable system response with no observable error for about 40% of the time (which is why I could not demonstrate it twice in a row!).

Along the more general lines, the Time-Lapse project not only exposed me to the real-world implementation and testing of a SRT system but also introduced me to the design issues and faulty assumptions that I could make which can result into negative outcomes in the real-world RT system designs if the services that I write for a particular RT applications are actually serving a great cause (i.e. human lives) – like the single point-of-failure RT issue with the Boeing MCAS.

References

Citations

[1]	Dr. Siewert's code for a Sequencer design
[2]	Linux man pages

Appendices

Due to the limitations for the length of this report, please refer to the attached zipped folder and the attachments link provided on the cover page of this report to access all the important files and documents.