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 11th, 2019***

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 which contains all the images and other files provided/explained in this report.

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.

Also, please note that the information provided in this report is consistent ONLY with the current development of the project. Further developments may see some modifications in the design to increase the responsiveness, reliability and efficiency of the SRT software system.

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 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 IO decoupling. 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. Further developments in this course project will see the operation of the board in a much resource-constrained SRT environment with limited error tolerance margin and with higher reliability on the user's end. The developments will also see extended features that can process the acquired frames from the camera device for logical operations and for stabilized frame storage before it is actually stored to the NAND Flash Memory. In this entire process, there are 2+ SRT services running on one CPU core in the specified board and further developments may depict additional complexities that exploit the high-end goals as mentioned in the project requirements here.

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. However, the current developmental phase of the project allows for using up to 4 other resolutions (see Appendix) with a common aspect ratio (i.e. 4:3).
- The SRT system acquires the frames from the camera device using the SRT environment that is supported by NPTL and processes it based on functional requirements like sharpening, continuous transformations (e.g. RGB to grayscale and vice versa).
- The current project developments allow for a reliable Time-Lapse capture of over 1800 frames (i.e. over 30 minutes of capturing time @ 1 Hz frame rate) with an error of about 3 seconds in the total timing calculation for approximately 1800 frames. 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 3 seconds error for a testing time of 30 minutes (i.e. 1800 frames) can be easily attributed to the memory erase latencies for its sectors which can be often as high as a couple of hundreds of milliseconds of execution time (i.e. 1000 milliseconds) for the board CPU resource. Existing developments also allow for a capturing frame rate of @ 10 Hz. The testing for this frame rate exposes higher latency issues and capturing issues as the error tolerance margin for the @ 10 Hz frame is greatly decimated. Upcoming developments will offload the SRT overload by running the image frame storage process as a best-effort service.
- The SRT system allows for testing the Time-Lapse acquisition using either RGB (.ppm) image frames or grayscale (.pgm) image frames. To reduce the amount of storage time latencies as well as the inevitable and often not very deterministic memory sector erase latencies, the use of grayscale image frames were used for acquisition. Also, the grayscale image frames (.pgm) consume about 300 KB of memory space vs 900 KB for the RGB (.ppm) image frames.
- The SRT system also provides time-stamps for every system software execution cycle and after every image frame is acquired and is stored. The timestamps are stored by the logger in the syslog file that keeps a record of all the execution instances.
- Further developments will showcase the inclusion of time-stamps in the header of every image frame (i.e. .ppm image frames) as a comment field along with the HOST name.
- Moreover, further developments will see additional features like CLI run-time modifications, offloaded processing onto non-RT CPU cores and logical comparisons for smart frame acquisition.

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. message queues), 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 IO-decoupling enables the SRT services to operate in a Non-blocking mode that is not affected by the failure of one system component for a SRT service deadline. The above mentioned SRT software elements come together as one SRT system which serves a Time-Lapse acquisition device.

The SRT software system uses 2+ SRT services on one CPU core of the board. For my project, the current development involves the implementation of 3 SRT services which are as under:

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 service 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. 100) image frames for allowing some time to stabilize the camera system 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 RGB or grayscale (.ppm or .pgm) file formats at a specified frame rate (i.e. 1 Hz or 10 Hz).

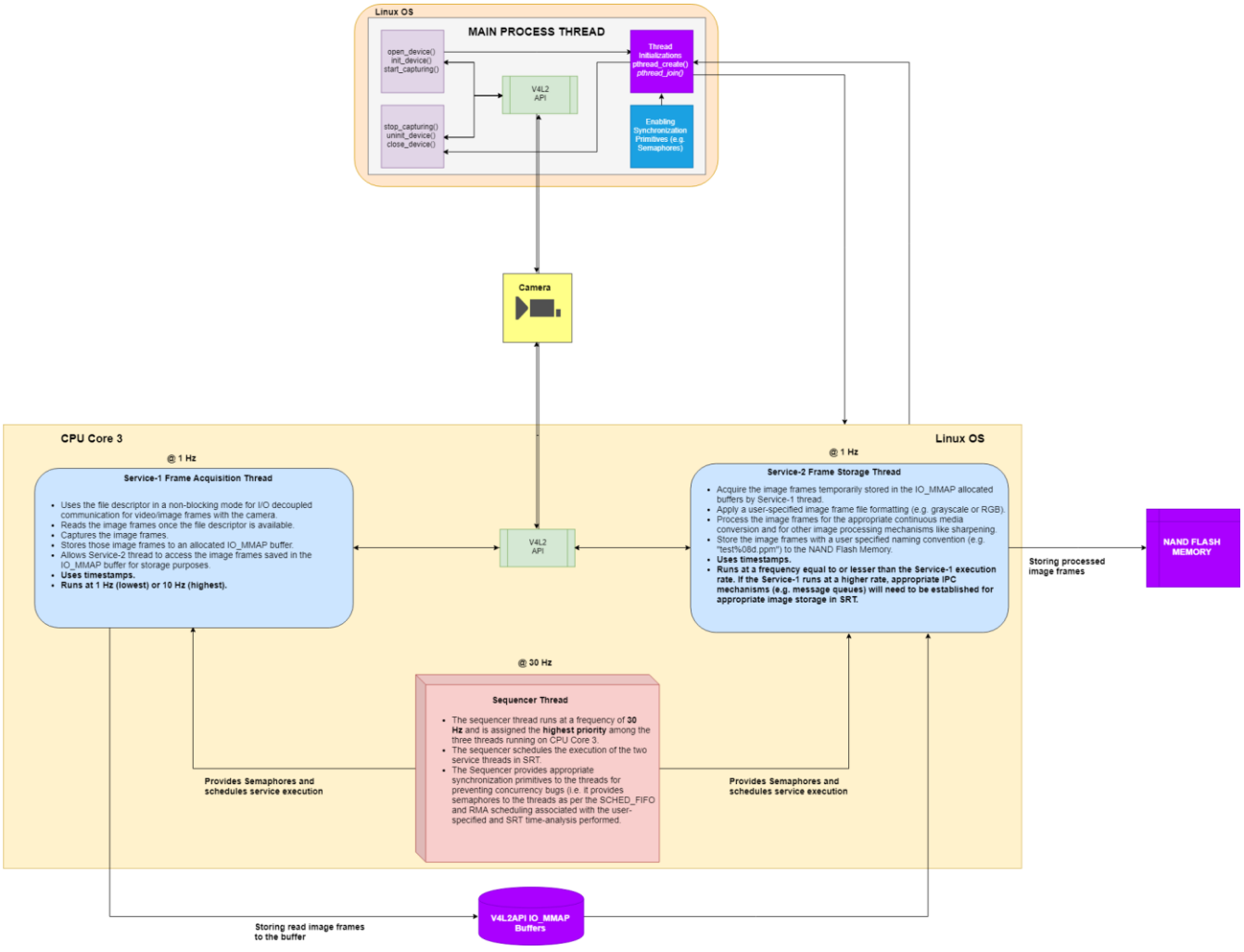
c. **Service-2 (Frame Storage Thread):** This is the second POSIX service thread that stores the image frames as the first service thread feeds it; ran at a similar or lower frequency than the first service thread. Message queues will be incorporated in the next development to increase the reliability, responsiveness and efficiency of the system by offloading the processing requirements of this service thread. 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).

The proposed SRT software system observes a well-tested set of timing values as discussed in the section on RT Analysis and Associated Design. Beyond that requirement, the SRT software architecture feeds in a set of parameters to the above POSIX threads in the RT execution. At this time, these parameters are dummy and do not serve any purpose but they are laid there in the program to reduce the latencies associated with the designing face later on. Further software developments will include the addition of features that will allow the service-2 thread to test the frames against the ones accumulated by the message queue buffer to pick the most stabilized frame and flush out the other ones as per necessity. Beyond these features, the 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. Upcoming developments will see additional best-effort service threads on a separate CPU core that will account for frame processing (i.e. sharpening), storage, ffmpeg video conversion, etc. The existing source code is not very CLI friendly but upcoming developments will see improvements in addition to extra stretch and target goals for the SRT system.

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



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
3. V4L2 API calls for camera setup and closing
4. The thread initialization block

5. The synchronization primitives initializer block
6. The SRT CPU Core block running 3 SRT services
7. V4L2 IO_MMAP allocated buffers
8. 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 that will run on the CPU Core 3. The thread parameters (although not utilized by the threads at this time) will be passed to the service threads. The parameters are added at this time and they are dummy but it is done so only to reduce the system upbringing time when new developments are introduced.

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 (i.e. the pthread_join() function is in the process of being executed which means that the 3 SRT services are being serviced by the OS), the OS shifts it focus to the CPU Core 3 where all the 3 SRT services will be executed. The Sequencer will have the highest priority and will run at 30 Hz while the service threads for image frame capturing and image frame storage will be run at around 1 Hz or 10 Hz depending on the need of the user.

Once the thread executions is done with the CPU Core 3 SRT services, 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.

Service-1 thread (for frame acquisition):

- 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.
- Stores those image frames to an allocated IO_MMAP buffer.
- Allows Service-2 thread to access the image frames saved in the IO_MMAP buffer 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 and storing it to V4L2 buf
```

Service-2 thread (for frame storing):

- Acquire the image frames temporarily stored in the IO_MMAP allocated buffers by Service-1 thread.
- Apply a user-specified image frame file formatting (e.g. grayscale or RGB).
- Process the image frames for the appropriate continuous media conversion and for other image processing mechanisms like sharpening.
- 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, appropriate IPC mechanisms (e.g. message queues) will need to be established for appropriate image storage in SRT.**

Functions used:

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

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 allocated IO_MMAP buffer with the image frames. 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 buffer memory and will dump the image file into a specified file format using the specified naming convention (e.g. test00001800.ppm) either in the user space or the root space in the NAND Flash Memory. Service-2 thread will consume about 5-6 milliseconds of execution time. 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 100 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.

There were however issues with the first image frame as was previously mentioned in this report which was satisfactorily solved by calibrating and stabilizing the camera by letting it read dummy frames for about 100 iterations after which the SRT service thread will begin processing it for storage purposes.

Apart from that, there were issues with the 10 Hz testing of this project. There were intermittent errors of about 1 second in the image capturing and storage for the SRT system environment. I suspect that this was because – I am requiring the second service thread to both process the image frame stored in the allocated IO_MMAP buffer (which was stored in the buffer by the first service thread) as well as make it store the image frame in the /root/… directory. At certain intervals, the frame storage V4L2 API may be required to erase sectors of memory which is often associated with the highest memory latency (often about a few hundreds of milliseconds). This could be the reason that the Service-2 thread would be missing the deadlines at intermittent instances where the NAND flash memory has to be erased. At those instances, the Service-2 thread would not access the shared memory buffer that the Service-1 thread is updating every time it is reading a frame and therefore although Service-1 thread may have reliably read the frame in SRT, the distorted execution of Service-2 thread would have resulted into it not storing the frames when it encounters the sector erase latency and that explains – the intermittent errors in the image frames captured at 10 Hz. Moreover, for the test performed at 1 Hz for a time of 30 minutes (i.e. 1800 frames), an error of 3 seconds was observed. The said image frames are submitted in the attached zipped folder. Again, I suspect that the sector erase latency may be the real reason behind the error of 3 seconds.

What strengthens the above observations is the fact that this 3 seconds error was attributed to a test performed for 1800 frames at 10 Hz. While the test frames were increased, this error rate increased linearly and this more than implies the underlying conviction that the sector erase or memory latencies were associated with the above error. As a token of improvement, in the upcoming developments, I will be shifting the frame storage process to a different CPU core and will be running it as a best-effort service thread. That way, the SRT system responsiveness and reliability will increase and the error tolerance margin will not be greatly affected.

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 of 20 test grayscale (.pgm) frames at 1 Hz and an acquisition of 290 test grayscale (.pgm) frames at a 10 Hz rate. The timestamp tracing shows that there is a slight anomalies in both the case for the first few frames.

For the 1 Hz tracing, it can be seen that there is an error of about 1 millisecond for the thread release timestamps as can be observed using the tracing PDF files. This 1 millisecond error is intermittent and again it can be as a result of the program code that isn't well modularized at this time (with excessive features and/or parameters that aren't actually used at this time but added to reduce the system design time for further developments). Moreover, the Sequencer thread can be seen do some task based on the return code values and that actually is slightly reducing the Sequencer frequency rate (with a period slightly lesser than 33.33… milliseconds). This is suspected to be a potential reason behind the 1 millisecond-backlog for the 1 Hz observation as can be seen from the thread release timestamps.

For the 10 Hz tracing, it can be seen that there is an intermittent error of a couple of hundreds of milliseconds in the thread release timestamps. Ideally, the gap between the thread release timestamps should be 100 exactly milli-seconds but often there is an error of about 1 millisecond in the thread release timestamps. Again, the suspect for this problem is assumed to be the slightly damped execution frequency of the Sequencer thread. In addition to what is observed here, there is an intermittent error in the thread release timestamps which can be around 1 second and as previously mentioned in the other sections of this report, the memory latencies is suspected to cause this issue (in addition to the not-well modularized existing codebase).

With further developments involving the below improvements, I am very firm that the system would be able to overcome these challenges and provide a completely reliable SRT responsive for the service threads:

1. Modularizing the code and reducing the heavy overload caused by unused thread processing elements (i.e. the function calls that the thread makes which are not utilized by the user at this time).
2. Offloading the frame storage process to a different CPU Core and making it to follow the best-effort approach to overcome non-deterministic memory latencies.
3. Overcoming the Sequencer period issues and testing for an exact Sequencer period of 33.33 milliseconds.
4. Use of message-queues and other IPC mechanisms to reduce the service thread execution interdependencies.
5. Introducing redundancy into the frame acquisition to reduce the jitter, blurring of some frames and involving more image processing logic to increase the system reliability and accuracy.
6. Involving the use of certain image storage pre-processing mechanism using the best-effort approach to aid the introduced redundancy for much sharp and desirable frames.
7. Introducing certain command line interface (CLI) features.
8. Reducing the presence of magic numbers in the code.
9. Burning more image frames in the beginning to reduce issues arising out of camera insensitivity and lack of responsiveness because of its instability during the first few frames that are sent.
10. Simplifying the software system architectural design even more to avoid inevitable and non-deterministic latencies that may not have been identified so far and can be potentially a reason behind the errors and challenges that are existing with the project at this time.

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) 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.
- The drifts associated with the SRT systems and how it impacts the overall system reliability and responsiveness. In my case, there was a 3-second drift for a testing of grayscale image frames at a 1 Hz frame rate for a total of 1800 frames. This exposed the potential latency issues with the memory and its presence in the SRT execution where the margin jitter test was ran only for 100 iterations and the error was not detected and accounted for in these 100 random iterations. But, there was a presence of this potential latency issue in 18 times the iterations that were performed (i.e. 1800 frames vs 100 iterations of 1 frame) and 3 frame issues were identified which is an error of 0.167% and can still have huge negative implications, although not catastrophic in case of SRT systems.

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 limitations for the length of this report, please refer to the attached zipped folder for the required files.