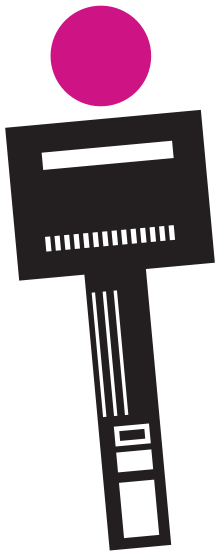
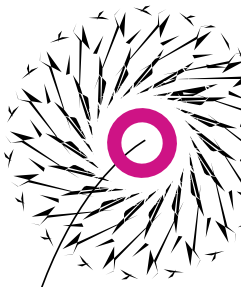




RAM ROBOTICS AND MECHATRONICS

A bare-metal microcontroller as a target for 20-sim-generated C code

B. (Berend) Visser

BSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
ing. M.H. Schwirtz
ir. E. Molenkamp

August, 2020

043RaM2020
Robotics and Mechatronics
EEMathCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Summary

By the use of 20-sim, real-time controllers for mechatronic systems can be developed. Using the 20-sim 4C software this controller can be run on actual hardware. Several development targets are by default supported by this software, it is also possible to make a custom target.

The goal of this project is to explore the capabilities of a microcontroller as a target for 20-sim-generated C code. A microcontroller is an interesting target as it can run without an operating system allowing for fast interrupt response times while having direct access to a wide range of peripherals. Also, it can be low energy, has a small footprint on a PCB and is relatively cheap. This comes at the cost of processing power.

We found that the Arduino Due board, featuring an ARM-M3 microcontroller, can run a PID controller with a 10th-order low-pass filter at a loop frequency of 100[Hz]. The combined maximum latency per sample is 0.6[ms]. We measured latencies for the step calculation, reading input, setting output and sending logging.

Extending the IO drivers would allow the system to be used on a wider range of plants. The latency can be reduced further by converting the control algorithm to work with fixed-point numbers. The data interface between the base station and the microcontroller should be improved as this is currently limiting the maximum sampling frequency.

Contents

1 Introduction

1.1 Context

The progress in digital computing over the last 50 years has allowed for increasingly advanced cyber-physical systems. Large and complex control systems require a structured way of design to keep them extendable and updatable. Simulation tools like 20-sim (Controllab Products, 2020a) offer a separation between control algorithm design and embedded control system implementation. In this research, we will focus on embedded control system implementation.

Embedded control system implementation means that the control algorithm, designed in simulation software, is converted into computer code. 20-sim offers C code generation with its real-time toolbox. This C code can than be configured and compiled for a specific control computer like a Raspberry pi.

Finding the ideal hardware target for a specific application can be challenging since the number of supported hardware targets is limited. The package 20-sim 4C (Controllab Products, 2020b) enables running C code on hardware using open-source real-time Linux.

Even real-time operating systems can suffer from excessive jitter due to task switching(Zagan, 2015). This has a significant effect on the predictability of hard real-time tasks. Also, the power consumption of a real-time operating system can be very high compared to the application code (Dick et al., 2003).

A microcontroller can be run without an operating system. It is an embedded system with its processor, memory and peripherals on a single integrated circuit. Making a microcontroller into an embedded control computer can be cost-effective, low power and have a small footprint as very few external components are needed.

1.2 Previous research

An organised design style is required to keep product development efficient, maintainable and safe.

In the paper by Broenink and Hilderink (2001), such an approach for embedded control systems is described. First, the physical system should be made into a model: this can be done via a bond-graph representation. Using this model, a control law can be designed. Then this control law is converted into computer code. All these steps are verified by simulation. Finally, in the realization step, the control algorithm is implemented on the target computer which is connected to the physical system. The general structure for the realization is divided into three parts: "The embedded computer and its software", "I/O interfacing" and "The appliance itself", Broenink and Hilderink (2001).

To select an appropriate embedded computer, it's real-time scheduling capabilities are vital. For some processes, missing deadlines can lead to catastrophic failure of the system, these are "hard" deadlines (Xu and Parnas, 1993). Deadlines are called "soft", when low and constant response time is desirable but not required.

The real-time performance of a *raspberry pi* is evaluated by (Johansson, 2018). A raspberry pi is a low-cost single-board computer. This research works with Linux loaded with a real-time extension called Xenomai. Linux is a General-purpose Operating system and has a large number of features but is unpredictable in timing. Xenomai opts to bound and reduce the latency of tasks that require real-time scheduling. This is achieved through using a co-kernel, which takes control of the hardware interrupt management (Gerum, 2004).

In the research of Dokter (2016), the raspberry pi was made into a target for 20-sim 4C. Here the target also runs Linux, but due to a compatibility issue, it could not be patched with Xenomai. The raspberry pi was found to be a valuable asset for quick prototyping with 20-sim. Adding a real-time extension was recommend, then the system would also be able to satisfy hard real-time requirements.

1.3 Problem definition

To satisfy the timing requirements, the target running the control law should possess enough processing power. The C code generated by 20-sim is based on floating-point numbers. Microcontrollers are cost-sensitive, therefore they do not always have a dedicated floating-point unit. Floating-point calculations then have to emulated, requiring more computation time.

Monitoring the target over the network is very convenient, but requires a TCP/IP stack. Therefore, decisions have to be made about which functionality can be implemented and which not.

1.4 Research Goal and requirements

In this research, a microcontroller is evaluated as a target for 20-sim generated C code. The microcontroller will run bare-metal, this means there is no operating system. The Arduino Due board is used as it is low cost, open-source and features a fast 32-bit microcontroller with diverse input and output peripherals.

The goal is to create a software framework that can run control algorithms exported from 20-sim. Features like automatic compilation and upload, parameter adjustment and data logging are added to make the software convenient to use. The system is tested using a test plant to find its real-world performance. An overview of the system functions is given in Figure 1.1.

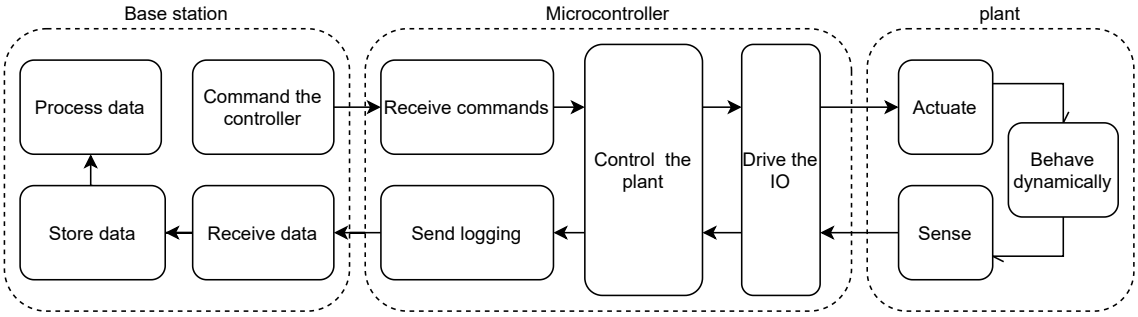


Figure 1.1: Functional overview control system

For the requirements we will follow the scheme as described by Waters (2009):

- Must have: the minimal functions required for the project to be useful
- Should have: functions that add a lot of value but are not critical
- Could have: functions that are nice to have, but not implemented when time becomes scarce.
- Won't have: Possible desirable functions, but these are outside of this project's scope

1. **Base station requirements**

   1.1. *The base station must be able to start the controller*
   The user must be able to start the controller remotely from the base station. This is required for a convenient prototyping workflow.

   1.2. *The base station must receive data from the microcontroller*

1.3. *The base station must store the received data*

1.4. *The base station must process the data stored*
The data stored from the experiments must be processed in order to evaluate the performance of the microcontroller.

1.5. *The base station should be able to change controller parameters during run time*
Being able to change parameters, like plant run time, set points and controller parameters speeds up development on the target significantly as recompiling and uploading is not needed.

1.6. *The base station could have an automated uploading scheme*
The user is protected from a lot of target specific technicalities and can instead focus on the control system itself.

2. **Micro controller requirements**

2.1. *The micro controller must receive commands*

2.2. *The micro controller must control the plant real time*

2.3. *The micro controller must have basic IO drivers*

2.4. *The micro controller must send logging*

2.5. *The micro controller could have IO drivers implemented for all the sensors available*
Having access to various IO drivers allows for more complex designs of the control system.

3. **Plant requirements**

3.1. *The plant must react on steering values by showing dynamic behaviour*

3.2. *The plant must provide its status via its sensor interface*

2 Background

2.1 20-sim

20-sim is a software tool for simulating the dynamic behaviour of cyber-physical systems (Broenink, 1999a). The tool facilitates model-driven design (Bezemer, 2013), allowing complex control algorithms to be created for multi-domain physical systems.

Physical systems can be modeled in 20-sim through bond-graphs (Broenink, 1999b). Bond-graphs are a graphical representation of domain-independent system components like capacitors, springs, inductors and inertial masses. The components are connected through ports that specify the transfer of energy between the components.

A control algorithm can be exported as C code (Kleijn, 2009) which can then be implemented on a hardware target which is connected to the actual physical system.

2.2 Real number representation

Representing real numbers in digital systems means trade-offs have to be made between precision, speed, power usage and development cost (Gaffar et al., 2004). Floating-point numbers representations are efficient when a large dynamic range is required. The IEEE 754 standard describes how floating-point numbers are represented. It can be implemented either in software, hardware or a combination of the two. Emulating floating-point arithmetic in software takes significantly more processing time (Iordache and Tang, 2003) compared to a dedicated floating-point unit.

Integer arithmetic is supported by most microcontrollers, minimizing processing time. Fixed point numbers are integer numbers scaled by a fixed factor (Yates, 2009). This makes efficient implementation in microcontrollers possible without requiring specialized hardware. Fixed point numbers suffer from a reduced dynamic range compared to a floating-point number with the same data size.

2.3 Arduino Due

The Arduino Due board features a 32-bit ARM Cortex M3 microcontroller, clocked at a frequency of 84$[MHz]$. It has no operating system, allowing for direct access to input and output peripherals. Among the supported peripherals are:

- 2 channel - 12 bit - 1 MSPS - Digital to analog converter
- 16 channel- 12 bit - 1 MSPS - Analog to digital converter
- 8 channel - 16 bit - Pulse width modulation
- 3 Quadrature Decoders
- USB 2.0
- 1 UART & 4 USART
- 103 GPIO lines

The microprocessor on the board has no dedicated Floating-point unit, but it supports the "UMULL" instruction. This multiplication instruction takes two 32-bit integers and produces a 64-bit output (Yiu, 2010), which allows for efficient implementation of fixed-point numbers.

2.4 Linix test plant

The Linix plant was designed as a mechatronic demonstrator for 20-sim software (Bax, 2006).

The Linix consists of a DC-motor and a flywheel. The flywheel and the DC-motor have differently sized pulleys, which are connected through an elastic band. It can be modeled as a 4th-order system. The 20-sim model is shown in Figure 2.1 and a 3D model is shown in Figure 2.2.

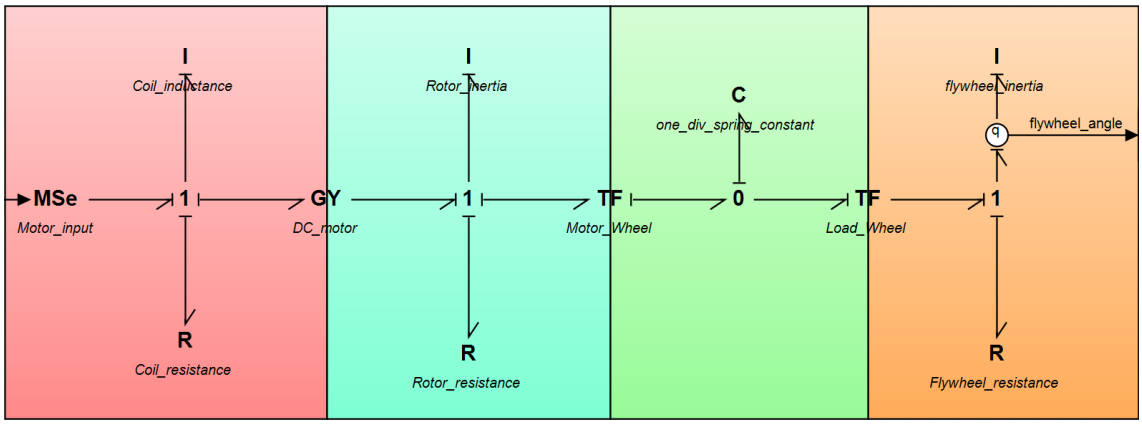


Figure 2.1: 20-sim model of the Linix test plant, Electrical part of the DC motor *(red)*, mechanical part DC-motor *(blue)*, belt and pulleys *(green)*, flywheel *(orange)*

A motor driver and angle sensor are connected to the Linix setup, these actuate and measure the system. The motor driver receives two digital signals. One sets the motor direction and the other is a square wave with a varying duty cycle which sets the motor speed. The position of the flywheel is measured with an angle sensor. The position is digitally encoded in 2 square waves through quadrature encoding.

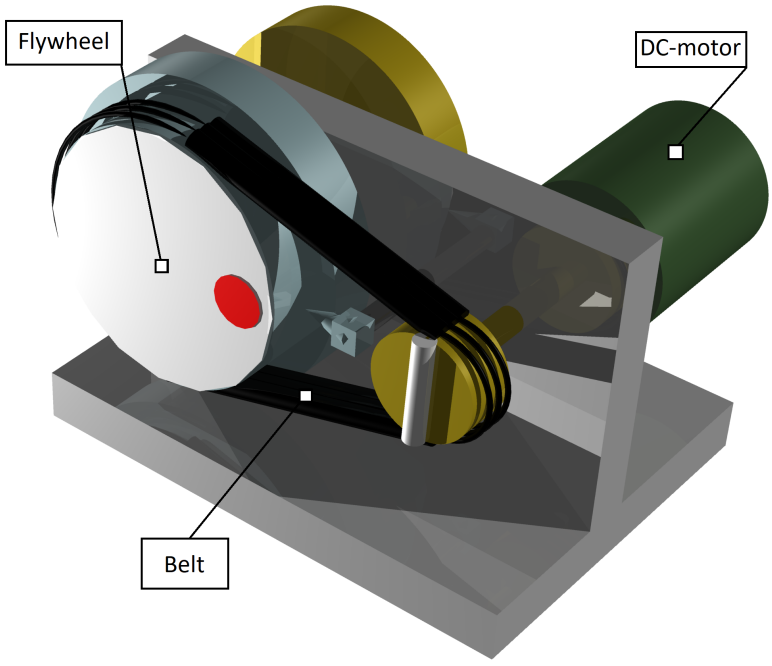


Figure 2.2: 3D-model of the Linix test plant

3 Design

A testbench is needed to evaluate the performance of the microcontroller. The testbench consists of a base station for commanding and monitoring, a test plant to find real-world performance, a measurement setup that can measure the actual system performance, and the Arduino due board itself (see Figure 3.1).

The C code generated by 20-sim only provides the control algorithm. A software framework is required that can handle other system tasks like variable logging and input and output control.

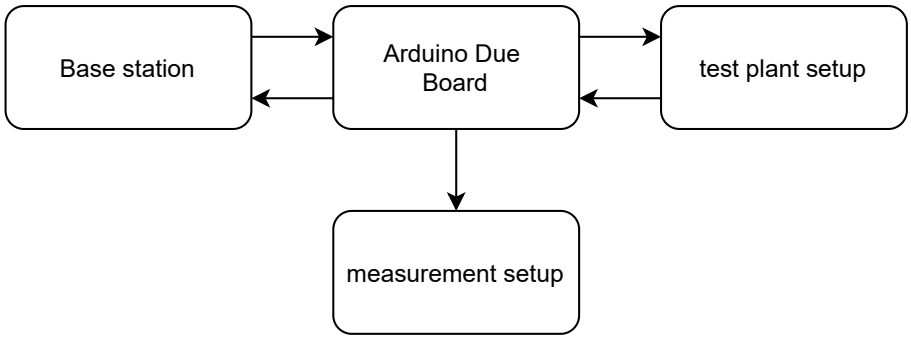


Figure 3.1: Overview of different subsystems

3.1 Software framework

The software framework has to perform a number of functions which are partly executed on the base station and partly on the micocontroller(see Figure 3.2). The basestation software is implemented using Matlab and the microcontroller software with C++.

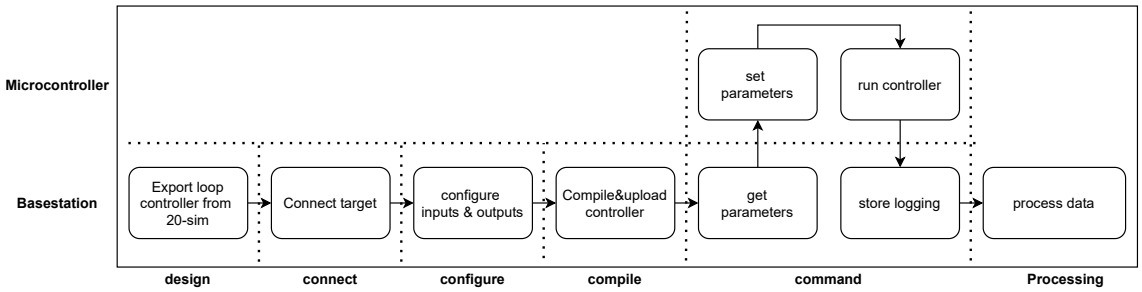


Figure 3.2: Overview of functions the software framework has to perform

3.1.1 Base station

First, a control algorithm is exported from 20-sim. The exported files consist of C code and a token file. The C code contains the controller algorithm and the token file contains information about the configuration. This includes the number of parameters, variables and states and their names.

After connecting the microcontroller to the base station the input and output ports should be configured. Using the token file, the base station can determine how many ports there are. Since only one test setup is used the ports are set by default and cannot be changed. The C code is then added to the target code and the base station compiles and uploads the target code to the microcontroller.

After uploading has finished the microcontroller reboots and goes into a state called "start-up". In the start-up state, a minimal number of peripherals is initialized to minimize power usage of the microcontroller. The base station sends the "INIT" command to initialize all in and output peripherals and the control algorithm. The microcontroller is now in the "initialized" state.

By sending the "GETPAR" command, the microcontroller transmits all current parameter values to the base station. Using parameter names provided by the token file, the base station maps the received values to the correct parameter names. This way the user knows which value corresponds to which parameter. The parameters can be changed by sending the "SETPAR" command.

With the "START" command, the microcontroller is put in the "running" state which starts the control algorithm on the microcontroller. During the running state, the microcontroller transmits variable and state data to the base station every sampling period. This data are stored until the "STOP" command is sent. The stop command puts the microcontroller in the "finished" state, this stops the control algorithm.

The stored logging is now copied into a data structure with field names corresponding to variables names. This data structure can then be used for further processing like transient plotting or frequency analysis.

3.1.2 Microcontroller

The paper of Broenink and Hilderink (2001) describes a general structured approach for embedded control software design. In Figure 3.3, an overview of the design is given. The microcontroller software is implemented using the C++ programming language, as it has extended support for classes compared to traditional C. Using component-based software design (Heineman and Councill, 2001), a clear separation of system functions is realized. This can aid development and make it easier to extend the system.

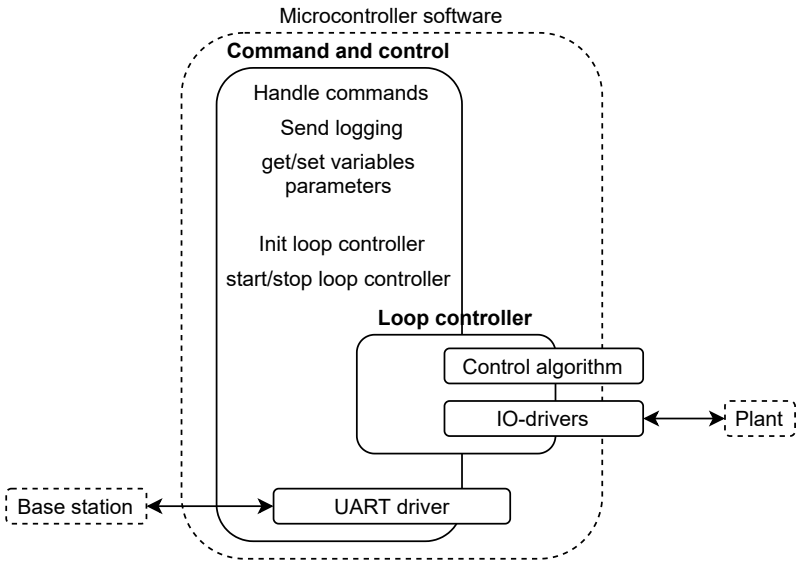


Figure 3.3: Layered microcontroller software structure

The loop controller is hard real-time as missing a sample deadline can lead to system failure. It contains the control algorithm. Before compilation, the C code files generated by 20-sim are copied into the source files of the microcontroller software. The C code files are according to a certain standard, therefore the microcontroller software is universal for different control algorithms. The control algorithm might miss certain elements for example, it could have zero states. This is handled through conditional compilation, this allows the compiler to ignore

parts of the code based on the definition of certain variables. The input and output drivers are also a component of the loop controller. The control algorithm works directly on actuators and sensors connected to the physical plant. Hence the IO-drivers also work in the hard real-time domain.

The control algorithm is initialized, started and stopped through the command and control code. Missing a deadline here decreases the quality of service, but it does not lead to system failure. The loop controller component is contained inside the command and control code. The commands are transmitted over a UART connection. The commands can start and stop the loop controller, and set the parameters of the control algorithm. Also, the variable logging is managed through the command and control code.

3.2 Test method

In this research, the real-world performance of the microcontroller is verified by testing the system with a physical plant. By measuring processing latencies of the system, we can determine for which applications the microcontroller is a suitable target. Each test is performed for 10 minutes to get reliable results of the maximum, minimum, average and standard deviation of the measured latencies.

The complexity of the control algorithm is varied to find the performance over a wider spectrum. The complexity of control algorithms created in 20-sim can be categorized by the number of independent states, equations and variables. The values of these criteria for a specific algorithm can be read from the token file that is generated together with the C code.

The computational precision of the algorithm is also varied by using single and double-precision floating-point numbers. Single-precision floating-point numbers only require half the memory. Since the microcontroller has a relatively low amount of memory, it could be useful to switch to single precision if the increased error does not cause any problems.

We also study the effect of different setpoint curves on the latency of the microcontroller. Each test scenario is executed with a sine wave curve, a periodic cycloidal motion profile and a square wave. In each of the tests, the same amplitude is used. It is expected that the system experiences different latencies under different setpoint curves.

Another important system aspect is the sampling frequency. We will measure the consistency of the sampling period for different loop frequencies. Sampling time jitter is an important system characteristic and should be taken into account for real-time systems as it can have a significant effect on control algorithm behaviour (Marti et al., 2001).

3.2.1 Measurement and plant setup

An overview of the measurement and plant setup is given in Figure 3.4. The input and output drivers of the microcontroller send and receive digital signals over wire connections.

The “Linix” is used as the physical plant.

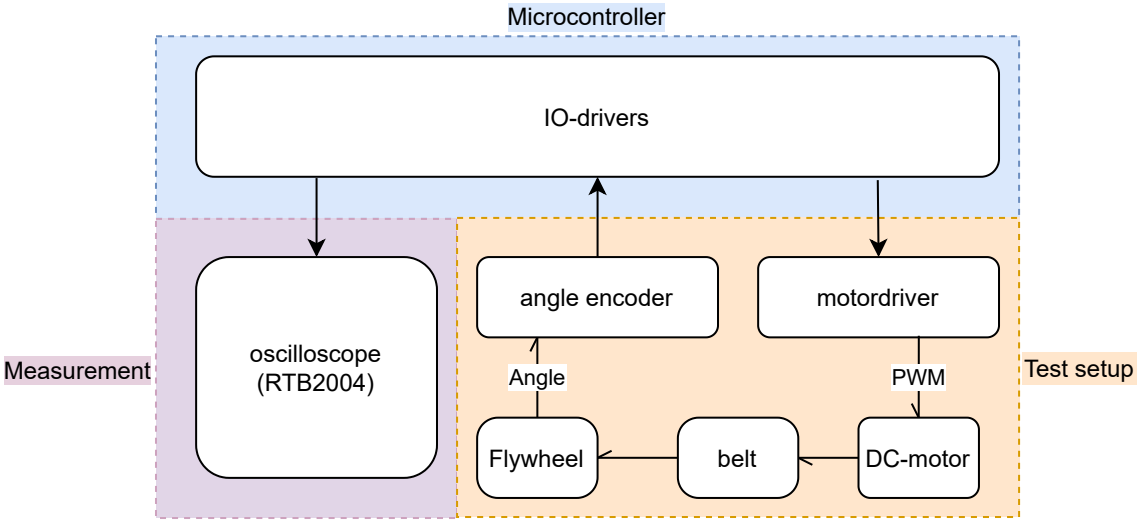


Figure 3.4: Overview of the measurement setup

The latency is measured by an oscilloscope with a logic analyzer. When a certain part of the control algorithm is started a digital output pin is set high and when it is finished the digital pin is set low again. This generates a square wave with a positive pulse width that corresponds to the execution time of the algorithm. Setting an output pin takes processing time and does influence the measurements. To minimize this effect the output pins are set by directly writing to the IO-pin peripheral registers.

3.2.2 Measurements

The microcontroller is tested using 3 different test algorithms with 3 different input signals and 2 precision levels each for 10 minutes. The test algorithms are implemented in the discrete domain and consist of a PID controller, a PID controller with a 5th-order low-pass filter and a PID controller with a 10th-order low-pass filter. The controller schematic is shown in Figure 3.5.

The input signals are a continuous pulse cycloid, a square wave and a sinewave, all with amplitude 2π. The amplitude was chosen to be 2π so that during real-world testing obvious setup problems can be spotted visually. The periods are respectively 8, 6.28 and 6 seconds. The periods are chosen 10 times larger than the biggest time constant of the system. This makes sure that the system has time to settle during the period of the setpoint curve and does not start to oscillate.

A sine wave was chosen as it contains only one frequency component. The square wave contains high-frequency components as well. The motion profile is a combination of the two. This allows us to study the effect of signal input on algorithm calculation time.

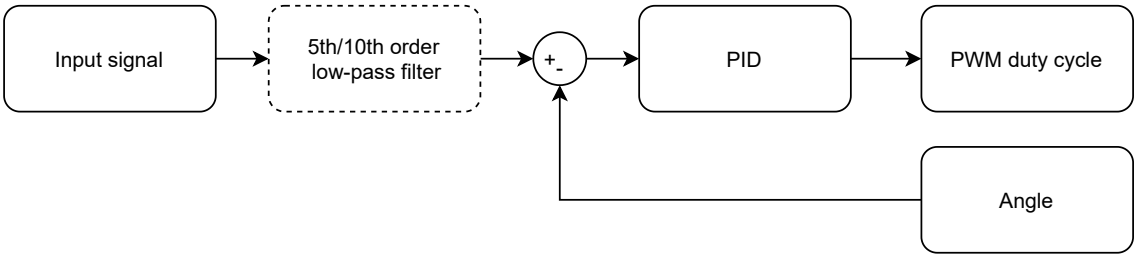


Figure 3.5: Control algorithm used for bench marking

The PID controller is a widely used control algorithm in industrial control systems (Knospe, 2006) hence it was chosen for the system performance test. To increase computational complexity a low-pass filter was added. The filter is placed after the input signal, to be able to verify the filter functionality. Since the input signal in simulation should be the same in the real world test, we can compare the signals check for any deviations.

The low-pass filter has a cut-off frequency of 0.5$[Hz]$, which is lower than the plant time constant but higher than the input signal frequency. This way the input signal is filtered partially and the loop can maintain its stability. The filter order was based on the amount of memory required to store all the system parameters. A 15th order filter was found to be the limit of the microcontroller. During measurements a reasonable margin of 5 orders lower was chosen to prevent any memory related bottlenecks.

The bandwidth of the data link is limited. For data monitoring purposes we want to log all variables and states. The sampling rate is selected such that the bandwidth of the data link is not exceeded. The complex algorithm has more variables and states and is tested at a lower sampling rate. The sampling rate should not affect measurements of individual samples.

It is important to examine sampling-time consistency. Therefore, the period between individual samples is also measured. We measure at a sampling rate of 100$[Hz]$ and a 1000$[Hz]$. The 100$[Hz]$ sampling rate is the maximum sampling frequency for which all the different control algorithms can still send all their logging data. The 1000$[Hz]$ is the maximum sampling frequency for which the algorithm with the least number of variables can still send all its data.

Finally, the effect of using single-precision and double-precision floating-point numbers is examined. As double-precision numbers require 64 bits of memory and single-precision only half, there should be a significant difference in calculation time between the different data types. The effect of the increased error will also be examined by comparing simulation data and logged data.

4 Results

An overview of the measurement setup is shown in Figure 4.1. The programming interface and data interface are connected to a base station running Matlab.

All the tests are executed for 600 seconds. The data is normalized per test set, to allow for easier comparison between subtests. The normalisation factor is based on the mean latency of the double-precision cycloid test. The sampling period is fixed per test. The double-precision measurements are abbreviated by 'DP' and the single-precision measurements by 'SP'.

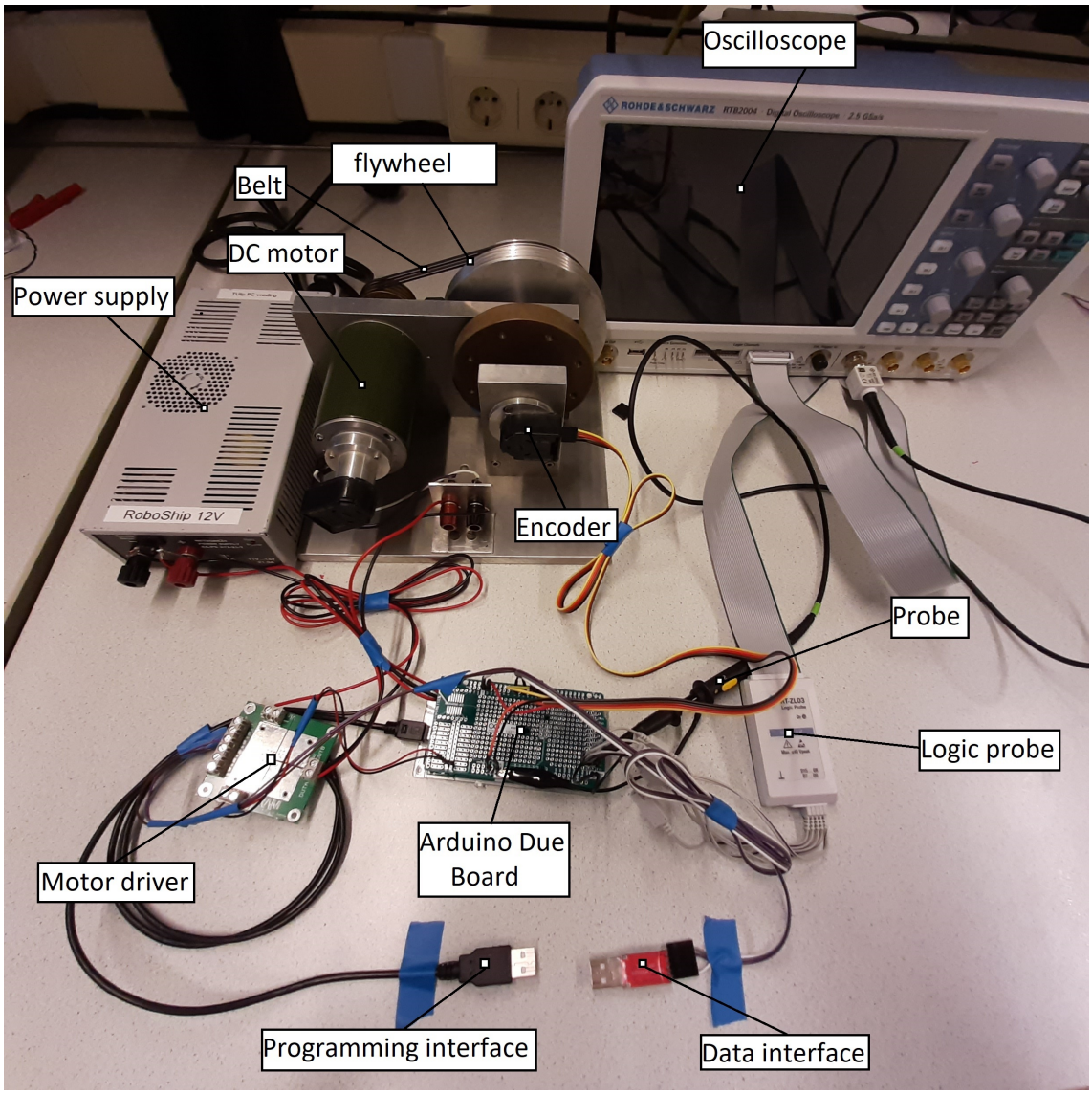


Figure 4.1: Complete setup

4.1 Sample period

The sample period of 1$[ms]$ has a maximum deviation of ±0.11%. The mean period has a positive offset of $2 \cdot 10^{-3}$%. The relative standard deviation is 0.04%.

The sample period of 10$[ms]$ has a maximum deviation of ±0.02% and the mean period has also a positive offset of $2 \cdot 10^{-3}$%. The relative standard deviation is ~ 10 times lower with 0.0044%.

The relative difference in the mean is equal for both measurements. The absolute standard deviation is also the same but the relative maximum-minimum and standard deviation are 10 times bigger for the shorter sampling period.

required period [ms]	min$[ms]$	max $[ms]$	mean$[ms]$	$\sigma[ms]$
1	0.9989	1.0011	1.00002	0.00042
10	9.999	10.002	10.0002	0.00044

Table 4.1: sample period measurement

4.2 Sensor input latency

There is no consistent difference in latency between double-precision tests and single-precision tests (see Figure 4.2). Over the results of different algorithms there is also no correlation between increased algorithm complexity and sensor input latency. The maximum latency is 20% higher and the minimum latency is 75% lower than the reference mean. The standard deviation is consistent over all tests at ~ 5%.

The absolute maximum sensor input latency over all tests is ~ 17.9$[\mu s]$.

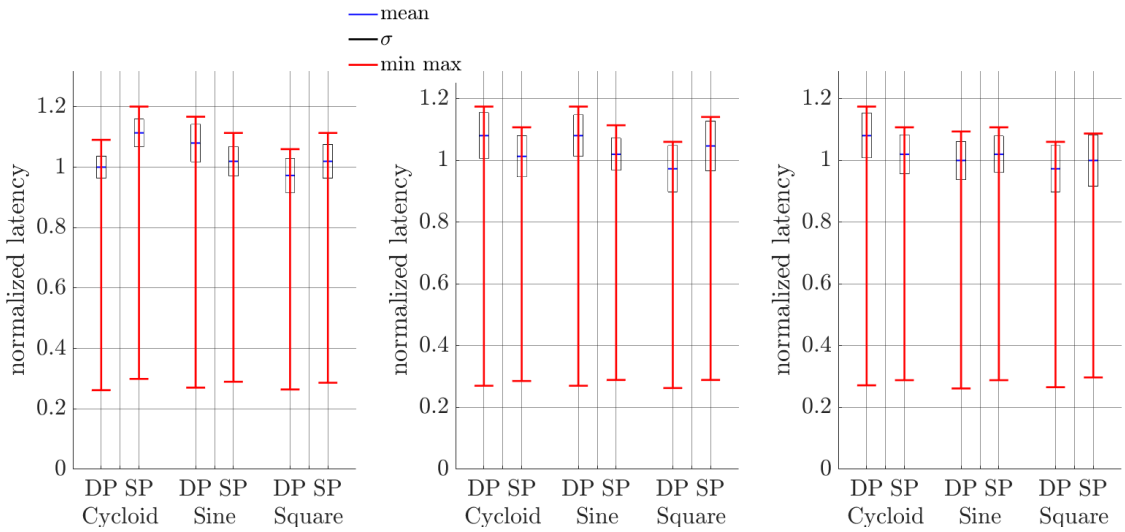
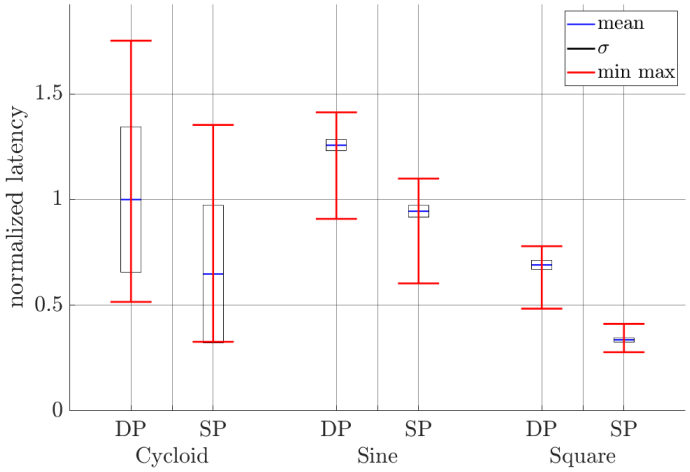


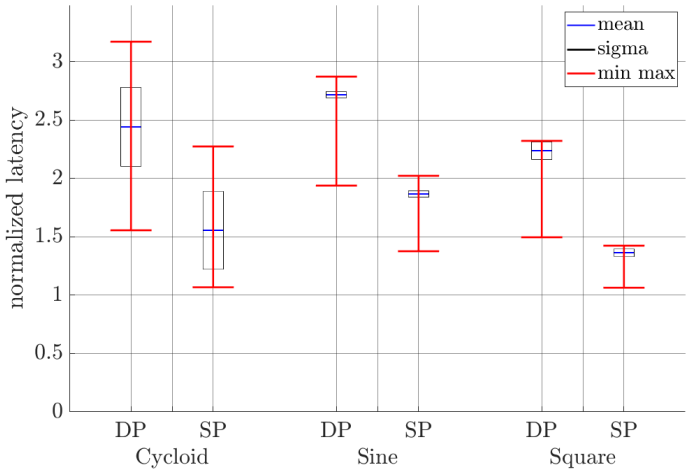
Figure 4.2: sensor input latency - normalization factor of 14.91$[\mu s]$
left:PID - 250[Hz] , **middle**:PID + 5th-order filter - 150[Hz], **right**:PID + 10th-order filter - 100[Hz]

4.3 Control loop calculation latency

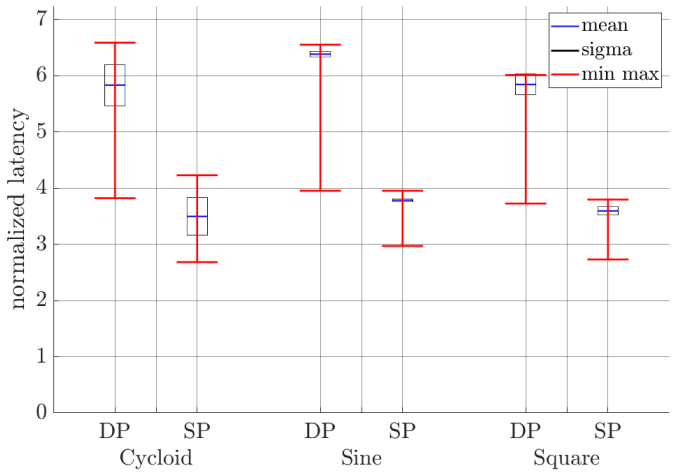
The maximum latency of the cycloid input signal for the PID algorithm is 75% higher than the reference mean (see Figure 4.3a). The minimum latency is 50% of the reference mean. The minimum and maximum latency are consistently lower for the single-precision calculation for all subtests. The maximum latency for the square wave input signal is 1.93 times lower for the single-precision test. The maximum latency of the sine wave input is lower than that of the cycloid but its mean is 25% higher. The square wave input signal has a significantly lower standard deviation and its mean is much lower than that of the cycloid input signal. The sine wave input is in between the two measurements.



(a) PID - 250[Hz[



(b) PID + 5th-order filter - 150 [Hz]



(c) PID + 10th-order filter - 100[Hz]

Figure 4.3: loop calculation latency - normalization factor of 83.51[μs]

For the algorithm with the added 5th-order low-pass filter, the relative standard deviation for all algorithms except the square wave becomes smaller. The maximum and minimum latencies become less spread for the different input signals. Also, the mean calculation time has overall increased by a factor of 2.5.

The mean calculation time for the PID + 10th-order low-pass filter algorithms has increased by a factor of 6 for all the tests. The difference in maximum latency has reduced even further.

The absolute maximum loop calculation latency over all the tests is ~ 550[μs].

4.4 Output actuation latency

The mean latency of setting the output is consistent over all three tests (see Figure 4.4). There is no trend between algorithm complexity and single or double-precision calculations.

The absolute maximum output setting latency over all tests is ~ 16.5[μs].

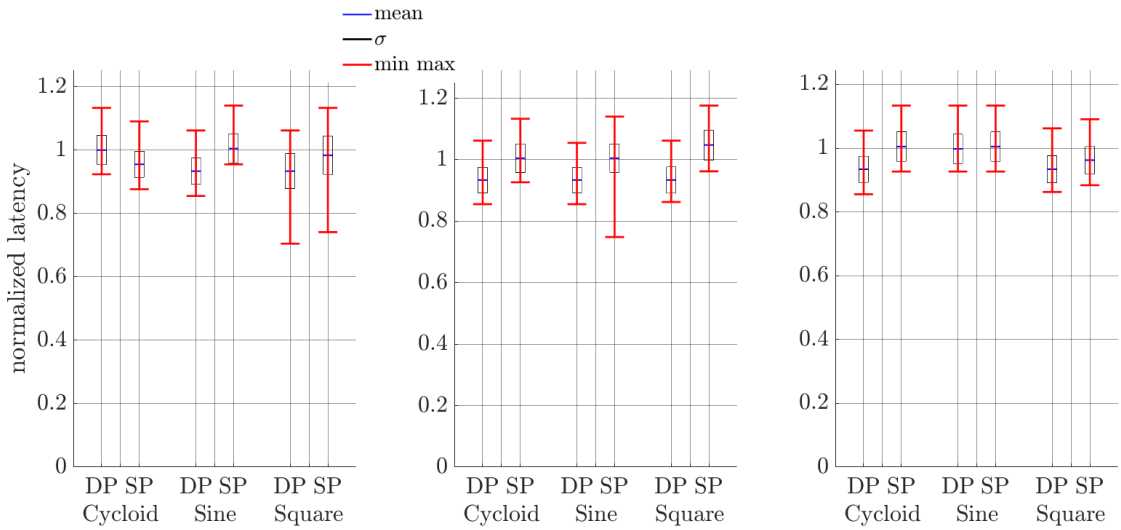


Figure 4.4: Set output actuators latency - normalization factor 14.03[μs]
left:PID - 250[Hz] , **middle**:PID+5th order filter - 150[Hz], **right**:PID + 10th order filter - 100[Hz]

4.5 Logging latency

The maximum logging latency is 10% lower for the single-precision compared to double-precision in the PID algorithm test (see Figure 4.5). The difference in logging latency between single and double-precision becomes bigger when the algorithm has an added low-pass filter.

The maximum absolute logging latency over all the tests is ~ 6.7[μsec].

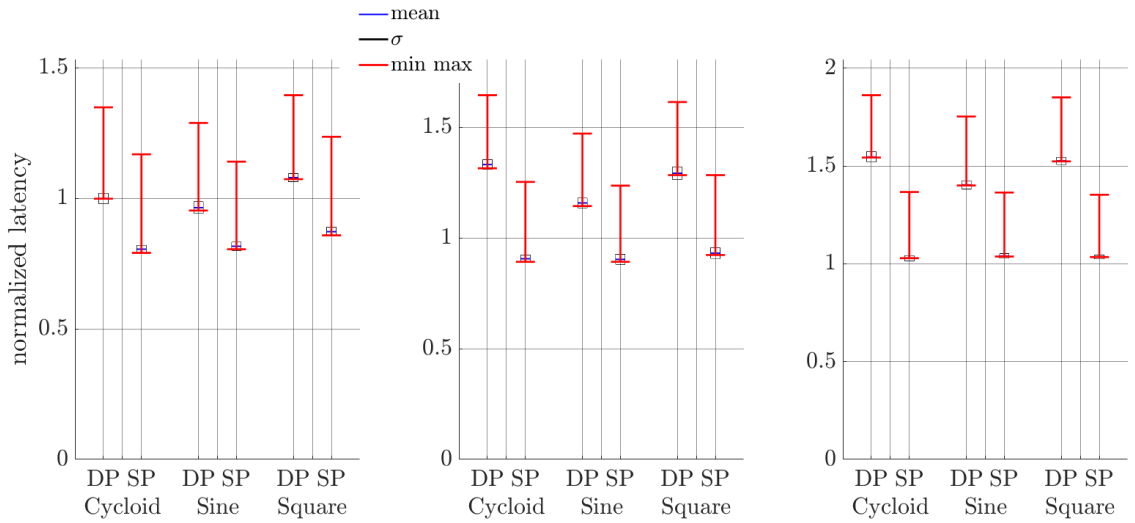


Figure 4.5: logging latency - normalization factor 3.577[μs]
left:PID - 250[Hz] , **middle**:PID+5th order filter - 150[Hz], **right**:PID + 10th order filter - 100[Hz]

4.6 Cumulative latency

In Figure 4.6 the timing diagram of the total maximum latency is shown.

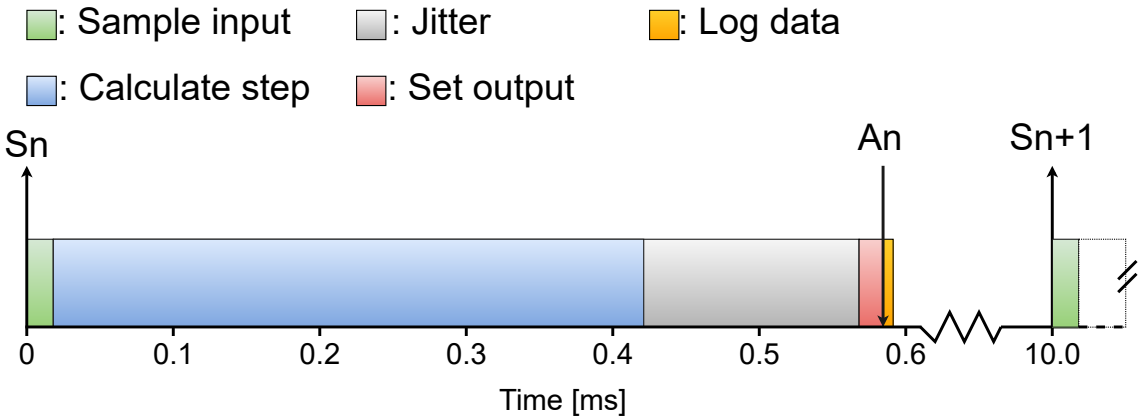


Figure 4.6: Timing diagram of the PID + 10th order low-pass filter algorithm with $F_{sample} = 100[hz]$, *Calculate step* represents the mean latency

The total maximum latency is ~ 0.59[ms], which is ~ 5.9% of T_{sample} at a frequency of 100[Hz].

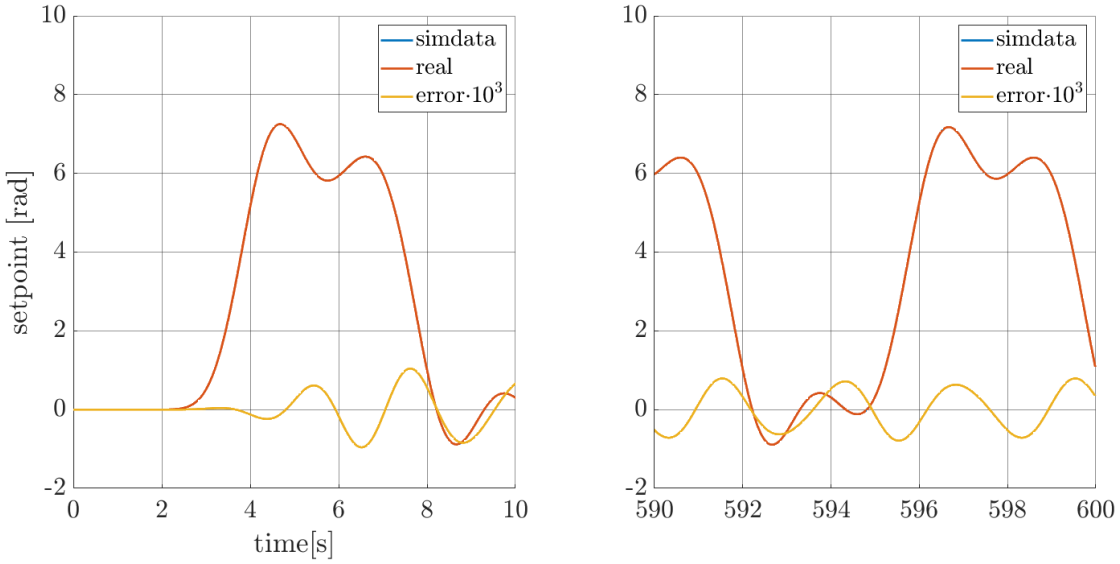
The combined maximum latency of getting the input and setting the output is ~ 0.034[ms]. This is ~ 0.34% of the total sampling period at a sampling frequency of 100[Hz].

The maximum latency of logging is ~ 0.007[ms] which is ~ 0.07% of the total sampling period at a sampling frequency of 100[Hz].

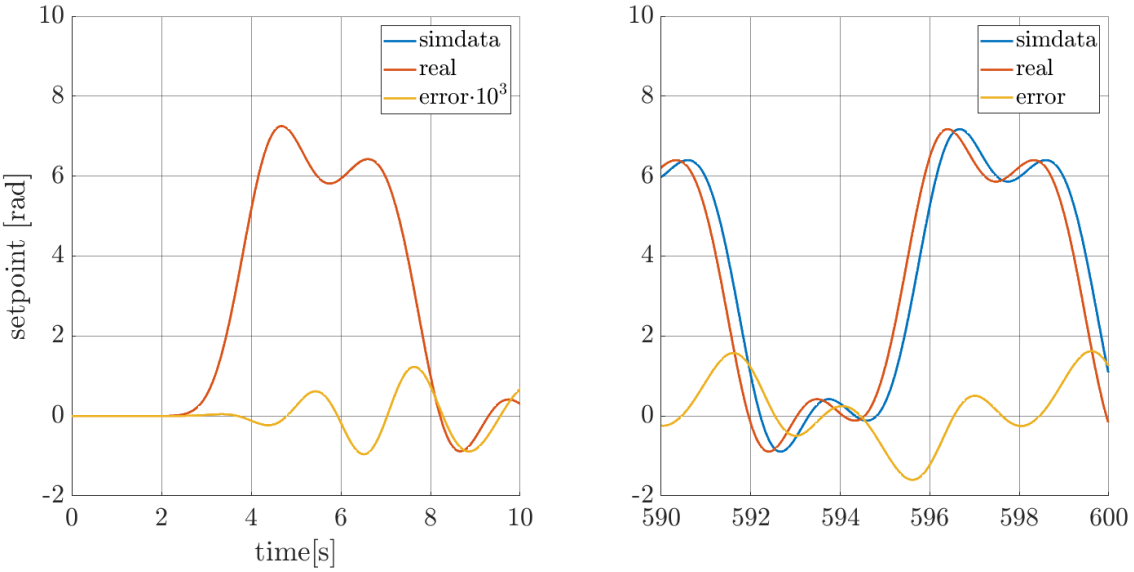
4.7 Test plant performance

The error for double-precision numbers stays constant throughout the experiment (see Figure 4.7a). The error oscillates with an amplitude of $0.1 \cdot 10^{-3}[rad]$ and has a period of 2.5 seconds.

The test with single-precision numbers starts with an error equal to that of double-precision. At the end of the test however, there is a time shift of ~ 0.28[s] between the simulated data and the experiment which leads to a maximum error of almost 2[rad].



(a) PID + 10th-order cycloid input DP



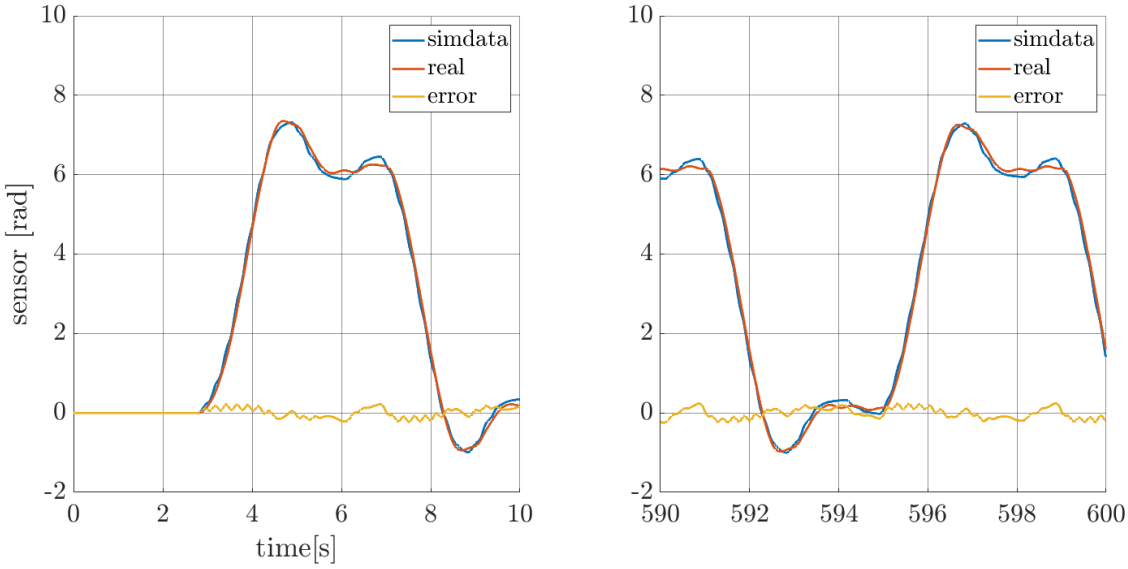
(b) PID + 10th-order cycloid input SP

Figure 4.7: setpoint error, the measurement is after the input signal has been filtered by the low-pass filter, the error is scaled for readability

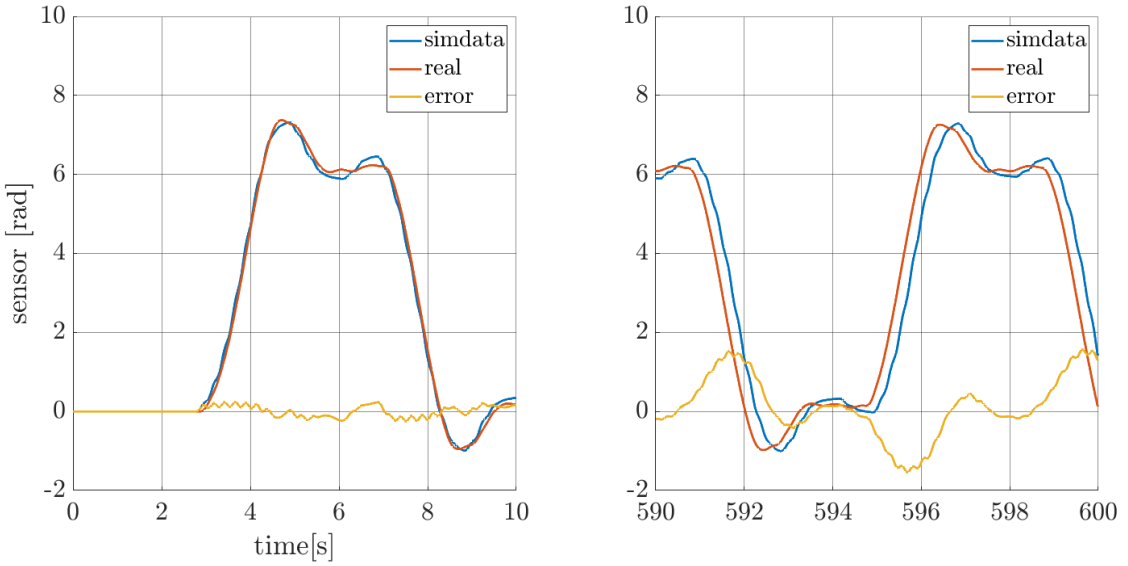
4.7.1 sensor signal

The absolute error for the double-precision measurement between the simulated sensor input and measured sensor input is below $0.2[rad]$ and is maintained during the entire length of the test run.

The single-precision error starts the same as the double-precision but becomes almost $2[rad]$ at the end of the test due to a time shift of $\sim 0.28[s]$.



(a) sensor input of PID + 10th-order low pass with a cycliod input signal double-precision - 100[Hz]



(b) sensor input of PID + 10th-order low pass with a cycliod input signal single-precision - 100[Hz]

Figure 4.8: error in setpoint between simulation and real measured data

5 Discussion

We tested the following characteristics:

- Sample period measurement
- Input latency
- step calculation latency
- Output latency
- Logging latency
- Test plant performance

The generated sampling period has a relative offset. This was expected as the oscillator crystal that generates the clock signal of the microcontroller has a finite precision causing the difference. A more precise oscillator like a temperature-controlled crystal oscillator can reduce the offset.

The control software is constantly polling for a new sample at a certain frequency. This implementation adds sampling time jitter. However, implementing the sampling generation with an interrupt instead of through polling could yield a much lower jitter.

The input latency should be constantly low relative to the sampling period to minimise sampling jitter. Choosing a high sampling frequency increases the relative jitter as this jitter is independent of the sampling period. The designer of the control loop should consider the maximum allowable delay bounds of their control algorithm to decide if this meets their specifications.

The step calculation latency shows a consistent performance gain for single-precision over double-precision. It is important to note that at the cost of half the precision we do not gain twice the performance. Since part of the algorithms does involve logic statements which see no benefit from single-precision over double-precision it is expected that the cumulative performance gain is less than two.

Maximum latency is of main interest as this describes the worst-case performance of the system. For hard real-time systems, this latency should always be bounded. The PID algorithm with a cycloid input shows a relatively large gap between mean and maximum calculation time. The sine and square wave inputs show a much smaller difference.

The motion profile block generates the cycloid signal and makes use of a lot of logic statements. The number of logic statements the algorithm has to check, varies greatly during run time. Hence a lot of jitter is generated. The implementation of the sine and square wave are simpler and therefore generate less jitter. For the PID with the 5th and 10th-order filter algorithms, the difference between mean and maximum becomes much smaller. Since the cycloid generation takes up a much smaller part of the total algorithm the relative jitter becomes smaller too.

The output latency shows similar results as the input latency. The time required to change the output can be improved by simplifying the driver design. Currently, the driver converts an output signal between -1 and 1 from the control algorithm to a suitable duty cycle for the PWM peripheral. By instead letting the control algorithm pre-calculate the duty cycle, this calculation time would be saved reducing the latency.

Logging variables is a monitoring task and is soft real-time. The direct memory access controller was used to minimize CPU overhead. The increase in latency for the more complex

algorithms is due to the increased number of variables and states. The monitoring algorithm has to copy more data, which increases latency.

The error in setpoint for the double-precision should be zero as the same data type and algorithm are used. The cycloid input signal generation, however, is dependent on trigonometric functions. These functions are not standardized among hardware platforms (Monniaux, 2008).

Every sample the current run time is increased by the sample step size. If the step size can not be exactly represented by the floating-point number a rounding error occurs. At the start of the test, this error might not be significant, however, at the end of the test this error has accumulated and becomes clearly visible for the single-precision data type. This issue can be resolved by using a higher precision data type or by selecting a time step that can be exactly represented and thus prevent rounding.

The error between simulation sensor data and measurement sensor data can occur due to an inaccurate simulation model of the mechanical system or if sensor readings do not match the actual state of the system. Bearing friction is highly non-linear at slow rotational speeds, but it is modeled as a linear fiction in simulation. The error is the largest when the angular velocity of the flywheel is zero, this coincides with the part where bearing fiction is very non-linear.

The increased error with single-precision numbers is again a result of an accumulating rounding error.

6 Conclusion

The goal of this project was to investigate if a bare-metal microcontroller is a suitable hardware target for 20-sim generated code. The Arduino Due microcontroller was used and a software framework was created to run generated code with a test plant. We measured latencies for getting sensor data, step calculation, setting actuators and logging variables. The maximum latency for a control algorithm consisting of a PID controller with a 10th-order low-pass filter is $\sim 600[\mu s]$. This is only 6% of the total sample period at a loop frequency of $100[Hz]$. The data connection with the base station has not enough bandwidth to keep up with variable logging at higher frequencies.

In conclusion, the Arduino Due can be a suitable target for a wide range of real-time control algorithms like PID controllers combined with digital filters with sampling frequencies up to about $100[Hz]$.

6.1 Recommendations

Use a faster communication channel for the data interface. Data logging is currently the limiting factor for the sampling frequency. The Arduino Due supports high-speed USB which has a much larger bandwidth than the current data connection over UART.

Add live variable plotting, so the variables can be viewed during a test run. Currently, the data can not be viewed until the test run has been finished. By adding live plotting the user can spot issues during a test saving a lot of time during controller development.

The Arduino Due has a lot of build-in IO-ports like an ADC, 2 DACs and GPIO. By adding drivers for these peripherals the system could be used for a wider range of test plants.

The control algorithm can be converted to work with fixed-point numbers. As there is no dedicated floating-point unit in the microcontroller, there could be a significant performance gain by using fixed-point numbers.

A Appendix: Demo instructions

Demo instructions

Setup

Prerequisites

The demo requires the following software tools listed below.

`20-sim - 4.8` `Matlab - 2020a` `Arduino IDE - 1.8.13`

Installing

1. Install the `Arduino SAM Boards(32-bits ARM Cortex-M3)` in `Tools->Board:->Boards manager...`.
2. Go to `C:\Users\[USERNAME]\AppData\Local\Arduino15\packages\arduino\hardware\sam\1.6.12\variants\arduino_due_x` and open `variant.cpp`

Comment out the following lines:

```
/*
void UART_Handler(void)
{
  Serial.IrqHandler();
}
*/
```

3. Unzip `Demo.zip` and copy the contents of `Demo\libraries` into `C:\Users\[USERNAME]\Documents\Arduino\libraries`.

Settings

4. Connect the Arduino Due programming interface and the Data interface
5. Open the `Arduino IDE` and select `Arduino Due Programming port` in `Tools->Board:->Arduino ARM (32-bit) Boards`
6. Select the Arduino Due programming port in `Tools->Port`
7. Open `Demo\Command_and_control\main.m` with Matlab and run `serialportlist` in the command window and set XX to the port of the Data interface

```
%% comports

comport = "COMXX";
```

Running the tests

1. Open `Demo\linix.emx` with `20-sim` and open the `20-sim Simulator` window.
2. Go to `Tools->Real Time Toolbox->C-Code generation` and select `C-Code for 20-sim submodel`. Set the output directory to `~\Demo\Command_and_control\ccode` and select `OK`.
3. Run `Demo\Command_and_control\main.m` with Matlab
4. Select `Yes` in the window `Upload controller?`
5. Set parameters and select `OK`
6. Set a runtime and select `OK`
7. After the run finishes the variable data is loading in the `Tokenstruct` variable, plotting is done with:

```
figure(1);
plot(tokensstruct.VARIABLE_NAMES2.[SOMEVAR].TIME,tokensstruct.VARIABLE_NAMES2.[SOMEVAR].DATA);
```

B Appendix: Wiring schematic

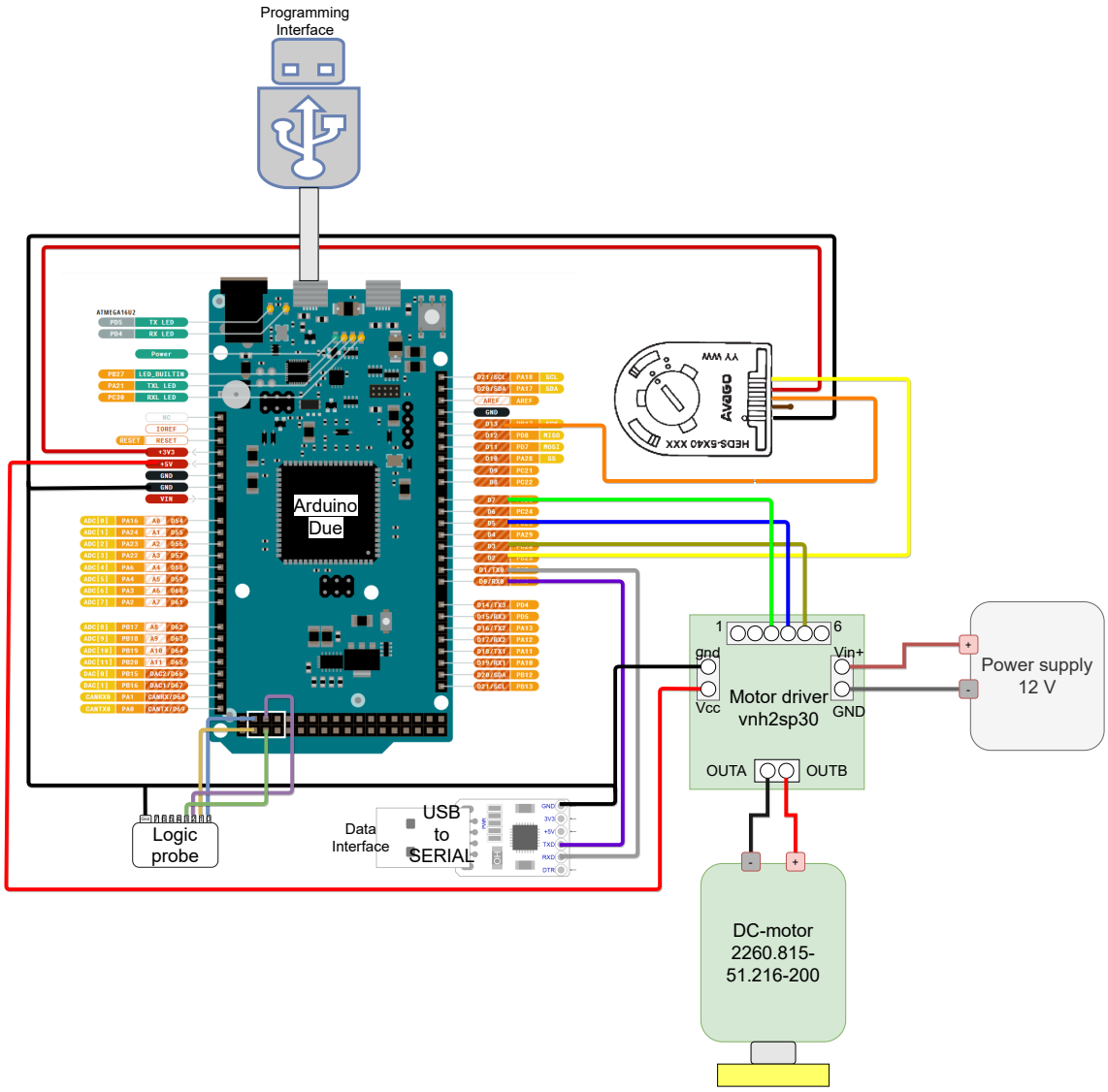


Figure B.1: Schematic of wiring connections

C Appendix: Measurement data

				Logging latency[s]			
Algorithm	Prec	F_sample	Duration	Min	Max	Mean	Sigma
PID	Double	250[Hz]	600[s]	3.572E-06	4.824E-06	3.577E-06	7.112E-08
PID	Single	250[Hz]	600[s]	2.83E-06	4.18E-06	2.88E-06	5.39E-08
PID + 5th order filter	Double	150[Hz]	600[s]	4.70E-06	5.88E-06	4.76E-06	7.83E-08
PID + 5th order filter	Single	150[Hz]	600[s]	3.19E-06	4.48E-06	3.24E-06	6.13E-08
PID + 10th order filter	Double	100[Hz]	600[s]	5.52E-06	6.66E-06	5.53E-06	9.46E-08
PID + 10th order filter	Single	100[Hz]	600[s]	3.68E-06	4.89E-06	3.68E-06	4.88E-08
PID	Double	250[Hz]	600[s]	3.41E-06	4.61E-06	3.45E-06	8.14E-08
PID	Single	250[Hz]	600[s]	2.88E-06	4.08E-06	2.92E-06	5.70E-08
PID + 5th order filter	Double	150[Hz]	600[s]	4.09E-06	5.26E-06	4.14E-06	7.81E-08
PID + 5th order filter	Single	150[Hz]	600[s]	3.19E-06	4.42E-06	3.23E-06	8.23E-08
PID + 10th order filter	Double	100[Hz]	600[s]	5.01E-06	6.27E-06	5.02E-06	7.63E-08
PID + 10th order filter	Single	100[Hz]	600[s]	3.71E-06	4.88E-06	3.72E-06	4.29E-08
PID	Double	250[Hz]	600[s]	3.84E-06	4.99E-06	3.86E-06	5.55E-08
PID	Single	250[Hz]	600[s]	3.07E-06	4.42E-06	3.12E-06	6.28E-08
PID + 5th order filter	Double	150[Hz]	600[s]	4.59E-06	5.77E-06	4.62E-06	1.04E-07
PID + 5th order filter	Single	150[Hz]	600[s]	3.30E-06	4.59E-06	3.33E-06	8.42E-08
PID + 10th order filter	Double	100[Hz]	600[s]	5.45E-06	6.62E-06	5.46E-06	6.67E-08
PID + 10th order filter	Single	100[Hz]	600[s]	3.70E-06	4.84E-06	3.70E-06	3.64E-08

				Input latency[s]			
Algorithm	Prec	F_sample	Duration	Min	Max	Mean	Sigma
PID	Double	250[Hz]	600[s]	3.904E-06	1.626E-05	1.491E-05	5.424E-07
PID	Single	250[Hz]	600[s]	4.46E-06	1.79E-05	1.66E-05	6.88E-07
PID + 5th order filter	Double	150[Hz]	600[s]	4.03E-06	1.75E-05	1.61E-05	1.11E-06
PID + 5th order filter	Single	150[Hz]	600[s]	4.26E-06	1.65E-05	1.51E-05	9.97E-07
PID + 10th order filter	Double	100[Hz]	600[s]	4.05E-06	1.75E-05	1.61E-05	1.08E-06
PID + 10th order filter	Single	100[Hz]	600[s]	4.30E-06	1.65E-05	1.52E-05	9.41E-07
PID	Double	250[Hz]	600[s]	4.03E-06	1.74E-05	1.61E-05	9.22E-07
PID	Single	250[Hz]	600[s]	4.32E-06	1.66E-05	1.52E-05	7.10E-07
PID + 5th order filter	Double	150[Hz]	600[s]	4.03E-06	1.75E-05	1.61E-05	1.00E-06
PID + 5th order filter	Single	150[Hz]	600[s]	4.31E-06	1.66E-05	1.52E-05	7.75E-07
PID + 10th order filter	Double	100[Hz]	600[s]	3.90E-06	1.63E-05	1.49E-05	9.10E-07
PID + 10th order filter	Single	100[Hz]	600[s]	4.30E-06	1.65E-05	1.52E-05	8.80E-07
PID	Double	250[Hz]	600[s]	3.94E-06	1.58E-05	1.45E-05	8.35E-07
PID	Single	250[Hz]	600[s]	4.27E-06	1.66E-05	1.52E-05	8.31E-07
PID + 5th order filter	Double	150[Hz]	600[s]	3.92E-06	1.58E-05	1.45E-05	1.12E-06
PID + 5th order filter	Single	150[Hz]	600[s]	4.31E-06	1.70E-05	1.56E-05	1.20E-06
PID + 10th order filter	Double	100[Hz]	600[s]	3.96E-06	1.58E-05	1.45E-05	1.12E-06
PID + 10th order filter	Single	100[Hz]	600[s]	4.43E-06	1.62E-05	1.49E-05	1.24E-06

				Calc loop latency[s]			
Algorithm	Prec	F_sample	Duration	Min	Max	Mean	Sigma
PID	Double	250[Hz]	600[s]	4.309E-05	1.463E-04	8.351E-05	2.879E-05
PID	Single	250[Hz]	600[s]	2.73E-05	1.13E-04	5.41E-05	2.73E-05
PID + 5th order filter	Double	150[Hz]	600[s]	1.30E-04	2.65E-04	2.04E-04	2.81E-05
PID + 5th order filter	Single	150[Hz]	600[s]	8.92E-05	1.90E-04	1.30E-04	2.80E-05
PID + 10th order filter	Double	100[Hz]	600[s]	3.19E-04	5.50E-04	4.87E-04	3.06E-05
PID + 10th order filter	Single	100[Hz]	600[s]	2.24E-04	3.53E-04	2.92E-04	2.80E-05
PID	Double	250[Hz]	600[s]	7.59E-05	1.18E-04	1.05E-04	2.18E-06
PID	Single	250[Hz]	600[s]	5.04E-05	9.18E-05	7.89E-05	2.29E-06
PID + 5th order filter	Double	150[Hz]	600[s]	1.62E-04	2.40E-04	2.27E-04	2.10E-06
PID + 5th order filter	Single	150[Hz]	600[s]	1.15E-04	1.69E-04	1.56E-04	2.18E-06
PID + 10th order filter	Double	100[Hz]	600[s]	3.30E-04	5.47E-04	5.33E-04	3.81E-06
PID + 10th order filter	Single	100[Hz]	600[s]	2.48E-04	3.30E-04	3.16E-04	2.57E-06
PID	Double	250[Hz]	600[s]	4.04E-05	6.51E-05	5.77E-05	1.87E-06
PID	Single	250[Hz]	600[s]	2.32E-05	3.44E-05	2.81E-05	8.65E-07
PID + 5th order filter	Double	150[Hz]	600[s]	1.25E-04	1.94E-04	1.87E-04	6.33E-06
PID + 5th order filter	Single	150[Hz]	600[s]	8.89E-05	1.19E-04	1.14E-04	2.51E-06
PID + 10th order filter	Double	100[Hz]	600[s]	3.11E-04	5.02E-04	4.88E-04	1.54E-05
PID + 10th order filter	Single	100[Hz]	600[s]	2.28E-04	3.17E-04	3.00E-04	6.25E-06

				Set output latency[s]			
Algorithm	Prec	F_sample	Duration	Min	Max	Mean	Sigma
PID	Double	250[Hz]	600[s]	1.296E-05	1.590E-05	1.403E-05	6.490E-07
PID	Single	250[Hz]	600[s]	1.23E-05	1.53E-05	1.34E-05	5.78E-07
PID + 5th order filter	Double	150[Hz]	600[s]	1.20E-05	1.49E-05	1.31E-05	5.87E-07
PID + 5th order filter	Single	150[Hz]	600[s]	1.30E-05	1.59E-05	1.41E-05	6.55E-07
PID + 10th order filter	Double	100[Hz]	600[s]	1.20E-05	1.48E-05	1.31E-05	5.78E-07
PID + 10th order filter	Single	100[Hz]	600[s]	1.30E-05	1.59E-05	1.41E-05	6.50E-07
PID	Double	250[Hz]	600[s]	1.20E-05	1.49E-05	1.31E-05	5.86E-07
PID	Single	250[Hz]	600[s]	1.34E-05	1.60E-05	1.41E-05	6.56E-07
PID + 5th order filter	Double	150[Hz]	600[s]	1.20E-05	1.48E-05	1.31E-05	5.89E-07
PID + 5th order filter	Single	150[Hz]	600[s]	1.05E-05	1.60E-05	1.41E-05	6.51E-07
PID + 10th order filter	Double	100[Hz]	600[s]	1.30E-05	1.59E-05	1.40E-05	6.49E-07
PID + 10th order filter	Single	100[Hz]	600[s]	1.30E-05	1.59E-05	1.41E-05	6.49E-07
PID	Double	250[Hz]	600[s]	9.89E-06	1.49E-05	1.31E-05	7.72E-07
PID	Single	250[Hz]	600[s]	1.04E-05	1.59E-05	1.38E-05	8.44E-07
PID + 5th order filter	Double	150[Hz]	600[s]	1.21E-05	1.49E-05	1.31E-05	6.00E-07
PID + 5th order filter	Single	150[Hz]	600[s]	1.35E-05	1.65E-05	1.47E-05	6.82E-07
PID + 10th order filter	Double	100[Hz]	600[s]	1.21E-05	1.49E-05	1.31E-05	5.95E-07
PID + 10th order filter	Single	100[Hz]	600[s]	1.24E-05	1.53E-05	1.35E-05	5.95E-07

Bibliography

Bax, M. (2006), The Linix2 mechatronic demonstrator, Pre-MSc Thesis 020CE2006, University of Twente, premsc.
https://pydio.ram.ewi.utwente.nl/ws-StudentTheses/2006/bax2006premsc.pdf

Bezemer, M. (2013), *Cyber-physical systems software development: way of working and tool suite*, Ph.D. thesis, University of Twente, Netherlands, doi:10.3990/1.9789036518796.

Broenink, J. F. (1999a), 20-sim software for hierarchical bond-graph/block-diagram models, *Simulation Practice and Theory*, **vol. 7**, no.5, pp. 481 – 492, ISSN 0928-4869, doi:https://doi.org/10.1016/S0928-4869(99)00018-X.
http://www.sciencedirect.com/science/article/pii/S092848699900018X

Broenink, J. F. (1999b), Introduction to physical systems modelling with bond graphs, *SiE whitebook on simulation methodologies*, **vol. 31**, p. 2.

Broenink, J. F. and G. H. Hilderink (2001), A structured approach to embedded control systems implementation, in *Proceedings of the 2001 IEEE International Conference on Control Applications (CCA'01)(Cat. No. 01CH37204)*, IEEE, pp. 761–766.

Controllab Products (2020a), 20-sim.
https://www.controllab.nl/product/software/20-sim/

Controllab Products (2020b), 20-sim4c.
https://www.controllab.nl/services/software/20-sim-4c/

Dick, R. P., G. Lakshminarayana, A. Raghunathan and N. K. Jha (2003), Analysis of power dissipation in embedded systems using real-time operating systems, **vol. 22**, no.5, pp. 615–627, ISSN 1937-4151, doi:10.1109/TCAD.2003.810745.

Dokter, J. (2016), 20-sim Template for Raspberry Pi 3, BSc Thesis 019RaM2016, University of Twente, bsc.
https://cloud.ram.eemcs.utwente.nl/index.php/s/SydM7FSnCiBiaYw

Gaffar, A. A., O. Mencer and W. Luk (2004), Unifying bit-width optimisation for fixed-point and floating-point designs, in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, IEEE, pp. 79–88.

Gerum, P. (2004), Xenomai-Implementing a RTOS emulation framework on GNU/Linux, *White Paper, Xenomai*, pp. 1–12.

Heineman, G. T. and W. T. Councill (2001), Component-based software engineering, *Putting the pieces together*, p. 5.

Iordache, C. and P. T. P. Tang (2003), An Overview of Floating-point Support and Math Library on the Intel/spl reg/XScale/spl trade/architecture, in *Proceedings 2003 16th IEEE Symposium on Computer Arithmetic*, IEEE, pp. 122–128.

Johansson, G. (2018), *Real-Time Linux Testbench on Raspberry Pi 3 using Xenomai*, Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS).

Kleijn, C. (2009), *20-sim 4.1 Reference Manual*, Getting Started with 20-sim.

Knospe, C. (2006), PID control, *IEEE Control Systems Magazine*, **vol. 26**, no.1, pp. 30–31.

Marti, P., J. M. Fuertes, G. Fohler and K. Ramamritham (2001), Jitter compensation for real-time control systems, in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, IEEE, pp. 39–48.

Monniaux, D. (2008), The pitfalls of verifying floating-point computations, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **vol. 30**, no.3, pp. 1–41.

Waters, K. (2009), Prioritization using moscow, *Agile Planning*, **vol. 12**, p. 31.

Xu, J. and D. L. Parnas (1993), On satisfying timing constraints in hard-real-time systems, *IEEE transactions on software engineering*, **vol. 19**, no.1, pp. 70–84.

Yates, R. (2009), Fixed-point arithmetic: An introduction, *Digital Signal Labs*, **vol. 81**, no.83, p. 198.

Yiu, J. (2010), APPENDIX A - The Cortex-M3 Instruction Set, Reference Material, in *The Definitive Guide to the ARM Cortex-M3 (Second Edition)*, Newnes, Oxford, pp. 349 – 403, second edition edition, ISBN 978-1-85617-963-8, doi:https://doi.org/10.1016/B978-1-85617-963-8.00025-9. http://www.sciencedirect.com/science/article/pii/B9781856179638000259

Zagan, I. (2015), Improving the performance of CPU architectures by reducing the Operating System overhead, in *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, IEEE, pp. 1–6.