# ASDR Assignment 2

Tjeerd Bakker(s2097966) Jonathan Schaaij(s2374862)

March 2023

## 1 Timing of periodic non-realtime thread

### 1.1 Timing Aspects for Robotics

The two main timing aspects regarding this assignment are the accuracy of the cycle time and the consistency of the cycle time. Other timing aspects such as the response time of sensors and latency of the actuator are also important but not applicable to his assignment, because we only measure the timing on the raspberry pi, which does not have any sensors or actuators attached to it.

The accuracy of the cycle time regards deviation between the actual and the desired cycle time. The consistency of the cycle time regards the variance of the time between each cycle.

There are two methods of plotting these aspects. The first method is by plotting the cycle number vs. the measured clock time. If the resulting plot is a perfectly linear function the cycles are consistent, however, a changing slope indicates a changing cycle time. In this plot the accuracy can be determined from the slope of the graph. The advantage of this plot is that developement over time can be seen, e.g. that the accuracy increased/decreases over time, and that the overall accuracy can be determined by looking at the last point, which should be : $t_{\text{final}} = N_{\text{cycles}} * t_{\text{cycle}}$. The disadvantage of this plot is that singular event might be hard to see. The second method is by plotting a histogram of the cycle duration, which directly shows the average cycle time as well as its variance. While a histogram does not show the historical data, it does make it easy to see the variance and it shows the number of timing failures.

### 1.2 Choosing between clock_nanosleep and create_timer/sigwait

There are two possible methods to achieve a loop with a desired consistent cycle time. The first method is by using the `clock_nanosleep` function. This way the process is paused for a consistent time after each computation. The advantage of this is that it is very easy to implement. However the disadvantage is that is is not accurate and the consistency depends on the consistency of the load. To make a loop which completes every millisecond, the process needs to sleep less than that:

$$1\text{ms} = t_{\text{computation}} + t_{\text{sleep}}$$

However, $t_{computation}$ is not known. For our implementation we simply waited for one millisecond after each computation, but to more accurately implement the `clock_nanosleep` function, the duration of the computation would need to be measured every time and the sleep time should be changed accordingly. However, this would add computational overhead, which would influence the timing as well, making this not an ideal solution.

The second method is by creating a timer and awaiting a signal each time the loop runs. This method is more accurate, since it uses an accurate timer which sends out a signal every millisecond. Even when the computation time varies the process only waits until the next signal. The disadvantage of this method is that in case the computation time exceeds one millisecond the process wait until the next signal is being send out.

### 1.3 Difference between CLOCK_MONOTONIC and CLOCK_REALTIME

`CLOCK_MONOTONIC` provides a monotonically increasing clock that is not affected by system time adjustments. It is primarily used for measuring elapsed time between two points in time and for implementing timeouts. This clock

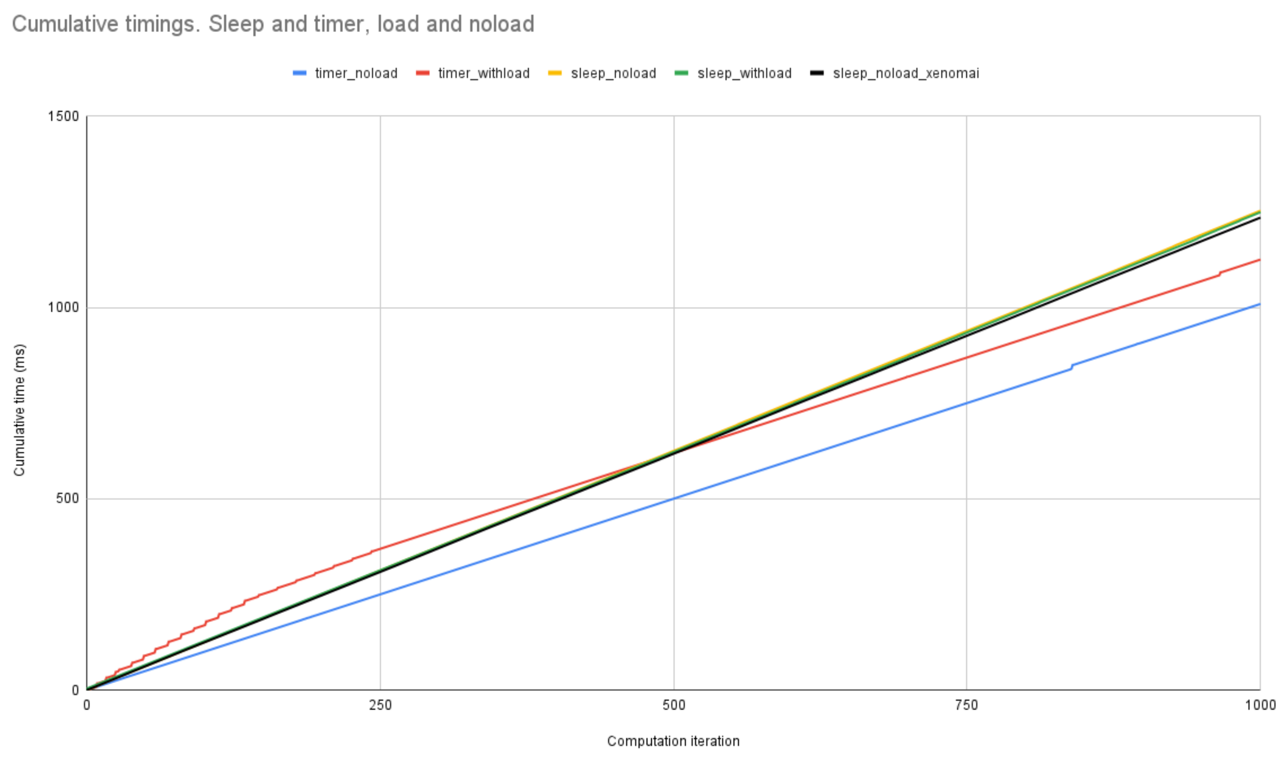

Figure 1: Timing measurement results

is suitable for timing a firm or hard real-time loop because it provides a consistent and stable measure of time that is not affected by changes in the system clock.

CLOCK_REALTIME provides a clock that is synchronized with the system clock and can be adjusted by the system time changes. This clock provides the current time and is suitable for non-real-time applications that require timestamps.

CLOCK_MONOTONIC is preferred for timing a firm or hard real-time loop because it provides a stable and consistent measure of time, CLOCK_REALTIME is suitable for non-real-time applications that require timestamps, e.g. such as the time a file was created.

### 1.4 Handling Initialized Timers in create_timer/sigwait

If the timer was created using `timer_create()` and was not set to be a "process-shared" timer, then the timer will be destroyed when the process that created it terminates, regardless of whether the program exits normally or due to a crash. If it was set to be a "process-shared" timer, it is important to explicitly call `timer_delete()` to destroy the timer before the program exits.

If the timer was set to be a "process-shared" timer using the TIMER_EXCLUDE_PID flag, then the timer will survive even if the process that created it terminates or crashes. Other processes can still access and use the timer, as long as they have the necessary permissions to do so.

If the timer was created using `setitimer()` or `signal()`, then it will be automatically cleaned up by the operating system when the program terminates or crashes.

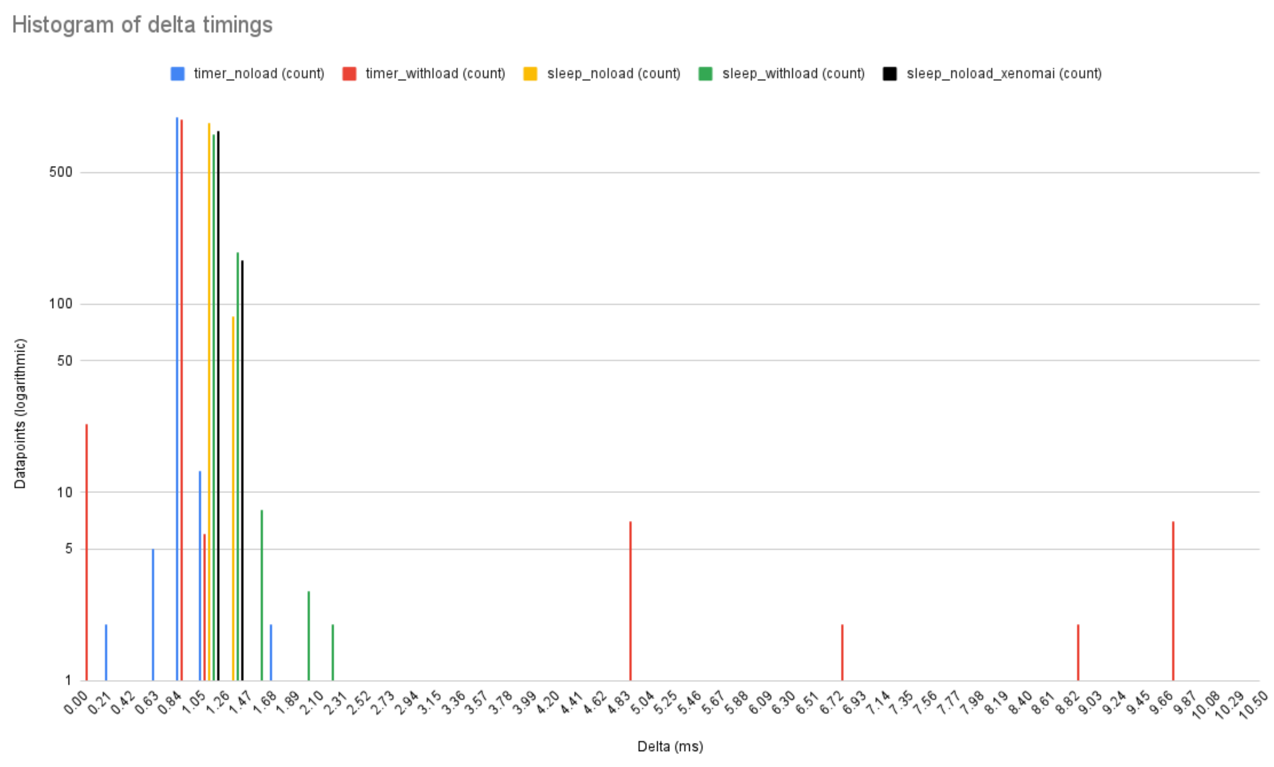

Figure 2: Histogram of durations

## 1.5 Suitability of Subassignment Approach for Closed-Loop Control of Robotic Systems

It is not a good idea to use the approach described in this subassignment for closed-loop control of a robotic system for the following reasons:

Timing Jitter: The POSIX threads and the methods mentioned (`clock_nanosleep` and `create_timer/sigwait`) can have varying levels of timing jitter. In a closed-loop control system, precise timing is crucial for accurate and stable control. Any inconsistency in timing can lead to poor performance or instability in the control system.

Unpredictable computational load: In the given assignment, the impact of extra computational load on timing performance is investigated. However, in a real-time robotic control system, unpredictable computational load from other processes can lead to variable execution times, which may cause control issues.

Non-real-time Operating System: The Raspberry Pi runs a non-real-time operating system (in our case Linux), which means that the operating system does not guarantee strict timing constraints. In a closed-loop control system, real-time constraints are crucial to ensure proper operation.

A better approach for closed-loop control of a robotic system would be to use a dedicated real-time operating system (RTOS) or real-time extensions for Linux (such as Xenomai), which can provide deterministic timing and better performance in control applications. Additionally, dedicated hardware, such as microcontrollers or FPGAs, can be used for time-critical tasks to achieve more accurate and reliable control.

## 2 Timing of periodic realtime thread (Xenomai)

### 2.1 POSIX Commands and Real-time Programming

POSIX commands used in a real-time program:

- `pthread_create`: Creates a new thread to run a function. Useful for handling multiple tasks at once in real-time programs.
- `pthread_join`: Waits for a thread to finish. Helps synchronize tasks in real-time programs.
- `clock_gettime`: Gets the current time from a specified clock. Used for measuring time in real-time programs. The chosen clock type, `CLOCK_MONOTONIC`, is suitable for real-time performance because it provides a consistent, non-adjustable time source.
- `nanosleep`: Pauses a thread for a specified time. Can introduce delays or pace tasks in real-time programs.
- `timer_create`: Creates a timer linked to a signal event. Used for timing tasks in real-time programs.
- `timer_settime`: Sets when a timer expires. Controls timers in real-time programs. The chosen timer interval (1ms) is suitable for real-time performance because it provides a balance between responsiveness and resource usage.
- `sigemptyset`: Initializes an empty set of signals. Helps manage signals in real-time programs.
- `sigaddset`: Adds a signal to a signal set. Specifies which signals a thread should handle in real-time programs.
- `sigaction`: Sets how a program responds to signals. Defines signal handling in real-time programs.

### 2.2 Comparing Timing Performance between Programs

Figure 1 shows the timing results for the both sleep and timer implementations with and without a load.

The timer without the load is the most accurate, since the duration of 1000 cycles is almost exactly one second. Only at $\approx 830$ cycles some signals are missed resulting in a discontinuity. When the load is attached the results are similar, however, during the first 200 cycles the signal is missed multiple times. After that the line continues parallel to the measurement without a load. The histogram shows that the timer is not very consistent. Especially with a load attached the durations vary from almost zero to 9 milliseconds. The close to zero millisecond durations are probably a result of race conditions in our code. We used the SIGUSR1 timer, which does not work with `sigwait` since the default action kills the process. Instead we made a signal handler which toggles a Boolean, activating the computation which could trigger a race condition causing the the Boolean to be triggered during the computation. This race condition could be resolves by using a Mutex, which only allows the boolean to be used by either the control loop or the signal handler at the same time.

The measurement using the `nanosleep()` function shows a linear result. The slope if larger for the sleep function since the average cycle duration is longer than one millisecond, because the computation time in not considered in the code.

In our sleep measurement the load slightly reduced the speed. The histogram in Figure 2 shows that a few cycles took slightly longer. One advantage of the sleep implementation is that it never starts the computation too early.

When compiling and running the timer implementation the process exist immediately. Unfortunately, we did not have enough time to investigate this issue and resolve it. Instead we compiled the sleep implementation for Xenomai. The figures 1 and 2 show that the xenomai implementation is very similar to the no-load implementation, with consistent timing. However, it still takes more than one milliseconds since the computation time was not taken into account.

### 2.3 Suitability of Xenomai Approach for Closed-Loop Control of Robotic Systems

Theoretically the approach used in this subassignment is suitable for closed-loop control of a robotic system because:

- The Xenomai real-time environment provides deterministic timing, ensuring that the control loop can maintain accurate and stable operation.
- The real-time scheduling policies and priorities help ensure that the control loop runs as required, even when other non-critical tasks are competing for system resources.
- The reduced timing jitter compared to a non-real-time environment ensures that the control system can respond more effectively to changes in the robot's state or environment.
- By utilizing real-time POSIX functions, the program can maintain data consistency and prevent race conditions, which are crucial for the reliable operation of a robotic system.

While our implementation shows a small improvement when a load is attached, it does not have a significant influence on the sleep implementation. We expect the bigger difference would be seen on xenomai using the timer implementation, because if could prevent missed/skipped timer signals. Unfortunately, it did not work for us as explained in the previous section.

## 3

Not for CBL students

## 4 Block diagram

Figure 3 show the high level block diagram for the code in Assignment 3. On the raspberry pi a ROS2 node converts the USB image to an image topic, which will be used by an other node to compute the setpoint based on the current position of the JIWY and the brightest point in the image. This node only needs to operate at 30Hz since that is the refresh rate of the camera. Otherwise the image will not have changed when the setpoints will have. This would result in the setpoint being set to overshoot the target, which is why it is important to only adjust the setpoint when a new image has arrived.

To control the motor and analyse the encoder data a realtime node is running using xenomai. This node implement a closed loop controller to make sure the motor power will definetely be turned off once the robot has arrived at the setpoint. If this node experiences timing delays, the robot could overshoot its maximum position and crash into itself.

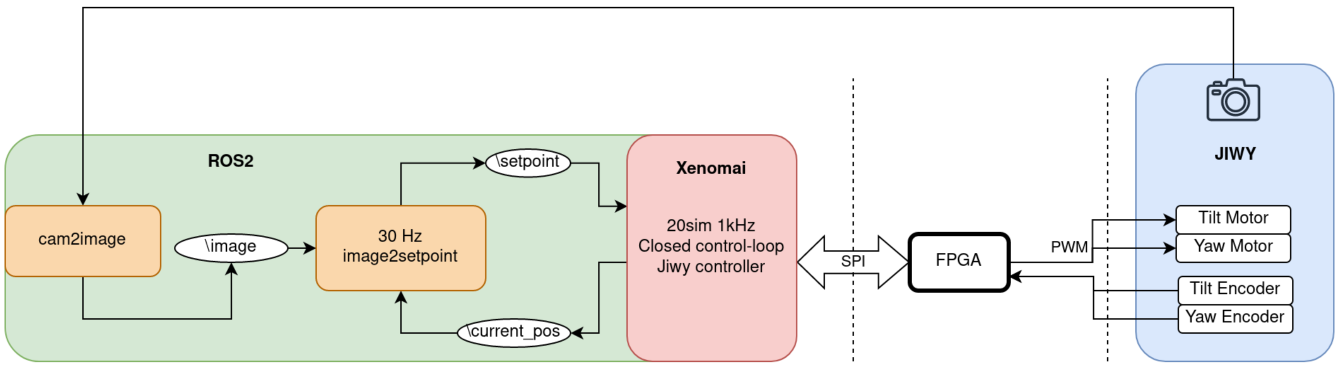

Figure 3: High level block diagram of the code for Assignment 3