Assignment 2: Timing in Robotic/ Cyber-Physical Systems

Bishoy Essam Samir Yassa Gerges, bishoy.gerges@student.utwente.nl, s2921073, M-ROB
Panteleimon Manouselis, p.manouselis@student.utwente.nl, s3084493, M-ROB

2.1 - TIMING OF PERIODIC NON-REALTIME THREAD

The objective of this sub assignment is to present the implementation and analysis of a program that creates a POSIX thread containing timed periodic code which will be executed on a Raspberry Pi 4. The primary objective of the program is to execute a while loop, spending exactly 1.0 ms on each iteration, by utilizing either `clock_nanosleep` or `create_timer/sigwait` (we selected `clock_nanosleep`). To ensure that the loop spends some time executing computational work, some mathematical calculations are performed within each iteration. More specifically, the mathematical work executed every iteration (i.e., computational load inside the while loop) is the following:

```
int j = rand() % 100;
double x = 0.0;
for (int k = 0; k < 10000; k++) {
x += (double)j / (double)(k+1);
}
```

The parameter "k" is adjustable to modify the time needed for the computational work to execute within each iteration of the while loop. After several trial-and-error attempts, an appropriate value for "k" was determined. A value greater than 10^5 will cause the loop to spend more than 1ms executing the mathematical calculations, while a value less than 10^3 will result in the computational work being too insignificant to affect the loop execution time. Therefore, we selected $k = 10^4$.

To understand the performance characteristics of the timed periodic execution of code on a Raspberry Pi 4 and investigate how extra computational load affects the timing performance of the loop we examine the timing of the loop in two different scenarios: without any additional processor load, and with additional processor load from the `stress` command. The results are presented graphically, in Figures 1, 2, 3, 4, 5 and 6.

From the Figures, we can see that the mean execution time of the loop is close to 1.2 ms both when we don't stress the Raspberry and when we execute `stress c -4 m -2`. By executing `stress c -4 m -2` we generate 4 CPU "workers" and 2 memory "workers. Each CPU worker generates CPU load by continuously computing the square root of a random number" whereas each memory "worker" hogs 256MB of free memory. From figures 1, 3 and 5, we can observe that the execution time varies around the mean value, indicating that there is some degree of jitter in the execution time. This jitter can be seen from the yellow line plots. As expected, the amount of jitter is directly proportional to the stress applied.

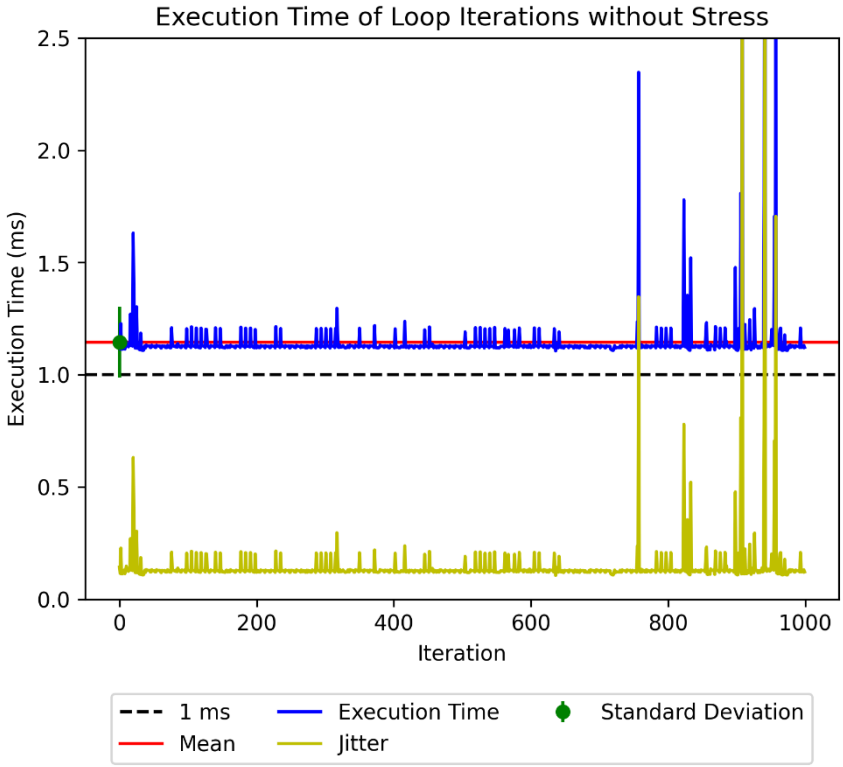

Figure 1: Execution time, mean, jitter and standard deviation of each iteration of while loop without stress

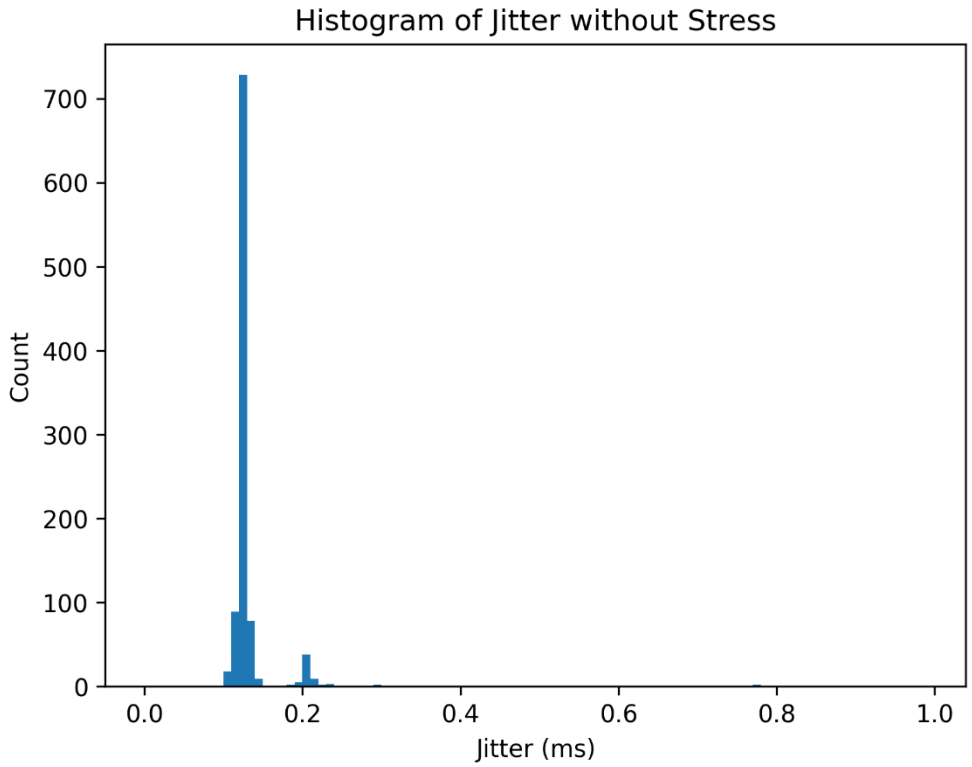

Figure 2: Histogram of jitter without stress

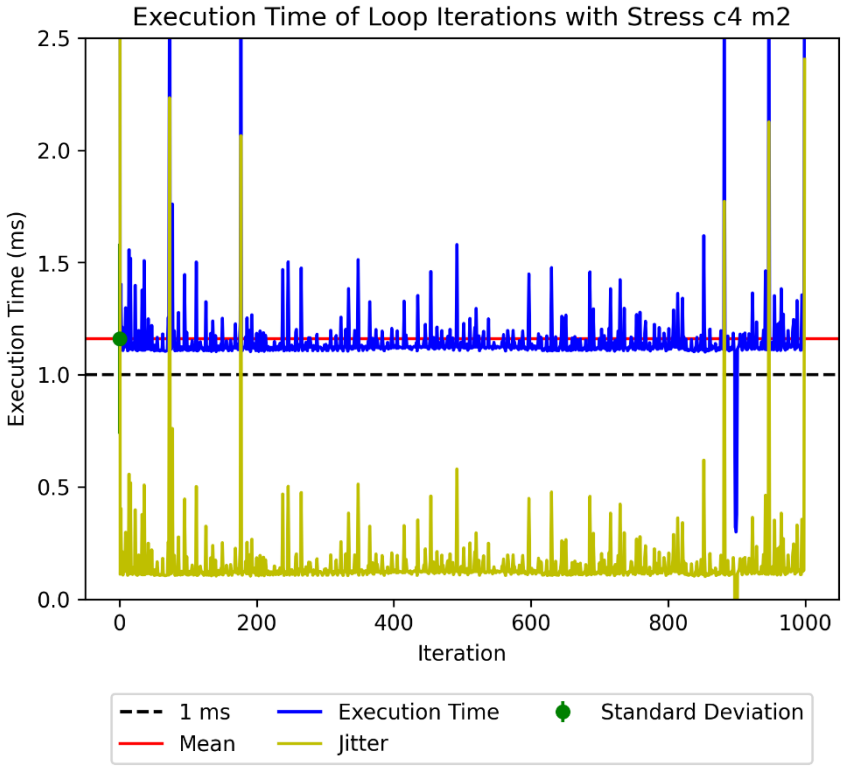

Figure 3: Execution time, mean, jitter and standard deviation of each iteration of while loop with stress c -4 m -2

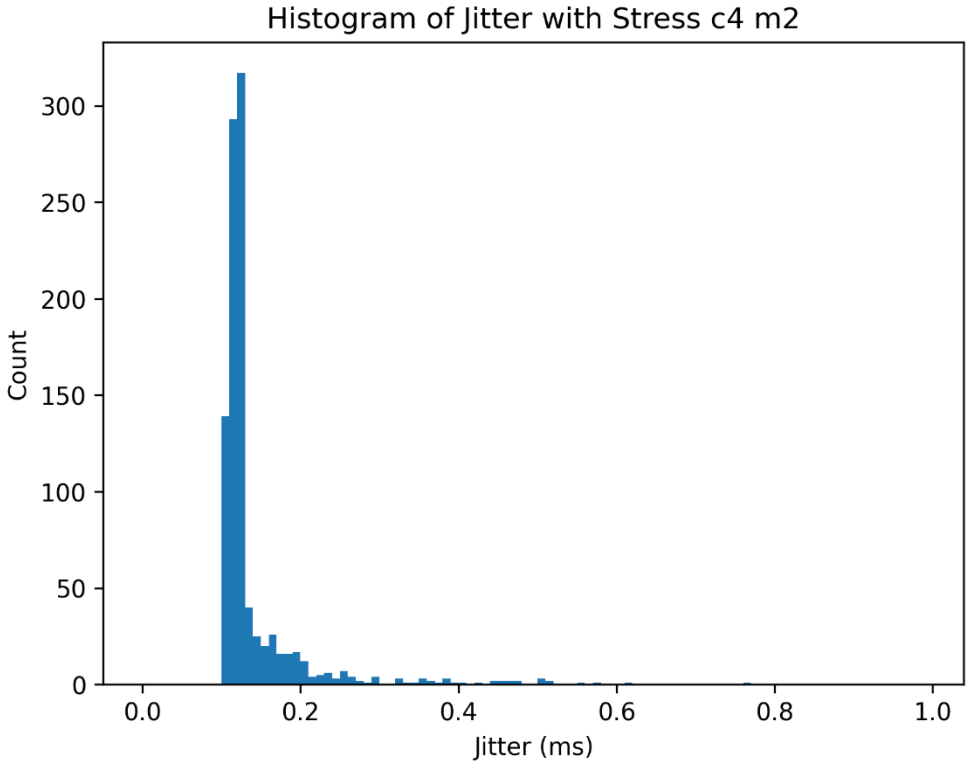

Figure 4: Histogram of jitter with stress c -4 m -2

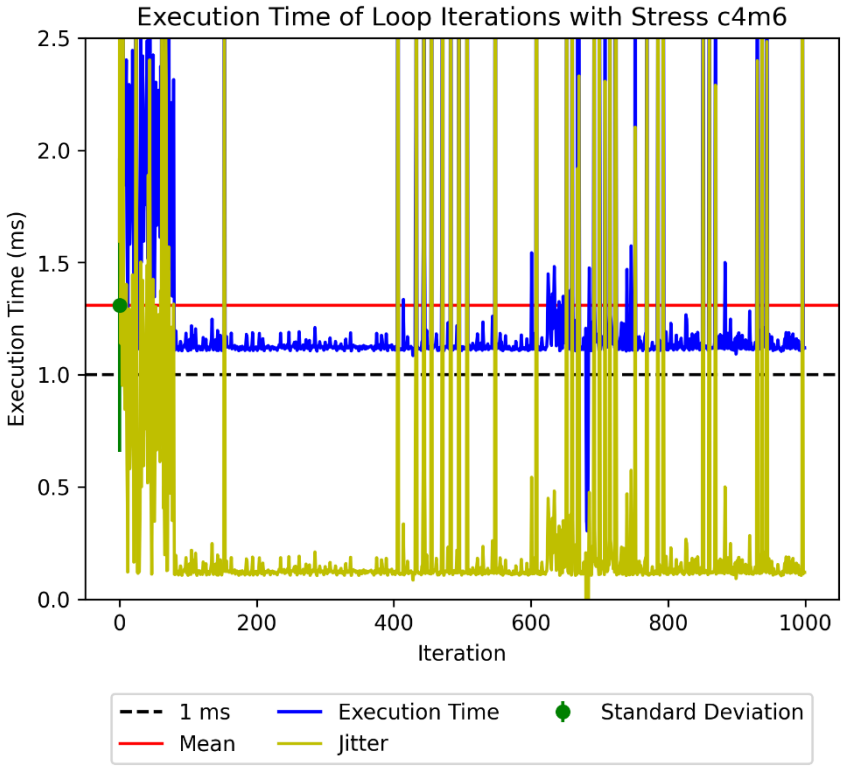

Figure 5: Execution time, mean, jitter and standard deviation of each iteration of while loop with stress c -4 m -6

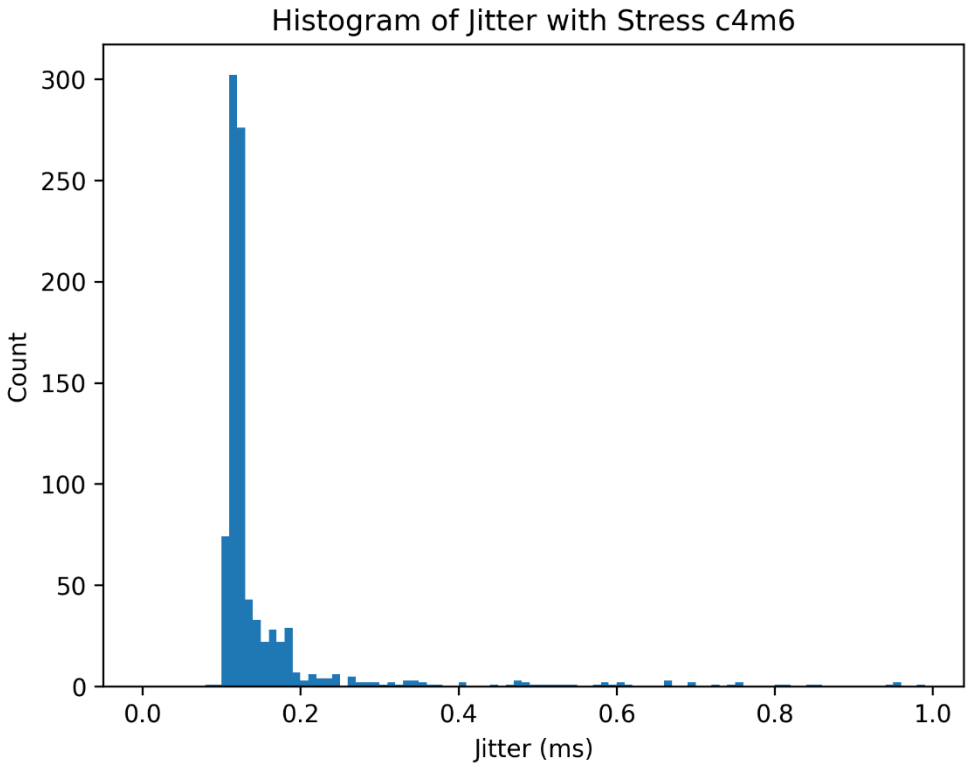

Figure 6: Histogram of jitter with stress c -4 m -6

In the first plot (without stress), the jitter appears to be relatively small, with the execution time mostly varying between $1.05\ and\ 1.15\ ms$. This suggests that the Raspberry Pi is able to execute the loop consistently under no additional load. In Figures 3 and 5 (with stress), the jitter appears to be larger, with the execution time mostly varying between $1.05\ to\ 1.4\ ms$ in the first stress test and $1.1\ to\ 1.5\ ms$ for the second stress test. This suggests that under additional load, the Raspberry Pi is less consistent in its execution of the loop, and that the extra computational load affects the timing performance of the loop. This is evident by the mean time which increases when we apply stress. More specifically, in the fifth plot (`stress c -4 m -6`) the mean time has jumped to $1.3\ ms$ (from the $1.15\ ms$ measured without stress).

Furthermore, we can observe that the standard deviation of the execution time (shown as a green error bar) is larger in the plots with stress than in the plot without stress, indicating that the extra computational load has increased the variability of the execution time. Overall, these results suggest that the Raspberry Pi's timing performance can be affected by extra computational load, and that the execution time of a loop can vary around its mean value.

2.1.1 - *EXPLAIN WHAT TIMING ASPECTS WE ARE INTERESTED IN AND WHY. GIVE AT LEAST TWO WAYS OF REPRESENTING THE MEASURED DATA. DISCUSS ADVANTAGES AND DISADVANTAGES OF THEM.*

In the software of robots, timing aspects play a critical role in achieving accurate and predictable closed-loop control. Two essential aspects of timing that we are interested in are the mean execution time and the jitter.

The mean execution time represents the average time taken to execute a (closed) loop iteration. It is a crucial parameter in closed-loop control as it affects the sample time, which is the time between consecutive measurements and control actions. A consistent and predictable mean execution time is necessary for stable and accurate control.

The jitter represents the variation in execution time between consecutive loop iterations. It is an important parameter in closed-loop control as it determines the maximum allowed delay in execution. If the jitter is too high, it can result in unstable and inaccurate control.

There are various ways to represent the measured data. Two common ways are:

- Line plot: A line plot shows the execution time of each loop iteration as a function of time. It is useful for visualizing the variation in execution time between consecutive iterations. However, it can be difficult to extract the jitter from a line plot (especially if the plot has been smoothened).
- Histogram: A histogram shows the frequency distribution of the execution time. It is useful for visualizing the distribution of execution time and identifying any outliers. It is also easy to extract the mean execution time and the jitter from a histogram.

The advantage of a line plot is that it provides a detailed view of the variation in execution time between consecutive iterations. However, it may not be easy to extract the jitter. The advantage of a histogram is that it provides a clear view of the distribution of execution time and allows easy extraction of the mean execution time and the mean jitter per bin. However, it will not provide a detailed view of the variation in execution time between consecutive iterations.

2.1.2 - EXPLAIN WHY YOU HAVE CHOSEN THE METHOD YOU USED (EITHER CLOCK_NANOSLEEP OR CREATE_TIMER/SIGWAIT). WHAT ARE ADVANTAGES AND DISADVANTAGES OF BOTH METHODS?

In our code we selected the method `clock_nanosleep()`. Advantages of using `clock_nanosleep()` over `create_timer()` and `sigwait()` methods include:

- Precise control over sleep time: `clock_nanosleep()` allows for accurate specification of the sleep time in nanoseconds, allowing for more precise timing of code execution.
- No overhead from signal handling: `clock_nanosleep()` does not require any signal handling overhead, unlike `create_timer()` and `sigwait()`.
- Portable across POSIX-compliant systems: `clock_nanosleep()` is a standard function available in the POSIX library, which means it is portable across POSIX-compliant systems.

Disadvantages of using `clock_nanosleep()` over `create_timer()` and `sigwait()` methods include:

- Limited functionality: `clock_nanosleep()` can only be used for sleeping the thread, whereas `create_timer()` and `sigwait()` methods can be used for a wider range of timing-related tasks.
- Not suitable for very long sleep times, as the `clock_nanosleep()` function may be interrupted by signals.

2.1.3 – WHAT IS THE DIFFERENCE BETWEEN CLOCK_MONOTONIC AND CLOCK_REALTIME? WHICH ONE IS BETTER FOR TIMING A FIRM/HARD REAL-TIME LOOP?

`CLOCK_REALTIME` represents the actual wall-clock time. This clock is affected by system time adjustments, such as NTP adjustments (i.e., adjustment of system time over a network like the internet). `CLOCK_REALTIME` can be used for determining the current time, measuring time intervals, or setting timeouts for system calls.

On the other hand, `CLOCK_MONOTONIC` represents the monotonic time, which is a time that is not affected by system time adjustments. This clock measures the time elapsed since some unspecified starting point, and it always moves forward, never backward. `CLOCK_MONOTONIC` is useful for measuring elapsed time intervals, as it provides a consistent time reference that does not jump backward or forward due to system time adjustments.

For timing a firm/hard real-time loop, `CLOCK_MONOTONIC` is the preferred choice, as it provides a more stable and accurate time reference that is not affected by system time adjustments. In real-time systems, it is essential to have a stable and predictable timing source, and `CLOCK_MONOTONIC` provides this stability by guaranteeing that time always moves forward, never backward. Therefore, it is important to use `CLOCK_MONOTONIC` when dealing with real-time systems, as it provides a more reliable and predictable timing source.

2.1.4 - WHEN USING `create_timer/sigwait`, WHAT HAPPENS WITH THE TIMER YOU INITIALIZED WHEN THE PROGRAM ENDS? AND WHAT HAPPENS WHEN IT CRASHES DUE TO SOME ERROR?

When the program ends, any active timers created using `timer_create()` will be destroyed automatically, along with any associated signal handlers. This is because the memory allocated to the timers and signal handlers is tied to the process that created them, and is released upon termination.

If the program crashes due to some error, any active timers created using `timer_create()` will not be automatically destroyed. This means that the memory allocated to these timers and associated signal handlers will not be released when the program stops. This can potentially cause problems if the program is restarted and tries to create new timers with the same names, as the old timers may still exist and cause conflicts or unexpected behavior.

2.1.5 – IS IT A GOOD IDEA TO USE THE APPROACH USED IN THIS SUBASSIGNMENT FOR CLOSED-LOOP CONTROL OF A ROBOTIC SYSTEM? WHY (NOT)?

It is generally not a good idea to use the approach used in this subassignment for closed-loop control of a robotic system. The reason is that the timing of the loop in this approach is not guaranteed to be precise and consistent (apparent from the jitter and mean execution time of each loop presented in Figures 1,2 and 3). The loop relies on a sleep call to delay the execution of each iteration, but the actual duration of the delay may vary depending on factors such as the load on the CPU (e.g., when stressed the sleep time was smaller), the scheduling of other processes, and the latency of the sleep call itself. As a result, the loop may not run at the desired frequency and may exhibit jitter in its timing.

As we mentioned previously, in closed-loop control of a robotic system, precise and consistent timing is critical for ensuring accuracy. The control loop must run at a fixed frequency that is synchronized with the physical dynamics of the system, and the timing of each iteration must be predictable and repeatable (minimum jitter). In addition, the loop must be able to respond quickly to changes in the system state or the control inputs, which requires a low latency and a high bandwidth. To achieve these requirements, one should use specialized real-time operating systems or real-time extensions to a normal OS (see Xenomai section below). These systems provide deterministic timing and low-latency communication between processes.

2.2 – TIMING OF PERIODIC REALTIME THREAD (XENOMAI)

In this sub assignment, the program was compiled to be run by Xenomai kernel instead of Linux kernel on Raspberry Pi. The Xenomai kernel runs on two cores and the Linux kernel runs on another two cores of the four cores of the Raspberry Pi. For comparison with the non-realtime case, the same analysis has been done here as that of (2.1). That is, we examined the timing of the loop in two different scenarios: without any additional processor load, and with additional processor load from the `stress` command. To assess the real-time capability of the periodic thread running on Xenomai, the Jitter and Loop Durations are going to be investigated.

The period (iteration) duration should be ideally equal to 1.0 ms. In Figure 7, Figure 9 and Figure 11 we can see the duration of each iterations and its corresponding Jitter duration without stress, with `stress c -4 m -2` and with `stress c -4 m -6` respectively. For the three cases, the mean and standard deviation of the total number of loops are: $1.080\ ms\ and\ 0.031\ ms$ for the first case in Figure 7, $1.072\ ms\ and\ 0.022\ ms$ for the second case in Figure 9 and $1.072\ ms\ and\ 0.024\ ms$ for the third case. It is seen that the extra computational load (stress) does not significantly affect the timing. In fact, the performance was better under stress but the difference is negligible.

As seen in the histograms in Figure 8, Figure 10 and Figure 12, the Jitter is not largely distributed but concentrated between values of 0.05 ms and 0.15 ms. That is why the standard deviation in the previous paragraph is very small. Again, the computational load does not significantly affect the timing of the loops.

Figure 7: Execution time, mean, jitter and standard deviation of loop durations without stress (Xenomai)

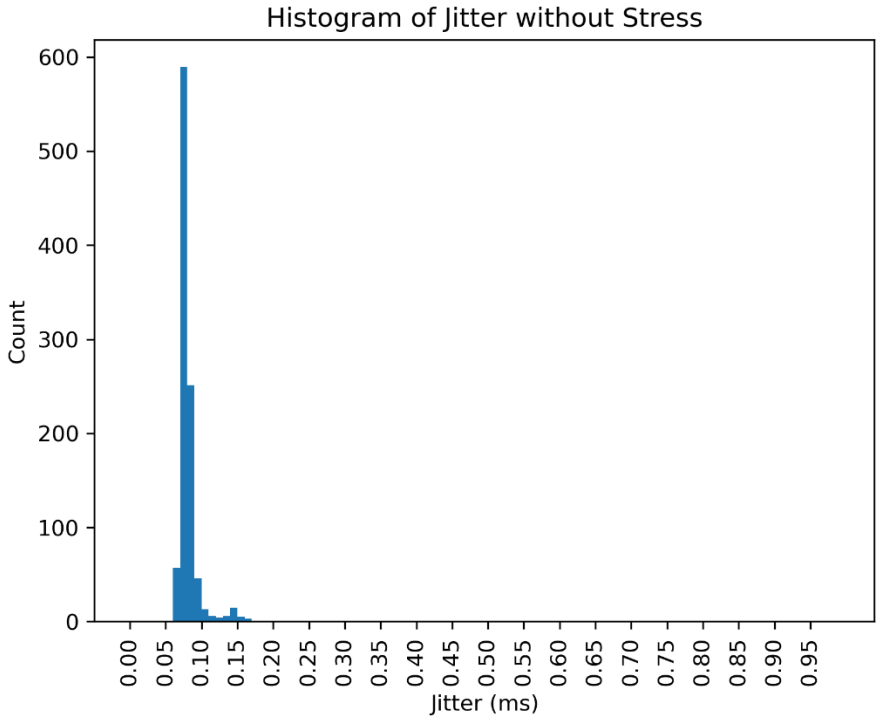

Figure 8: Histogram showing Jitter count with bins of 0.1 ms width. The processor is not stressed (Xenomai)

Figure 9: Execution time, mean, jitter and standard deviation of loop durations with stress c -4 m -2 (Xenomai)

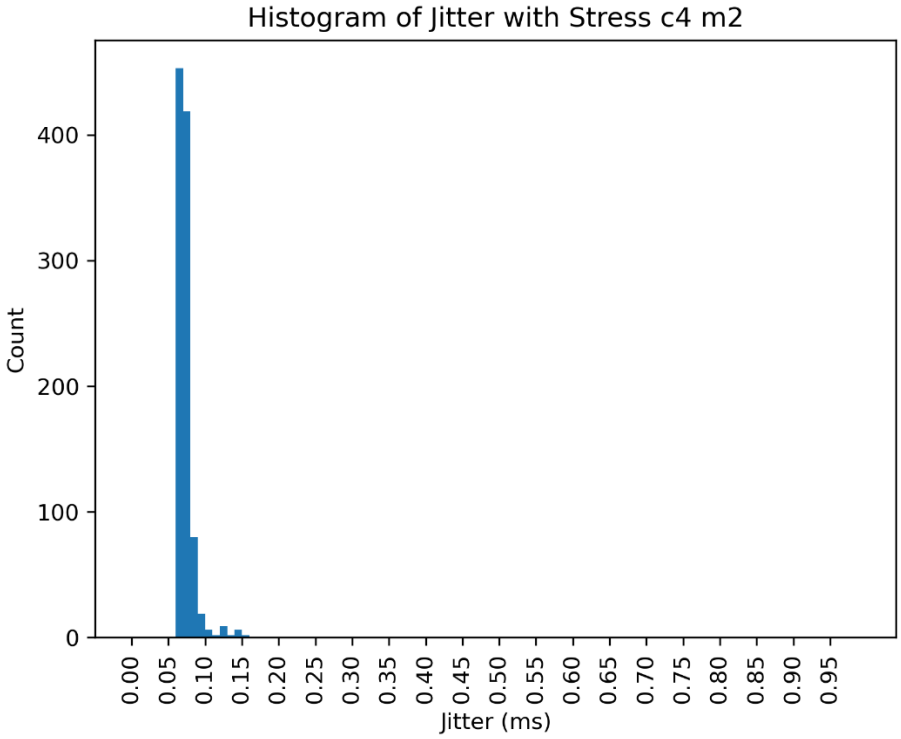

Figure 10: Histogram (bins of 0.1 ms width) showing Jitter count with stress -c 4 -m 2 (Xenomai)

Figure 11: Execution time, mean, jitter and standard deviation of loop durations with stress c -4 m -6 (Xenomai)

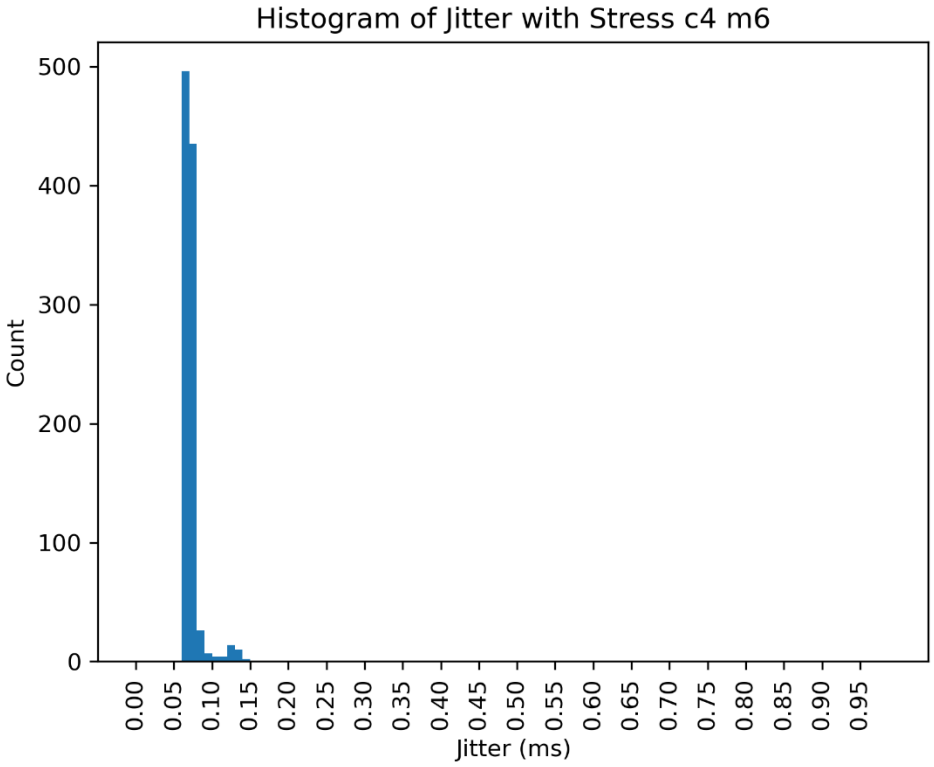

Figure 12: Histogram (bins of 0.1 ms width) showing Jitter count with stress -c 4 -m 6 (Xenomai)

2.2.1 – FOR EACH POSIX COMMAND USED IN YOUR PROGRAM, EXPLAIN IN YOUR OWN WORDS WHAT IT DOES AND WHY YOU NEED IT FOR A REAL-TIME PROGRAM.

- **`pthread_create(&thread_id, NULL, &timed_loop, NULL);`**
  This command creates a new POSIX thread and assigns it a unique thread ID, which is stored in the variable `thread_id`.
  The `pthread_create()` function here takes four arguments (in order from left to right):
  - `thread_id`: a pointer to a `pthread_t` variable that will store the ID of the newly created thread.
  - `NULL`: a pointer to a `pthread_attr_t` structure that specifies the attributes for the thread. In this case, `NULL` means that the thread will be created with default attributes.
  - `&timed_loop`: a pointer to the function that the new thread will execute. In this case, it is a function named `timed_loop`.
  - `NULL`: an optional pointer that can be used to pass arguments to the thread function. In this case, there are no arguments being passed, so `NULL` is used.

  The choice of arguments passed to `pthread_create()` can influence the real-time behavior of the program. In particular, the choice of thread attributes can affect the scheduling policy and priority of the thread, which in turn can impact the thread's ability to meet its real-time deadlines.

  In lab work on the Raspberry Pi, for time efficiency, we tested the thread with the default values of the thread attributes, and we did not have time to test and compare different combinations of scheduling parameters. However, the default values are not the ideal values for the best real-time performance. We can choose the `SCHED_FIFO` scheduling policy with the highest possible priority, which is typically `99` on Linux systems. The `SCHED_FIFO` policy uses a priority-based scheduling algorithm, where threads with higher priorities preempt threads with lower priorities. This policy guarantees that higher-priority threads will always execute before lower-priority threads, providing predictable and deterministic behavior. However, setting the thread priority to the highest possible can potentially affect the system's overall performance, as other threads might not get enough CPU time. To achieve this scheduling policy, we would use:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
      struct sched_param sched_params;
sched_params.sched_priority = 99
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
pthread_attr_setschedparam(&attr, &sched_params);
pthread_t thread_id;
pthread_create(&thread_id, &attr, &thread_func,
NULL);
```

- **`pthread_join(thread_id, NULL);`**
  This function is used to wait for a thread to terminate and obtain its exit status. When a thread is created with `pthread_create`, it runs independently of the thread that created it. The `pthread_join` function provides a way for the creating thread to synchronize with the created thread and wait for it to complete before continuing execution.
- **`clock_gettime(CLOCK_MONOTONIC, &start_time);`**

The `clock_gettime` function is used to retrieve the current time with a specified clock. `CLOCK_MONOTONIC` is one of the clock type and it represents the monotonic time since some unspecified starting point, and is not affected by changes in the system time. The monotonic nature of `CLOCK_MONOTONIC` ensures that the time measurements are accurate and not affected by changes in the system time. On the contrary, `CLOCK_REAL_TIME` is affected by changes in the system time (such as adjustments due to time synchronization or daylight saving time). Unlike `CLOCK_MONOTONIC`, `CLOCK_REAL_TIME` is not guaranteed to be monotonic, and its values can be adjusted by the system. This makes it less suitable for measuring time intervals and calculating timeouts in real-time systems, since changes in the system time can affect the accuracy of the measurements.
However, `CLOCK_REAL_TIME` can be useful in situations where the absolute time is important, such as in scheduling events that are tied to real-world time (such as alarms or timers). So, the choice of clock type depends on the specific requirements of the application. For real-time systems, where accurate and predictable timing is critical, `CLOCK_MONOTONIC` is often the preferred choice, that is why it was chosen here.

- **`clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &sleep_time, NULL);`**
  This is used to sleep the thread until the absolute time specified in the `sleep_time` variable is reached. `CLOCK_MONOTONIC` is one of the clock types and it was explained and motivated in the previous bullet point. Here, the same clock type is used to be consistent. `TIMER_ABSTIME` is a flag that specifies that the sleep time should be interpreted as an absolute time rather than a time interval. This is the best option for best real-time performance because `TIMER_ABSTIME` allows for more precise timing than using an interval by setting a waking absolute time. On the contrast, if an interval is used, this interval may not be accurate because it requires calculating the time consumed in computation of the thread task and subtract it from the desired period time to calculate the sleeping time. This can be affected, for example, by not accounting for all the computational time done by the thread before commanding the sleep.

2.2.2 – COMPARE THE TIMING PERFORMANCE OF THIS PROGRAM WITH THE PROGRAM FROM SECTION 2.1. WHAT CAN YOU CONCLUDE ON TIMING WITH RESPECT TO REAL-TIME CAPABILITY?

By looking at the results of realtime (Xenomai) explained in this Section 2.2 and comparing them with that of 2.1, it is shown that the program here is more real-time capable than the program in Section 2.1.

- By comparing Figure 1 with Figure 7, we can see that the thread period is met with more accuracy in case of using Xenomai than that in case of Section 2.1. This can be observed in the mean and standard deviation values:
  - ***Section* 2.1** (Figure 1):

$$Mean = 1.15\,ms$$
$$Standard\ Deviation = 0.022\,ms$$

  - ***Section* 2.2** (Figure 7):

$$Mean = 1.072\,ms$$
$$Standard\ Deviation = 0.30\,ms$$

- By comparing the Jitter histograms in Figure 2 with Figure 8, it is seen that Jitter in case of using Xenomai is much lower than that in Section 2.1.

- Additionally, by comparing the performance in case of stress, that is Figure 3, Figure 4, Figure 5 and Figure 6 against Figure 9, Figure 10, Figure 11 and Figure 12 respectively, we can see that the performance in case of using Xenomai is not affected under processor loading which is not the case in Section 2.1

2.2.3 – IS IT A GOOD IDEA TO USE THE APPROACH USED IN THIS SUBASSIGNMENT FOR CLOSED-LOOP CONTROL OF A ROBOTIC SYSTEM? WHY (NOT)?

Contrary to the previous answer to the question 2.1.5, the approach used in this subassignment is suitable for closed-loop control of a robotic system. This is because Xenomai provided enough real time capabilities that Section 2.1 could not deliver.

For closed-loop control of a robotic system, real-time performance is critical, and Xenomai can provide a high level of real-time performance that is difficult to achieve with standard Linux (as seen by the results and comparisons between the two approaches). By using Xenomai, one can create real-time tasks that execute with high priority and low latency, which is important for controlling the movements of the robot in real-time.

3.1.1 – JIWY SYSTEM ARCHITECTURE : BLOCK DIAGRAM

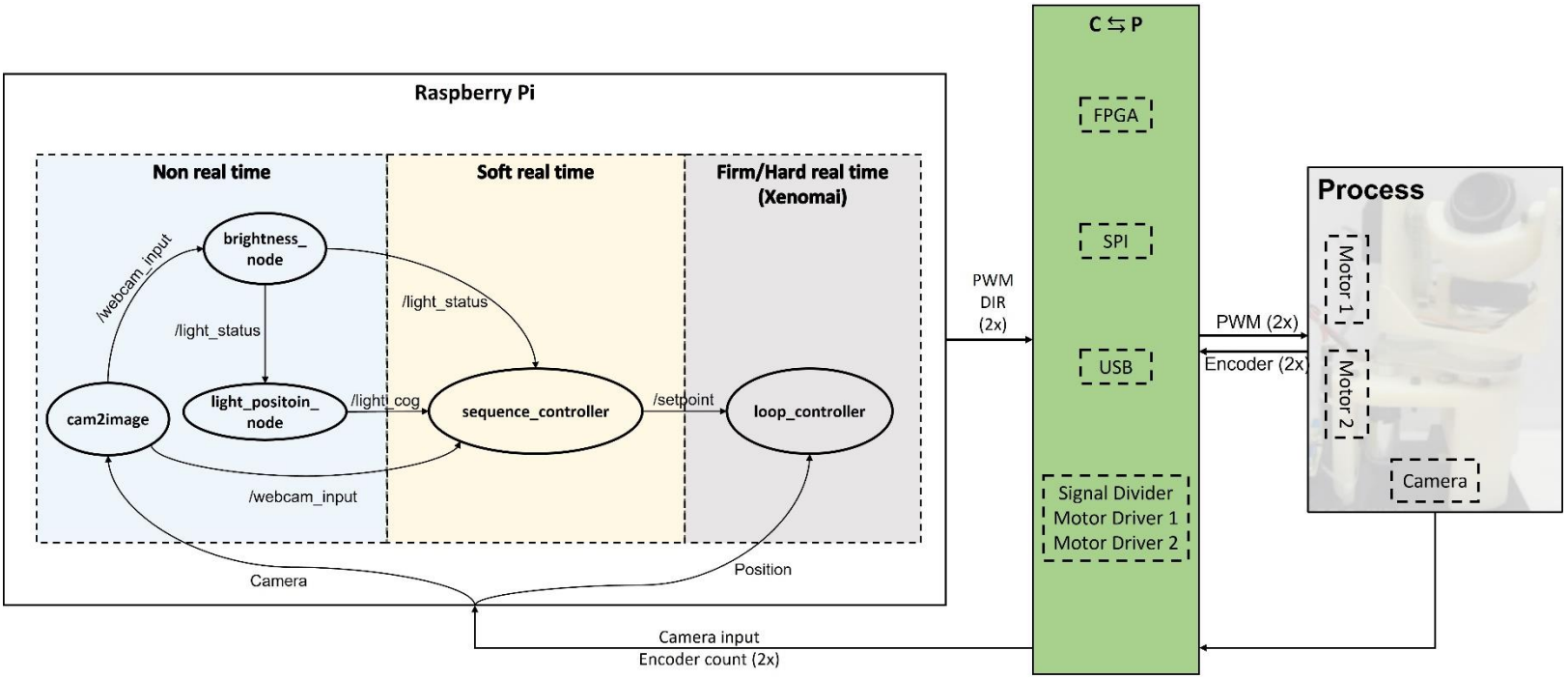

Figure 13: Block Diagram of the Jiwy System Architecture