CONTROL OF THE PRODUCTION CELL ON RASPBERRY PI USING A REAL-TIME ROBOT-SOFTWARE FRAMEWORK

N.E.D. (Nick) in het Veld

MSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
dr. ir. G. van Oort
H.H. Folmer, MSc

December, 2023

056RaM2023
Robotics and Mechatronics
EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

This report describes the realisation of an embedded-control-software architecture on the Raspberry Pi 4 for the Production Cell demonstrator machine. The embedded-control-software architecture is built with a real-time robot-software framework that has been developed at the Robotics and Mechatronics (RaM) group. The robot-software framework utilises ROS2, 20-sim and Xenomai to build an embedded-control-software architecture. The design of the embedded-control-software architecture is based on the layered-controller-structure concept. The choices of allocations concerning the embedded-control-software architecture have been made based on design space explorations and set requirements. The chosen allocations have been implemented for the embedded-control-software architecture using a model-driven design approach.

The implemented embedded-control-software architecture utilises three Raspberry Pi boards to control the Production-Cell setup. The various software components, which comprise the layered controller structure, run on each Raspberry Pi board. Each Raspberry Pi board is responsible for the control of two Production-Cell Units. The embedded-control-software architecture for a board is divided into a firm-real-time and soft-real-time part. For the firm-real-time part, a single Xenomai-based real-time task runs a 20-sim-code-generated motion controller. For the soft-real-time part, multiple ROS2-based tasks are responsible for the discrete-event control. A ROS2-based bridge is used to allow communication between the firm-real-time and soft-real-time part. Communication between boards is done by using ROS2's networking capabilities over Ethernet.

Different aspects of the implemented embedded-control-software architecture have been characterised by performing measurements with a test bed. A test scenario is used to represent the nominal-load scenario of the Production-Cell setup. The control performance, load balancing and real-timeness of the embedded-control-software architecture have been characterised during the nominal-load scenario. For the control-performance aspect, the setpoint and position error of each unit has been captured. For the load-balancing aspect, the load per Raspberry Pi core has been measured by evaluating kernel-related information of Linux and Xenomai. For the real-timeness aspect, the jitter of each embedded-control-software-architecture component has been measured using a POSIX-based clock command. The processor-load results show that CPU load of each Raspberry Pi core stays below the recommended maximum load (< 90%). The jitter results show that the ROS2-based embedded-control-software-architecture components can have significant high jitter (< 90% of period), while the Xenomai-based components have relatively low jitter (< 5% of period).

The test results indicate that load balancing in the embedded-control-software architecture meets the set requirements and that the architecture can distribute the CPU load well over the Raspberry Pi cores. Regarding the control performance, the test results show that it falls short according to the steady-state-error criterion of prior Production-Cell-related work (> 0.5 mm).

In future work, further investigations should be done regarding the system performance during high system-stress scenarios. Furthermore, investigations should be done regarding the relationship between control performance and real-timeness.

Contents

1 Introduction

1.1 Context

The demand for systems utilising small but powerful computer boards increases day by day. Here, the Robotics and Mechatronics (RaM) group of University of Twente is active in the research about how to efficiently realise control software on embedded computer boards. To adopt to the latest trends, the RaM-group uses the Raspberry Pi (RPi) computer board to prototype control software, which is a widely-adopted computer board that enjoys community support due to its open-sourceness (University of Cambridge, 2023).

At the RaM group, a real-time robot-software framework has been developed for the Raspberry Pi 4 (Meijer, 2021). In combination with two tools, the framework allows the design of a full-fledged embedded-control-software architecture. The framework relies on two software tools: ROS2 and 20-sim. The first tool, ROS2, is a commonly-used software development kit for robotics applications (ROS, 2023b). The second tool, 20-sim, is a modeling and simulation program which is used for modeling complex multi-domain systems and for the development of control systems (20-sim, 2021).

While the Raspberry Pi and its robot-software framework have been used for the control of simple demonstrator systems at RaM (Meijer, 2021), their capabilities with respect to complex systems remain to be seen. Thus, an opportunity to utilise both on a complex demonstrator machine would showcase their capabilities.

At the RaM group, a Production-Cell demonstrator machine exists (Van den Berg, 2006). It is modelled after a plastic-injection molding machine, where multiple units require to work together to fulfill its objective (Figure 1.1). For the Production Cell, discrete-event control and motion control play a significant role for its operations.

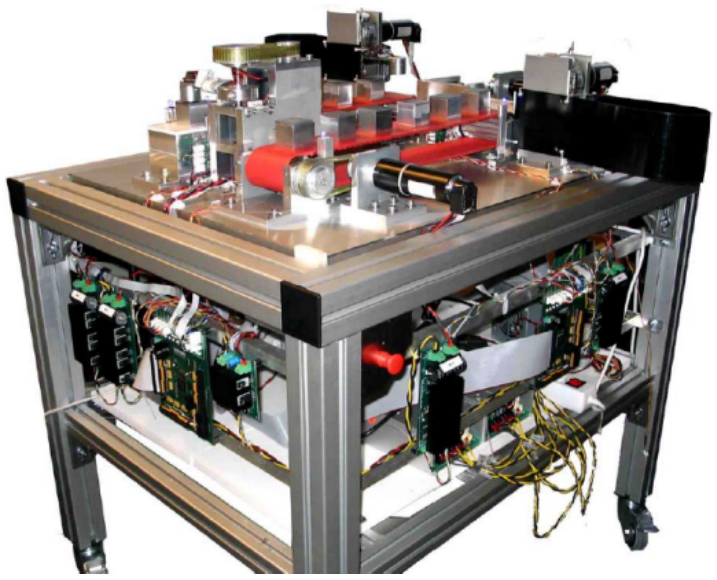


Figure 1.1: A spectators's view of the Production Cell (Ridder, 2018).

The Production Cell and robot-software framework provide the opportunity to develop and deploy an embedded-control-software architecture on the Raspberry Pi (Figure 1.2). The main issues herein lie that a performant embedded-control-software architecture must be realised, while taking the available resources of the Raspberry Pi into account.

The Production Cell provides an interface for three embedded-control-software implementations via dipswitches (Figure 1.2), enabling the prototyping of different embedded-control-software architectures. An implementation, consisting of the work of Sassen (2009), currently exists, which is the fifth full demonstrator after the work of Maljaars (2006). The implementation of Sassen (2009) can be shown when the dip switches are in position 1, see Figure 1.2 and Figure A.6.

Two implementations can still be attached to the setup. In this thesis, a Raspberry-Pi-based implementation will be realised and connected to the setup when the dipswitches are in position 2, see Figure 1.2 and Figure A.6.

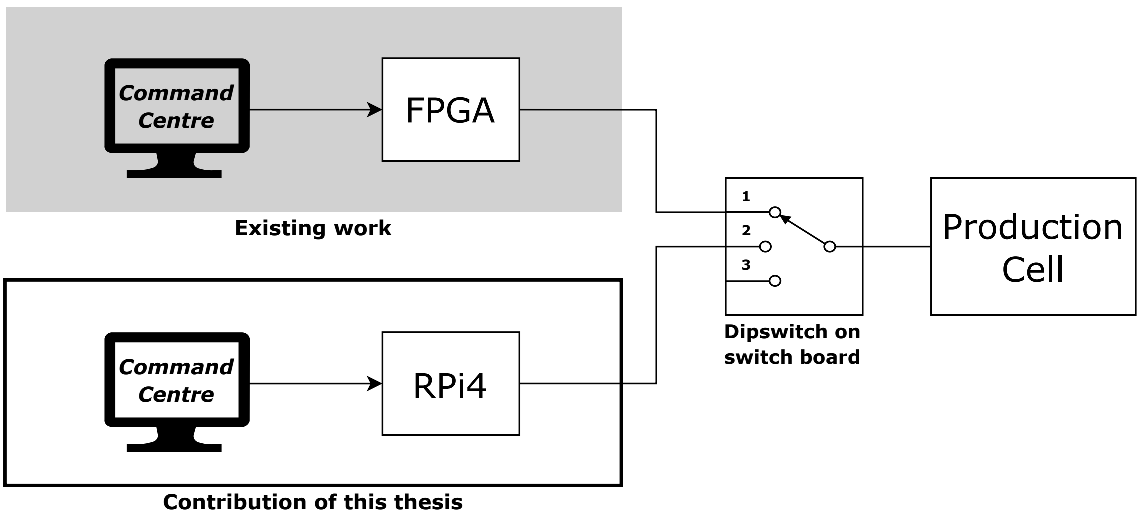


Figure 1.2: Context of this thesis

1.2 Design objectives

The main goals of this thesis are formulated as follows:

1. Realisation of a performant demonstrator for the Production Cell using the Raspberry Pi 4 .
2. Characterisation of the implemented embedded-control-software architecture in terms of control performance, load balancing, and real-timeness.

The project constraints of this thesis are formulated as follows:

- One or more Raspberry Pi boards are used for the design of the embedded-control-software architecture.
- The robot-software framework is used as is and will not be modified, unless it is deemed necessary.
- The Production Cell is not altered such to keep the existing implementation running (Figure 1.2).

1.3 Approach

To realise a performant demonstrator, different configurations for the embedded control software on the Raspberry Pi will be evaluated for its effectiveness. This is done using a design space exploration. Based on the results of the design space exploration, the best configuration is chosen and implemented as the embedded-control-software architecture for the Raspberry Pi.

The implemented embedded-control-software architecture will be characterised by performing measurements while the demonstrator is active. The results of the measurements will be used to determine whether the implemented embedded-control-software architecture is effective or not. This will be done in comparison with the results of prior Production-Cell-related work.

1.4 Report outline

Chapter 2, focusses on the background of the Production Cell, the embedded control system and the embedded-control-software design approach. Chapter 3 applies this knowledge in deciding which embedded-control-software architecture is best to implement with respect to the Production-Cell demonstrator. Chapter 4 describes the design of the implemented embedded-control-software architecture on both functional and implementation level. Chapter 5 elaborates on the used test bed and ways of measuring the variables of interest, concluding with a discussion of the presented measurement results. The last chapter, Chapter 6, concludes the work of this thesis, where the project goals and future work are discussed.

2 Background

2.1 Introduction

This chapter introduces the necessary background information for understanding the following chapters of this thesis. First, it covers important details of the Production Cell, elaborating on its design and purpose, and ending with a brief overview of Production-Cell-related work. After this, an explanation of the embedded control system of this thesis is given, which describe the Raspberry Pi and the real-time robot-software framework. Lastly, the aspects of the embedded-control-software design approach are explained. Here, the layered-controller-structure approach, real-timeness and the model-driven design approach are described.

2.2 Production Cell

The Production-Cell demonstrator is modelled after a plastic-injection moulding machine creating buckets (Figure 2.1). Instead of creating buckets, the Production Cell mimics the bucket-creation process by transferring and performing actions on small metal blocks, symbolising the bucket. After a block is "moulded" in the Production Cell, the block is transferred away, after which the block is being fed back again into the system. This results in initiating the moulding process once more for a block.

Different disciplines play a role in the control of the Production Cell, such as motion control and discrete-event control. The latter discipline particularly plays a prominent role, since the handlings among the units have to be synchronised for the Production Cell to function properly (Figure A.3). In other words, each unit requires to communicate with its nearest neighbours to fulfill its own objective.

There are six units in the Production Cell, each unit having one degree of freedom for its motion (Figure 2.1). For a block to complete one cycle, the following units are involved. First, a *Feeder Belt* brings a block towards the moulding section. When arrived at the belt's end, the block is pushed by the *Feeder Robot* towards the *Molder Door*. After opening the door, the *Extraction Robot* extracts the block from the moulding section by lifting it with a magnet and placing it on a belt afterwards. This *Extraction Belt* then transports it away from the moulding section. Lastly, the *Rotation Robot* residing at the end of the belt feeds the block back into the system by lifting it with a magnet and passing the block to the other belt, thus completing the cycle.

To be able to switch between different embedded-control-software implementations, there are electrical switch boards to which up to three implementations can be connected (Figure 2.2; Figure A.6). A dip switch exists on each board. When the dip switch on each switch board is flipped to the same configuration, it enables one of the three implementations to take full control of the setup. To interface computer boards with the setup, a 50-pin connector is present on each switch board (Figure A.7). Each switch board covers the control of two units of the Production Cell (Figure A.4). An FPGA-based embedded-control-software implementation (Figure 2.2d) already exists and can be used when the dip switches are in position 1, see Figure 1.2 and Figure A.6.

Other work has been done on the Production Cell, each work utilising a different set of tools and frameworks to control the Production Cell. An overview of related work with respect to the Production Cell is shown in Table 2.1.

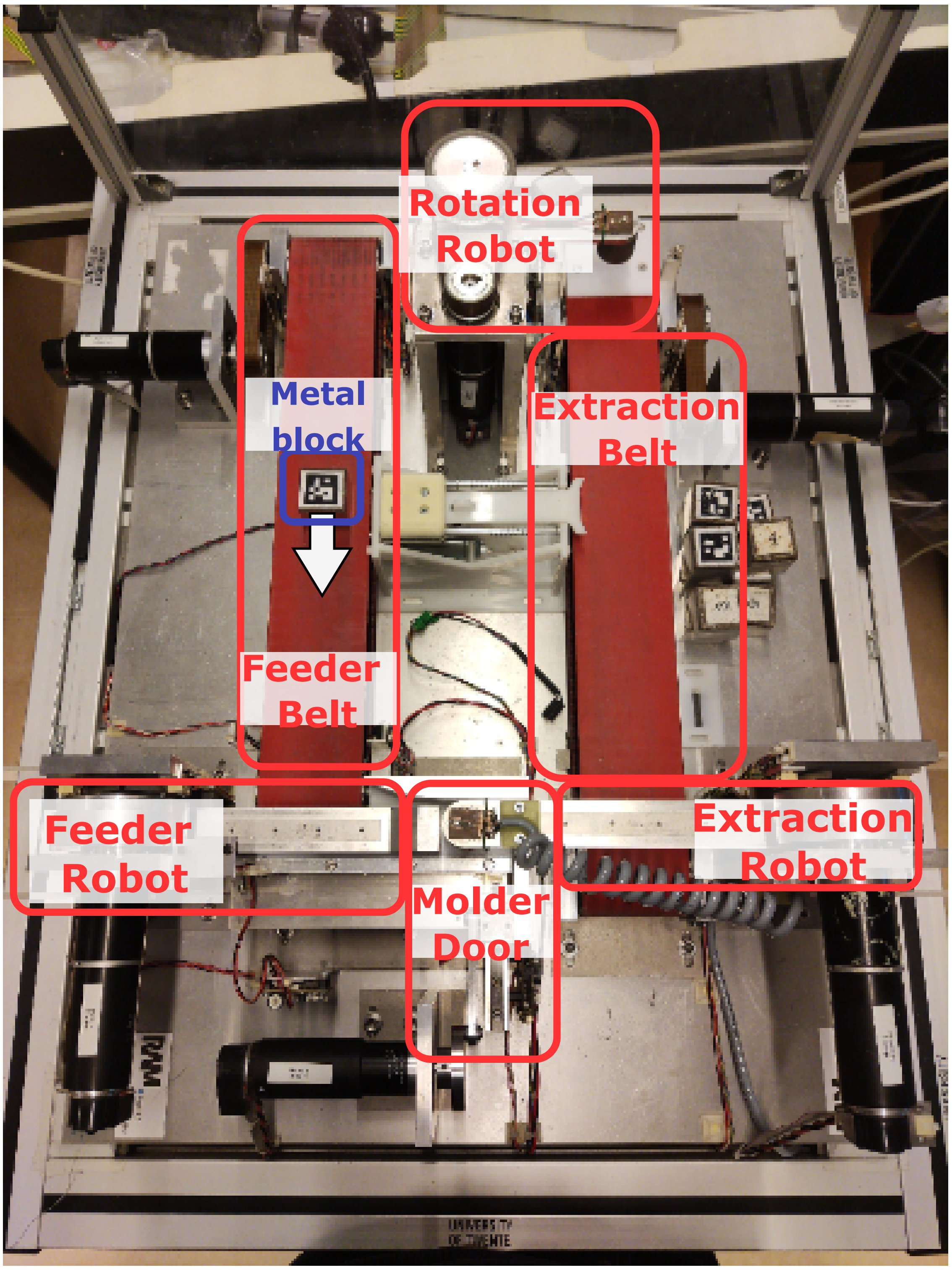
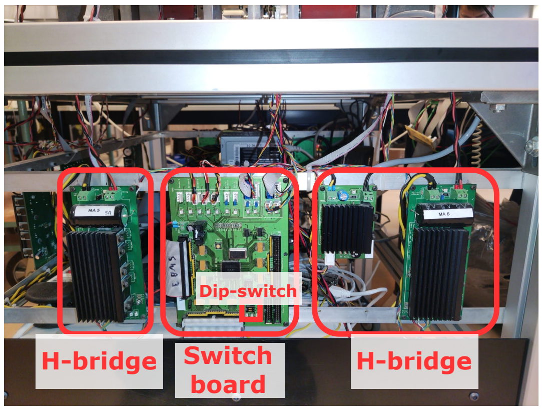
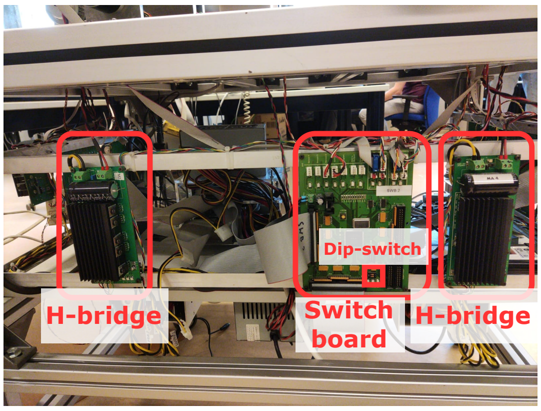


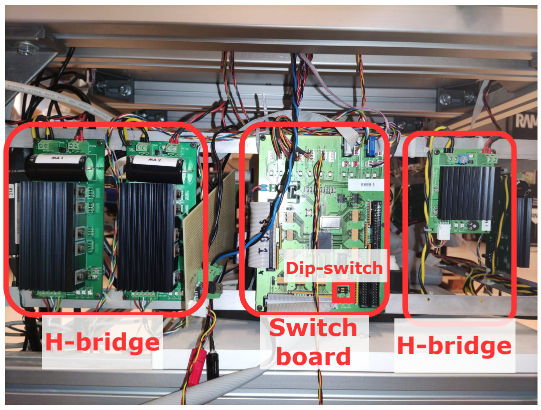
Figure 2.1: Overview of the units of the Production Cell



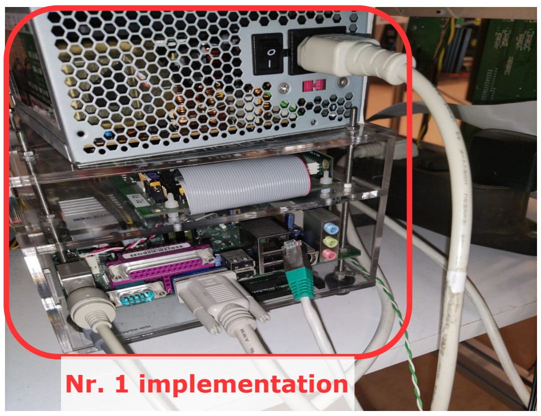
(a) Switch board 1 plus H-bridges



(b) Switch board 2 plus H-bridges



(c) Switch board 3 plus H-bridges



(d) Inside view

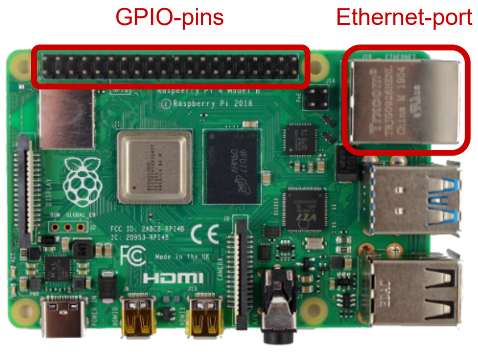
Figure 2.2: Different views of the Production Cell's internals

Table 2.1: Overview of Production-Cell-related work. The checkmark indicates the existing implementation in FPGA.

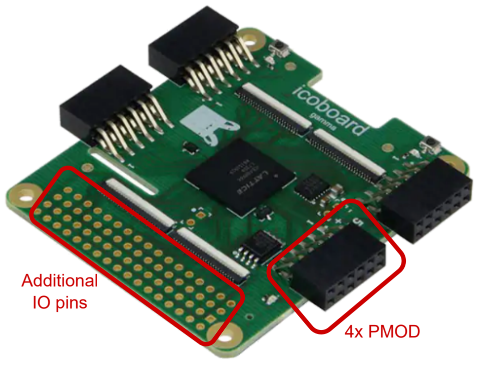
Work	Embedded board	RTOS	Tooling
Maljaars (2006)	PC104 (CPU)	RTAI	20-sim, gCSP, UPPAAL, CT-library
Huang et al. (2007)	PC104 (CPU)	RTAI	POOSL
Zuijlen (2008)	Xilinx Spartan III (FPGA)	-	20-sim, gCSP, Handel-C
Verhaar (2008)	PC104 (CPU)	RTAI	Ptolemy II
Veldhuijzen (2009)	PC104 (CPU)	QNX	20-sim, gCSP
Sassen (2009) ✓	Xilinx Spartan III (FPGA)	-	20-sim, gCSP, Handel-C
Vos (2015)	PC104 (CPU)	QNX	20-sim, ROS, LUNA, TERRA
Ridder (2018)	RaMstix (CPU)	QNX	20-sim, IBM Rhapsody, ROS, LUNA, TERRA

2.3 Embedded control system

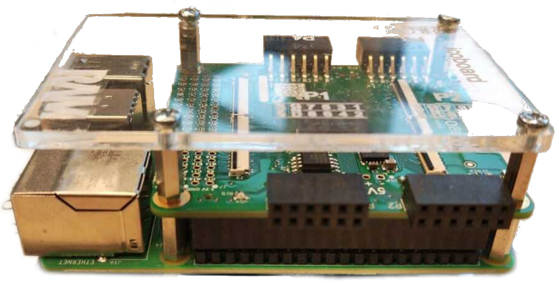
2.3.1 Raspberry Pi



(a) Raspberry Pi 4B



(b) icoBoard



(c) Board stack

Figure 2.3: The available computer boards in this thesis

The Raspberry Pi version 4B (Figure 2.3a) is a single-board computer that houses an ARM processor containing four cores (Raspberry Pi Foundation, 2014). It is capable of running different OSes, but it is standardly shipped with its default Linux operating system, Raspberry OS. For connectivity purposes, the Raspberry Pi has a 40-pin GPIO interface with which it can interface with peripherals. However, it can be argued that the capabilities of its pins are inadequate for interfacing with mechatronic systems. This is due to its capable I/O operations being limited. To address this issue, at RaM a Raspberry-Pi-compatible icoBoard (Figure 2.3b) is used (Icoboard, 2016). The icoBoard is an FPGA-based I/O board that can be stacked on the Raspberry Pi as a HAT (Figure 2.3c). By default it has four PMOD-connectors with which it can interface with peripherals. More connectors can be added to the icoBoard by utilising its additional I/O pins.

2.3.2 Robot-software framework

The robot-software framework developed by Meijer (2021) relies on two software tools to build up an embedded-control-software architecture, which are ROS2 and 20-sim (Figure 2.4).

ROS2 is an open-source software framework for robotics applications (ROS, 2023b). Its component-based design philosophy enables the partition of user code into seperate components, which are called *nodes*. To facilitate communication between nodes, ROS2 relies on third-party Data Distribution Services (DDS) and Real-Time Publish Subscribe (RTPS) communication protocols (ROS, 2023a). ROS2 provides an abstract middleware interface to the user, which has been built on top of DDS/RTPS implementations' APIs and tools. Using the middleware interface, nodes can communicate with each other using a publish-subscribe pattern. A

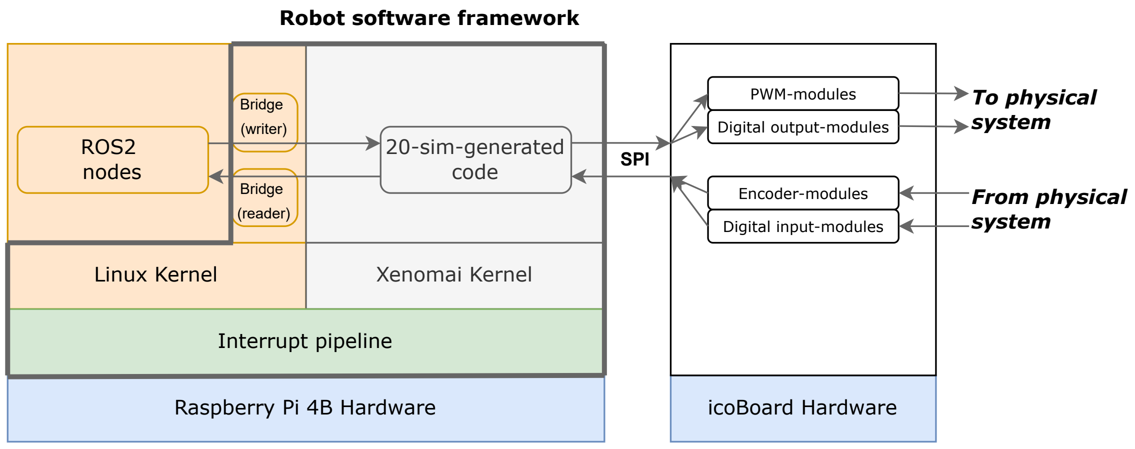


Figure 2.4: Robot-software framework and the icoBoard-based interface

node has the option to be a publisher and/or subscriber. A *publisher* puts information onto a *topic*, while a *subscriber* retrieves information from a topic. The name of a topic is always denoted with a forward slash plus its name, for example */name*.

20-sim is a modeling and simulation software package for mechatronic systems (20-sim, 2021). Inside the modelling environment of 20-sim, systems are modelled using equations, bond graphs and/or block diagrams. Included with 20-sim are various toolboxes, one of them being the Code Generation toolbox. This toolbox enables the generation of C/C++ code out of any 20-sim model. By creating 20-sim models and mapping these to code, rapid prototyping of control systems can be done on embedded hardware.

Included with the robot-software framework is a Linux operating system that has been patched with Xenomai on which the framework runs (Xenomai, 2023a). Xenomai is a real-time operating-system framework for Linux, which enables running real-time applications in Linux using a dual-kernel approach.

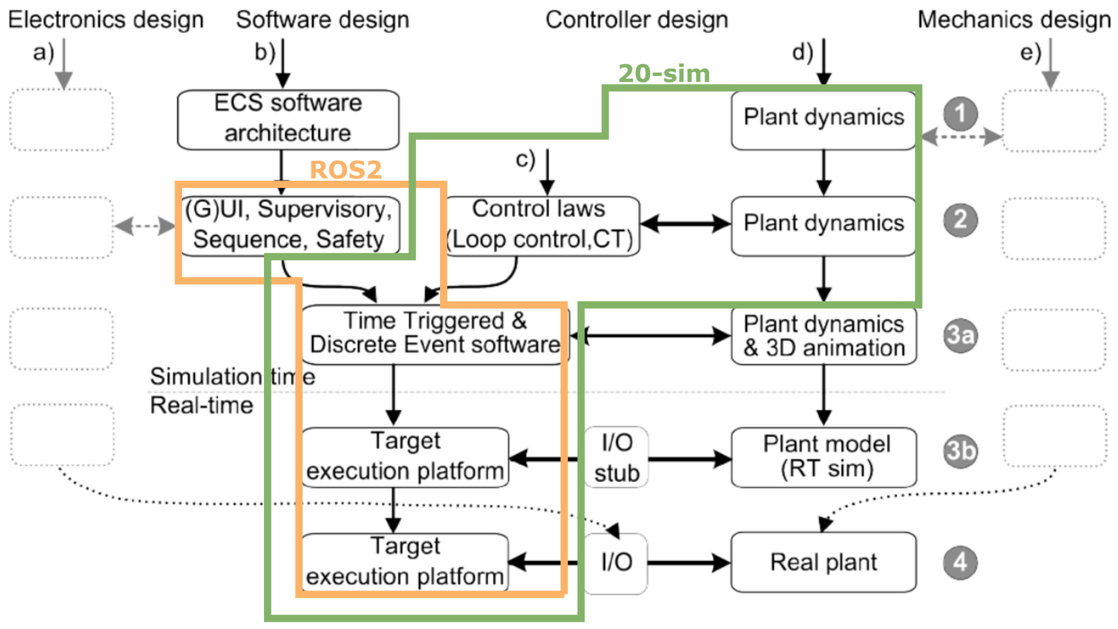
In the framework, a ROS2 node runs on the Linux kernel, while 20-sim-generated code runs on the Xenomai kernel. To allow communication between components running on either kernel, a bridge has also been developed by Meijer (2021). The bridge utilises the XDDP protocol of Xenomai for cross-kernel communication. The XDDP protocol has 32 ports by default over which data can be sent between Linux and Xenomai (Xenomai, 2023b). The framework divides the bridge into two seperate ROS2 nodes. One node reads data from XDDP ports based on a periodic timer, while the other node writes data to XDDP ports based on topic callbacks. Regarding the 20-sim-generated code, an interface is attached to the 20-sim-generated code component. With this interface, the component reads from and writes to the XDDP ports every tick of its period.

To allow the real-time task running the 20-sim-generated code to interface with the physical system, it can communicate with the interface running on an icoBoard via the SPI protocol (Figure 2.4). Using an SPI-based interface which can be attached to a 20-sim-generated code component, the component reads from and writes to the icoBoard FPGA every tick of its period. The icoBoard runs Verilog code on its FPGA which is dedicated to interfacing with physical systems. The interface consists of digital inputs, digital outputs, encoders and PWMs. With each PMOD-connector of the icoBoard two digital inputs, one digital output, one PWM and one encoder is associated (Figure B.1).

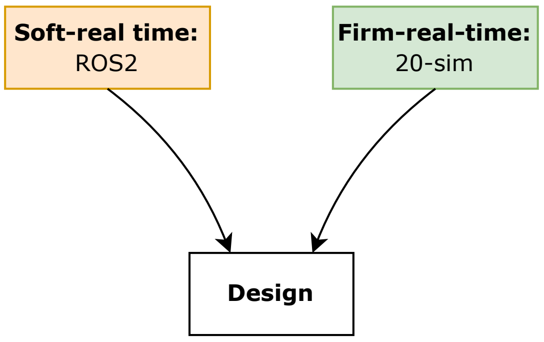
2.4 Embedded-control-software design procedure

2.4.1 Model-driven design approach

The workflow underlying the embedded-control-software design procedure is the model-driven design approach (Broenink and Ni, 2012). An overview of this workflow with respect to the robot-software framework is shown in Figure 2.5. By following the principles of model-driven development, a first-time-right approach for the design of the embedded software is strived for.


Electronics design
Software design
Controller design
Mechanics design
a)
b)
c)
d)
e)
20-sim
ROS2
ECS software architecture
(G)UI, Supervisory, Sequence, Safety
Control laws (Loop control,CT)
Plant dynamics
Plant dynamics
Time Triggered & Discrete Event software
Plant dynamics & 3D animation
Simulation time
Real-time
Target execution platform
I/O stub
Plant model (RT sim)
Target execution platform
I/O
Real plant
1
2
3a
3b
4

(a) General view of the workflow and the covered aspects by the framework



(b) Abstract view of the workflow in the framework

Figure 2.5: The model-driven design workflow (Broenink and Ni, 2012) and its relations with the robot-software framework. For real-timeness, see Section 2.4.3.

2.4.2 Layered controller structure

With the robot-software framework a *layered controller structure* can be built to realise an embedded-control-software architecture (Figure 2.6). The layered controller structure embodies the functional and timing partition of controllers. In this structure, each controller requires a certain degree of real-timeness for its tasks.

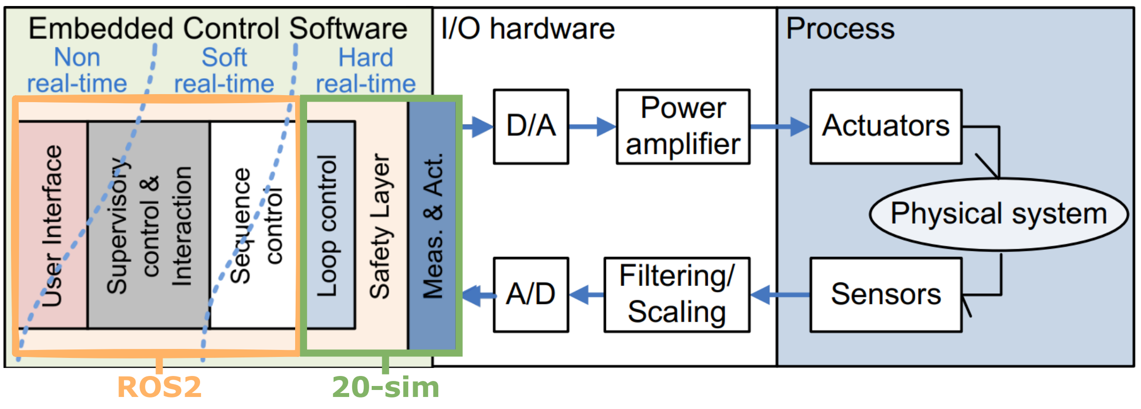


Figure 2.6: The functional and timing partition of an embedded control system (Broenink et al., 2010). The partition of controllers with respect to the robot-software framework is also shown.

2.4.3 Real-timeness

Real-time tasks have to adhere to a deadline, unlike non-real time tasks (Liu et al., 2015). There are different degrees of real-time, which are described below (Boode, 2018; Bezemer et al., 2011):

- **Hard real-time (HRT):** Whenever a deadline is missed, the consequences are catastrophic.
- **Firm real-time (FRT):** Infrequent deadline misses, less than k deadline misses in a given time frame of t s, will not be catastrophic for the system.
- **Soft real-time (SRT):** After a deadline miss the system may still execute, though possibly with less functionality, but may recover to normal operation.
- **Non real-time (NRT):** Deadlines do not have to be met, so the response time of a task does not have to be guaranteed.

Associated with deadlines is the utility function $u(\tau)$, describing the usability of the system when a deadline has been missed. Depending on the degree, the utility function can drop to 0 or $-\infty$ when one or more deadlines have been missed (Figure 2.7).

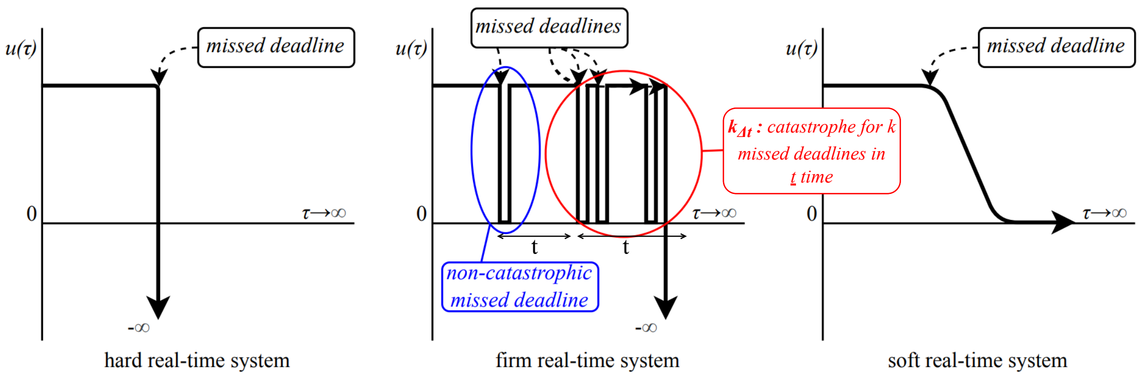


Figure 2.7: The utility function $u(\tau)$ for hard, firm and soft real-time systems. Adapted from Boode (2018).

3 Analysis

3.1 Introduction

Goal 1 is to realise a performant demonstrator for the Production Cell using one or more Raspberry Pi boards. The design aspects that are involved in which embedded-control-software architecture to implement are the following:

- **Number of Raspberry Pi boards.** One can choose which Raspberry Pi board controls which part of the Production-Cell setup.
- **Communication method between Raspberry Pi boards.** One can choose over which available communication protocols of the Raspberry Pi to communicate with other Raspberry Pi boards.
- **Core partition of a Raspberry Pi board over Linux and Xenomai.** Within each Raspberry Pi one can choose the partition of soft-real-time and firm-real-time cores.
- **Mapping of components to a Raspberry Pi board's cores.** Within each Raspberry Pi one can choose which processes and components to run on which core.

Since the core partition and mapping of components is dependent on the number of boards used, the number of boards is preselected.

On the other hand, the main functional requirements of the embedded-control-software-architecture realisation are the following:

- It must adhere to **required timing**.
- It must have appropiate **load balancing**.

Further requirements of the realisation are described in this chapter.

Lastly, a summary of the chosen embedded-control-software architecture is given. Related to this, additional information is given about the design approach that is used for the realisation of the embedded-control-software architecture.

3.2 Requirements

3.2.1 Use cases

A working demonstrator is the use case for goal 1. The working demonstrator will be such that spectators see that the demonstrator performs its operations effectively. Safety measures should also be incorporated for the demonstrator to avoid potential harm to spectators.

For goal 2 the system needs to be analysed while it is running. For this, a test bed should be used. The test bed must be able to test the embedded-control-software architecture and to show whether the requirements are met or not. This implies that the implemented embedded-control-software architecture must allow enough headroom to run software-based instrumentation, without significantly influencing the system's performance. Therefore, an implementation that distributes CPU load well and does not fully utilise the Raspberry Pi's cores is necessary.

3.2.2 Functional requirements

1. **Architectural requirements**
   (a) *The embedded-control-software architecture **must** incorporate the layerered controller structure*
   Goal 1 is realise a performant demonstrator by means of a well-thought-out layered controller structure (Section 2.4.2).

   (b) *The embedded-control-software architecture **must** make use of both the Linux and Xenomai kernel*

To uphold the timing partition of the layered controller structure, it must be the case that at least both kernel types are used for the embedded-control-software architecture, since both are required to map the controllers to their required degree of real-timeness.

(c) *The embedded-control-software architecture **must** at least use ROS2 and 20-sim for its components*
One of the project constraints is that the robot-software framework must be used, therefore ROS2 and 20-sim must be utilised for the embedded-control-software architecture. Though, other tools which can integrate with either tool to ease the development process of components are useful (Table C.1).

(d) *The embedded-control-software architecture **must** take the existing bridge into account for its design*
While not formally a controller layer, the bridge of the robot-software framework is considered as a layer in this thesis. Since the bridge is necessary for the embedded-control-software architecture to function, it must therefore be taken into account in the architecture deisgn.

2. **Communication requirements**
   (a) *Used communication protocols **must** be supported by the Raspberry Pi*
   It is possible that software components on a Raspberry Pi board need to communicate with components running on another Raspberry Pi board, in case multiple Raspberry Pi boards are used. To allow communication between components and boards, a communication protocol must be used which is supported by the Raspberry Pi.

   (b) *Used communication protocols **must** at least be soft-real-time capable*
   It must be ensured that communication between the units of the Production Cell happens with a certain degree of real-timeness, such to avoid system malfunctions due to desynchronization issues.

3. **Performance requirements**
   (a) *The system **must** be able to balance CPU load well*
   Since multiple processes have to run on a Raspberry Pi board, the system must be able to balance these processes on the processor well, such to avoid quality-of-service degradation due to a processor working at maximum capacity. For a real system the CPU utilization should range from 40% to 90% (Silberschatz et al., 2014).

   (b) *The system **should** perform equal or better than previous work*
   It has been reported by Ridder (2018) that the round-trip time of one block is approximately 8 seconds in prior Production-Cell-related work. On the other hand, in the work of Maljaars (2006) it is said that the steady-state error of the units should not exceed 0.5 mm. To evaluate the performance of the realised demonstrator in this thesis, the same metrics will be used.

   (c) *The system **will not** solve deadlock situations*
   Similar to the existing demonstrator for the Production Cell, the realised demonstrator in this thesis will not solve a deadlock situation. An added benefit of this is that the concept of deadlock can be visually demonstrated to spectators.

4. **Safety requirements**
   (a) *The system **must** include a emergency button to allow proper outside intervention*
   In prior work an emergency stop button has not been included in the design of

the demonstrator. For safety purposes an emergency button must be taken into account for the design of the demonstrator in this thesis.

(b) *The system **should** keep the metal blocks inside its boundaries at all times*
The units of the Production Cell can perform their motions rapidly, with the possibility of metal blocks leaving the Production Cell's intended perimeters. To avoid metal blocks being flung towards spectators, the demonstrator should be made such to avoid these situations from happening.

5. **Testing requirements**
   (a) *The test bed **must** analyse the performance and real-time capability of the system*
   Goal 2 is to characterise the embedded-control-software architecture, therefore the test bed must be able to measure the variables of interest.

   (b) *The used instrumentation **should** have minimal influence on the system*
   Running software-based instrumentation alongside the embedded-control-software-architecture components and/or processes should not significantly affect the system's performance. The instrumentation should take only a small share of the CPU load, such to avoid degradation of the system's quality of service.

3.2.3 Non-functional requirements

1. *The existing demonstrator **must** not be affected*
   An existing demonstrator using the FPGA is present (Figure 1.2). To keep this demonstrator running, no significant changes must be made to the setup (e.g. cabling).

2. *The base Production Cell demonstrator **must** be realised*
   Goal 1 is to realise the base Production-Cell demonstrator using the Raspberry Pi. Here, base refers to the Production-Cell setup containing the six units and not more.

3. *The demonstrator **will not** be extended with image processing and/or sorting functionality*
   In prior Production-Cell-related work, work has been done on including image processing and block-sorting functionality in the Production-Cell setup. In this work, additional units have been made for and have been incorporated into the Production Cell. These will not be taken into consideration for the demonstrator in this thesis, mainly due to project-time constraints.

3.3 Approach to allocation

In the following sections the different design aspects are discussed which comprise the constituents of an embedded-control-software-architecture alternative. Using a design space exploration for these design aspects, the best alternative is chosen and implemented in the final design. For an overview of the used design-space-exploration scoring system, see Appendix E.

First, a design space exploration is done for the selection of the number of Raspberry Pi boards. Connected to this, a design space exploration is also done for the interboard communication method. By means of a combined design space exploration, the best alternative setup is chosen with respect to the number of Raspberry Pi boards and the interboard communication method. After this, the allocation of Production-Cell Units to the chosen number of Raspberry Pi boards is discussed and chosen.

Subsequently, a design space exploration for the allocation of controllers to Raspberry Pi cores is done. The best alternative is chosen with respect to the partition of cores over Linux and Xenomai and the mapping of controllers to these cores.

Lastly, a design space exploration is done for the allocation of a safety circuit to Raspberry Pi boards and the best alternative is chosen.

3.4 Selection of number of Raspberry Pi boards

The selection criteria used in Table 3.1, Table 3.2 and Table 3.3 are discussed in this section. For the described alternatives in this section many configurations exist. However, some configurations are unsuitable due to set requirements and/or existing constraints. These are discussed in the following subsections.

3.4.1 Number of Raspberry Pi boards

Table 3.1: Design space exploration for the number of Raspberry Pi boards

		Alternative: *Number of boards*			
Criterion	Weight	One	Two	Three	Six
Development time	2	++	+	+/-	−−
Load distribution	2	−−	+/-	+	++
IO coverage	1	−−	-	+/-	+
Total	**5**	-2	1	**2**	1

The following selection criteria are related to Table 3.1.

Development time for number of boards. With some options a long development time is associated. One example is the case for using multiple Raspberry Pi boards. While the load can be distributed better when opting to use more boards, the development time will increase, since it will take more time to deploy software to multiple target systems. Due to the project-time constraints, a high importance is attached to the development-time criterion.

Load distribution. An ideal number is where the Production-Cell Units are divided evenly over the Raspberry Pi boards. In this case, either 1, 2, 3, or 6 boards should be used. By doing so, the processes associated with the units are evenly distributed as well. Based on goal 1, a high importance is attached to the load-distribution criterion.

IO coverage. Considering the Raspberry Pi boards need to cover the required inputs and outputs of the Production Cell's peripherals (Figure A.4), it is of importance that the allocated Raspberry Pi boards have enough connections to do so. To achieve a minimal fit for the peripherals with respect to the boards, at least three Raspberry Pi boards need to be used (Figure B.2). Some modifications or workarounds still require to be made to achieve the minimal fit, which consists of abusing the encoder inputs as digital inputs. When fewer boards than three boards are used, to increase IO capability it will be required to solder more PMOD-connectors to the icoBoard and/or to modify the robot-software framework significanty. While the IO-coverage criterion is important, it does not directly contribute to goal 1, so less importance is attached to it.

3.4.2 Interboard communication

Table 3.2: Design space exploration for interboard communication

		Alternative: *Communication*		
Criterion	Weight	In software	Pins	Ethernet
Development time	2	+/-	-	+
Real-timeness	1	+	+	+/-
Total	**3**	1	-1	**2**

The following selection criteria are related to Table 3.2.

Development time for communication. If only one board is used, communication between units can take place on the board itself via code-based linkage. In case two or more boards are used, there are two options to consider for interboard communication. The first option is to allow communication to go over the PMOD-connectors via pin cables. The second option is to allow communication to go over Ethernet cables, since the Raspberry Pi has an Ethernet-port. Other options such as USB-based communication are not considered, since this type of connection is based on a master/slave communications protocol.

Since Ethernet-based communication is supported by ROS2 out of the box, it is favoured over other communication methods from the development-time point of view. For pin-based communication, the interface on the icoBoard has to be modified significantly such to fit the needs of the communication. This, however, will require significant development time to set up. This also holds for communication via code-based linkage, which is referred to as in-software communication in this context. In this case, a custom and well-thought-out communication structure must be made between the units, which will take more development time than the out-of-the-box solution for Ethernet. Due to the project-time constraints, a high importance is attached to the development-time criterion.

Real-timeness. One of the requirements is that communication between units needs to be at least soft-real-time. While Ethernet-based communication is not as real-time capable as pin-based communication, it is considered to be effective for soft-real-time communication (Kweon and Cho, 2004). When using Ethernet-based communication, an Ethernet-switch can be used such that the Raspberry Pi boards are on the same network. Regarding in-software communication, it is considered to be more soft-real-time capable than Ethernet-based communication. With respect to development time, less importance is attached to the real-timeness criterion.

3.4.3 Conclusion

A combined design-space-exploration table of the design space explorations done in Section 3.4.1 and Section 3.4.2 is shown in Table 3.3. The design space explorations show that a setup with three Raspberry Pi boards and Ethernet-based interboard communication is the best alternative.

Table 3.3: Combined design-space-exploration table of Table 3.1 and Table 3.2

	Alternative						
Criterion	Setup 1	Setup 2	Setup 3	Setup 4	Setup 5	Setup 6	Setup 7
Number of boards	One: -2	Two: 1	Two: 1	Three: 2	Three: 2	Six: 1	Six: 1
Communication	Software: 1	Pins: -1	Ethernet: 2	Pins: -1	Ethernet: 2	Pins: -1	Ethernet: 2
Total	-1	0	3	1	**4**	0	3

3.5 Allocation of Production-Cell Units to Raspberry Pi boards

The electrical switch boards of the Production Cell have an existing partition of the Production-Cell Units (Figure A.4). The switch boards divide the Production-Cell Units into three pairs, where each pair is associated with a switch board. In Section 3.4, three Raspberry Pi boards are chosen for the embedded-control-software architecture. The Production-Cell-Unit partition of the switch boards is utilised for the Raspberry Pi boards as well. Therefore, the following mappings of Production-Cell Units to Raspberry Pi boards are done:

- **Board 1:** *Extraction Belt, Rotation Robot*
- **Board 2:** *Feeder Belt, Feeder Robot*
- **Board 3:** *Molder Door, Extraction Robot*

The control stack of each Production-Cell Unit, consisting of the controllers of the layered controller structure, is run on their assigned board. A reliable cable interface between the Raspberry Pi board and the Production Cell is necessary, which should also include probing options for debugging purposes.

3.6 Allocation of controllers to Raspberry Pi cores

There are many alternatives to consider when mapping controllers to cores. For the purpose of constraining the number of alternatives, requirements are set for an alternative to be viable (Table 3.4).

Table 3.4: Set requirements for an embedded-control-software architecture to be viable

Requirement	Description
If multiple layers are mapped as a single component onto a core, then they must be adjacent to each other	Mapped controller layers must be layers that are computed subsequently without any layers in between. By doing so, the information flow of the layered controller structure is upheld. For example, the loop controller should not be mapped together with the measurement-and-actuation layer if the safety layer is not mapped as well, since the loop controller and measurement-and-actuation layer are not directly adjacent to each other.
A computationally-expensive user-interface layer must not be run on a board	A user-interface layer with high overhead must not be run on the already resource-constrained boards. Instead, these processes should be off-loaded to a system which has more resources available, if possible. A minimal user-interface layer must still be considered for the embedded-control-software architecture. It is assumed that, due to its low computational cost for the board, the minimal user interface does not warrant seperate resources to be reserved for it.
The measurement-and-actuation layer is always coupled with the safety layer	It is assumed that the measurement-and-actuation layer does not warrant its own seperate component and/or core. With respect to the other layers, it is assumed to be less computationally expensive and to have less overhead. For the sake of structure simplification, the measurement-and-actuation layer always accompanies the safety layer.
A single core is always reserved for the bridge, unless it is not possible	Since the architecture is heavily dependent on communication taking place between Linux and Xenomai, it is warranted to reserve seperate resources for the bridge to avoid a bottleneck situation.
The supervisory, sequence and loop controller can be split, provided more than one core is available for them	These controllers run relatively computational-heavy processes. Therefore, one should respectively consider to split these controllers over multiple cores, if possible. For example, in case two Production-Cell Units reside on a board, then both their loop controllers can be mapped onto their own core, provided that there are two Xenomai cores available. Likewise, this proposition is applicable for the supervisory plus sequence controller, provided that there are two Linux cores available.

A visual representation of the alternatives adhering to the requirements in Table 3.4 is shown in Table 3.5. The selection criteria used in Table 3.6 are discussed further in this section.

Table 3.5: Different configurations to allocate controller layers to Raspberry Pi cores

	Cores			
Alternative	**Core 0**	**Core 1**	**Core 2**	**Core 3**
A	🟧 (2x) Supervisory •	🟧 (2x) Sequence •	🟧 Bridge •	⬜ (2x) Loop (2x) Safety (2x) Meas. & Act.
B	🟧 (1x) Supervisory (1x) Sequence	🟧 (1x) Supervisory (1x) Sequence	🟧 Bridge •	⬜ (2x) Loop (2x) Safety (2x) Meas. & Act.
C	🟧 (2x) Supervisory (2x) Sequence	🟧 Bridge •	⬜ (2x) Loop •	⬜ (2x) Safety (2x) Meas. & Act.
D	🟧 (2x) Supervisory (2x) Sequence Bridge	⬜ (1x) Loop	⬜ (1x) Loop	⬜ (2x) Safety (2x) Meas. & Act.

🟧	=	**Linux core**
⬜	=	**Xenomai core**
•	=	**Functional-to-implementation mapping**

Table 3.6: Design space exploration for the controller-to-core allocations of Table 3.5

		Alternative			
Criterion	Weight	A	B	C	D
Quality of service	2	-	+	+/-	-
Functional-to-implementation mapping	1	+	-	+/-	−−
Total	**3**	-1	**1**	0	-4

The following selection criteria are related to Table 3.6.

Quality of service. To support the execution point of view, it is beneficial to map multiple controllers onto cores. By doing so, communication time between controllers is improved, such that the quality of service of the control stack is improved. Another consideration to improve the quality of service, is to allocate more Linux cores than Xenomai cores. Since Linux has to schedule its own processes as well, it is beneficial to leave headroom for Linux by means of using more Linux cores. Based on goal 1, a high importance is attached to the quality-of-service criterion.

Functional-to-implementation mapping. To support the modelling point of view, it is beneficial to do mappings from the functional design to the implementation design. In this context, the best functional-to-implementation mapping is achieved when mapping one controller to one core, such that each core shares only one computing responsibility. While it is beneficial to do functional-to-implementation mappings, it does not directly contribute to goal 1, so less importance is attached to the functional-to-implementation-mapping criterion.

3.7 Allocation of safety circuit to Raspberry Pi boards

The set safety requirement in Section 3.2.2 is that an emergency button must be included in the embedded-control-software-architecture design. There are different ways to propagate the emergency signal to the Raspberry Pi boards, which can be either done through Ethernet-based or pin-based communication. Safety must adhere to hard-real-time constraints, therefore this leaves only pin-based communication to be an option, as it is closest to the hardware. A design space exploration is done for the allocation of a safety-circuit on the Raspberry Pi boards (Table 3.7).

Table 3.7: Design space exploration for safety-circuit allocation to Raspberry Pi boards

		Alternative: *Safety circuit*	
Criterion	Weight	Leader-follower	Daisy-chain
Worst-case prevention	2	+/-	++
Development time	1	++	+/-
Total	**3**	2	**4**

The following selection criteria are related to Table 3.7.

Worst-case prevention. A safety-circuit alternative is that one board is the leader and the other boards are its followers. By doing so, the emergency button only needs to be connected to one Raspberry Pi board. The followers actively listen whether the leader has received an emergency signal or not. Another safety-circuit alternative is to daisy chain a safety circuit through each Raspberry Pi board. The emergency button when pressed will then propagate its signal to every board at once. If a disconnect occurs in the daisy-chained safety circuit, the circuit will indicate that the emergency signal is present.

The leader-follower alternative has a worse worst-case scenario than the daisy-chain alternative. A situation could occur that the leader does not respond to the emergency button. This results in the emergency signal not being propagated across the boards. A high importance is attached to the worst-case-prevention criterion, since safety is a high priority of the demonstrator.

Development time for the safety circuit. The leader-follower alternative requires less development time than the daisy-chain alternative. For the daisy-chain alternative an external safety circuit must be made, since the current configuration of the Raspberry Pi boards does not support this structure. With respect to safety, less importance is attached to the development-time criterion.

3.8 Conclusion

Concluding, three Raspberry Pi boards are being used together with Ethernet as the interboard communication medium. The Production-Cell Units communicate with each other by making use of ROS2's networking capabilities.

To each Raspberry Pi board two Production-Cell Units are mapped. The mapping that is used is in line with the existing Production-Cell-Unit partition of the switch boards (Figure A.4).

A safety circuit is created for the setup by daisy chaining an emergency-signal line through the Raspberry Pi boards. An emergency button is connected to the safety circuit, propagating its signal to each Raspberry Pi board when pressed.

Alternative B of Table 3.5 is chosen for the controller-to-core allocation. A three-to-one core partition is used for each Raspberry Pi board. Three Linux-cores are dedicated to discrete-event control, while one Xenomai-core is dedicated to motion control.

In the Linux domain, two supervisory controllers, two sequence controllers and the bridge are run. Two supervisory-sequence pairs are developed, where each pair is a ROS2 node. Each of these nodes run on a seperate Linux-core. The ROS2-based supervisory-sequence component encapsulates the discrete-event-control logic necessary for one unit. For the bridge, a seperate Linux-core is reserved as well.

In the Xenomai domain, the loop controller, safety layer and measurement-and-actuation layer are split up into two seperate but identical 20-sim models. The same 20-sim model is used for each Production-Cell Unit. All together they are run as one code component, which runs on the sole Xenomai-core. This component is then responsible for the motion control of two Production-Cell Units.

To follow the workflow of the model-driven design approach, first the motion controller has been designed in 20-sim (Figure 3.1). Here, the necessary verification steps have been done for the motion controller. After this, the discrete-event-control part of the embedded-control-software architecture has been created using ROS2 nodes. Likewise, verification steps have been done for the discrete-event-control components as well.

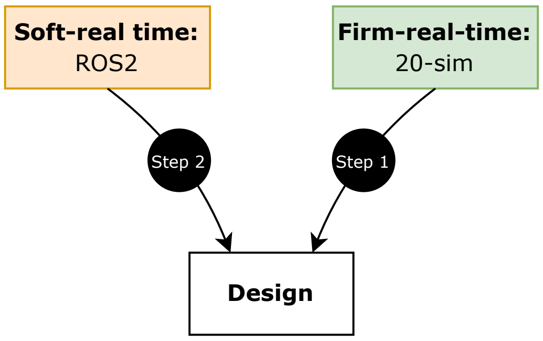


Figure 3.1: The sequential ordering that is followed in the design of the embedded-control-software-architecture design, which is based on the model-driven design approach shown in Figure 2.5.

4 Design

4.1 Introduction

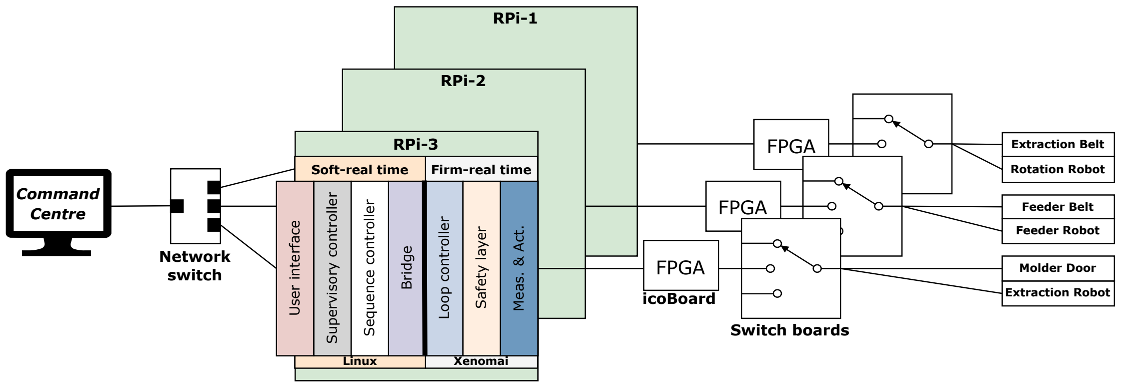


Figure 4.1: Overview of the implemented embedded-control-software architecture

This chapter contains a description of the firm-real-time and soft-real-time design of the implemented embedded-control-software architecture (Figure 4.1). First, the design is explained on a functional level, after which details are highlighted in the design at the implementation level. Lastly, a brief description is given of the designed PCB interface board between the Raspberry Pi board and Production Cell.

For the physical connections associated with the implementation, see Figure C.9. For the design of the implemented safety circuit, see Section D.2.

4.2 Firm-real-time design

The motion controller for each Production-Cell Unit is modelled in 20-sim. Each motion controller contains a loop controller, safety layer and measurement-and-actuation layer (Figure 4.2).

Since two Production-Cell Units reside on one board, two motion controllers are modelled in 20-sim. The structure of this 20-sim model is reused for the other two boards as well. This model is code generated using 20-sim's Code Generation toolbox and mapped as a task to the sole Xenomai core (Figure 4.2). In the work of Van den Berg (2006), it is stated that the loop controller should be executed in real-time with a frequency in-between 100 Hz and 5 kHz. Based on this criterion, it is chosen to run the motion controller with a period of 1 ms, which is equal to 1 kHz. More importantly however is that by choosing this period, jitter of the motion controller can be compared with the jitter results of Meijer (2021). In the work of Meijer (2021) the same period of 1 ms is used for the jitter experiments. To ensure the motion-controller task runs as real-time as possible, a priority of 99 is granted to the task, which is the highest priority that can be given for a task.

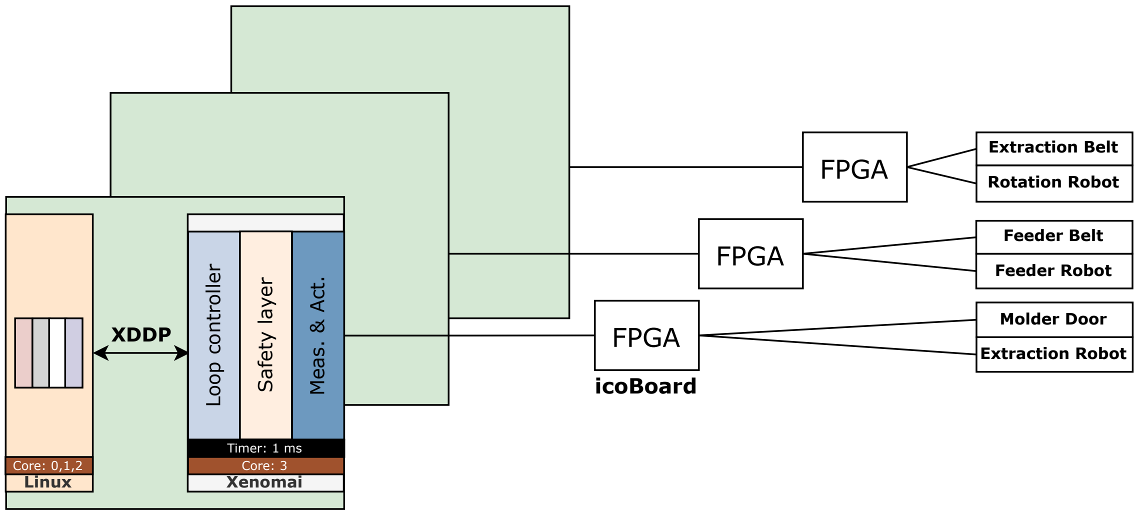


Figure 4.2: Overview of the firm-real-time design. For simplicity the switch board is omitted.

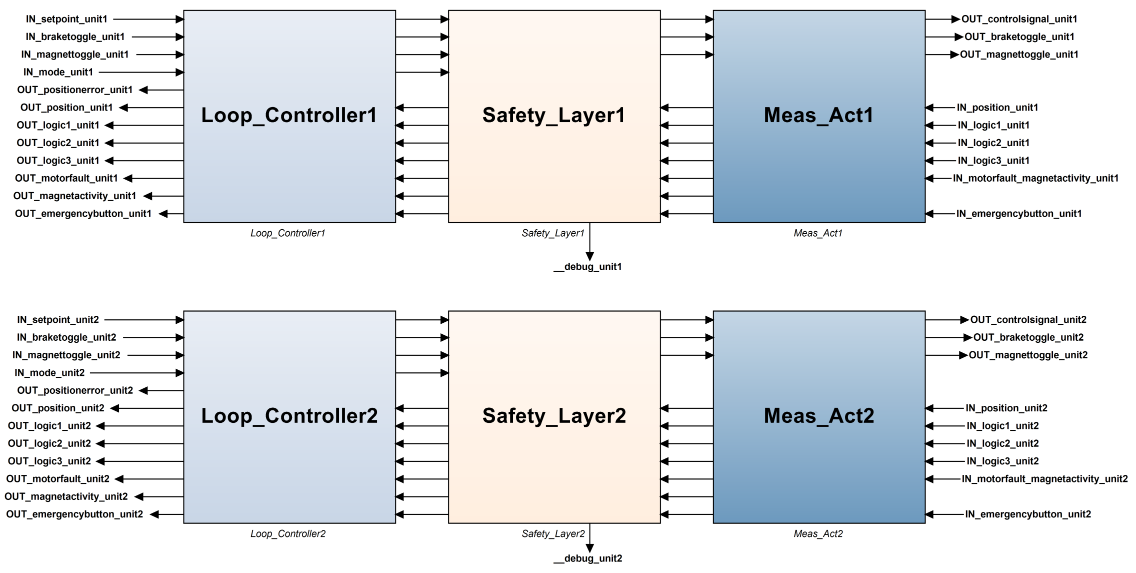


Figure 4.3: The motion controllers modelled in 20-sim for a single board. For an overview of the model blocks see **Loop Controller**: Figure C.1; **Safety Layer**: Figure C.2; **Meas**urement & **Act**uation: Figure C.5.

The loop-controller block is responsible for calculating the correct control signal. A custom 20-sim PD-controller model block is used, since controllers in 20-sim's standard library did not fulfil controller-parameter-tuning requirements. These requirements are that the controllers are tunable with the system-identification data of the work of Ridder (2018). In the work of Ridder (2018), system-identification tests have been done for each Production-Cell Unit. With the parameters extracted from the system-identification data, the controller for each Production-Cell Unit is tuned accordingly.

The safety-layer block consists of an error detector, safety controllers and a decision maker as suggested in Ni (2015). This structure provides a modular approach to safety, where the checked error, safeguarding strategy and safety controllers can be modelled separately from each other. The following design considerations are present for the safety-layer block:

- With the error detector, different types of errors that may be present in the system are checked. Some errors that are checked describe whether the Production-Cell Unit is out-

of-bounds, which is based on physical limits and/or virtual limits. Virtual limits are utilised such to improve the safety of the setup even further. Other errors that are checked are related to the peripherals, such as the magnet, the motors and the emergency button. For the magnet, it is checked whether the magnet is active when it is turned on. For the motors, it is checked whether it experiences an undervoltage situation. For the emergency button, it is checked whether an emergency situation has occurred according to the emergency-button operator.
- To allow the decision maker to execute context-aware safeguarding strategies, a 'mode' signal from the discrete-event-control component of the Production-Cell Unit is relayed to the decision maker. The mode signal describes the current operation state of the Production-Cell Unit.
- When an emergency occurs during normal operations, the decision maker will decide to brake the Production-Cell Unit and keep its magnet active when present. The purpose of keeping the magnets active is to avoid flinging a metal block during the motion of the magnet-based units.

The measurement-and-actuation block is used to filter and scale signals coming from and going to the Production Cell. Other types of operations which do not fall in the category of filtering and scaling are done as well. The following operations are done on signals passing through the measurement-and-actuation block:
- All signals are logically inverted where applicable, since the signals coming from and going to the peripherals of the Production-Cell Units are defined as active-low. By logically inverting these signals, a positive-logic-based environment is created for the embedded-control-software-architecture components which utilise these signals.
- The calculated control signal of the loop controller is scaled into the range accepted by the PWM-module on the icoBoard (range between -2047 and 2047). Control signals which exceed the extrema of this PWM-range are filtered such that these do not exceed the extrema. Braking-mode control (Van den Berg, 2006) is used for the motors, therefore the PWM-range-scaled control signals are mirrored with respect to the PWM-range extrema. For example, when it is required to actuate a motor with a control effort of 10%, a PWM duty cycle of 90% must be sent to the brake input of the motor's H-bridge.
- Due to the limited number of bits reserved for the encoder count on the icoBoard, the encoder count can overflow. This results in reading a discontinuous encoder count for a Production-Cell Unit. To solve this issue, a custom 20-sim model block is used which tracks the encoder-count overflows. The necessary offset for the encoder count is calculated in this model block and added on top of the current encoder-count reading. By doing so, the encoder count is made continuous. This encoder count is then scaled into a meter-based position, which describes the end-effector position of the unit.
- To solve the issue of the limited number of digital inputs for the system, the encoder inputs are abused on the icoBoard. To this end, some of the encoder-modules on the icoBoard have been adjusted. The adjusted encoder-module reads two digital inputs and describes their state with a fixed set of encoder-count numbers. This encoder-count number is then read by the measurement-and-actuation layer. A custom 20-sim model block is used to convert the read encoder-count number into two seperate signals. Each signal describes the state of one of the two digital inputs.
- When the emergency button is pressed it generates a pulse signal for the emergency signal. Due to the pulse being in its active state for only a short period of time, the emergency signal will become inactive shortly afterwards. To save the state of the emergency signal, a custom 20-sim model block is used to detect the edge of the pulse. When an edge-event has been detected by the model block, it keeps outputting an active signal.

Related to the measurement-and-actuation block, the SPI-based interface for a 20-sim-generated code component is attached to the motion-control component. Since the motion-control component is run with a period of 1 ms, values are read from and sent to the FPGA with a period of 1 ms respectively. With the SPI-based interface, values on the FPGA are written to the 20-sim-code-component inputs, while values on the 20-sim-code-component outputs are written to the FPGA.

For in-depth implementation details of the motion controller see Section C.1.

4.3 Soft-real-time design

The discrete-event-control logic for each Production-Cell Unit is run with ROS2 nodes. It consists of a supervisory controller and sequence controller, of which their information flow is propagated through a ROS2-based bridge to the motion controller (Figure 4.4). Using publish-subscribe relations, discrete-event-relevant information is exchanged between ROS2 nodes. In total, there are four ROS2 nodes involved in the discrete-event control of one board.

For the discrete-event-control logic of the two Production-Cell Units residing on one board, two ROS2 nodes are realised (Figure 4.5). One ROS2 node contains the discrete-event-control logic of the first Production-Cell Unit, while the other ROS2 node contains the discrete-event-control logic of the second Production-Cell Unit. This arrangement is used for the other boards as well. For these ROS2 nodes the same code structure is used. The code structure consists of one ROS2 node executing the discrete-event-control model of one Production-Cell Unit. This discrete-event-control model contains the supervisory-controller and sequence-controller functionality. Every tick of the ROS2-node timer, the discrete-event-control model reads the values on subscribed topics, calculates values and puts these values on its outputs. The ROS2 node then publishes the values on these outputs into the ROS2 network (denoted as */write*). Each of the two Production-Cell-Unit ROS2 nodes are mapped to a seperate Linux core.

Two ROS2 nodes are used to establish the information flow between the discrete-event-control component and the motion-control component (Figure 4.5). These two ROS2 nodes represent the bridge. The first bridge ROS2 node is responsible for sending information from the discrete-event-control component to the motion-control component. This ROS2 node sends the information based on ROS2-topic callbacks. When a ROS2-topic callback occurs, values from a Production-Cell-Unit ROS2 node are written to XDDP ports. The second bridge ROS2 node is responsible for sending information from the motion-control component to the discrete-event-control component. This ROS2 node sends the information based on the tick of its ROS2-node timer. When this timer is raised, values from the motion-control component are published into the ROS2 network (denoted as */read*). These values are read by the Production-Cell-Unit ROS2 nodes. The arrangement of the bridge ROS2 nodes is used for the other boards as well. Since two Linux cores are reserved for the Production-Cell-Unit ROS2 nodes, the two bridge ROS2 nodes are mapped to the remaining Linux core.

The handlings among the Production-Cell Units are synchronised by means of exchanging information between their control stacks. To this end, a Production-Cell-Unit ROS2 node is allowed to read the motion-control-component measurement data of its adjacent Production-Cell Units. Furthermore, a Production-Cell-Unit ROS2 node is allowed to read the current state of its adjacent Production-Cell Units as well (denoted as */state*).

A rule of thumb is used for the period of the ROS2 nodes involved in the discrete-event control. All together the ROS2 nodes are run with a period of 10 ms, thus resulting in the motion control running ten times faster than the discrete-event control does.

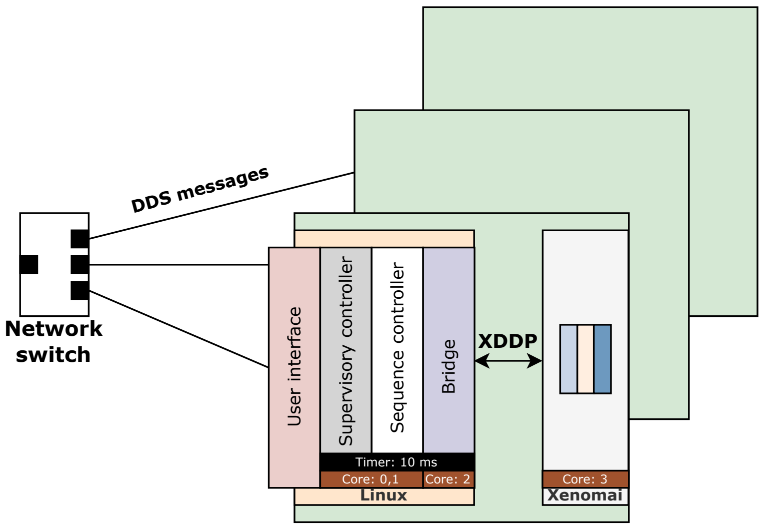


Figure 4.4: Overview of the soft-real-time design. For simplicity the command centre is omitted.

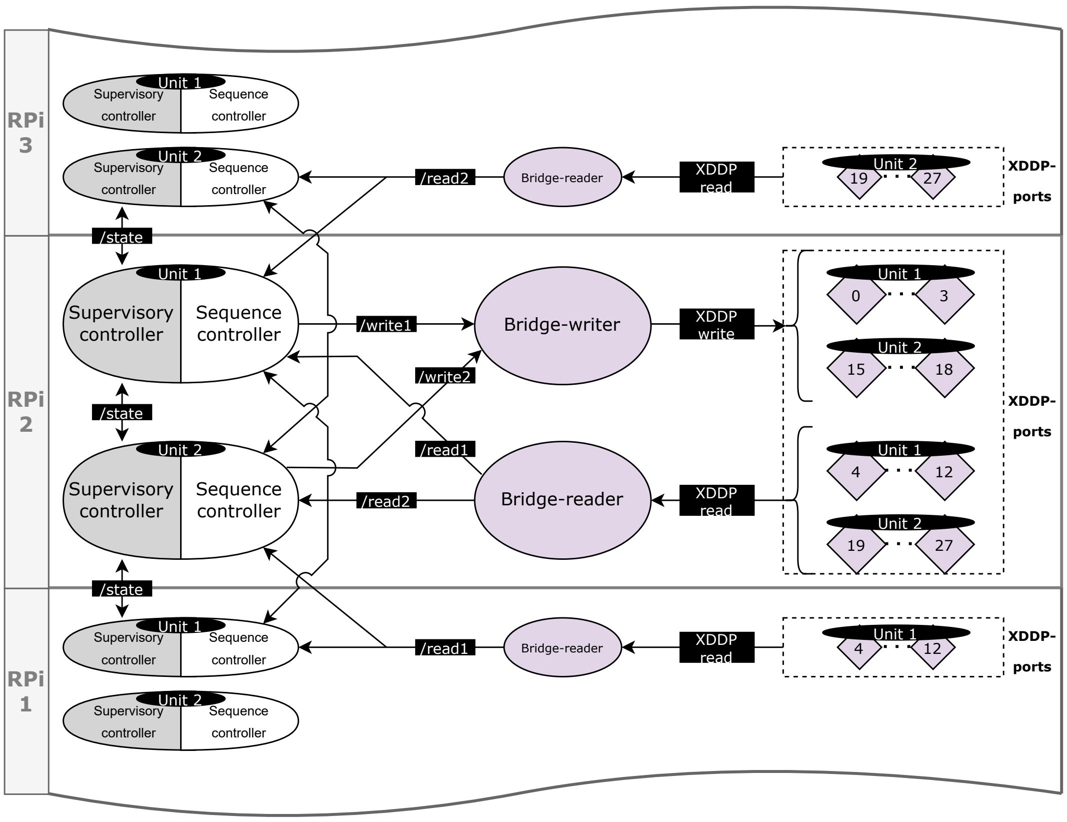


Figure 4.5: The general ROS2-based structure running the discrete-event logic for a single board. For an overview of the models underlying the components see **Supervisory Controller**: Figure C.6; **Sequence Controller**: Figure C.7; **Bridge-reader/-writer**: Figure C.8.

The supervisory-controller model is based on a finite-state-machine model. With each Production-Cell Unit the same high-level finite-state-machine structure is associated, consisting of a *Startup, Idle, Action* and *Shutdown* state. By reusing the same structure, development time has been reduced and the possible superstates per unit is standardized. The *Startup* state contains substates which are responsible for the homing procedures and motion profile generation for a unit. The *Idle* and *Action* state describe the normal operations of a unit when the demonstrator is active. Since the performed handlings by the Production-Cell Units differ, the *Action* state is described by a different set of substates depending on the unit. Lastly, the *Shut-down* state contains substates that brings the unit home to its start position and disables the peripherals for safety purposes.

The sequence-controller model is based on a motion profile generator which creates a setpoint lookup table after homing procedures. The setpoint lookup table ensures that a Production-Cell Unit is bounded to a static range. This range is based on the position of the limit switches of a unit, if these are present for the unit. By utilising a lookup table, setpoints do not require to be calculated online, which reduces the load generated by the sequence controller.

The original bridge nodes that exist in the robot-software framework have been modified. In this case, the modified bridge nodes merge functionality to improve system performance. While the original bridge is based on a single value being propagated by respectively one node, the modified bridge is based on multiple values being propagated by one node instead. For an overview of the differences, see Figure 4.6.

For in-depth implementation details of the discrete-event control see Section C.2.

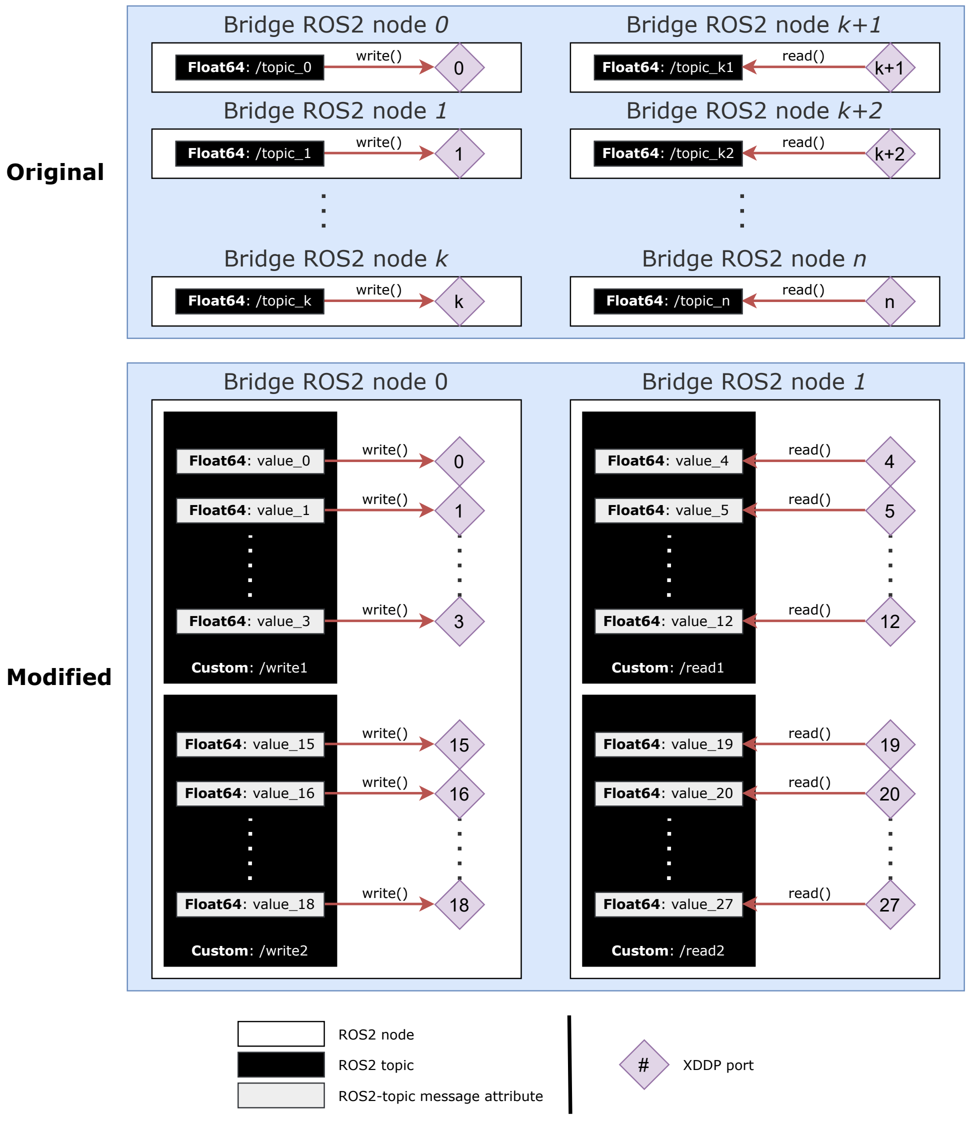


Figure 4.6: The modified bridge nodes with respect to the original bridge nodes

4.4 PCB interface board

Each Raspberry Pi board needs to interface with the 50-pin header of the switch board. To avoid extensive cabling, a PCB board has been developed that maps 50-pin-header signals to the PMOD-connectors of an icoBoard. The PCB design is chosen such that the three needed PCB boards are identical. Furthermore, additional functionality is added to the PCB board, such as probing points for signals coming from and going to the setup.

For information about the design of the PCB interface board, see Section D.1.

5 Testing

5.1 Introduction

Goal 2 is to characterise the control performance, load balancing and real-timeness of the implemented embedded-control-software architecture. This chapter discusses the characterisation. First, a description of the test scenario and the test bed is given. This is followed up by an explanation of the way of measuring the variables of interest. After this, the results obtained from the test scenario are shown in plots. where observations are made per aspect of goal 2. Lastly, both the test bed and the results are discussed.

5.2 Setup

For the test scenario it is desired to capture the variables of interest when the system has nominal load. One can define nominal load as a scenario where only one block makes a full round trip. Alternatively, one can also define nominal load as a scenario where it circulates more than one block, which is more similar to the real world process where many blocks circulate simultaneously. One basic cycle in the Production-Cell setup is achieved with one block. Therefore, the one-block scenario is chosen to represent the nominal load for the system. For the test scenario the block is placed on the *Extraction Belt* and in front of the *Extraction Robot* (Figure 5.1). This location is chosen over other possible locations, since other locations do not allow easy operations regarding the scenario start. Furthermore, this location is one of the closing points in the Production-Cell setup, where the round-trip time of a block can be determined when the *Extraction Robot* finishes its motion.

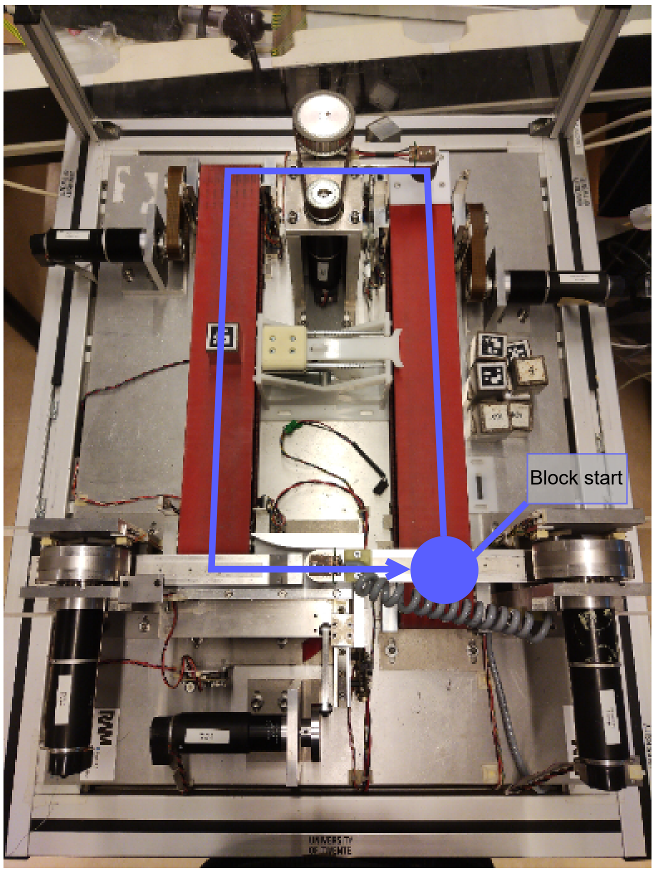


Figure 5.1: The test scenario where one block starts the round trip

To characterise the control performance, load balancing and real-timeness as stated in goal 2, it is also of importance to know the variables of interest associated with these aspects. The variables of interest include the following:

- **Setpoint, position error**: characterise the control performance during a unit's motion.
- **CPU load**: characterises the load balancing with respect to the Raspberry Pi cores.
- **Timing**: characterises the real-timeness of a component's operations.

Regarding the instrumentation for the variables of interest, the following design considerations are present:

- **Setpoint, position error, and timing with existing components.** To capture these variables of interest, it is necessary to build logging capabilities into some existing components of the embedded-control-software architecture (Table 5.1). For the control-performance aspect, the variables of interest are already present in the system for the logger to capture. For the real-timeness aspect, components must send their timing to the logger at hand.
- **Timing of motion-control component.** Since the Xenomai-side runs quicker than the Linux-side, the issue is that timing-related variables in the former are too quick for the latter to capture per sample. To circumvent this issue, one logger has to run in Linux and one logger has to run in Xenomai. For the Linux-side logger, a ROS2-node-based logging component is created which obtains motion-control-component timing variables over XDDP ports. This logger is denoted as *Xenomai logger* (Table 5.1). This ROS2-based logger functions as a bridge which publishes these variables into the ROS2 environment. For the Xenomai-side logger, the variables are stored per sample, which eventually are written to a CSV-file. For the ROS2-based logger, a slower sample time is used with respect to the Xenomai-side logger. With the two-logger approach, the low-resolution data obtained with the ROS2-based logger can be overlaid with the high-resolution data that has been obtained with the Xenomai-side logger (Figure C.10).
- **CPU load of the Raspberry Pi cores.** The CPU-load-related variables are not yet captured by the existing components. To this end, an additional logging component is made. This logging component is a ROS2 node and is denoted as *CPU logger* (Table 5.1).

For the components involved in the logging process, the following design decisions are present:

- **Core assignment for the additional ROS2-based loggers.** Since additional ROS2-based logging components are created, it is necessary to assign on which Linux-core these will run. The problem, however, is that to each Linux-core a component from the embedded-control-software architecture is already mapped. The result is that this embedded-control-sofware-architecture component needs to share a core with the ROS2-based logging components. Thus, one should choose which of the existing ROS2-based embedded-control-software-architecture components are the most crucial. An appropiate decision must be made such to minimise degradation of the quality of service of the system. To this end, it is chosen to avoid running the logging components on the core which run the bridge ROS2 nodes. With respect to the Production-Cell-Unit ROS2 nodes, the bridge ROS2 nodes run more computationally-expensive operations. The reason is the bridge ROS2 nodes utilise many system calls for the data transport. Hence, the logging components are mapped to the cores on which the existing Production-Cell-Unit ROS2 nodes are run. To divide the CPU load generated by the ROS2-based logging components, each of these components are mapped to a seperate core (Table 5.1).
- **Period of the additional ROS2-based loggers.** The period of the additional ROS2-based logging components must be chosen appropiately. Running these quickly will result in a higher resolution for the variables of interest, but possibly results in their associated core becoming overloaded. On the other hand, running these slowly minimises their influence on the CPU load, but might result in the data being not useful due to its low resolution. Different periods have been tested: a 1 ms and 10 ms period would overload

the core to 100%, while a 100 ms period would not. A period of 100 ms is thus chosen for the logging components (Table 5.1). The 100 ms period resulted in the logging-responsible cores not being overloaded to their maximum capacities. Furthermore, the 100 ms period provides a sufficient resolution to see the general trend of the variables of interest.

- **Across-the-board timeline for logged data.** A central point of logging needs to capture the data on the boards with an across-the-board timeline, since the data captured from components are asynchronous with respect to each other. It can be chosen such that one of the boards is the central point of logging, but this results in an asymmetrical load distribution across the boards. Therefore, it is chosen to offload this logging responsibility to another device, which is the command centre. The command centre is the central point of logging, on which a ROS2 node will capture the available data on the boards via subscriptions. The ROS2-based command-centre logger is subscribed to the logging-related ROS2 topics in Table 5.1. When a topic callback occurs in the command-centre logger, the topic values are written to one of the logger's temporary buffers. The topic values are stored in this buffer until the next topic callback writes new values in this buffer. The values in this buffer are therefore updated based on the publishing rate of the topic. While the command-centre logger is running, it stores the buffer values into a tabular data container with a period of 1 ms (Figure C.11). A period of 1 ms is used to detect the change in buffer values as fast as possible. The contents of the tabular data container is printed to a CSV-file when the command-centre logger is stopped. Based on the timeline of the command-centre logger, the data captured from each board is thus respectively synchronised with each other.

Table 5.1: Overview of the components involved in the logging process. **C & C**: Command & Control centre

Component			Runs on	Timer	Variables	Outgoing channel	To component
Unit 1	Supervisory	Sequence	(1x) RPi 1, core 0 (1x) RPi 2, core 0 (1x) RPi 3, core 0	10ms	Setpoint Timing	/write1 /timing1	Logger (C&C) Logger (C&C)
Unit 2	Supervisory	Sequence	(1x) RPi 1, core 1 (1x) RPi 2, core 1 (1x) RPi 3, core 1	10ms	Setpoint Timing	/write2 /timing2	Logger (C&C) Logger (C&C)
	Bridge-reader		(1x) RPi 1, core 2 (1x) RPi 2, core 2 (1x) RPi 3, core 2	10ms	Position error	/read1 /read2	Logger (C&C) Logger (C&C)
Loop	Safety	Meas. & Act.	(1x) RPi 1, core 3 (1x) RPi 2, core 3 (1x) RPi 3, core 3	1ms	Timing	Internal XDDP	CSV-file Xenomai logger
	Xenomai logger		(1x) RPi 1, core 0 (1x) RPi 2, core 0 (1x) RPi 3, core 0	100ms	Timing	/timing_20simtask	Logger (C&C)
	CPU logger		(1x) RPi 1, core 1 (1x) RPi 2, core 1 (1x) RPi 3, core 1	100ms	Processor load	/cpuload	Logger (C&C)
	Logger (C&C)		(1x) C&C cores	1ms	Setpoint Position error Processor load Timing	Internal	CSV-file

5.3 Way of measuring

To obtain the timing in each component, the POSIX-based function `clock_gettime()` is used to capture the cycle-to-cycle time of a component and of certain segments of code. In the function call the `CLOCK_MONOTONIC_RAW` argument is given, which uses a raw hardware-based monotonic time that is not subject to NTP-based clock adjustments (Die.net, 2023).

The CPU load is obtained by parsing device files in the `/proc/` directory of Linux. For reading the CPU load of the Linux cores, the `cpp-linux-system-stats` library on GitHub is used (improvess, 2023). This library allows one to read the current CPU load in Linux by parsing the `/proc/stat` device file. For reading the CPU load of the Xenomai cores, a derivative of this library is made. The library is modified such that it parses the `/proc/xenomai/sched/stat` device file, from which it reads the CPU load of a Xenomai core.

5.4 Results

In this section the data obtained from the test scenario is shown. First, observations are made concerning the performance of the system, which includes both the control performance and the load balancing. After this, observations are made concerning real-timeness-related aspects.

5.4.1 Performance

An overview of performance-related plots can be seen in Figure 5.2. First, observations are made about the controller-performance results. Afterwards, observations are made about the CPU-load results.

Controller performance

The plots concerning the *Extraction Robot* indicate that the block makes one full round trip in approximately 9 s, as it shows that the *Extraction Robot* finishes its movement around this time. Compared to previous work, the round-trip time of a block is slower, which is reported to be around 8 s.

When looking at the steady-state error of the units, units generally do not violate the 0.5 mm steady-state-error criterion of the requirements, except for the *Rotation Robot* and *Feeder Belt*. The results show that the steady-state error of the *Rotation Robot* after transporting a block is 1.0 mm. On the other hand, at the begin phase of the *Feeder Belt* a steady-state error of approximately 2.0 mm is found, which in this context is the steady-state error of its prior-made motion. Looking at the position error when a unit is in motion, the unit which has the best control performance is the *Feeder Belt*, while the unit with the worst control performance is the *Rotation Robot*.

CPU load

Regarding the load balancing per core, none of the cores exceed the 90% processor-load boundary that was described in the requirements. Still, noteworthy is that the core which runs the ROS2 node of the *Rotation Robot* reaches a load of 80%. At this 7.5 s timestamp, a sudden change in CPU load can be seen for the other cores on the board as well. For the Linux cores the CPU load increases, while for the Xenomai core the CPU load decreases, albeit little. When looking at the same timestamp in the position-error plot of the *Extraction Belt*, some points indicate that the position error decreases suddenly, albeit little as well.

Overall, the CPU load per board is comparable over the other boards, where they mainly differ in maxima but not significantly in mean. This applies to both the Linux cores and the Xenomai core. For the Xenomai core, the CPU load ranges from 38% to 41%. For the Linux cores, the CPU load ranges from 25% to 80%.

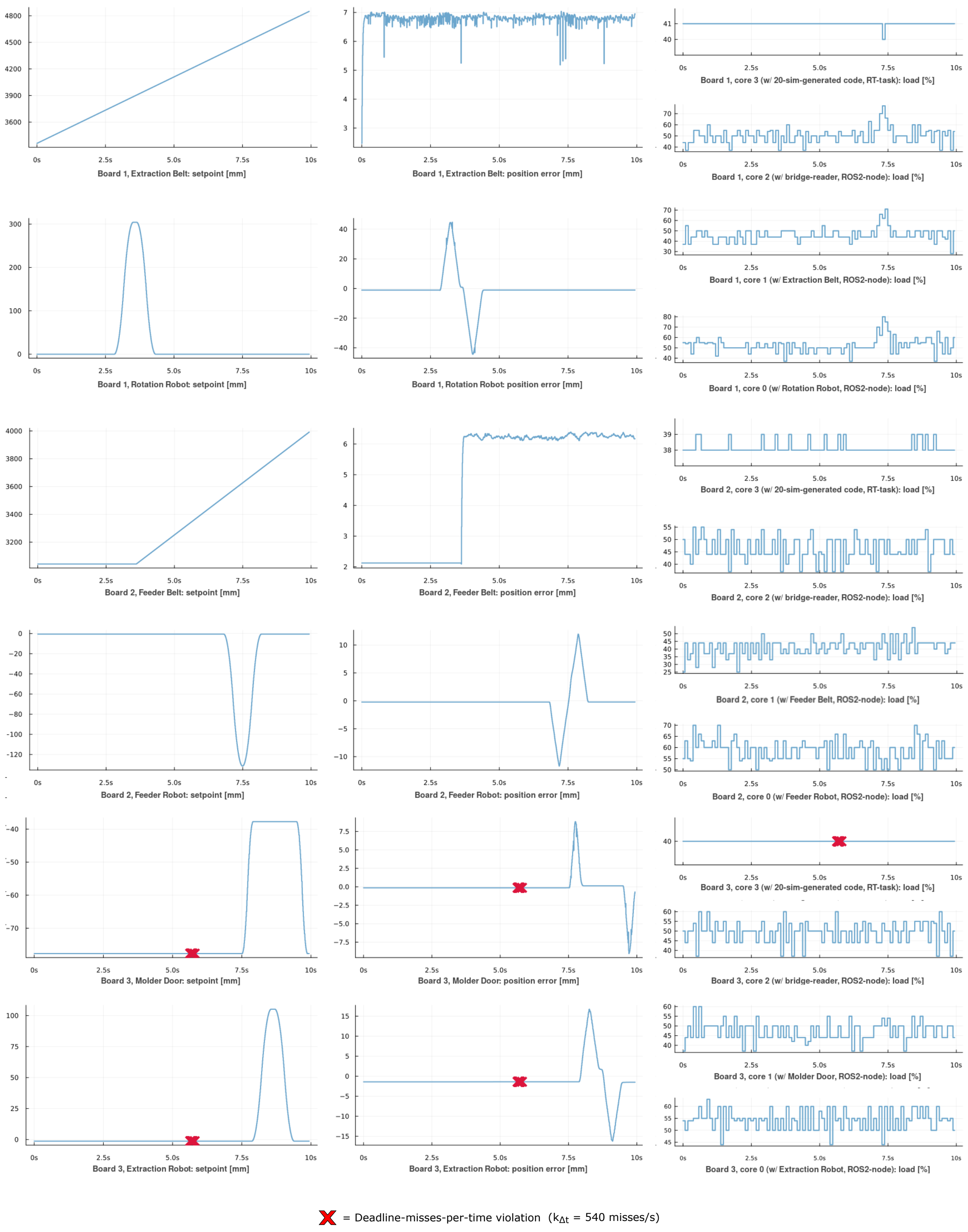


Figure 5.2: Overview of the control performance and load balancing during one round-trip time of a block.

5.4.2 Real-timeness

An overview of the jitter per component in relation to Figure 5.2 is shown in Figure 5.3. A closer look of the timing of the real-time task running 20-sim-generated code is shown in Figure 5.4.

It can be observed that the jitter differs per component (Figure 5.3). Specifically, the magnitude of jitter differs between the ROS2 nodes and the real-time task running the 20-sim-generated code, which is approximately a difference of 10^2 to 10^3.

Regarding the ROS2 nodes, the jitter range varies from 0 ms to 8.7 ms. The maximum of this range indicates that a node is approximately 90% over its intended deadline, since the ROS2 nodes run with a period of 10 ms. Overall, most of the jitter of ROS2 nodes is clustered around the 0 ms mark, where eventually not so much jitter is observed after the 3 ms mark. However, noteworthy are the jitter plots for the *Extraction Belt* and the *Molder Door*, where a significant cluster of jitter can be observed around the 4 ms mark. Also noteworthy is that these plots show higher worst-case jitter than the other ROS2-node-based plots, but these are mostly one-off values.

For the real-time task running the 20-sim-generated code, the jitter range varies from 0 μs to 48.7 μs. This indicates that the real-time task is at maximum approximately 5% over its intended deadline, since the real-time task is run with a period of 1 ms. Noteworthy is that most of the jitter in the higher range are one-off values or close to being one-off values. Instead, most of the jitter is clustered around the 0 μs mark.

The execution cycle of the real-time task running 20-sim-generated code consists of different phases (Figure 5.5). When taking a closer look at the timing concerning these different phases (Figure 5.4), one can observe that a significant part of the cycle is spent on receiving data and sending data. In contrast with these communication-related phases, the actual calculations done by the 20-sim model is relatively a small segment of the cycle. Overall, the job of the real-time task running the 20-sim-generated code takes approximately 0.3 ms to complete, with some outliers reaching towards the 0.5 ms range. This indicates that the job for its worst case requires approximately 50% of the period of 1 ms. These observations are applicable to each board.

Deadline-misses-per-time violations $k_{\Delta t}$ are searched for in the data (Figure 5.2). Using different values for this violation criterion, timestamps are found on which the criterion is violated. These violations are based on the time evolution of deadline misses of the real-time task running the 20-sim-generated code (Figure C.13). The first violations are found when $k_{\Delta t} = 540\ misses/s$, which is shown in Figure 5.2. However, it must be noted that these violations occur while the units are idling, so its implications on the results cannot be observed here. For the purpose of looking for a relation between the violations and the system performance, the violation criterion is adjusted to $k_{\Delta t} = 520\ misses/s$ (Figure C.12). Using this value for the criterion, it can be observed that violations occur during the motion of units. However, no significant decrease in system stability or position error is observed. Making the criterion stricter eventually results in all points violating the criterion, resulting in the plot not being able to provide useful insights. Therefore, stricter values for the criterion are not evaluated for the plots.

A direct relationship between CPU load (Figure 5.2) and deadline misses (Figure C.13) is not visible. As had been described before, a significant CPU load fluctuation can be observed at the 7.5 s mark for the components on board 1. Though, the influence of this fluctuation cannot be observed in the deadline-misses plots of board 1. Overall, no steep increases in deadline misses are observed in the deadline-misses plot of the components.

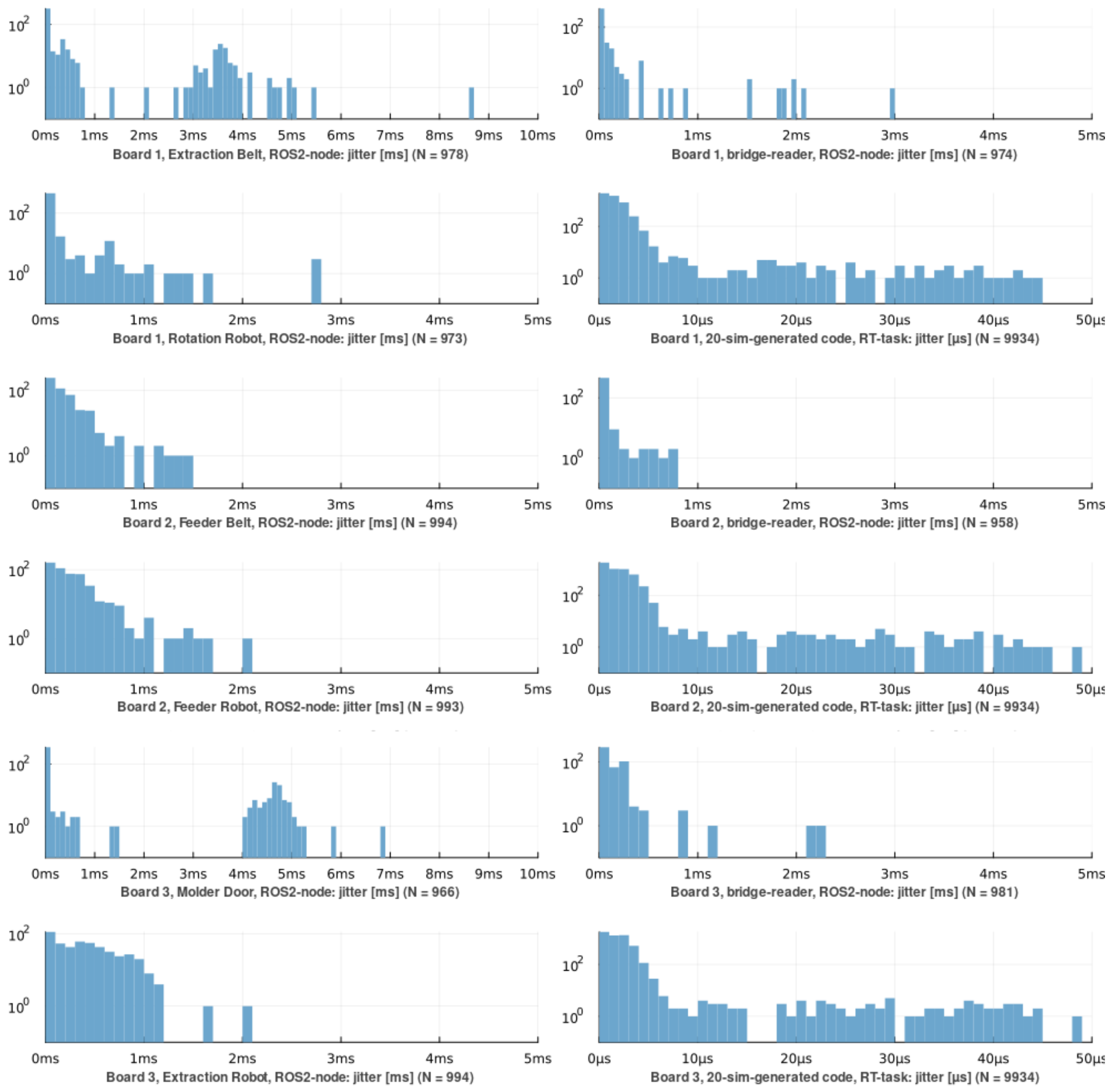


Figure 5.3: Overview of the jitter per component during one round-trip time of a block

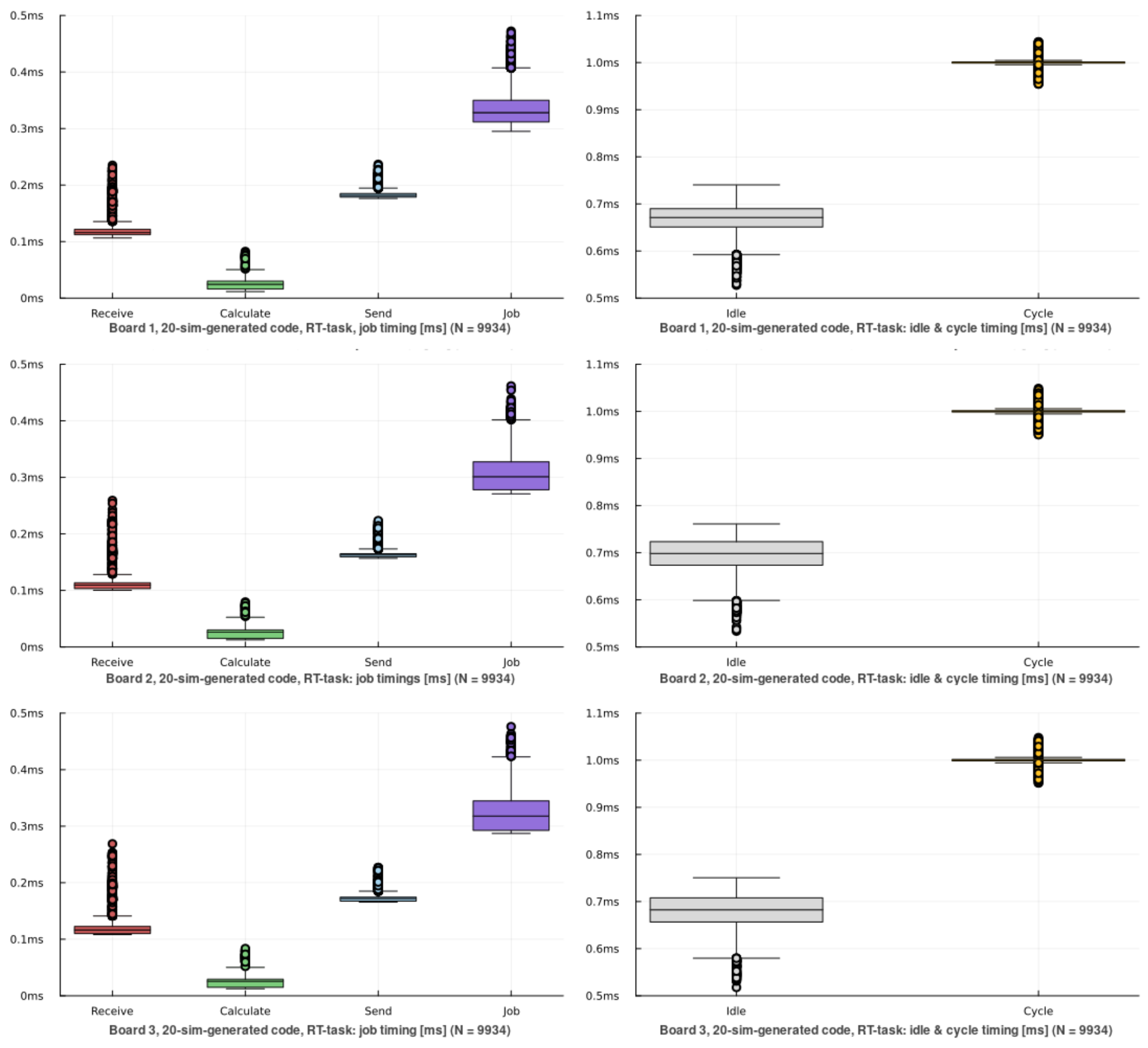


Figure 5.4: Closer look at the timing of the real-time task running the 20-sim-generated code. For an overview of the execution timeline see Figure 5.5.

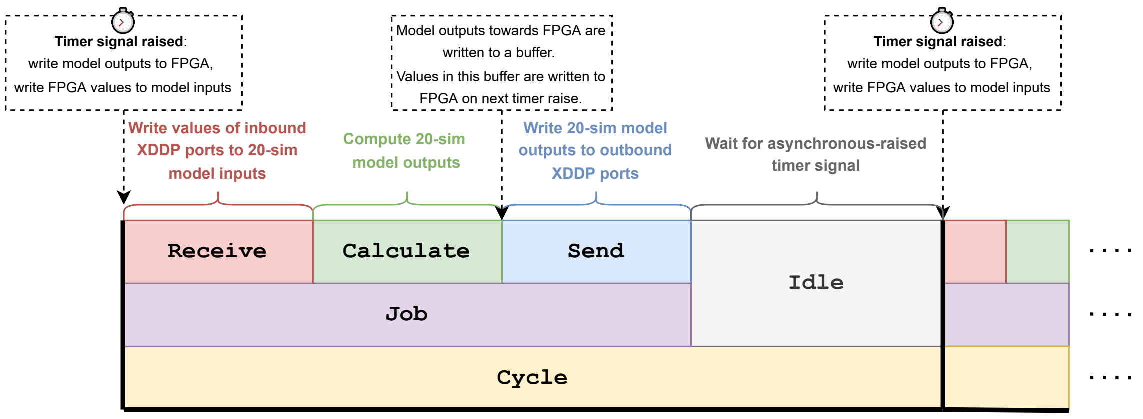

Timer signal raised: write model outputs to FPGA, write FPGA values to model inputs
Model outputs towards FPGA are written to a buffer. Values in this buffer are written to FPGA on next timer raise.
Timer signal raised: write model outputs to FPGA, write FPGA values to model inputs
Write values of inbound XDDP ports to 20-sim model inputs
Compute 20-sim model outputs
Write 20-sim model outputs to outbound XDDP ports
Wait for asynchronous-raised timer signal
Receive
Calculate
Send
Idle
Job
Cycle
....

Figure 5.5: Execution cycle of the task running the 20-sim-generated code. For an overview of the underlying code see Listing C.12.

5.5 Discussion

For ROS2 nodes, it can be argued that they can adhere to soft-real-time constraints most of the time, as most of their jitter is bounded between an acceptable range. In this case, jitter is mostly centered around the 0 s mark. Beyond that, jitter starts to taper off and eventually become one-off values, where jitter is bounded to be 30% of the period at most. However, in the results it also has been observed that ROS2 nodes can have a large jitter, which in the worst-case is approximately 90% of the period. This indicates that ROS2 nodes can potentially have jitter which makes them less soft-real-time-capable than desired.

For the real-time task running 20-sim-generated code in Xenomai, it can be argued that it adheres to its firm-real-time constraints. While jitter can be found away from the 0 μs mark, which can reach up to 5% of its period, most of these jitter values are one-off values. In that regard, the results show that Xenomai-based tasks are always ensured a timely execution time with respect to jitter, unlike the ROS2 nodes. Noteworthy is that the worse-case jitter of the real-time task running 20-sim-generated code is worse than has been found in the work of Meijer (2021). In the work of Meijer (2021) it is reported that jitter is bounded to be 3% of the period at most, but where an one-off value for the jitter has been found which reached into the 4% range. A possible explanation for the differences in results is that the context underlying the results in this work and Meijer's work differs. While in Meijer's results the system load is likely to be relatively low, the system load underlying the results of this thesis is not. It is probable that due to the many processes being run during system testing, a difference in jitter can be observed. This argument is based on the assumption that Xenomai is actually affected by the processes that run in Linux.

Related to the real-time task running the 20-sim-generated code, it was expected that the time needed for its calculations would be a significant section of its period. This expectation was based on the fact that both the model in 20-sim and the size of the resulting code are relatively large. However, the results show that a significant part of the period is spent on communication-related aspects instead of the calculations. This indicates that the code generated from 20-sim can be performant regardless of its size, and that most of the time gains can be achieved by optimizing the communication instead.

In the work of Meijer (2021) a criterion is set for the jitter, where sufficient real-time performance is achieved when jitter is below 3% of the period. This criterion is used as a relaxed measure of what a deadline constitutes, since it differs per application what magnitude of jitter is acceptable. However, in the deadline-misses results of this thesis (Figure C.13) the strictest criterion is adhered to, where a deadline miss is considered anything that is positive jitter. It is likely that this criterion is too strict with respect to the setup. This argument is based on the fact that no significant control performance drops or signs of instability were observed with respect to the deadline-misses-per-time violations. This indicates that while adhering to real-time constraints is important, the non-zero jitter is not critical enough to significantly influence the control in this application. With respect to the results, a statement can therefore be made that jitter up till 5% is acceptable for the motion control of the Production-Cell setup.

To measure the variables of interest, it was chosen to run additional logging components alongside the embedded-control-software-architecture components. However, it is obviously the case that the former have influenced the results of the latter. It was observed that a significant cluster of jitter was located away from the 0 ms mark. This possibly indicates that the embedded-control-software-architecture component shares a significant amount of processor time with the logging component. Furthermore, the worst-case jitter were found for the embedded-control-software-architecture components which run alongside the logging components.

For some units, the control performance did not satisfy the requirement for the steady-state error. A reason for the poor performance is that the controllers for the units are not tuned well enough. Furthermore, for some units the motion profile may have been set to be too aggressive. Due to project-time constraints, further tuning of the system was not done.

6 Conclusions and Recommendations

6.1 Conclusions

The first goal of this thesis is achieved within the project constraints of this thesis. The first goal describes that a Production-Cell demonstrator must be realised by means of a performant embedded-control-software architecture on the Raspberry Pi. Different alternatives for the architecture are evaluated and the best alternative is chosen. Using a real-time robot-software framework and a model-driven-design workflow, a layered controller structure is built based on the chosen embedded-control-software-architecture alternative. The real-time robot-software framework uses 20-sim for motion control and ROS2 for discrete-event control, where both tools are adequate to build the architecture. The Raspberry-Pi-based demonstrator is able to circulate a block, thus being a next addition to the working Production-Cell demonstrators.

The second goal of this thesis is achieved as well. The second goal describes that the realised embedded-control-software architecture must be characterised in terms of performance and real-timeness. This characterisation is shown in Chapter 5. The variables of interest of the performance aspect and real-timeness aspect are logged using a test bed. This test bed is run during a nominal-load test scenario. In this scenario, one block makes one full round trip. For the performance aspect, results are shown of the control performance of each Production-Cell Unit and the load balancing with respect to the Raspberry Pi cores. For the real-timeness aspect, results are shown of the timing and deadline misses of the embedded-control-software-architecture components. According to the processor-load requirements, the results show that the architecture is well-performant in terms of load balancing (< 90% CPU load). According to the control-performance requirements, the results show that in terms of the steady-state error for a Production-Cell Unit, the architecture does not perform as well as prior work (> 0.5 mm). For the nominal-load scenario, the results have shown that worst-case jitter of ROS2-based tasks can be approximately 90% of their period. For the same scenario, is is shown that the worst-case jitter of the real-time task in Xenomai running 20-sim-generated code is approximately 5% of its period at most.

6.2 Recommendations

In future work, a relationship between deadline-misses-per-time violations and control performance should be investigated. With respect to the results of Chapter 5, it is unclear if the deadline-misses-per-time violations can be related to the control performance.

Additionally, in future work stress tests should be done for both the processor-load aspect and the networking aspect of the embedded-control-software architecture. The effects of overloading the Raspberry-Pi processor cores are not investigated in this thesis. Furthermore, the influence of network interference between the boards is not investigated as well. Due to project-time constraints these extra tests were not done.

Furthermore, in future work other embedded-control-software architectures should be explored using newer components. At the moment of writing the Raspberry Pi company has started the roll-out of the Raspberry Pi version 5. Furthermore, the Xenomai project is active with developing the newest version of Xenomai, which is Xenomai EVL (version 4). These components provide the opportunity to create another demonstrator for the Production-Cell setup, which eventually can be compared to the demonstrator of this thesis.

Lastly, in future work the functioning of the safety circuit with the emergency button must be tested. Due to project time constraints, this functioning has not been tested.

A Production Cell

This chapter of the appendix describes different aspects concerning the Production Cell. Section A.1 until Section A.7 serve as documentation for the Production Cell. These sections describe the following:

- The Production-Cell Units and their associated peripherals
- Details of the switch boards and the per-pin definition of its onboard 50-pin header

For more information about the Production-Cell setup see the electronically-handed-in documentation.

Lastly, Section A.8 describes the realised embedded-control-software architectures for the Production Cell in related work.

A.1 Real world picture

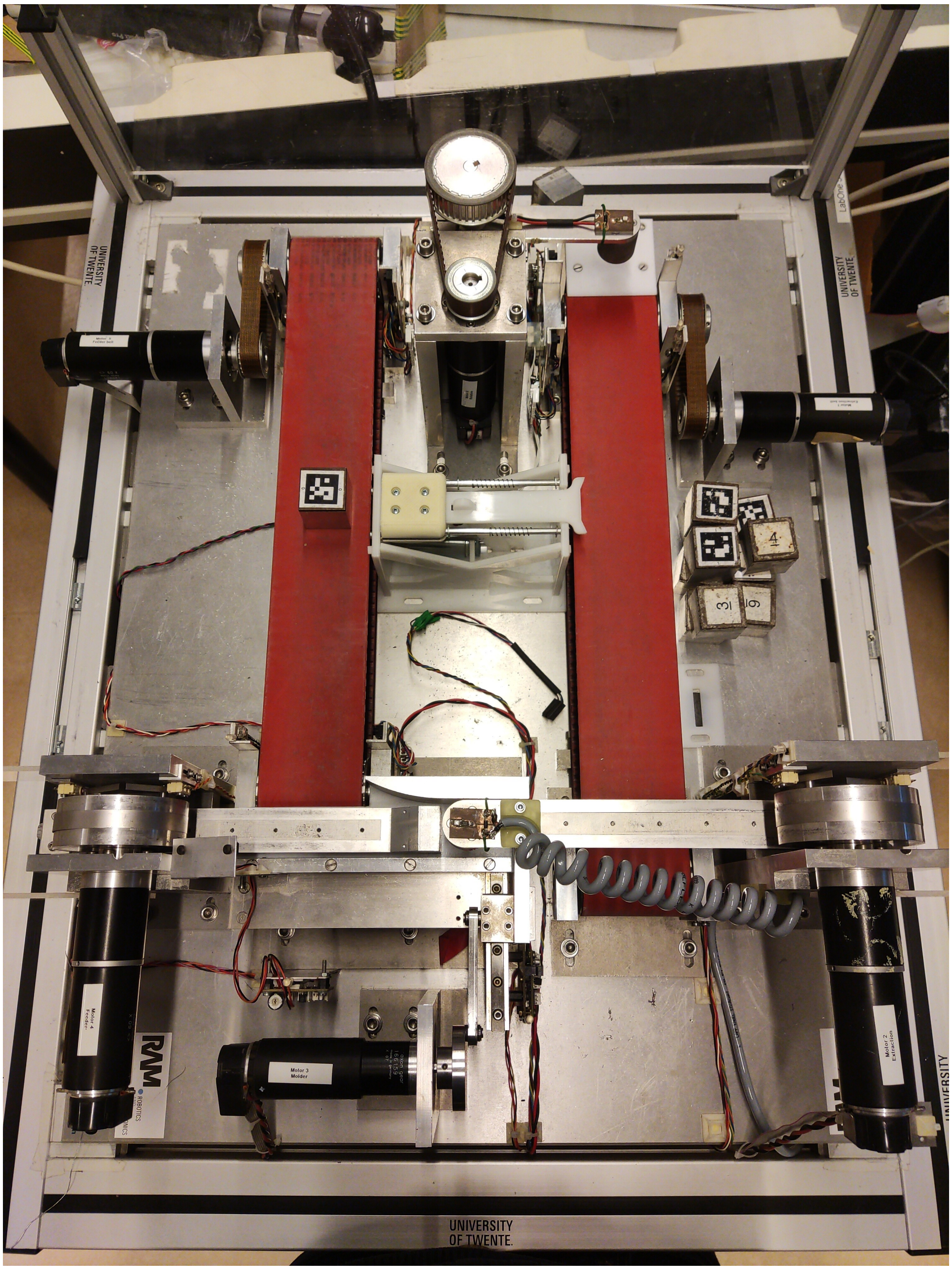


Figure A.1: Top-down view of the Production Cell

A.2 Units

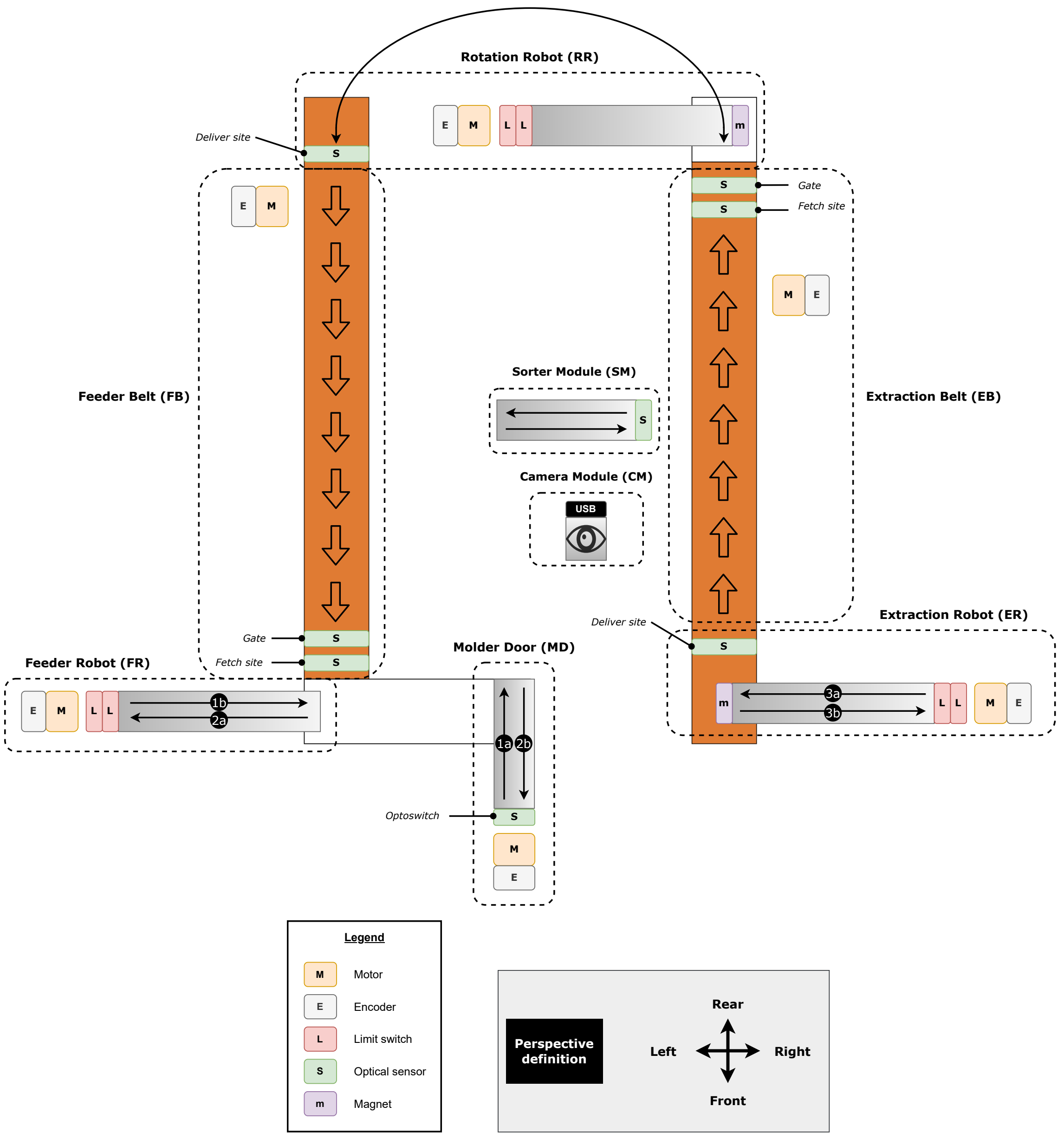


Figure A.2: Functional overview of the units of the Production Cell. The arrows indicate the allowed directions for a unit's motion. The number-letter markings symbolise the sequence of motion handlings at the moulding section.

A.3 Sequence diagram

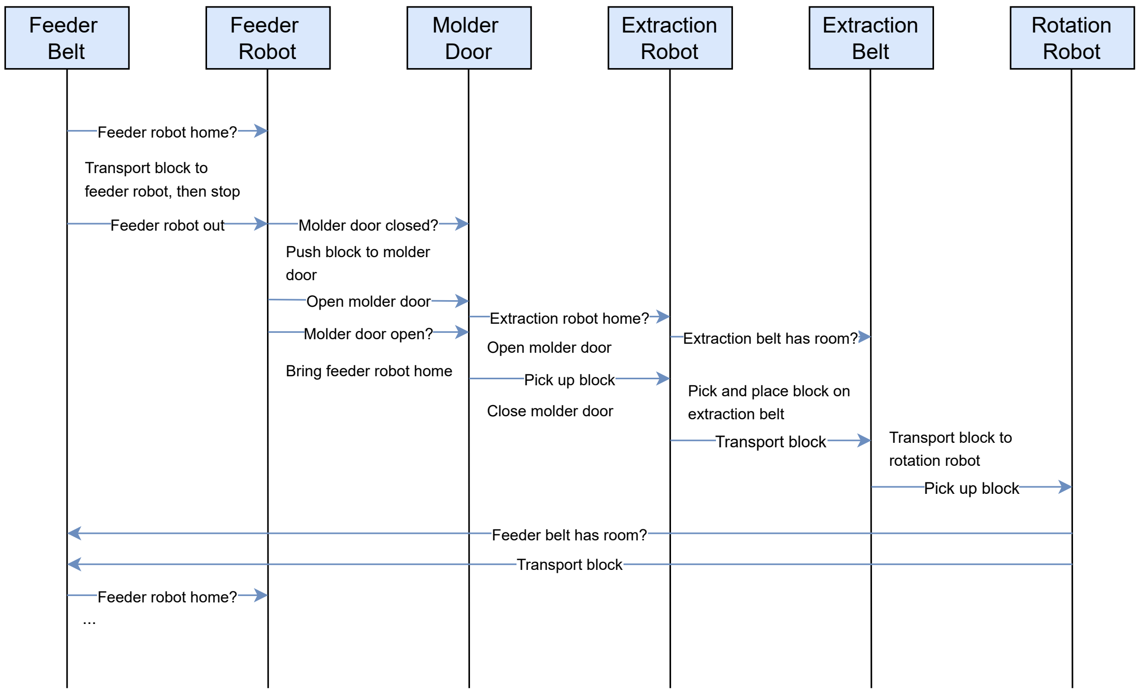


Figure A.3: Normal operation sequence diagram of the Production Cell. Adapted from Groothuis et al. (2008).

A.4 Peripherals

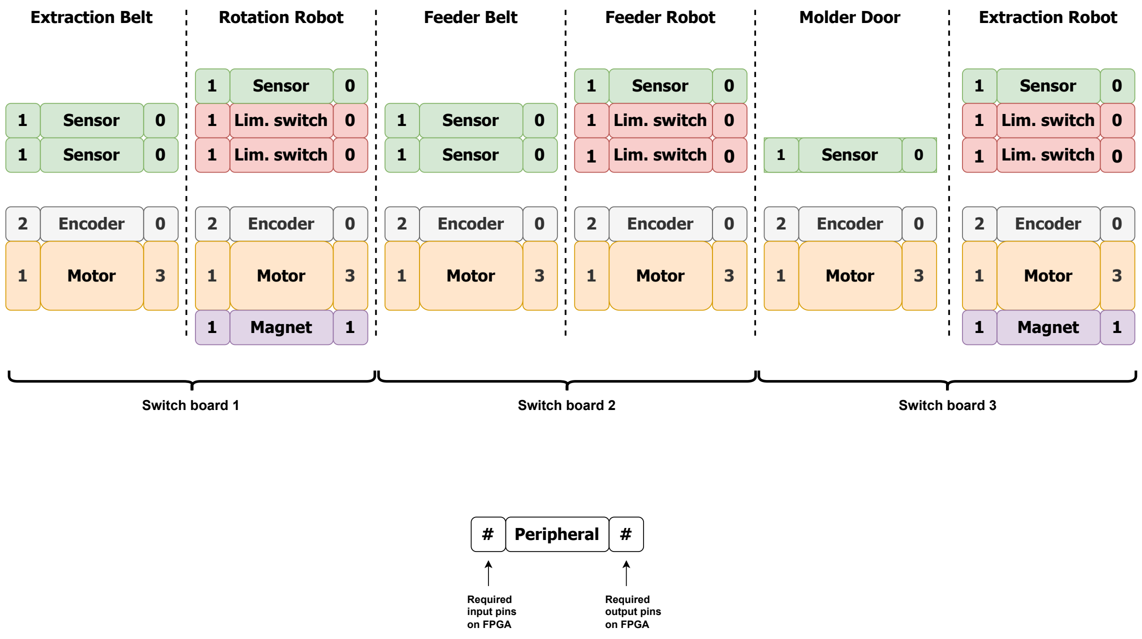


Figure A.4: Overview of the required IO per peripheral of the Production-Cell Units. To each electrical switch board the peripherals of two Production-Cell Units are connected. For the specific signal definitions per peripheral see Table A.1.

A.5 PCB boards

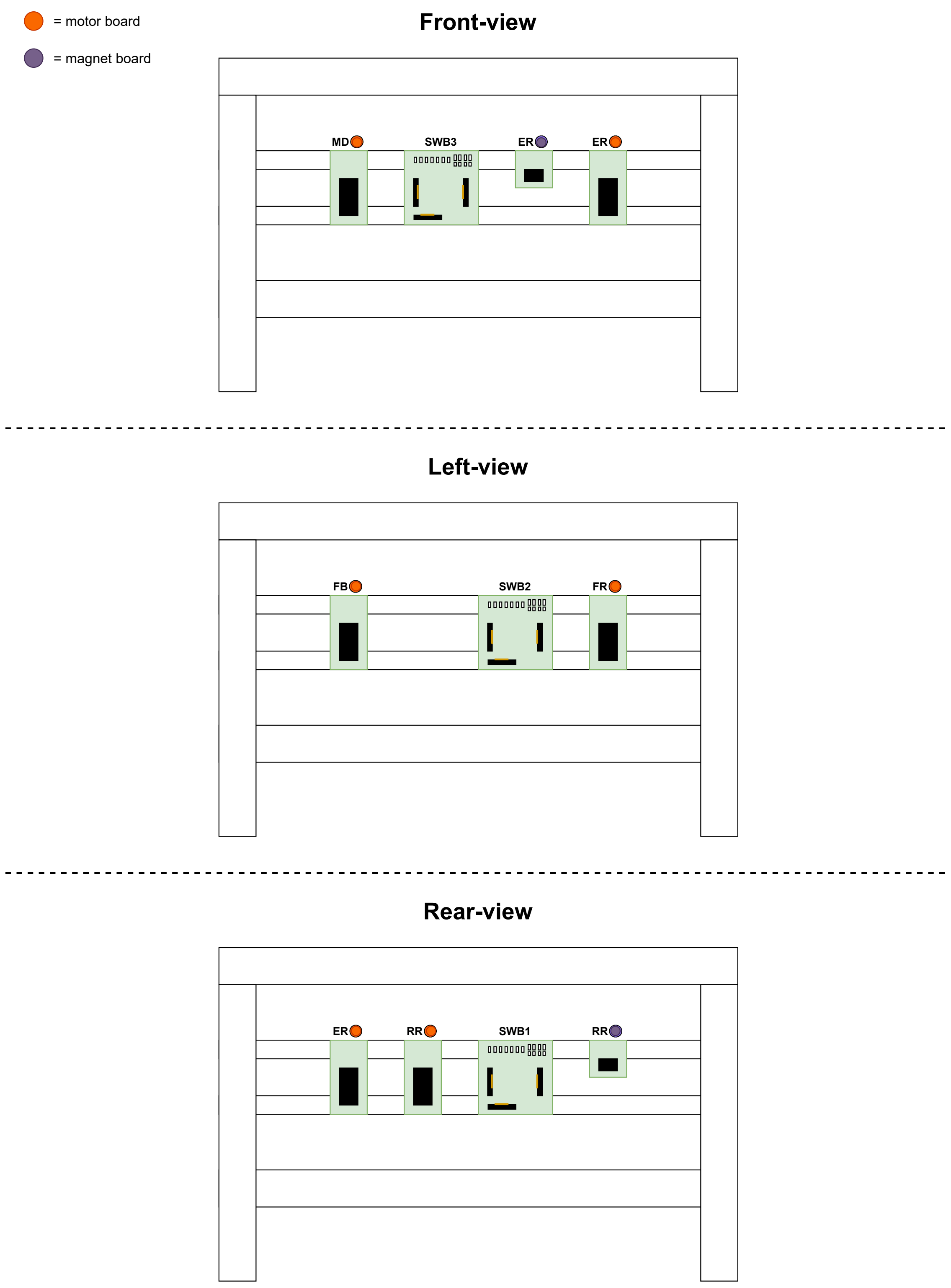


Figure A.5: Abstract overview of the PCB boards residing on the Production Cell's sides. For the abbreviation definitions see Figure A.2 and Figure A.6. The motor boards use the Si9978DW H-bridge-driver chip.

A.6 Electrical switch board

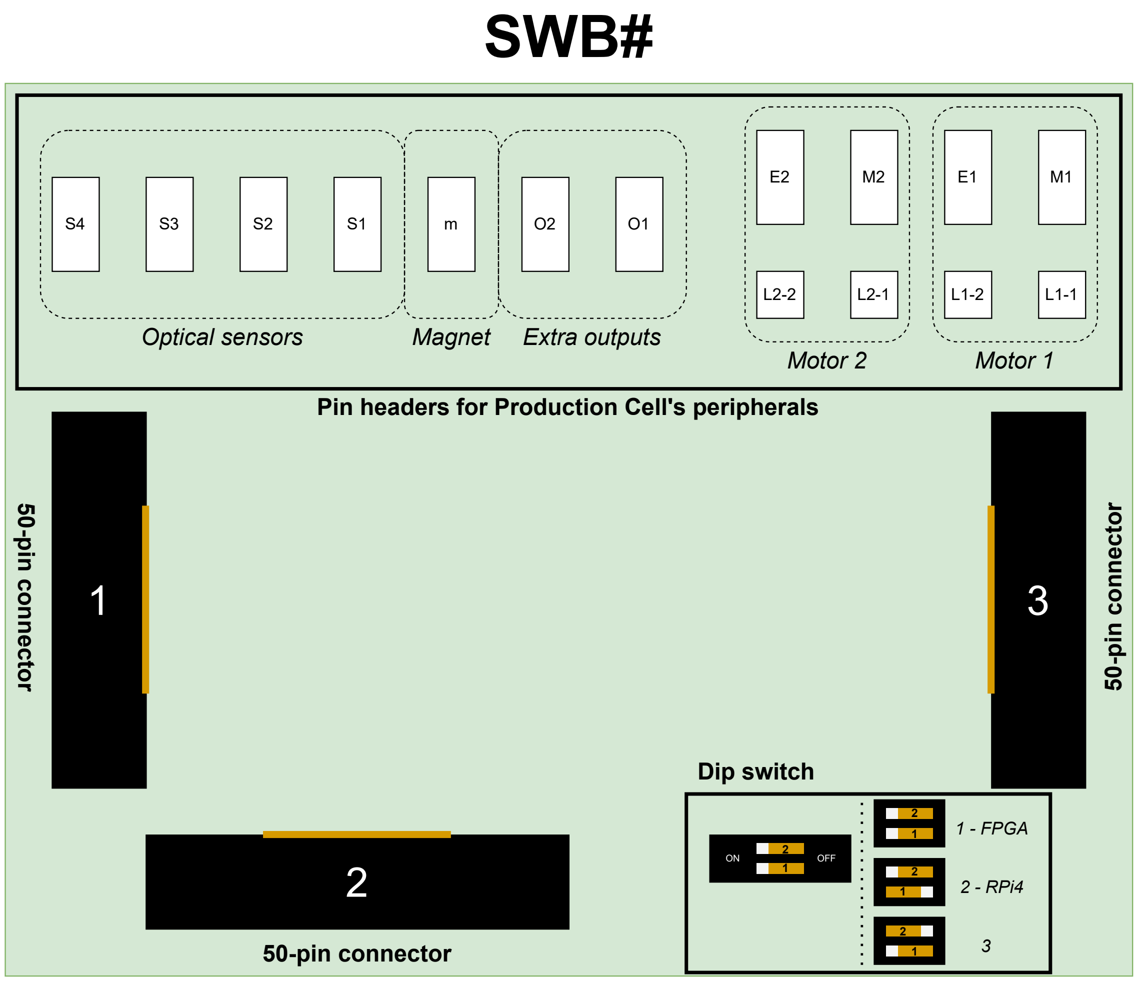
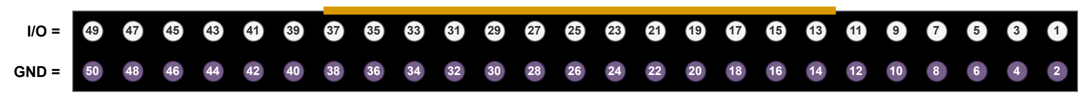


Figure A.6: Abstract overview of the Production Cell's electronic switch board

A.7 50-pin connector definitions



SWB#

1	M1 motor fault	11	M2 motor fault	21	Magnet current sensor	31	M1 direction	41	M2 brake
3	M1 limit switch A	13	M2 limit switch 1	23	Sensor 1	33	M1 PWM	43	Magnet toggle
5	M1 limit switch B	15	M2 limit switch 2	25	Sensor 2	35	M1 brake	45	Extra output 1
7	M1 encoder channel A	17	M2 encoder channel A	27	Sensor 3	37	M2 direction	47	Extra output 2
9	M1 encoder channel B	19	M2 encoder channel B	29	Sensor 4	39	M2 PWM	49	VCC controller node

SWB1

1	RR motor fault	11	EB motor fault	21	RR magnet sensor	31	RR direction	41	EB brake
3	RR limit switch ←	13		23	RR deliver site	33	RR PWM	43	RR magnet toggle
5	RR limit switch →	15		25	EB fetch site	35	RR brake	45	
7	RR encoder channel A	17	EB encoder channel A	27	EB gate	37	EB direction	47	SM toggle
9	RR encoder channel B	19	EB encoder channel B	29		39	EB PWM	49	VCC controller node

SWB2

1	FR motor fault	11	FB motor fault	21		31	FR direction	41	FB brake
3	FR limit switch ←	13		23	FB fetch site	33	FR PWM	43	
5	FR limit switch →	15		25	FB gate	35	FR brake	45	
7	FR encoder channel A	17	FB encoder channel A	27		37	FB direction	47	
9	FR encoder channel B	19	FB encoder channel B	29		39	FB PWM	49	VCC controller node

SWB3

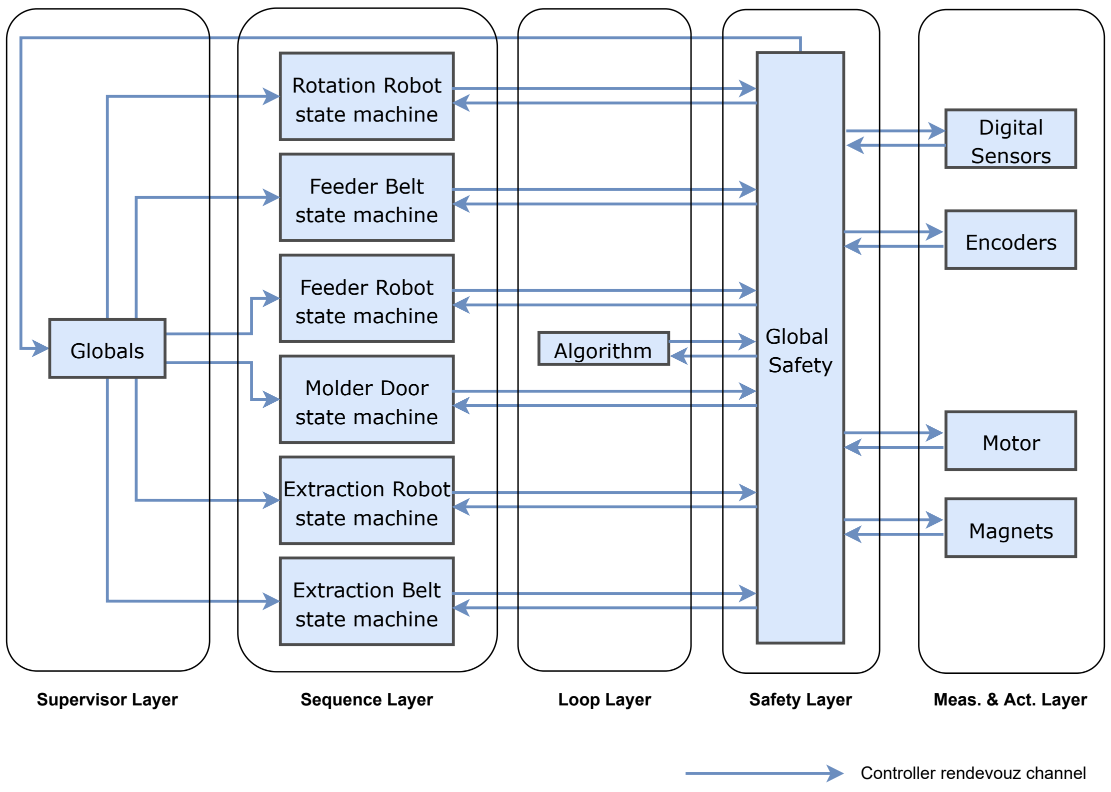
1	ER motor fault	11	MD motor fault	21	ER magnet sensor	31	ER direction	41	MD brake
3	ER limit switch ←	13		23	ER deliver site	33	ER PWM	43	ER magnet toggle
5	ER limit switch →	15		25	MD optoswitch	35	ER brake	45	
7	ER encoder channel A	17	MD encoder channel A	27		37	MD direction	47	
9	ER encoder channel B	19	MD encoder channel B	29		39	MD PWM	49	VCC controller node

Figure A.7: Definitions of the 50-pin connector on the switch board. For the abbreviation definitions see Figure A.2.

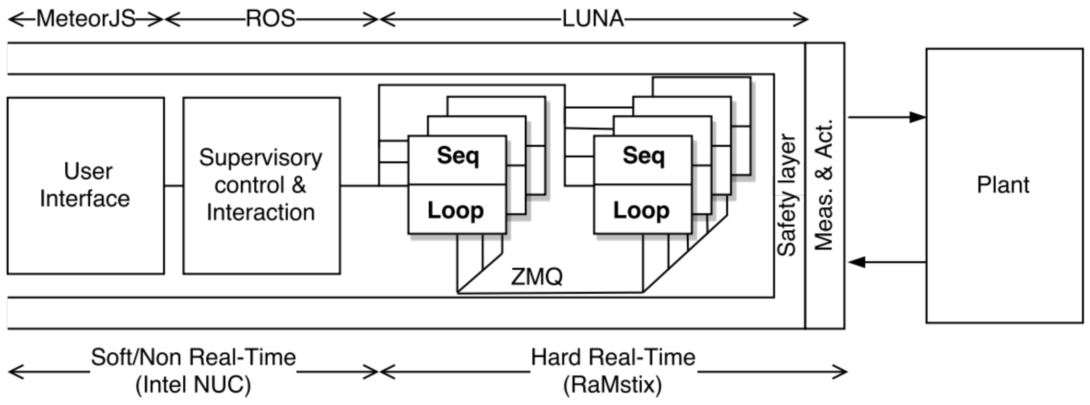
Table A.1: The 50-pin-connector definitions in Figure A.7 with respect to the peripherals of the Production-Units in Figure A.4.

Peripheral	**Pin numbers on 50-pin header**	**Is input or output (FPGA perspective)**	**Description**
Limit switch	3, 13	Input	The Production-Cell Unit is out-of-bounds (left-hand side)
	5, 15	Input	The Production-Cell Unit is out- of-bounds (right-hand side)
Encoder	7, 17	Input	Channel-A signal of the encoder
	9, 19	Input	Channel-B signal of the encoder
Motor	1, 11	Input	The motor experiences a fault (zero pulse is overcurrent, zero signal is undervoltage)
	31, 37	Output	The actuation direction of the motor (active-low 0 is counterclockwise, active-low 1 is clockwise)
	33, 39	Output	**Freerunning-mode control**: duty cycle to H-bridge (0% is stand still, 100% is max. torque) **Braking-mode control**: enable signal to H-bridge (active-high 1 is enable)
	35, 41	Output	**Freerunning-mode control**: enable signal to H-bridge (active-low 0 is enable) **Braking-mode control**: duty cycle to H-bridge (0% is max. torque, 100% is stand still)
Magnet	21	Input	Current is flowing through the magnet
	43	Output	Activates the magnet
Sensor	23, 25, 27, 29	Input	Entity is detected (metal block or door). See Figure A.7.

A.8 Related work


Rotation Robot state machine
Feeder Belt state machine
Feeder Robot state machine
Molder Door state machine
Extraction Robot state machine
Extraction Belt state machine
Globals
Algorithm
Global Safety
Digital Sensors
Encoders
Motor
Magnets
Supervisor Layer
Sequence Layer
Loop Layer
Safety Layer
Meas. & Act. Layer
Controller rendevouz channel

(a) Sassen (2009) on Xilinx FPGA



(b) Ridder (2018) on Intel NUC and RaMStix

Figure A.8: Embedded-control-software architectures that have been realised for the Production Cell in prior work.

B icoBoard

This chapter of the appendix describes the details of the icoBoard, which is used as an extension board for the Raspberry Pi. In Section B.1 the I/O associated with the icoBoard's PMOD-connectors is shown, showing the per-signal definition of a PMOD-connector. Afterwards, in Section B.2 the peripheral mapping of the Production-Cell Units onto the PMOD-connectors is illustrated. Lastly, Section B.3 describes the required toolchain for the icoBoard that is used for compiling FPGA code and running it on the icoBoard.

B.1 PMOD connector

The interface running on the icoBoard is developed by Hofstede (2022). This interface has predefined signal definitions for the PMOD-connectors of the icoBoard. A visual overview of the mapping of the pin definitions of Hofstede (2022) to the pin definitions of the PCB interface board (Appendix D) is shown in Figure B.1. For a tabular overview of this pin-definition mapping, see Table B.1.

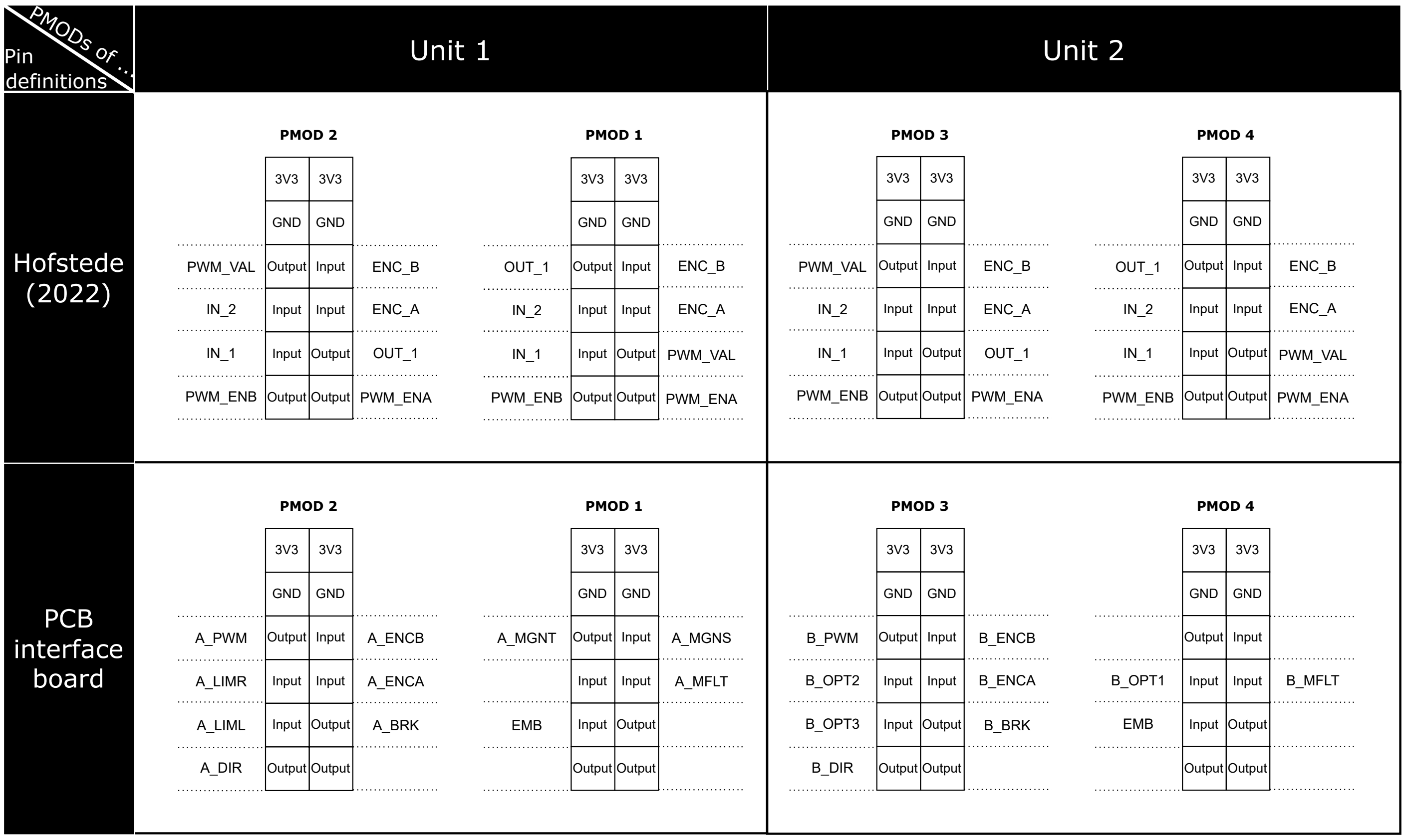

PMODs of ...
Pin definitions
Unit 1
Unit 2
Hofstede (2022)
PCB interface board
PMOD 2
PMOD 1
PMOD 3
PMOD 4
3V3 3V3
GND GND
PWM_VAL Output Input ENC_B
IN_2 Input Input ENC_A
IN_1 Input Output OUT_1
PWM_ENB Output Output PWM_ENA
OUT_1 Output Input ENC_B
IN_2 Input Input ENC_A
IN_1 Input Output PWM_VAL
PWM_ENB Output Output PWM_ENA
PWM_VAL Output Input ENC_B
IN_2 Input Input ENC_A
IN_1 Input Output OUT_1
PWM_ENB Output Output PWM_ENA
OUT_1 Output Input ENC_B
IN_2 Input Input ENC_A
IN_1 Input Output PWM_VAL
PWM_ENB Output Output PWM_ENA
A_PWM Output Input A_ENCB
A_LIMR Input Input A_ENCA
A_LIML Input Output A_BRK
A_DIR Output Output
A_MGNT Output Input A_MGNS
Input Input A_MFLT
EMB Input Output
Output Output
B_PWM Output Input B_ENCB
B_OPT2 Input Input B_ENCA
B_OPT3 Input Output B_BRK
B_DIR Output Output
Output Input
B_OPT1 Input Input B_MFLT
EMB Input Output
Output Output

Figure B.1: The per-pin definition of the PMOD-connectors on the icoBoard

Table B.1: A tabular overview of the mapping of PMOD-pin definitions of Figure B.1.

PMOD number	**Pin definition**	
	Hofstede (2022)	**PCB-interface board**
1	OUT_1	A_MGNT
	IN_2	-
	IN_1	EMB
	PWM_ENB	-
	PWM_ENA	-
	PWM_VAL	-
	ENC_B	A_MGNS
	ENC_A	A_MFLT
2	OUT_1	A_BRK
	IN_2	A_LIMR
	IN_1	A_LIML
	PWM_ENB	A_DIR
	PWM_ENA	-
	PWM_VAL	A_PWM
	ENC_B	A_ENCB
	ENC_A	A_ENCA
3	OUT_1	B_BRK
	IN_2	B_OPT2
	IN_1	B_OPT3
	PWM_ENB	B_DIR
	PWM_ENA	-
	PWM_VAL	B_PWM
	ENC_B	B_ENCB
	ENC_A	B_ENCA
4	OUT_1	-
	IN_2	B_OPT1
	IN_1	EMB
	PWM_ENB	-
	PWM_ENA	-
	PWM_VAL	-
	ENC_B	-
	ENC_A	B_MFLT

B.2 Mapping Production Cell's peripherals

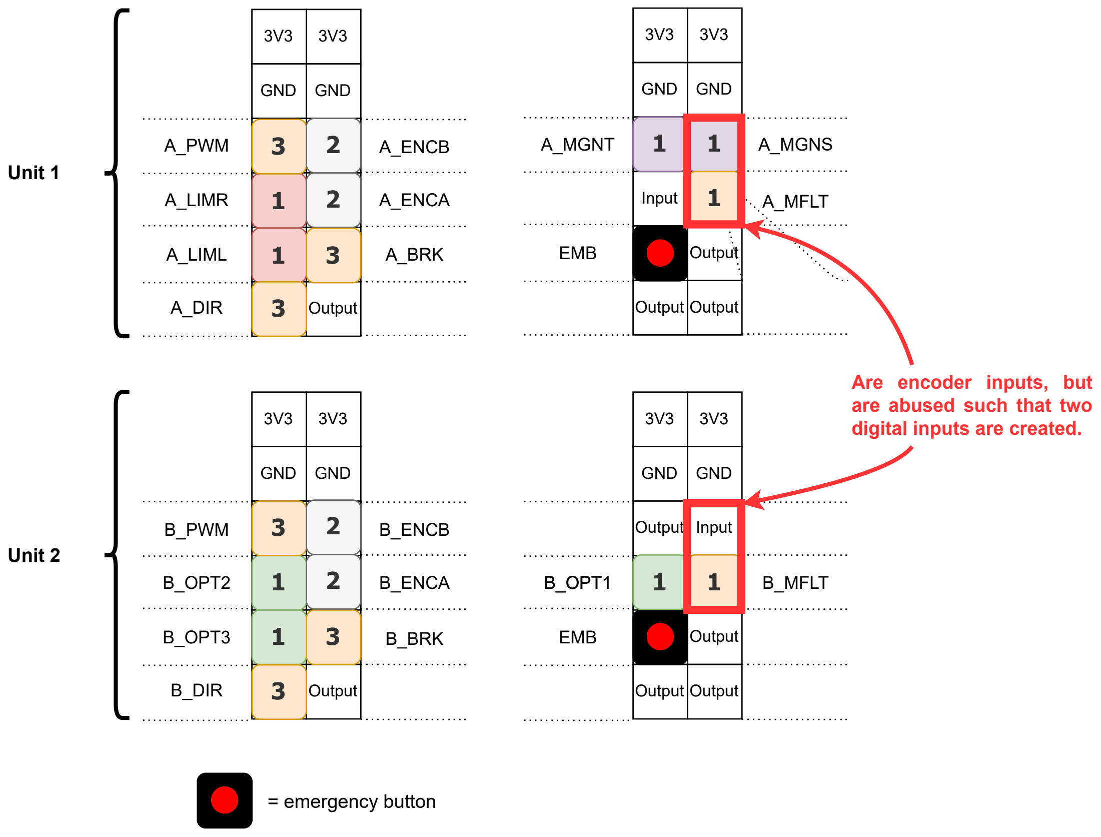


Figure B.2: Per-board mapping of two Production-Cell-Units' peripherals (Figure A.4) to the PMOD-connectors (Figure B.1). The pin definitions of the PCB interface board are used in this figure, see Figure B.1.

B.3 Toolchain

The necessary tools to program the FPGA are shown in Table B.2. These are run on the computer, where the toolchain compiles Verilog code into a .bin file. This .bin file can be used by the Raspberry Pi to flash the icoBoard.

Table B.2: The open source FPGA toolchain for the Icoboard. Adapted from Vinkenvleugel (2022).

Tool	Description
Yosys	An open synthesis suite for Verilog, used to convert the initial Verilog code to an intermediate representation in JSON.
Nextpnr	A place and route tool that accepts the JSON file that was generated by Yosys and converts it to a hardware-specific layout in an ASC file.
Icestorm	A package that contains hardware support for the particular Lattice iCE40- HX8K FPGA that is used in the Icoboard. A program called Icepack is used to convert the ASC file to a binary file that can be flashed on the FPGA chip.
Icotools	Tools to flash the binary file on the FPGA, either to flash or to run it as a bitstream.
Icarus Verilog	A tool to simulate Verilog code on a computer.

C Embedded-control-software architecture

This chapter of the appendix describes different aspects concerning the implemented embedded-control-software architecture. Section C.1 until Section C.3 describe the in-depth implementation details of the embedded-control-software architecture. These sections describe the following:

- Overview of the 20-sim models of the motion-control layers
- Models used for the discrete-event-control layers
- Custom 20-sim model blocks that are used for the motion-control layers

In Section C.4, the physical connections underlying the implemented embedded-control-software architecture is shown. Section C.5 describes additional diagrams and plots concerning the embedded-control-software architecture, primarily serving as an addendum to Chapter 5. Lastly, Section C.6 describes the run commands to activate and deactivate the Raspberry-Pi-based demonstrator.

C.1 Motion-control model

This section describes the 20-sim model blocks of the motion-control layers, which consist of the loop controller, safety layer and measurement-and-actuation layer (Figure 4.3). In Section C.1.1, an overview of the elements of the loop-controller model block in 20-sim is shown. After this, in Section C.1.2, an overview of the elements of the safety-layer model block in 20-sim is shown. Next, in Section C.1.3 an overview of the elements of the measurement-and-acuation-layer model block in 20-sim is shown. In Section C.3 an overview of the used custom 20-sim model blocks is given. These blocks have been used in the motion-controller-layer model blocks and are not present in 20-sim's standard library.

C.1.1 Loop-controller model

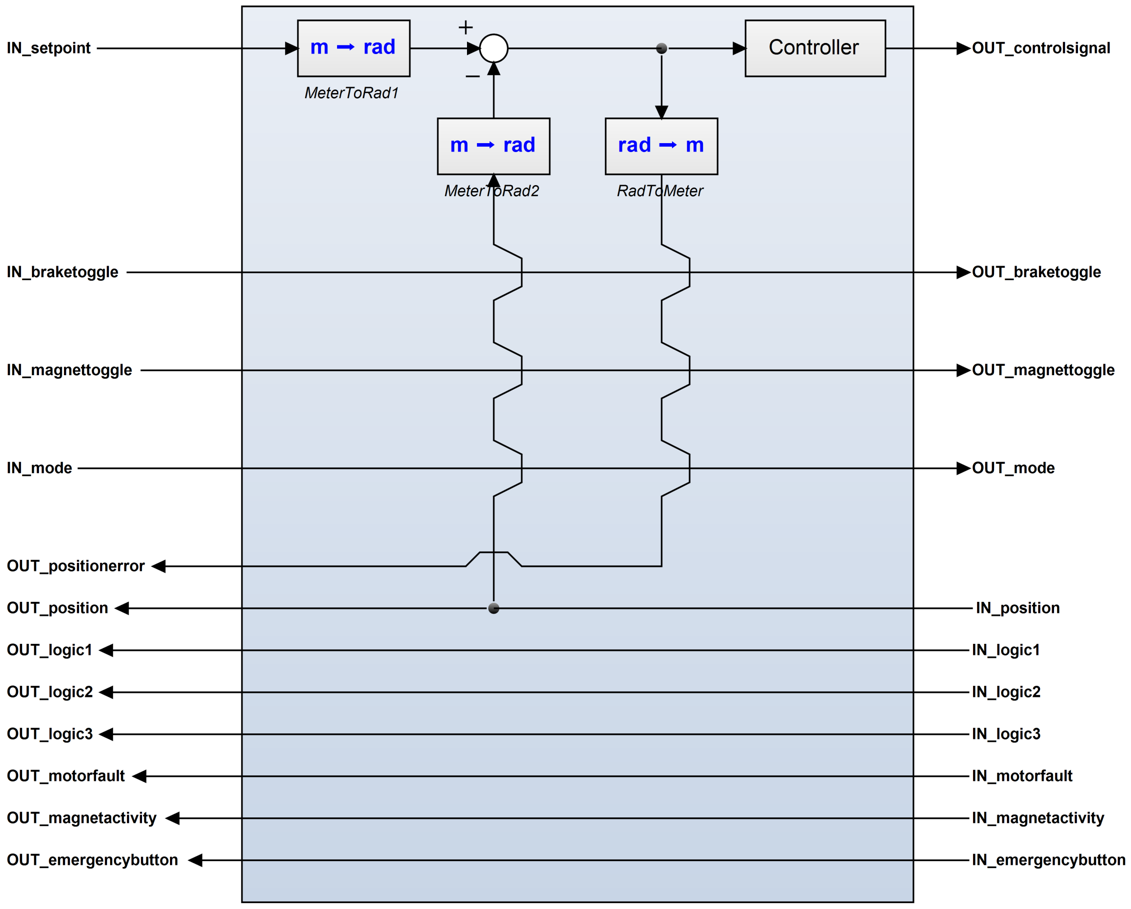


Figure C.1: Loop controller model in 20-sim. For descriptions of the used model blocks see for **Controller:** Listing C.1; **MeterToRad/RadToMeter:** Listing C.2.

C.1.2 Safety-layer model

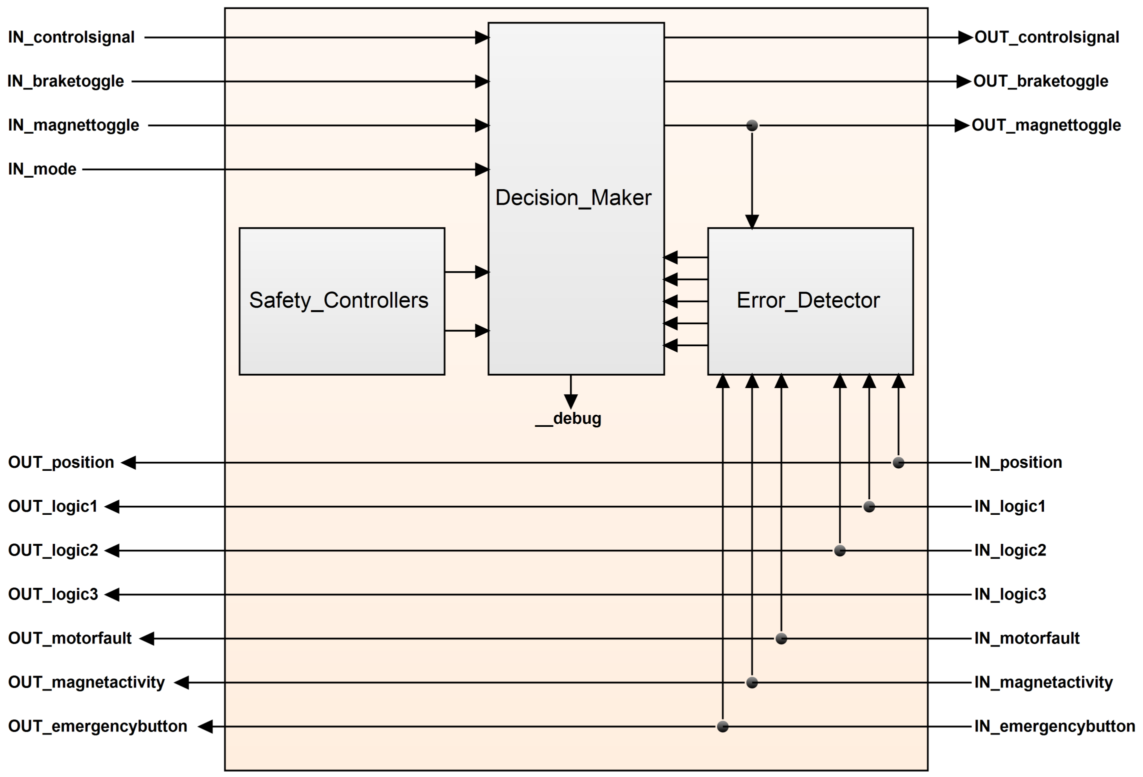


Figure C.2: Safety layer model in 20-sim. For descriptions of the used model blocks see for **Decision Maker:** Table C.2; **Error Detector:** Figure C.3, Table C.3; **Safety Controllers:** Figure C.4.

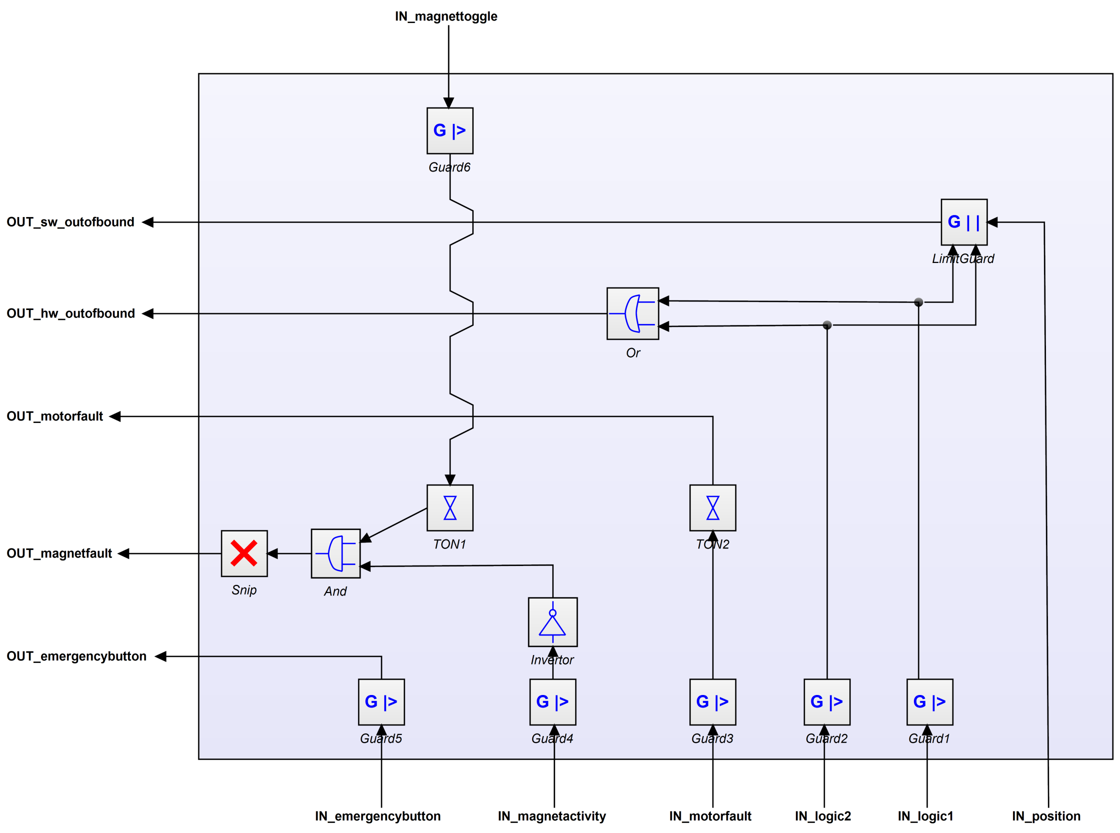


Figure C.3: 20-sim model of *Error_Detector* in Figure C.2. Depending on the Production-Cell Unit some error signals are disabled with a *Snip* block, since they are either not applicable to the unit or are considered unneccesary for the unit. For descriptions of the used model blocks see for **Guard:** Listing C.3; **LimitGuard:** Listing C.4; **TON:** Listing C.5.

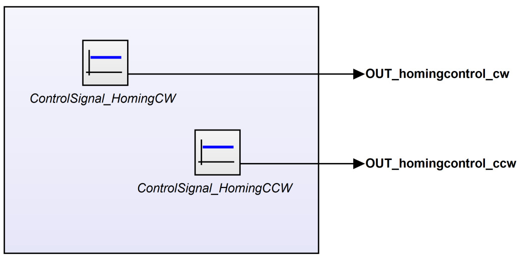


Figure C.4: 20-sim model of *Safety_Controllers* in Figure C.2. The homing controllers output low-voltage control signals which rotates the motor of the unit either clock-wise or counter-clockwise. The safety-controller block only covers homing procedures.

C.1.3 Measurement-and-actuation model

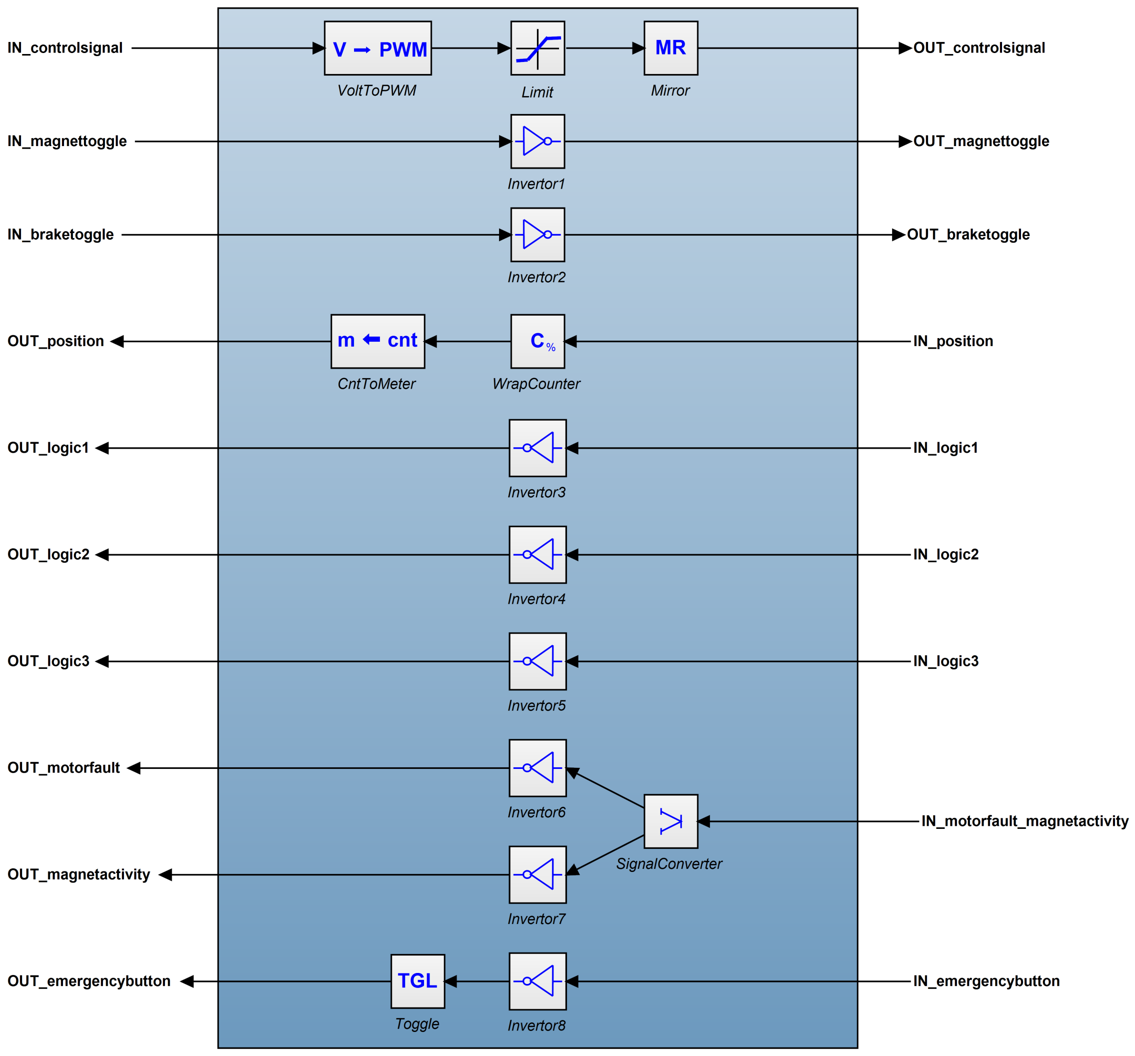


Figure C.5: Measurement-and-actuation model in 20-sim. For descriptions of the used model blocks see for **VoltToPWM:** Listing C.6; **Mirror:** Listing C.7; **SignalConverter:** Listing C.8; **CntToMeter:** Listing C.9; **WrapCounter:** Listing C.10; **Toggle:** Listing C.11.

C.2 Discrete-event-control model

This section describes the models that are used for the discrete-event-control layers, which consist of the supervisory controller, supervisory controller and bridge (Figure 4.5). In Section C.2.1, the finite-state-machine model underlying the supervisory controller is shown. After this, in Section C.2.1 the trajectory-generation model that is used for the sequence controller is shown. Lastly, in Section C.2.3 an overview of the signal lines in the embedded-control-sofware architecture is shown. In this signal-line overview, it is illustrated how the bridge components transports the information between Linux and Xenomai.

C.2.1 Supervisory-controller model

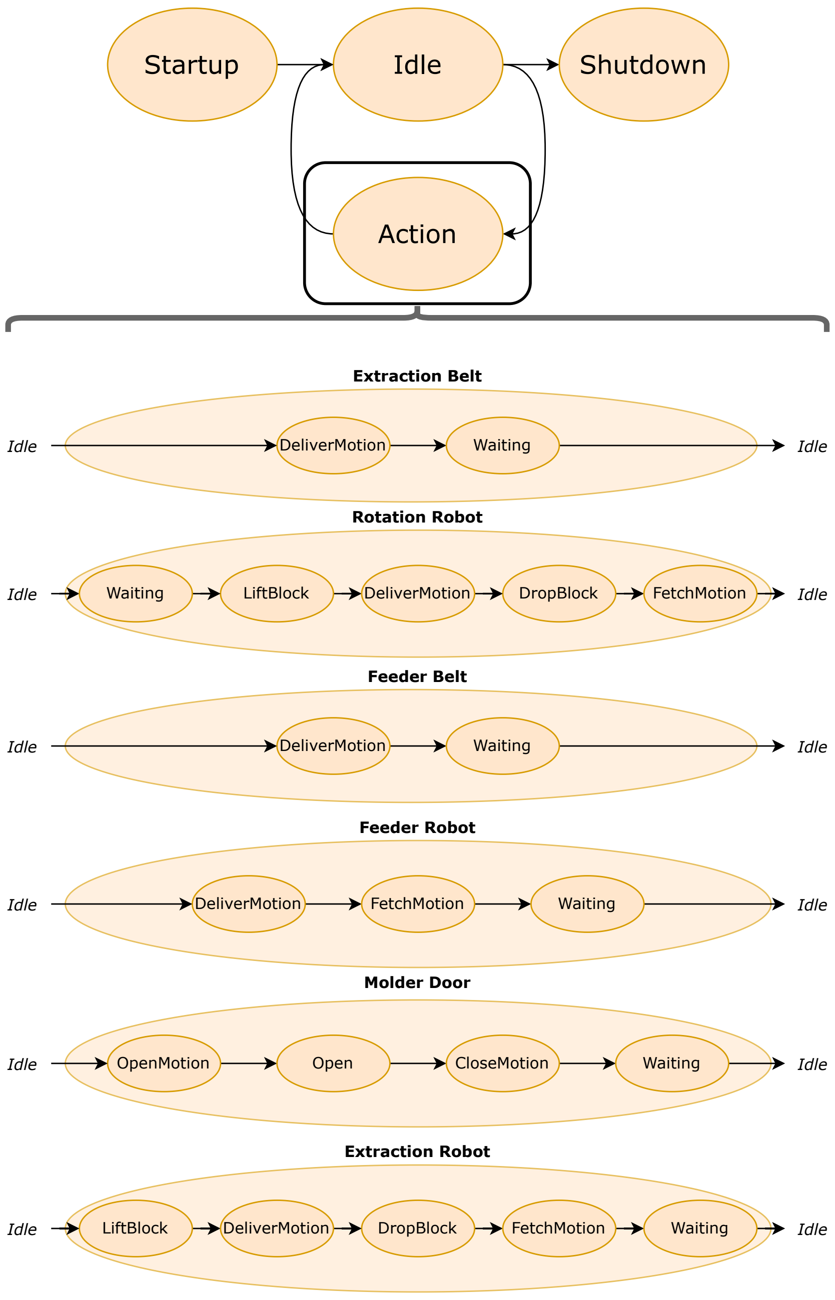


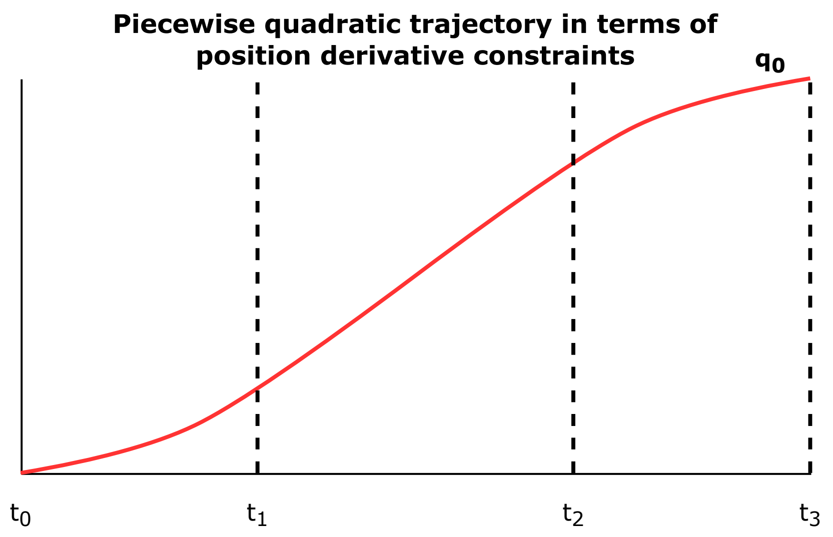
Figure C.6: Supervisory controller functionality is based on finite state machines. In the bottom of the figure the substates of the *Action*-state is shown for each unit.

The information in Table C.1 serves as additional information regarding supervisory controllers. In this table different tools are shown which can create supervisory-like code components using certain models of computation (finite-state machines or behaviour trees). These tools have the potential to be integrated in the model-driven design workflow of the robot-software framework. Some of the tools support a ROS2-based solution out of the box and/or promise that it is adequate for real-time systems.

Table C.1: Existing tools that can create supervisory-like code components

Tool	Description
Simulink	Simulink (Stateflow) provides a graphical language that includes state transition diagrams, flow charts, state transition tables, and truth tables. Stateflow enables you to design and develop supervisory control, task scheduling, fault management, communication protocols, user interfaces, and hybrid systems. Here, Simulink Coder allows you to generate C and C++ code from models that contain Stateflow charts. You can then use the generated code for real-time and non-real-time applications.
20-sim	20-sim is modeling and simulation software package for mechatronic systems. With 20-sim you can enter models graphically, similar to drawing an engineering scheme. With these models you can simulate and analyse the behaviour of multi-domain dynamic systems and create control systems. The Real Time toolbox of 20-sim allows you to create C-code out of any 20-sim model for the use in real-time applications.
UPPAAL	Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.).
Rhapsody	Rhapsody is part of the IBM Engineering portfolio that provides a collaborative design development, and test environment for systems engineers that supports UML, SysML, UAF as well as AUTOSAR import and export capabilities.
Groot	Groot is the Integrated Development Environment to build and debug Behavior Trees. It allows you to create and edit trees, using a drag and drop interface, monitor the state of a tree remotely in real-time and debug and test your behaviors.
Papyrus	Papyrus for Robotics is graphical editing tool for robotic applications that complies with the RobMoSys approach. It manages complexity of robotics development by supporting composition-oriented engineering of robotics systems and separating the task into multiple tiers executed by different roles.
SmartMDSD	SmartMDSD is an Eclipse-based IDE for software development and system composition in a robotics software business ecosystem. It supports the different roles that act around the development of robotics systems to offer software components and/or use software components to build systems.
SMACC2	SMACC2 is an event-driven, asynchronous, behavioral state machine library for real-time ROS 2 (Robotic Operating System) applications written in C++, designed to allow programmers to build robot control applications for multicomponent robots, in an intuitive and systematic manner.
YASMIN	YASMIN is a project focused on implementing robot behaviors using Finite State Machines (FSM). It is fully integrated into ROS2, allows fast prototyping and a webviewer is included which allows the monitoring of the execution of the state machines.
MERLIN2	MERLIN2 is a hybrid cognitive architecture based on symbolic planning and state machine decision-making systems that allows performing robot behaviors. The architecture can run in any robot running ROS2 , the latest version of the Robot Operative System. The latest version of MERLIN only supports ROS2 using Python.
TinyFSM	TinyFSM is a simple finite state machine library for C++, designed for optimal performance and low memory footprint. This makes it ideal for real-time operating systems. The concept is very simple, allowing the programmer to fully understand what is happening behind the scenes. It provides a straightforward way of mapping your state machine charts into source code.

C.2.2 Sequence-controller model



Tunable parameters

$\mathbf{q_{0,\,max}}$	: position maximum (stroke)
$\mathbf{q_{1,\,max}}$	: velocity maximum
$\mathbf{q_{2,\,max}}$	: acceleration maximum

From this follows ...

$$\overbrace{\hspace{10cm}}$$

$\mathbf{t_0} = 0$

$\mathbf{t_1} = q_{1,\,max} \,/\, q_{2,\,max}$

$\mathbf{t_2} = q_{0,\,max} \,/\, q_{1,\,max}$

$\mathbf{t_3} = (q_{1,\,max} \,/\, q_{2,\,max}) + (q_{0,\,max} \,/\, q_{1,\,max})$

<u>**Traverse the following polynomial per time segment**</u>

$$\mathbf{q_0} = (1/2) \cdot q_{2,i} \cdot t^2 + q_{1,i} \cdot t + q_{0,i}$$

$t_0 < t < t_1$:

$\mathbf{q_{0,i}} = 0$

$\mathbf{q_{1,i}} = 0$

$\mathbf{q_{2,i}} = q_{2,max}$

$t_1 \leq t < t_2$:

$\mathbf{q_{0,i}} = -q_{1,max}^2 \,/\, (2 \cdot q_{2,max})$

$\mathbf{q_{1,i}} = q_{1,max}$

$\mathbf{q_{2,i}} = 0$

$t_2 \leq t < t_3$:

$\mathbf{q_{0,i}} = (-q_{0,max}^2 \cdot q_{2,max}^2 - q_{1,max}^4) \,/\, (2 \cdot q_{1,max}^2 \cdot q_{2,max})$

$\mathbf{q_{1,i}} = (q_{0,max} \cdot q_{2,max} \,/\, q_{1,max}) + q_{1,max}$

$\mathbf{q_{2,i}} = -q_{2,max}$

Figure C.7: Sequence controller functionality is based on second-order motion-profile equations

C.2.3 Bridge model

In Figure C.8, an overview is shown of the signal lines that are present in the embedded-control-software architecture. The overview describes the information flow between:

- The discrete-event-control component (*ROS2-node*) and the motion-control component (*20-sim model*)
- The motion-control component (*20-sim model*) and the icoBoard-based interface (*icoBoard-PMOD*)

In this overview, the functionality of the bridge nodes is shown with the black rectangles.

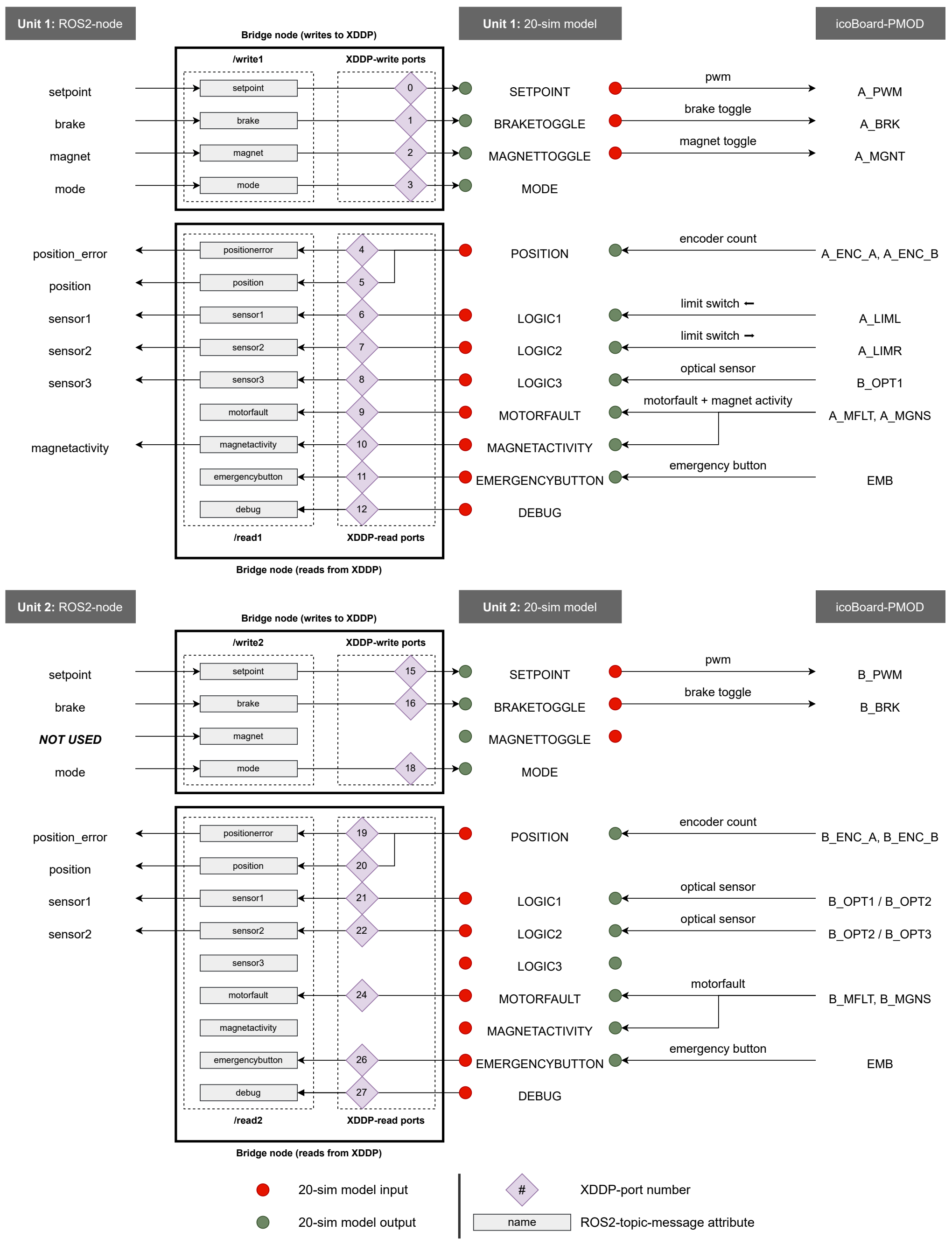


Figure C.8: An overview of the general structure of the embedded-control-software-architecture signal lines. The entries under a component represent the attributes of the component. The arrows indicate how the component attributes are interrelated via the signal lines. The bridge nodes transport signals between Linux and Xenomai over the available XDDP ports.

C.3 Custom 20-sim model blocks

This section describes the custom 20-sim model blocks that are used for the motion-control layers in Section C.1. For each custom 20-sim model block its respective code is shown, except for the *Decision_Maker* model block of the safety-layer model block (Figure C.2). Instead of the code, an overview of the *Decision_Maker*'s strategies is shown. Similarly, an overview of the checked errors by the *Error_Detector* model block of the safety-layer model block is shown.

```
parameters
    // See global editor for values
    real global beta; // Tameness factor
    real global zeta; // Damping factor
    real global wc;   // Cutoff frequency
    real global Z;    // Total moving mass
variables
    real hidden Kc, tau_z, tau_p;    // Parameters for controller
    real hidden _yk2_, _yk1_;        // Delay variable coefficients (output)
    real hidden _uk2_, _uk1_, _uk0_; // Delay variable coefficients (input)
    real hidden yk2, yk1;            // Delay variables (output)
    real hidden uk2, uk1;            // Delay variables (input)
initialequations
    // Controller coefficients
    Kc = (Z*(wc)^2) / sqrt(beta);
    tau_z = sqrt(beta) * (1/(wc));
    tau_p = 1 / (sqrt(beta)*(wc));
    // Delay variable coefficients
    _yk2_ = (-4*sampletime^2*tau_p^2 + 4*sampletime*tau_p*zeta -
     1)/(4*sampletime^2*tau_p^2 + 4*sampletime*tau_p*zeta + 1);
    _yk1_ = 2*(4*sampletime^2*tau_p^2 - 1)/(4*sampletime^2*tau_p^2 +
     4*sampletime*tau_p*zeta + 1);
    _uk2_ = Kc*(-2*sampletime*tau_z + 1)/(4*sampletime^2*tau_p^2 +
     4*sampletime*tau_p*zeta + 1);
    _uk1_ = 2*Kc/(4*sampletime^2*tau_p^2 + 4*sampletime*tau_p*zeta + 1);
    _uk0_ = Kc*(2*sampletime*tau_z + 1)/(4*sampletime^2*tau_p^2 +
     4*sampletime*tau_p*zeta + 1);
equations
    // Output difference equation (controller output)
    yk0 = _yk2_ * yk2 + _yk1_ * yk1 + _uk2_ * uk2 + _uk1_ * uk1 + _uk0_ * uk0;

    // Store current samples to "previous" samples
    yk2 = previous(yk1);
    yk1 = previous(yk0);
    uk2 = previous(uk1);
    uk1 = previous(uk0);
```

Listing C.1: 20-sim model of *Controller* in Figure C.1. This custom 20-sim model block describes a PD-controller-based transfer function (Equation C.1). The controller can be tuned with the total moving mass (Z) and cutoff frequency (ω_c) of a system (Equation C.2). The transfer function has been discretized with the bilinear transform and is modelled as the *Controller* block.

$$C(s) = K_c \cdot \frac{s\tau_z + 1}{(s\tau_p)^2 + 2\zeta\tau_p s + 1} \tag{C.1}$$

$$K_c = \frac{Z \cdot \omega_c^2}{\sqrt{\beta}} \qquad \tau_z = \sqrt{\beta} \cdot \frac{1}{\omega_c} \qquad \tau_p = \frac{1}{\sqrt{\beta} \cdot \omega_c} \qquad \beta = 10 \qquad \zeta = 0.8 \tag{C.2}$$

```
// Description:
// - This block transforms a "meter"-based position of the end-effector into
    an equivalent "radial" encoder position.
parameters
    real counts_per_m = 77986; // See Figure 28 of (Van den Berg, 2006)
    real counts_per_rev = 2000;
variables
    real counts;
    real revs;
    real rev_per_counts;
initialequations
    rev_per_counts = 1 / counts_per_rev;
equations
    // Convert meters to equivalent count (see Figure 28 of {Van den Berg,
     2006})
    counts = m * counts_per_m;
    // Convert counts to revolutions needed by motor to achieve this
     translational position
    revs = counts * rev_per_counts;
    // Convert revolutions to radians needed by motor to achieve this
     translational position
    rad = revs * (2*pi);
```

Listing C.2: 20-sim model of *MeterToRad* and *RadToMeter* in Figure C.1.

```
// Description:
// - This "guard" block is primarily used to check what the incoming signal
     value should NOT be, reporting an error if this is the case. A
     larger-than statement instead of an is-equal statement is used here as
     replacement for floating-point comparisons (x > 0.5 instead of x == 1.0)
parameters
    real boundary = 0.5;
equations
    if input > boundary then
        output = 1.0;
    else
        output = 0.0;
    end;
```

Listing C.3: 20-sim model of *Guard* in Figure C.3.

```
// Description
// - This block takes snapshots of the inputted position when the physical
    position of the limit switches are found. After this event, the block
    will output a fault signal when the current position is out-of-bound
    w.r.t. to the clipped "software" limits.
parameters
   real clipping_value = 0.0005; // context dependent; could be either in
    [m] or [counts] --> in this context in [m]
variables
   real boundary_lower;
   real boundary_upper;
   real swap_variable;
   boolean clipping_done;
initialequations
   boundary_lower = 0;
   boundary_upper = 0;
   swap_variable = 0;
   clipping_done = false;
code
   // If limit switch (1) is hit, take a snapshot of the position; this will
    be the first boundary
   if (limit1 > 0.5) then
   // If boundary is zero, then value is still uninitialized
   if (boundary_lower == 0) then
      boundary_lower = position;
   end;
   end;

   // If limit switch (2) is hit, take a snapshot of the position; this will
    be the second boundary
   if (limit2 > 0.5) then
   // If boundary is zero, then value is still uninitialized
   if (boundary_upper == 0) then
      boundary_upper = position;
   end;
   end;

   // Swap the boundary values, in case it is found that the definitions are
    flipped; do this only when boundaries are non-zero
   if (boundary_lower <> 0) and (boundary_upper <> 0) then
   if (boundary_upper < boundary_lower) then
      swap_variable = boundary_upper;
      boundary_upper = boundary_lower;
      boundary_lower = swap_variable;
   end;
   // Clip the values with a fixed user-defined distance value
   if (not clipping_done) then
      boundary_lower = boundary_lower + clipping_value;
      boundary_upper = boundary_upper - clipping_value;
      clipping_done = true;
   end;
   end;

   // When both limit switches have been initialized, allow this guard block
    to output a fault signal
   if (boundary_lower <> 0) and (boundary_upper <> 0) then
   // When outside the clipped boundary range, return a fault signal
   if (position < boundary_lower) or (position > boundary_upper) then
```

```
      output = 1;
   // ... zero signal in all other cases
   else
      output = 0;
   end;
   else
      output = 0;
   end;
```

Listing C.4: 20-sim model of *LimitGuard* in Figure C.3.

```
// Description:
// - This block describes a TON element, which is commonly used in PLC-logic
parameters
   real limit_wait_time = 10.0; // in [s]
variables
   real hidden counter;
   real hidden time_count_equivalent;
initialequations
   counter = 0;
   time_count_equivalent = limit_wait_time / sampletime;
code
   // If the enable signal is HIGH, start the TON and the counting process
   if (previous(enable) > 0.5) then
      counter = counter + 1;
   // If the enable signal is LOW, reset the TON and its counter
   else
      counter = 0;
   end;

   // If the count, which is equal to a certain time period (since the
    sample time is known), is exceeded, then output a signal
   if (counter >= time_count_equivalent) then
      output = 1.0;
   else
      output = 0.0;
   end;
```

Listing C.5: 20-sim model of *TON* in Figure C.3.

```
// Description:
// - This block transforms a voltage of the controller in [V] into an
    equivalent PWM-signal in [1] (PWM module on icoBoard accepts range from
    -2047 to +2047)
parameters
   real motor_voltage_rating = 24;  // in [V]
   real pwm_limit = 2047;
equations
   // Scale the input based on the range of allowed PWM-signals
   pwm = (controlsignal / motor_voltage_rating) * pwm_limit;
```

Listing C.6: 20-sim model of *VoltToPWM* in Figure C.5.

```
// Description:
// - This block transforms a PWM-signal into its mirrored equivalent, since
    inverted definitions are required (e.g. braking-mode control). Allows
    the option to flip the sign of the control effort
parameters
   // The boundary for the PWM value according to RPi-Icoboard framework
    (Hofstede, 2022); in other words 11-bit value
   real maximum = 2047;
   real minimum = -2047;
variables
   real shifted_value;      // Shift to [max - N] or [min - N]
   real corrected_value;    // Correct to a value that always has a sign bit
    (issue is basically: "which sign bit to use, if the sent PWM value is
    0?")
   boolean flip_sign;
initialequations
   shifted_value = 0;
   flip_sign = true;
equations
   if (input > 0) then

      // Shift
      shifted_value = maximum - input;

      // Correct
      corrected_value = if (input == maximum) then
         1.0    // Small POSITIVE offset to acquire the proper sign bit ...
      else
         shifted_value
      end;

      // Invert
      if (flip_sign) then
         output = (-1) * corrected_value;
      else
         output = corrected_value;
      end;

   else if (input < 0) then

      // Shift
      shifted_value = minimum - input;

      // Correct
      corrected_value = if (input == minimum) then
         -1.0   // Small NEGATIVE offset to acquire the proper sign bit ...
      else
         shifted_value
      end;

      // Invert
      if (flip_sign) then
         output = (-1) * corrected_value;
      else
         output = corrected_value;
      end;

   else
```

```
        // Input is 0, or an error occurred
        output = 2047; // Can also be -2047; both are equivalent to zero output

    end;
    end;
```

Listing C.7: 20-sim model of *Mirror* in Figure C.5.

```
// Description:
// - This block decodes the muxed encoder value, now functionally as digital
    inputs, into seperate signals (signal from A, signal from B). Here, it
    is assumed: {A}: Motor fault, {B}: Magnet activity
equations
    switch signal

        case 0 do
            signalA = 0;
            signalB = 0;

        case 1 do
            signalA = 1;
            signalB = 0;

        case 2 do
            signalA = 0;
            signalB = 1;

        case 3 do
            signalA = 1;
            signalB = 1;

        // Error
        default do
            signalA = 0;
            signalB = 0;

    end;
```

Listing C.8: 20-sim model of *SignalConverter* in Figure C.5.

```
// Description:
// - This block transforms a "count" position of the encoder into an
    equivalent "meter"-based position of the end-effector
parameters
    real counts_per_m = 77986; // See Figure 28 of (Van den Berg, 2006)
variables
    real m_per_count;
initialequations
    m_per_count = (1 / counts_per_m);
equations
    // Convert counts to equivalent meter (see Figure 28 of {Van den Berg,
    2006})
    m = counts * m_per_count;
```

Listing C.9: 20-sim model of *CntToMeter* in Figure C.5.

```
// Description:
// - This block counts the wraparounds that occur in the encoder, and saves
    this information. With the counted wraparounds, the necessary offset for
    the current encoder-count is computed.
parameters
   // The limit for the encoder value, where the encoder value starts
    wrapping around; upper boundary -->  2^(n-1) - 1
   integer encoder_boundary_upper = 16383;

   // The limit for the encoder value, where the encoder value starts
    wrapping around; lower boundary --> -2^(n)
   integer encoder_boundary_lower = 0;

   // The difference at which a wraparound "occurs" according to this "wrap
    counter" block
   integer wrap_threshold = 8000;

variables
   integer last_input;
   integer wrap_count;
initialequations
   current_output = 0;  // Current corrected output
   last_input = 0;      // Previous non-corrected input
   wrap_count = 0;      // Counter for the wraparounds
code
   // Store the "current" encoder value as "previous" encoder value;
    eventually check wraparound by comparing these
   last_input = previous(current_input);

   // Wraparound [n ... -n-1] has occurred (sudden switch in value)
   if (last_input - current_input) > 2*wrap_threshold then
      wrap_count = wrap_count + 1;
   else
   // Wraparound [-n-1 ... n] has occurred (sudden switch in value)
   if (last_input - current_input) < -2*wrap_threshold then
      wrap_count = wrap_count - 1;
   end;
   end;

   // Determine the correct format depending on the wrap count

   // Travelled {half} map --> upper
   if wrap_count == 1 then
      current_output = (wrap_count * encoder_boundary_upper) +
    abs(encoder_boundary_lower - current_input);

   // Travelled at least {half + N * whole} map --> upper
   else if wrap_count >= 2 then
      current_output = encoder_boundary_upper + (wrap_count-1) *
    (encoder_boundary_upper - encoder_boundary_lower) +
    abs(encoder_boundary_lower - current_input);

   // Travelled {half} map --> lower
   else if wrap_count == -1 then
      current_output = (-wrap_count * encoder_boundary_lower) -
    abs(encoder_boundary_upper - current_input);

   // Travelled at least {-half - N * whole} map --> lower
```

```
    else if wrap_count <= -2 then
        current_output = encoder_boundary_lower + (wrap_count+1) *
     (encoder_boundary_upper - encoder_boundary_lower) -
     abs(encoder_boundary_upper - current_input);

    else
        current_output = current_input;
    end;
    end;
    end;
    end;
```

Listing C.10: 20-sim model of *WrapCounter* in Figure C.5.

```
// Description:
// - This block describes a 'toggle' element which is equivalent to a
     push-button-like block which saves the button press state. It switches
     state when an edge-event has been detected.
variables
    real prev_input;
    integer state; // OFF: 0 / ON: 1
initialequations
    prev_input = 0;
    state = 0;
equations
    prev_input = previous(input);

    switch (state)

        case 0 do
            if (prev_input == 0) and (input == 1) then
                state = 1;
            end;
            output = 0;

        case 1 do
            output = 1;

    end;
```

Listing C.11: 20-sim model of *Toggle* in Figure C.5.

Table C.2: Safeguarding strategies of the *Decision_Maker* 20-sim model in Figure C.2. The strategy differs per state of a Production Cell's unit state machine (Figure C.6) and the present errors (Table C.3).

	Checked errors					**Safeguarding strategy**		
State	**Hardware boundary**	**Software boundary**	**Motor fault**	**Magnet fault**	**Emergency button**	**Control signal**	**Brake**	**Magnet**
Startup			X		X	0	X	
Idle, Action	X	X	X	X	X	0	X	X
Shutdown	X	X	X		X	0	X	

Table C.3: Explanations of the checked errors in the *Error_Detector* 20-sim model of Figure C.3

Error	Name in 20-sim model	Functionality
Hardware boundary	*OUT_hw_outofbound*	Checks if the unit is out-of-bound by looking at the limit-switch signals.
Software boundary	*OUT_sw_outofbound*	Checks if the unit is out-of-bound by looking at virtual-set boundaries. These virtual boundaries are derived from the limit-switch positions, but their range is defined narrower using some clipping value.
Motorfault	*OUT_motorfault*	Checks if the unit's motor experiences a fault, which describes an undervoltage situation. If there is an undervoltage situation for 10 seconds, the motor fault is reported.
Magnetfault	*OUT_magnetfault*	Checks if the magnet is indeed active when it is turned on. If there is no magnet activity after 10 seconds, the magnet fault is reported.
Emergency button	*OUT_emergencybutton*	Checks if the emergency button signal is active.

C.4 Physical connections

This section describes the physical connections that are present in the embedded-control-sofware architecture realisation (Figure C.9). In this figure, a complete overview is given of the mapping of the Production-Cell-Units' peripherals to the icoBoards' PMOD-connectors. On each board two PMOD-connectors cover the inputs/outputs associated with one Production-Cell Unit (Section B.2). A functional partition is made for the PMOD-connectors. PMOD-connectors 1 and 2 are responsible for one Production-Cell Unit, while PMOD-connectors 3 and 4 are responsible for the other Production-Cell Unit. Due to project constraints (pertaining cabling) and the design of the PCB interface board (Appendix D), some of the inputs are assigned to one unit, but belong to the other unit on the board from a functional perspective however.

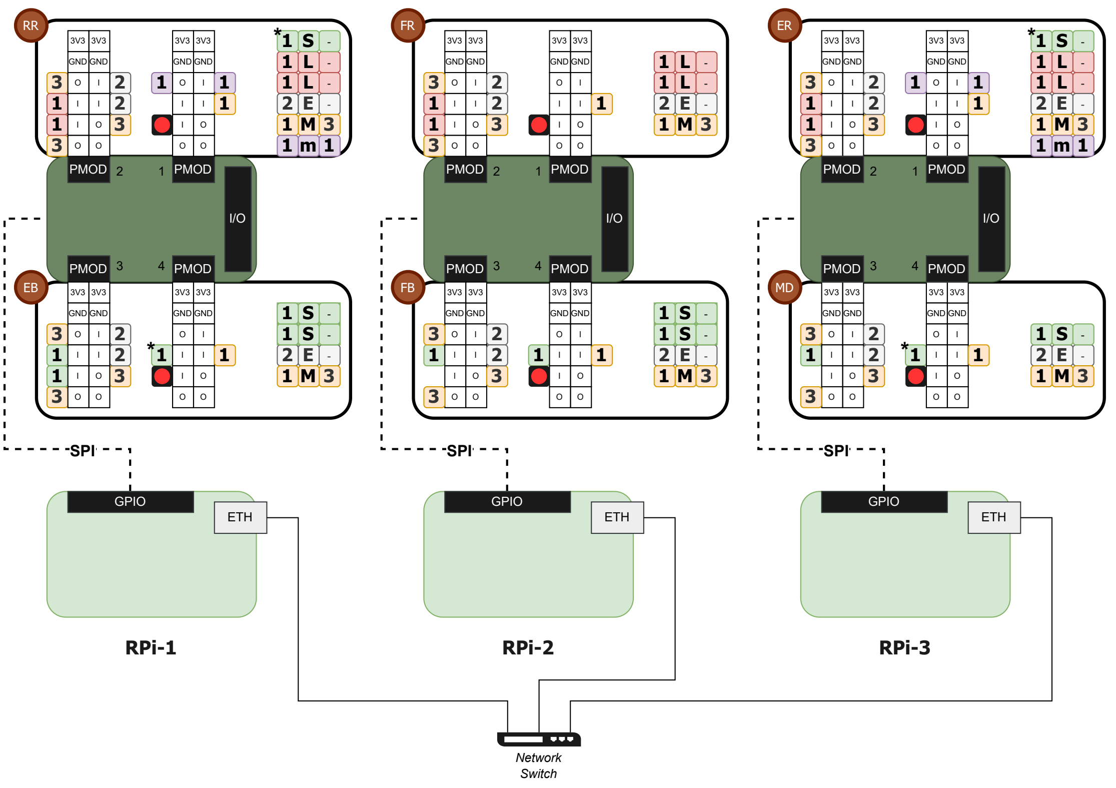


Figure C.9: Physical connections of the embedded-control-software architecture with respect to an icoBoard-extended Raspberry Pi board. For definitions of elements in this figure see Figure A.2, Figure A.4 and Figure B.2. The asterisk denotes that the peripheral input or output is owned by the other unit on the board from a functional perspective.

C.5 Additional diagrams and plots

In this section additional diagrams and plots are shown that are relevant to Chapter 5. In Figure C.10 and Figure C.11, descriptions are given about the datasets generated from the test bed. In Figure C.12, the plot that has been shown in Figure 5.2 is replotted with a different value for the deadline-misses-per-time criterion. In Figure C.13, the deadline-miss evolution per embedded-control-software-architecture component during the test scenario is shown. In Listing C.12, the code that is used for measuring the timing of the motion-controller task is shown.

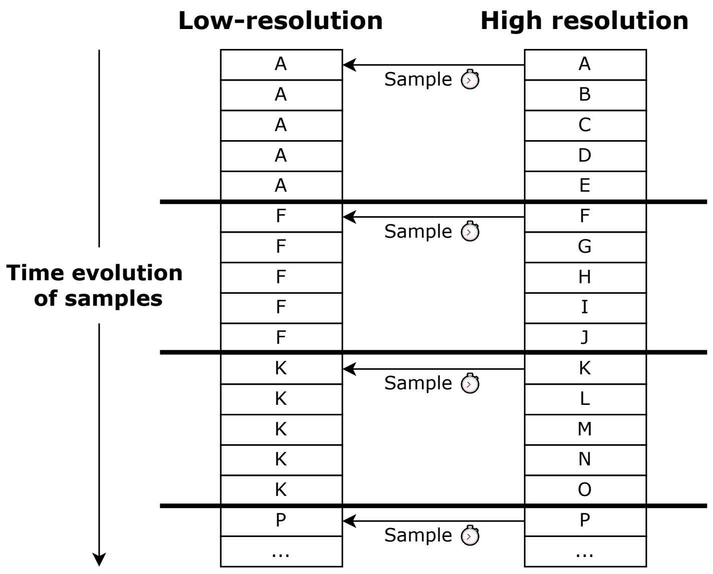

Low-resolution
High resolution
Time evolution of samples
Sample
A A A A A F F F F F K K K K K P ...
A B C D E F G H I J K L M N O P ...

Figure C.10: Symbolic representation of how low-resolution data is related to high-resolution data

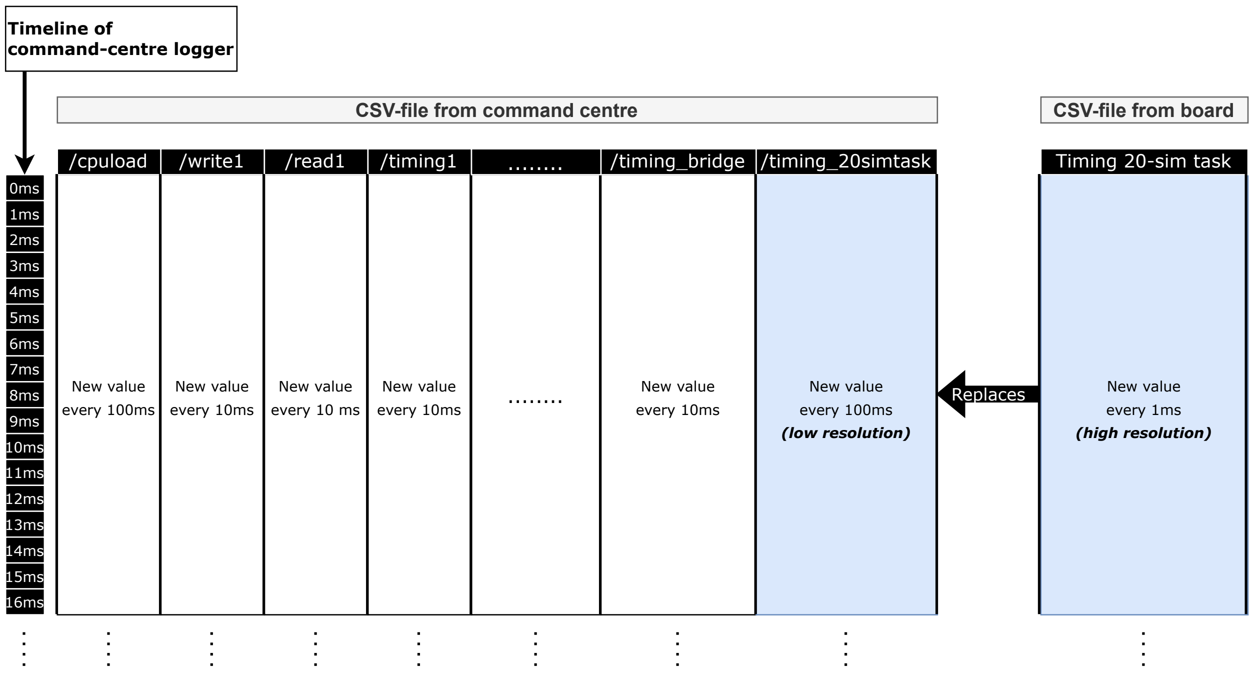


Figure C.11: Overview of the CSV-based datasets of Table 5.1. The CSV-based datasets from the command centre and the Raspberry Pi board are merged based on the scheme of Figure C.10.

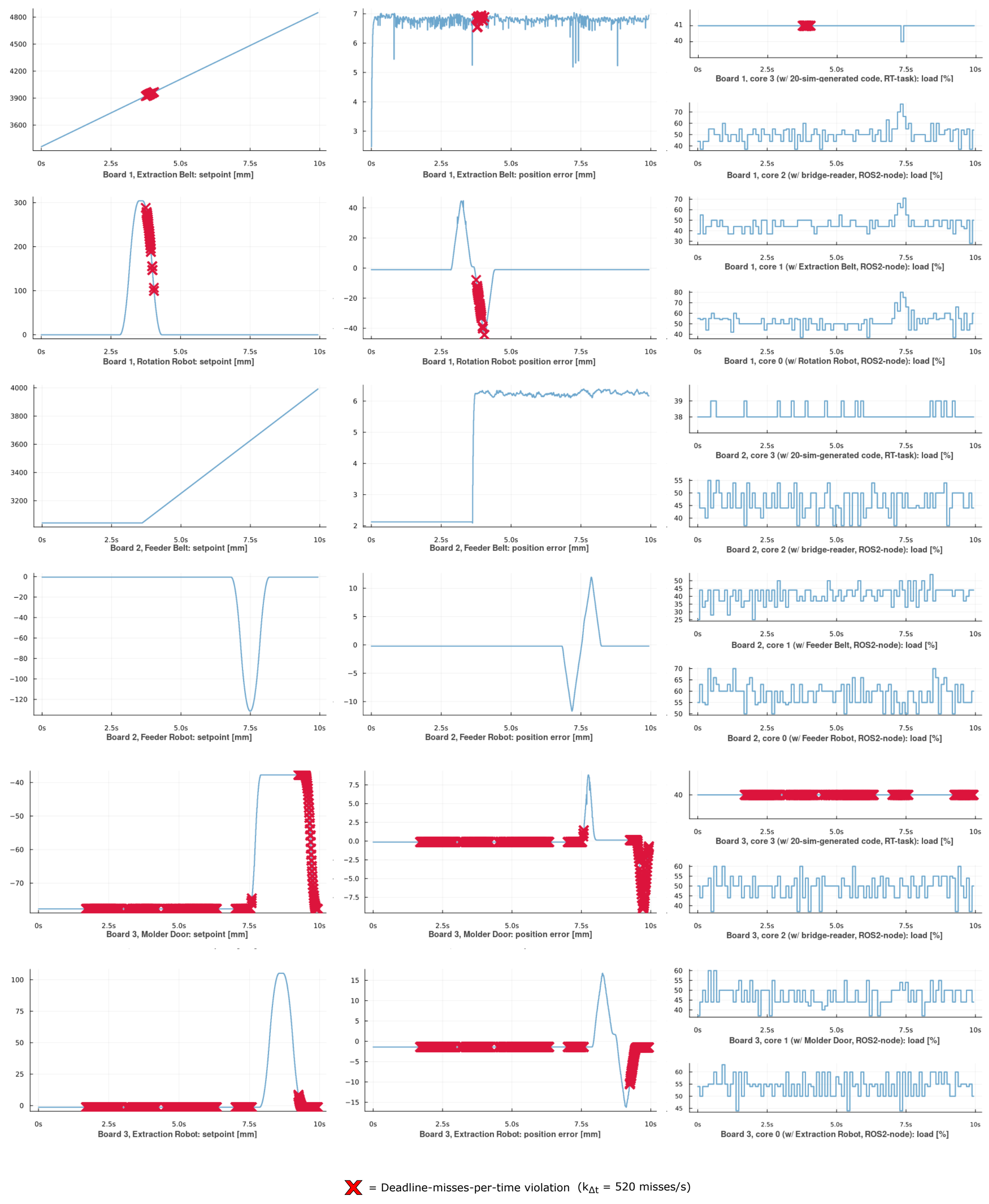


Figure C.12: Overview of the control performance and load balancing during one round-trip time of a block. See Section 5.4.

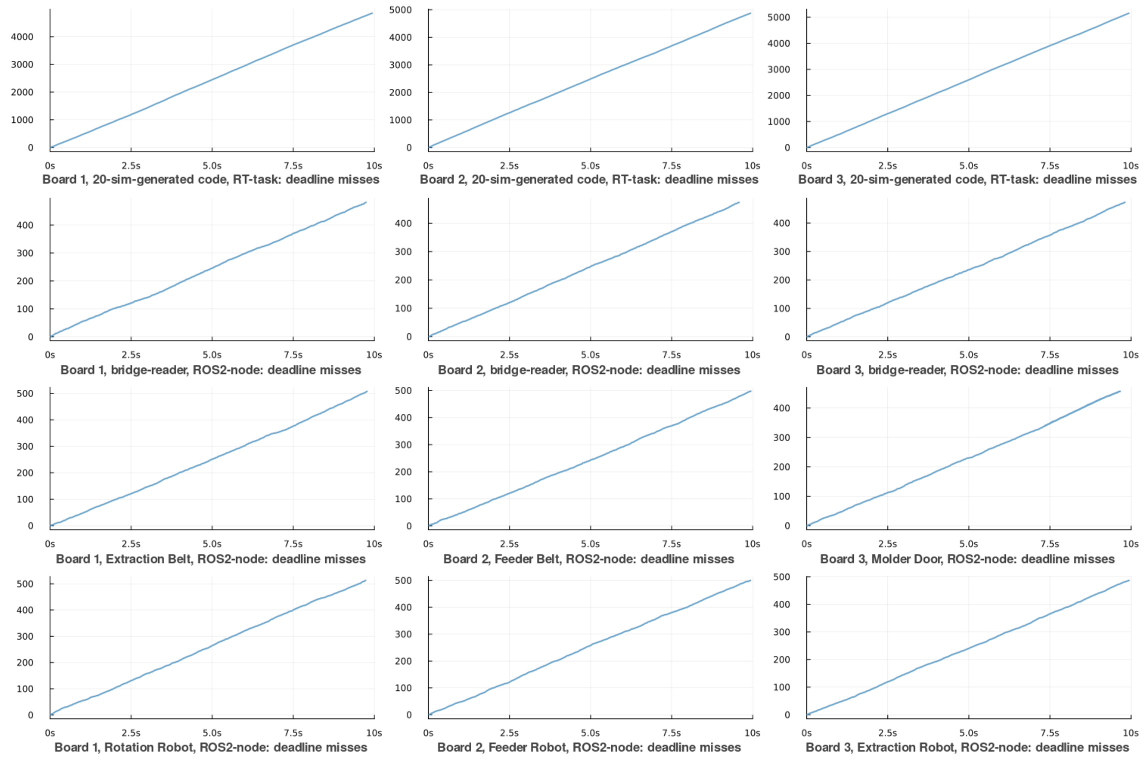


Figure C.13: The evolution of deadline misses per component during one round trip time of a block. A deadline miss is considered a sample which has positive jitter. See Section 5.4.

```
while (1)
{
    t0 = getTimestamp();                              // --------| t0
                                                      //         |
    for (int i = 0; i < receiveArraySize; i++)        //      receive
        receiveClass[i]->receive(u);                  //         |
                                                      //         |
    t1 = getTimestamp();                              // --------| t1
                                                      //         |
    simclass20->Calculate(u, y);                      //     calculate
                                                      //         |
    t2 = getTimestamp();                              // --------| t2
                                                      //         |
    for (int i = 0; i < sendArraySize; i++)           //       send
        sendClass[i]->send(y);                        //         |
                                                      //         |
    t3 = getTimestamp();                              // --------| t3
                                                      //         |
    logTimingToXDDP(receive, calculate, send);        //         |
                                                      //         |
    // Wait until timer raise                         //         |
    sigwait(&set, &signum);                           //       idle
                                                      //         |
    t4 = getCycleTime();                              // --------| t4 ==> cycle

    logTimingToXDDP(cycle);

    // This array is printed to CSV at program exit
    logTimingToArray(receive, calculate, send, cycle);
}
```

Listing C.12: Inline-code view of the code measuring the execution cycle in Figure 5.5

C.6 Running the demo

In this section, the prerequisites and step-by-step plan to run the Raspberry-Pi-based demonstrator are described. Also, a troubleshooting guide is given. For the mentioned Production-Cell-Unit names in this section, see Figure A.2.

Prerequisites

Before running the system, check the following points:

- The Raspberry Pi boards are on the same local network. This is done by connecting the boards to the network switch via Ethernet cables.
- A command centre (laptop or PC) is connected to the same network of the Raspberry Pi boards. This command centre can be connected to the network switch via an Ethernet cable as well.
- No blocks are present at the moulding section such that a situation could occur that two blocks are pushed against the *Molder Door*.

Step-by-step plan

1. Plug in the connector of the network switch into an available wall socket.
2. Plug in the connector of the black power strip of the Production Cell into a wall socket. This power strip is responsible for powering the whole Production-Cell setup and the Raspberry Pi boards.
3. Boot up a command centre. In case the PC command centre near the Production-Cell setup is used, see the sticker on the keyboard for the login password.
4. Before running the demo, make sure that the Raspberry Pi boards are present on the local network. Open up a terminal and check if they give a response by pinging each of the three boards.

```
ping prodcell-rpi1
ping prodcell-rpi2
ping prodcell-rpi3
```

5. To START the demonstrator, run the following commands in a terminal.

```
sshpass -p ramforpresident ssh pi@prodcell-rpi1 '/bin/bash -s <
    ~/demo/start_board1.bash'
sshpass -p ramforpresident ssh pi@prodcell-rpi2 '/bin/bash -s <
    ~/demo/start_board2.bash'
sshpass -p ramforpresident ssh pi@prodcell-rpi3 '/bin/bash -s <
    ~/demo/start_board3.bash'
```

For the PC command centre mentioned in step 3, run the following command instead.

```
. ~/Desktop/demo.bash
```

6. Wait approximately 15 seconds until the homing procedures by the units are finished.
7. Blocks can be placed on top of the belts from this point on. Ensure that there is proper spacing between the blocks. At least 1 cm of spacing between blocks is ideal.
8. To STOP the demonstrator, run the following commands in a terminal.

```
sshpass -p ramforpresident ssh pi@prodcell-rpi1 '/bin/bash -s <
    ~/demo/stop_board1.bash'
sshpass -p ramforpresident ssh pi@prodcell-rpi2 '/bin/bash -s <
    ~/demo/stop_board2.bash'
sshpass -p ramforpresident ssh pi@prodcell-rpi3 '/bin/bash -s <
    ~/demo/stop_board3.bash'
```

For the PC command centre mentioned in step 3, see the stop instructions in terminal.

9. To shutdown the system, unplug the connectors mentioned in step 1 and step 2.

Troubleshooting

- **One of the boards does not respond to pings.** Check if the Ethernet cable is properly slotted into the Raspberry Pi board and/or network switch. It may be the case that it is not fully inserted.
- **A board does not want to respond via pings, but the LEDs indicate that it is powered.** It is possible that the SD card of the Raspberry Pi board is not slotted properly. In the worst-case scenario, however, it is possible that the board's SD card and/or its system image is broken. As a last resort, one can reflash the SD card of the Raspberry Pi board with a backup image. For this, see the electronically-handed-in files.
- **The belts (*Extraction Belt, Feeder Belt*) started moving initially, but now suddenly stopped moving.** A watchdog timer has been implemented for the belts. If the belts have not detected a block moving through the block-detecting sensors for ten seconds, they stop moving. To trigger movement again, one of these sensors must be triggered.
- **One of the robots (*Rotation Robot, Extraction Robot, Feeder Robot*) does not want to move.** Inspect if one of the limit switches of the robot has been hit. For the magnet-based robots (*Rotation Robot, Extraction Robot*) this can be visually inspected, since the red magnet LED on top of the robot will become active when a limit switch has been hit. If a limit switch been hit, gently move the robot away from the limit switch to allow the robot to move to its necessary position.
- **The robot does not make contact with its limit switches, but it still does not move.** It may be the case that the robot violates virtual boundaries which have been placed nearby the limit switches. Try to move the robot further away from the limit switches and see if it then moves to its necessary position.
- **The magnet-based robots (*Rotation Robot, Extraction Robot*) after transporting a block look like they are stuck.** After transporting a block, these robots expect that the block arrives at the target location. If no block is detected, it continues to await the signal that the block has arrived. To allow these robots to continue their operations, trigger the block-detecting sensor at the target location.
- **No matter what is tried, one of the units still does not want to move.** It is possible that a fault has occurred with respect to the peripherals of the unit. Either an undervoltage situation is occurring for its motor, or no magnet activity is found after the unit tries to turn on its magnet (Table C.3). Alternatively, it is possible that the Raspberry Pi, icoBoard and/or PCB board do not function properly. In this scenario consult a technician.
- **Debugging has to be done, but is unclear how to do so.** Two debugging approaches for the setup are described hereafter.
  1. The first approach that should be taken is to read out the ROS2 topics that describe the motion-control-component measurement data of the units. When the PC command centre nearby the Production-Cell setup is used, a ROS2-topic monitoring service (Venkatraman, 2023) is started when running the demonstrator script `demo.bash` (see step 5 of the step-by-step plan). After clicking the hamburger menu at the top-left of the monitoring service and obtaining a list of active ROS2 topics, click the topics denoted as */**_RD* in this list to read out the motion-control-component measurement data of a unit. These values describe the state of the peripherals and motion-control component. A value denoted as *debug* can be found among these values. This value denotes the current state of the safety layer of the unit's motion-control component. The fractional part of this value should be checked in case a unit does not do its operations: a value of 10X.1 represents that no error is detected, while a value of 10X.2 represents that an error is detected.
  2. The second approach that could be taken is to debug the signals coming from and going to the Production-Cell setup with the probing points of the PCB interface board mentioned in Section D.1. Use the appropiate tools to do so.

D PCB interface board

This chapter describes the design of the PCB board which interfaces the PMOD-connectors of the icoBoard (Figure B.1) with the 50-pin header of the Production-Cell switch boards (Figure A.7). In Section D.1, the schematics and design considerations of the PCB board are described. Related to this, in Section D.2 the chosen safety circuit of Section 3.7 is illustrated.

D.1 Design

For the design of the PCB board, the following design considerations are present:

- Existing cabling must persist to keep the first demonstrator running. This constraint means that definitions of the 50-pin connector cannot be altered, and that the PCB design must adhere to the existing definitions.
- The same signal template is used per two PMOD connectors (Section C.4). In Figure D.1, a limit-switch-based Production-Cell Unit (*Rotation Robot, Feeder Robot, Extraction Robot*) is denoted as *A_***, while an optical-sensor-based Production Unit (*Extraction Belt, Feeder Belt, Molder Door*) is denoted as *B_***.
- It has the Raspberry-Pi-board format such that it can be stacked with the other embedded boards.
- For debugging purposes, it has probing points for signals coming from and going to the Production Cell's peripherals.

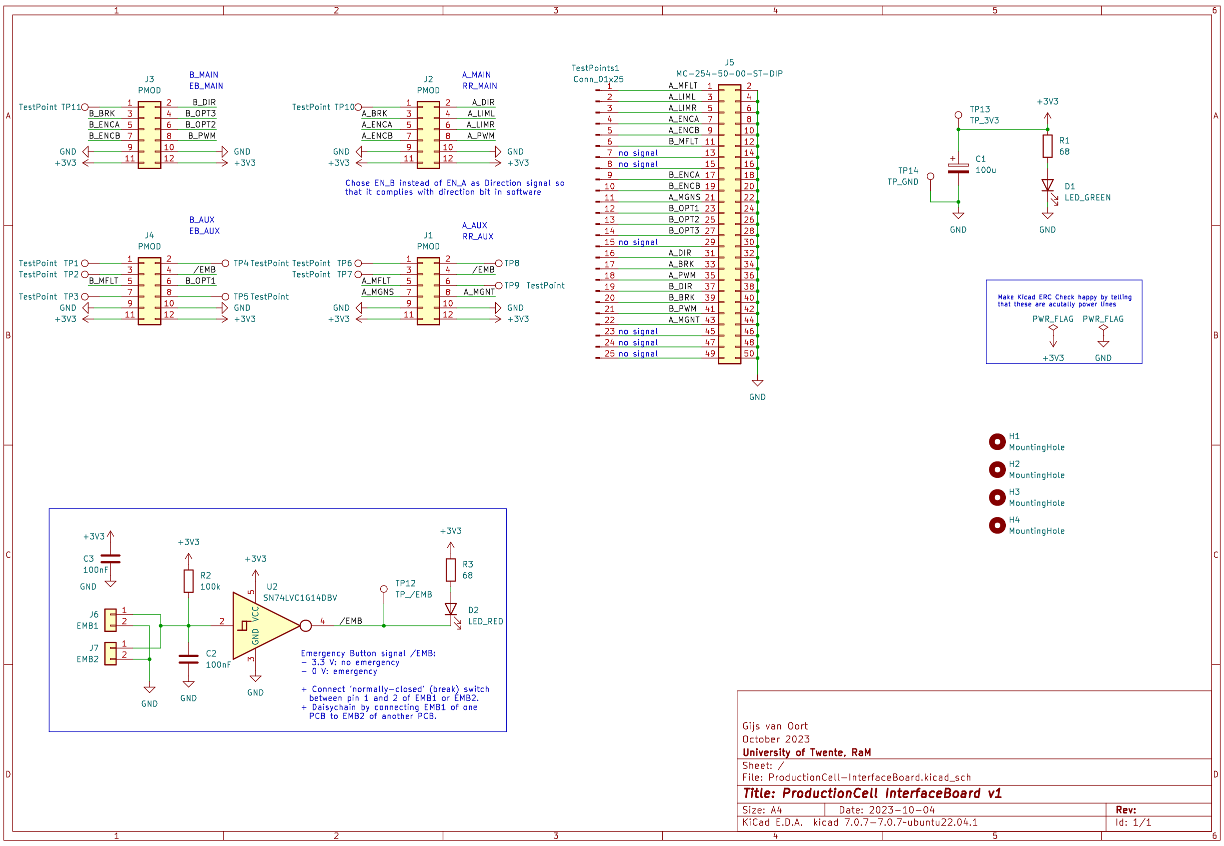

J3 PMOD
B_MAIN EB_MAIN
J2 PMOD
A_MAIN RR_MAIN
J4 PMOD
B_AUX EB_AUX
J1 PMOD
A_AUX RR_AUX
TestPoints1 Conn_01x25
J5 MC-254-50-00-ST-DIP
Chose EN_B instead of EN_A as Direction signal so that it complies with direction bit in software
TP13 TP_3V3
TP14 TP_GND
C1 100u
R1 68
D1 LED_GREEN
Make Kicad ERC Check happy by telling that these are acutally power lines
PWR_FLAG
H1 MountingHole
H2 MountingHole
H3 MountingHole
H4 MountingHole
C3 100nF
R2 100k
U2 SN74LVC1G14DBV
TP12 TP_/EMB
R3 68
D2 LED_RED
J6 EMB1
J7 EMB2
C2 100nF
Emergency Button signal /EMB:
- 3.3 V: no emergency
- 0 V: emergency
+ Connect 'normally-closed' (break) switch between pin 1 and 2 of EMB1 or EMB2.
+ Daisychain by connecting EMB1 of one PCB to EMB2 of another PCB.
Gijs van Oort
October 2023
University of Twente, RaM
Sheet: /
File: ProductionCell-InterfaceBoard.kicad_sch
Title: ProductionCell InterfaceBoard v1
Size: A4
Date: 2023-10-04
Rev:
KiCad E.D.A. kicad 7.0.7-7.0.7-ubuntu22.04.1
Id: 1/1

Figure D.1: Schematic of the PCB board. For more information about the signal definitions see the electronically-handed-in documentation.

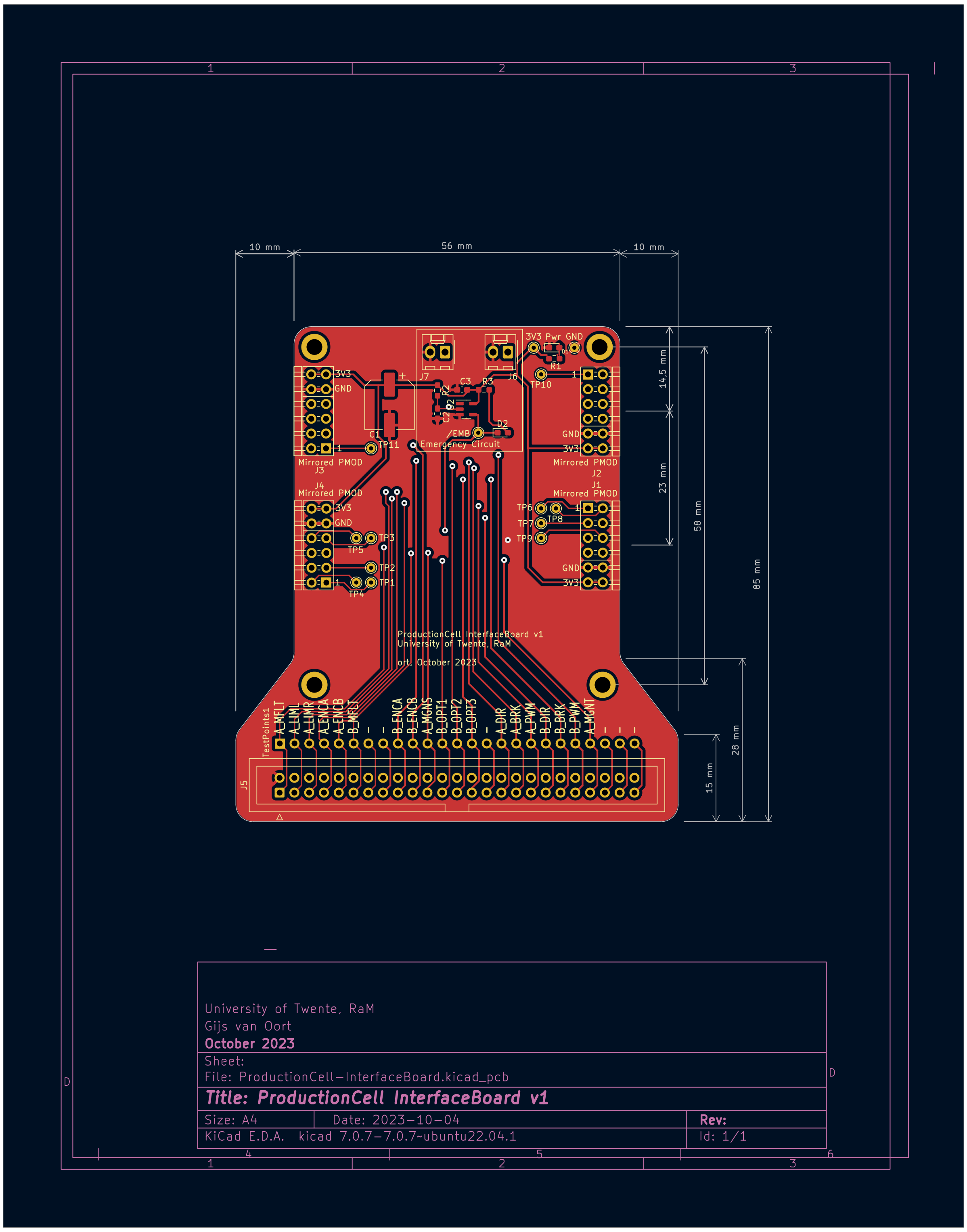


Figure D.2: Front view of the PCB board

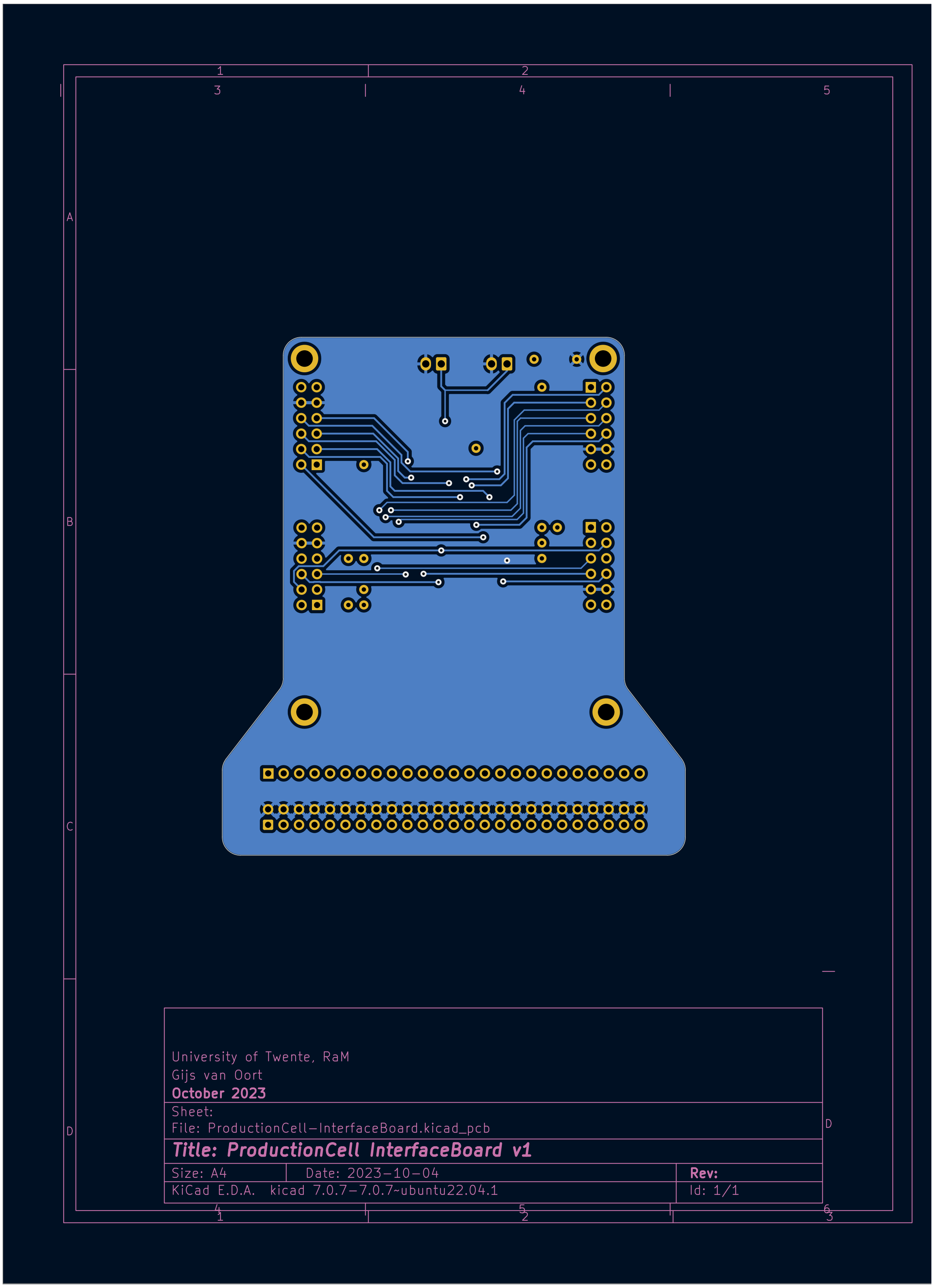

1
2
3
4
5
A
B
C
D
D
University of Twente, RaM
Gijs van Oort
October 2023
Sheet:
File: ProductionCell–InterfaceBoard.kicad_pcb
Title: ProductionCell InterfaceBoard v1
Size: A4
Date: 2023–10–04
Rev:
KiCad E.D.A. kicad 7.0.7–7.0.7~ubuntu22.04.1
Id: 1/1

Figure D.3: Back view of the PCB board

D.2 Safety circuit

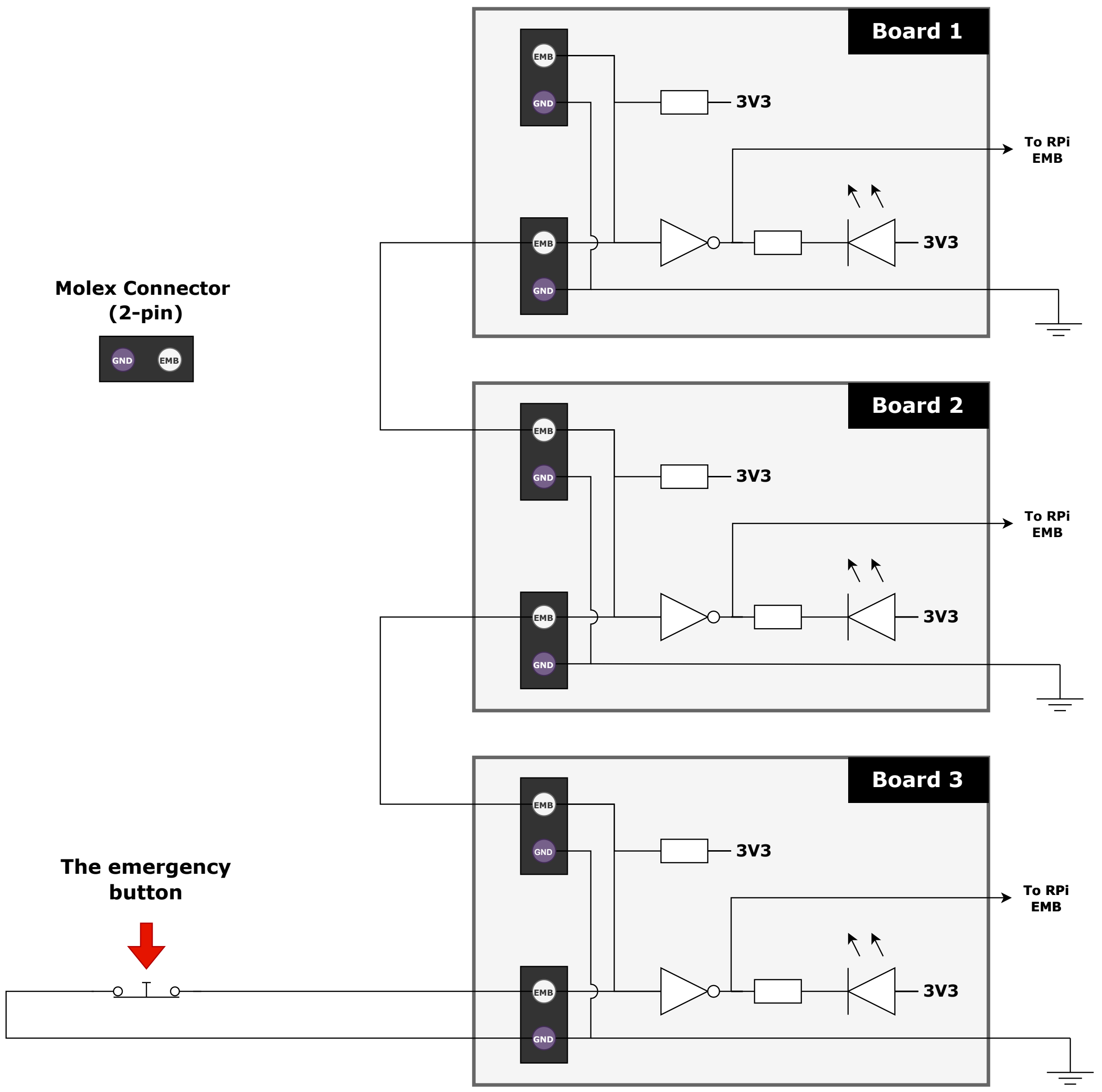


Figure D.4: The safety circuit between the interface boards. See Section 3.7.

E Design-space-exploration scoring system

This appendix chapter is adapted from the work of Hoorweg et al. (2023).

The design-space-exploration scoring is done via the following formula, where S is derived from the score following Table E.1.

$$T_{weighted\_score} = \sum_{categories} (W_{weight} \cdot S_{score\_value}) \tag{E.1}$$

Table E.1: Design-space-exploration scoring system

Score	−−	-	+/-	+	++
Value	-2	-1	0	1	2

For example, if category 1 has a weight 2 and scored ++, and category 2 has a weight 3 and scored −, the weighted score would be:

$$\begin{aligned} T_{weighted\_score} &= 2 \cdot (++) + 3 \cdot (-) = 1 \\ &= (2 \cdot 2) + (3 \cdot -1) \\ &= 1 \end{aligned} \tag{E.2}$$

Bibliography

20-sim (2021), Overview - 20-sim.
https://www.20sim.com/

Van den Berg, B. (2006), *Design of a Production Cell Setup*, Ph.D. thesis, University of Twente, Enschede.

Bezemer, M., M. Groothuis and J. Broenink (2011), Way of Working for Embedded Control Software using Model-Driven Development Techniques, *IEEE ICRA Workshop on Software Development and Integration in Robotics, SDIR VI.*

Boode, A. H. (2018), *On the automation of periodic hard real-time processes: a graph-theoretical approach*, Ph.D. thesis, University of Twente.
https://research.utwente.nl/en/publications/on-the-automation-of-periodic-hard-real-time-processes-a-graph-th

Broenink, J., M. Groothuis, P. Visser and M. Bezemer (2010), Model-driven robot-software design using template-based target descriptions, *ICRA 2010 Workshop*.

Broenink, J. F. and Y. Ni (2012), Model-driven robot-software design using integrated models and co-simulation, in *2012 International Conference on Embedded Computer Systems (SAMOS)*, pp. 339–344, doi:10.1109/SAMOS.2012.6404197.

Die.net (2023), clock_gettime(2): clock/time functions - Linux man page.
https://linux.die.net/man/2/clock_gettime

Groothuis, M., J. Zuijlen and J. Broenink (2008), FPGA based Control of a Production Cell System, CPA 2008, pp. 135–148, doi:10.3233/978-1-58603-907-3-135.

Hofstede, A. (2022), pi4-icoboard · GitLab.
https://git.ram.eemcs.utwente.nl/hofstedea/pi4-icoboard

Hoorweg, H., B. Wesselink and P. Kingma (2023), *Hardware and software design space for a small pan-tilt robot*, Individual report, University of Twente, Enschede.

Huang, J., J. Voeten, M. Groothuis, J. Broenink and H. Corporaal (2007), A model-driven design approach for mechatronic systems, in *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007)*, pp. 127–136, doi:10.1109/ACSD.2007.40, iSSN: 1550-4808.

Icoboard (2016), icoBoard - Main Page.
http://icoboard.org/

improvess (2023), cpp-linux-system-stats, original-date: 2021-04-02T11:44:22Z.
https://github.com/improvess/cpp-linux-system-stats

Kweon, S.-K. and M. Cho (2004), Soft Real-Time Communication over Ethernet with Adaptive Traffic Smoothing., *IEEE Trans. Parallel Distrib. Syst.*, **vol. 15**, pp. 946–959, doi:10.1109/TPDS.2004.59.

Liu, X., X. Chen and F. Kong (2015), Utilization Control and Optimization of Real-Time Embedded Systems, *Foundations and Trends in Electronic Design Automation*, **vol. 9**, pp. 211–307, doi:10.1561/1000000042.

Maljaars, P. (2006), *Controllers for the Production Cell Set Up*, Ph.D. thesis, University of Twente, Enschede.

Meijer, A. (2021), *Real-time robot software framework on Raspberry PI using Xenomai and ROS2*, info:eu-repo/semantics/masterThesis, University of Twente, publisher: University of Twente.
https://essay.utwente.nl/88952/

Ni, Y. (2015), *System design support of cyber-physical systems: a co-simulation and co-modelling approach*, Ph.D. thesis, University of Twente.
https://research.utwente.nl/en/publications/system-design-support-of-cyber-physical-systems-a-co-simulation-a

Raspberry Pi Foundation (2014), Raspberry Pi 4 Model B specifications.
https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

Ridder, L. W. v. d. (2018), *Improvements to a tool-chain for model-driven design of Embedded Control Software*, info:eu-repo/semantics/masterThesis, University of Twente, publisher: University of Twente.
https://essay.utwente.nl/77036/

ROS (2023a), Different ROS 2 middleware vendors — ROS 2 Documentation: Rolling documentation.
https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Different-Middleware-Vendors.html

ROS (2023b), ROS: Home.
https://www.ros.org/

Sassen, T. (2009), *Floating-point based control of the production cell using an FPGA with Handel-C*, info:eu-repo/semantics/masterThesis, University of Twente, publisher: University of Twente.
https://essay.utwente.nl/59191/

Silberschatz, A., P. B. Galvin and G. Gagne (2014), *Operating System Concepts*, Wiley, ISBN 978-1-118-09375-7, google-Books-ID: 2STYMwEACAAJ.

University of Cambridge (2023), The life of Pi: Ten years of Raspberry Pi.
https://www.cam.ac.uk/stories/raspberrypi

Veldhuijzen, B. (2009), *Redesign of the CSP execution engine*, info:eu-repo/semantics/masterThesis, University of Twente, publisher: University of Twente.
https://essay.utwente.nl/58514/

Venkatraman, D. (2023), ROSboard, original-date: 2019-04-03T07:20:03Z.
https://github.com/dheera/rosboard

Verhaar, K. (2008), *An integrated embedded control software design case study using Ptolemy II*, info:eu-repo/semantics/masterThesis, University of Twente, publisher: University of Twente.
https://essay.utwente.nl/58154/

Vinkenvleugel, J. T. (2022), *Designing an embedded software architecture for a mobile education robot with real-time control on a Raspberry Pi 4 with FPGA-based I/O*, info:eu-repo/semantics/bachelorThesis, University of Twente, publisher: University of Twente.
https://essay.utwente.nl/92427/

Vos, P.-J. (2015), *Demonstrator combining ROS/TERRA-LUNA*, info:eu-repo/semantics/masterThesis, University of Twente, publisher: University of Twente.
https://essay.utwente.nl/69403/

Xenomai (2023a), Xenomai 3 :: Xenomai 3.
https://v3.xenomai.org/

Xenomai (2023b), Xenomai: Real-time IPC.
https://www.cs.ru.nl/lab/xenomai/api3/group__rtdm__ipc.html#xddp_label_binding

Zuijlen, J. v. (2008), *FPGA-based control of the production cell using Handel-C*, info:eu-repo/semantics/masterThesis, University of Twente, publisher: University of Twente.
https://essay.utwente.nl/58152/