Ulm University | 89069 Ulm | Germany

Faculty of Engineering, Computer Science and Psychology
Institute of Databases and Information Systems

Concept and Implementation of a Factory Simulation

Bachelor Thesis at Ulm University

Submitted by:
Manuel Göster
manuel.goester@uni-ulm.de

Reviewers:
Prof. Dr. Manfred Reichert

Advisor:
Klaus Kammerer

2017

Revision November 17, 2017

Abstract

Current technological trends, such as cyber-physical systems and the industrial internet of things (IIoT), blur boundaries between software and hardware development. Industrial software systems control whole production factories by using advanced information technology approaches. Hence, the development of such systems requires a tight integration of data and processes between the different software systems, e.g., programmable logic controllers, production planning systems, or enterprise resource planning software.

This thesis provides the concept and implementation of a factory simulation including the development of a programmable logic controller (PLC) application based on PLC programming languages. The PLC application controls the execution of customizable production processes and generates process data, e.g., sensor data, production logs, or alarm events, which can be further analyzed, for example, in condition monitoring applications.

Glossary

BPMN Business Process Model and Notation. A graphical business process notation..

CAN Controller Area Network. A serial bus system, rated as a fieldbus.

CANopen CAN based communication protocol for automation technology.

CRUD Create, Read, Update, Delete.

EEPROM Electrically Erasable Programmable Read-Only Memory.

EtherCAT Ethernet for Control Automation Technology. An Ethernet-based field bus system.

FB IEC 61131-3 Function Block. It is a program organization unit having access to peripheral equipment..

FBD IEC 61131-3 Function Block Diagram. It is a graphical PLC programming language..

FRAM Ferroelectric Random Access Memory.

FUN IEC 61131-3 Function. It is a program organization unit having no access to peripheral equipment..

FUR Furnace. A component of the Fischertechnik factory.

GLV IEC 61131-3 Global Variable List.

GUI Graphical User Interface.

HR High Rack. A component of the Fischertechnik factory.

IC Integrated Circuit.

IDE Integrated Development Environment.

IEC International Electrotechnical Commission.

IL IEC 61131-3 Instruction List. It is a textual PLC programming language and related to assembler..

LD IEC 61131-3 Ladder Diagram. It is a graphical PLC programming language..

LED Light Emitting Diodes.

Modbus TCP Modbus Transport Layer Protocol.

Modbus RTU Modbus Remote Terminal Unit.

OPC UA Open Platform Communications Unified Architecture.

PLC Programmable Logic Controller.

POU IEC 61131-3 Program Organization Unit, being part of the PLC application..

PRG IEC 61131-3 Program. It is the most commonly used program organization unit having access to peripheral equipment and which can be assigned to a task..

Profinet Process Field Network.

RS232 Recommended Standard 232: Standardized serial interface.

SFC IEC 61131-3 Sequential Function Chart. It is a graphical PLC programming language..

SL Sorting Line. A component of the Fischertechnik factory.

SNMP Simple Network Management Protocol.

SRAM Static Random Access Memory.

ST IEC 61131-3 Structured Text. It is a textual PLC programming language that is related to standard programming languages..

SVN Subversion system.

TC Task Configuration.

VG Vacuum Gripper. A component of the Fischertechnik factory.

XES eXtensible Event Stream. A unified and extensible methodology for capturing systems behaviors by means of event logs and event streams.

Contents

1 Introduction

Current technological trends, such as cyber-physical systems and the industrial internet of things (IIoT), blur boundaries between software development and underlying hardware [1]. Industrial software systems are tightly integrated into factory processes. Hence, knowledge about machine engineering becomes crucial to computer scientists. Usually, they do not gain knowledge about processes interacting with real world actuators and electrical machines. Machine engineering knowledge also barely exists among computer scientists. To counteract to these trends, a factory simulation is developed to be a basis for further considerations, as for example process mining and process execution control. Data is generated by the simulation in order to feed higher layer applications with it. As for example, a web visualization of process data of the factory can be realized. Furthermore, it may be used in academic education as a showcase to teach students about processes being close to machines.

The factory simulation is the first step to merge low and high level software components, building on the hardware layer. It is a composition of different soft- and hardware components. Based on a construction kit by Fischertechnik, production processes are illustrated. A programmable logic controller (PLC) is used to control the factories actuators and to react to sensor signals.

The goal of this thesis is to define a concept of the factory simulation and to realize it by connecting a PLC to the Fischertechnik factory that runs a PLC application implementing the technical processes being defined in the concept. The concept is designed to integrate the simulation into a greater system architecture concept.

The remainder of this thesis is organized as follows. Chapter 2 introduces the hardware components of the factory simulation, PLC programming languages, and PLC integrated development environments. Chapter 3 describes the concept of the factory simulation, including use cases, requirements, the software architecture, and technical processes. In chapter 4, the proof-of-concept implementation is explained. Chapter 5 illustrates the conducted code testing, and evaluates the processes and requirement realizations. Chapter 7 concludes the thesis, and gives an outlook on possible extensions.

2 Fundamentals

In the following, general concepts and technologies are introduced, which are necessary for the creation of the factory simulation. At first, the Fischertechnik factory simulation modules, the (PLC) and further hardware are described. Afterwards, some PLC programming languages are introduced, followed by an overview of the integrated development environment (IDE) for PLC code.

2.1 Hardware Components

In this section, main hardware components, being used to realize the factory simulation, are described. The hardware setup includes a Fischertechnik factory construction kit consisting of four stations which all have certain sensors and actuators, such as *light barriers*, *switches*, *motors*, *compressors*, and *valves*. To control these sensors and actuators, each station is connected to an I/O-board. The I/O-board is an interface that translates sensor values and actuator control commands of the controller. The controller is a PLC. It runs beforehand implemented PLC-applications and consists of special purpose hardware and a real-time operating system. In order to send signals to actuators and to receive sensor signals, the PLC has both analog and digital I/O-pins which can be accessed by the application running on it. Moreover, there are certain *communication interfaces* available. For example, a programmer is able to load compiled applications into the PLCs memory via Ethernet. Applications are typically developed in special PLC-IDEs, such as *Codesys IDE*, using certain PLC programming languages. During the execution of a PLC application, the PLC constantly sends current values of program variables to the IDE to support debugging (cf., Figure 2.1).

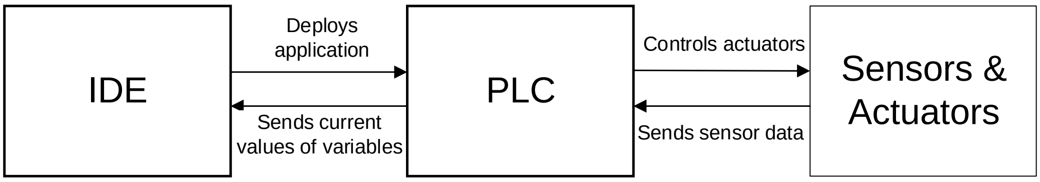


Figure 2.1: Factory Simulation Component Interaction

2.1.1 Fischertechnik Factory Simulation

The Fischertechnik factory simulation consists of the following *stations*: a *high rack (HR)*, *vacuum gripper (VG)*, *furnace (FUR)* and *sorting line (SL)* (cf., Figure 2.2). Every station consists of different actuators, e.g., *s-motors*, *encoder motors*, *compressors*, *pneumatic cylinders*, *magnetic valves*, and *LEDs*. Each of these actuators run with 24V DC. S-motors and encoder motors can move in both directions. Additionally, encoder motors send encoding signals during movement. Thereby, three impulses per rotation of the motor shaft are sent while the encoder motor is rotating. The source of pneumatic compressed air are compressors which are realized by *membrane pumps*. Each compressor can create an overpressure of 0.7 bar. To get depression in order to suck in a workpiece, the pneumatic cylinders use overpressure, created by compressors, and magnetic valves. Light barriers are realized with LEDs, that are positioned on the opposite side of photo-transistors. To be more precise, if the light hits the photo-transistor, it will transmit electricity. Further sensors, being provided by the Fischertechnik construction kit, are switches, which are used in many stations to detect the positions of movable parts, e.g., the vacuum gripper.

The Fischertechnik stations consist of different machines. HR consists of a *high rack*, a *moveable tower* with a *cantilever* and a *conveyor belt*. The high rack has nine places for boxes, which can hold one workpiece each. The three axes movable tower has one switch per axis, which will be pressed if the tower is in its default position at the corresponding axis. The tower stores workpieces and takes them out of stock by picking up boxes, and carrying them to the conveyor belt to put them down. The conveyor belt

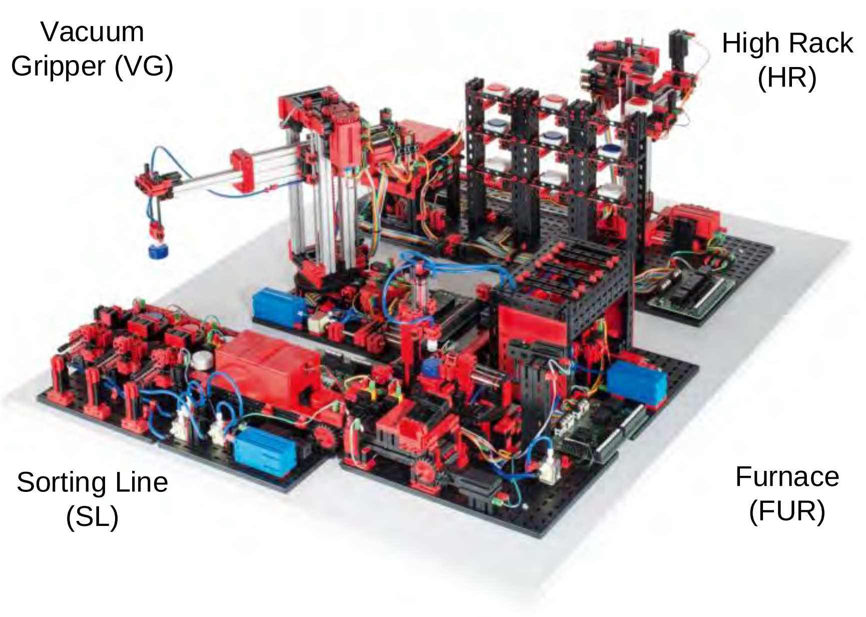


Figure 2.2: Fischertechnik Factory Simulation

can be moved in both directions (forward/backwards) and comprises two light barriers — one at the beginning, and one at the end of itself. Thus, the position of a box at the conveyor can always be determined.

The station VG is a vacuum gripper being mounted onto a three-axis tower. In order to take a workpiece out of stock, the vacuum gripper is moved to the conveyor belt of the HR. The tower can be rotated both clockwise and counterclockwise for about 300 degrees. Furthermore, it can be moved up and down, as well as forward and backwards. By using a compressor and a valve, it is possible to create a vacuum to suck in a workpiece in order to stick it to the gripper. A workpiece can be dropped in place by opening the valve. Similar to the station HR, VG also has three switches stating the default position of one axis each, when pressed.

The VG is able to drop workpieces onto a wagon at the station FUR. In order to detect, if a workpiece rests on the wagon, a light barrier is mounted onto the wagon. The latter can be driven inside the furnace by a simple motor in order to simulate burning the workpiece. Beside these machines, FUR consists of a vacuum gripper, which can transport a workpiece from the furnace to a rotary desk. A rotary cultivator is placed there, which simulates milling the workpiece. A pneumatic slider at the end of the station ejects workpieces to a conveyor belt, which transports the workpiece to the sorting line. At the end of this conveyor belt, there is a light barrier. For pneumatic functionality, one compressor, several valves and pneumatic loaders are used.

The fourth station is SL. Its purpose is to determine the color of a workpiece, which can be either white, red or blue. It is directly connected to FUR by a conveyor belt, having a light barrier at its beginning, a color sensor in the middle, a light barrier after the color sensor and three pneumatic sliders at the end, which can eject a workpiece to three different positions. The light barriers determine the position of a workpiece on the conveyor. Furthermore, the conveyor comprises a switch, which is pressed after every step of the conveyor. Similar to station FUR, one compressor, several valves and pneumatic loaders are installed for pneumatic functionality.

All presented sensors and actuators are controlled by a PLC.

2.1.2 Programmable Logic Controller (PLC)

A *programmable logic controller (PLC)* is a computer that consists of a processor, memory and I/O components [2]. The CPU executes an application that alters the output memory depending on the input signals. Beside that, it includes communication interfaces, such as field buses or Ethernet to communicate with PCs or external hardware modules. Typically, PLCs have a modular structure. Thus, they can be extended after deployment to comply with changing requirements, e.g., by additional I/O-pins. PLCs are primarily used in industrial environments where specific tasks have to be executed periodically — typically within a few milliseconds. Special purpose PLCs comprise additional co-processors to compute complex functions. Often, a PLC is just one component within an automation system, beside sensors, actuators and other control and feedback control systems [3]. In contrast to hard-wired relay-controlled systems, PLC controlled systems are microprocessor-controlled systems that are easily changeable by reprogramming the application code [4]. Since no rewiring is needed, PLCs are flexible, compact, fast and cost-effective control systems.

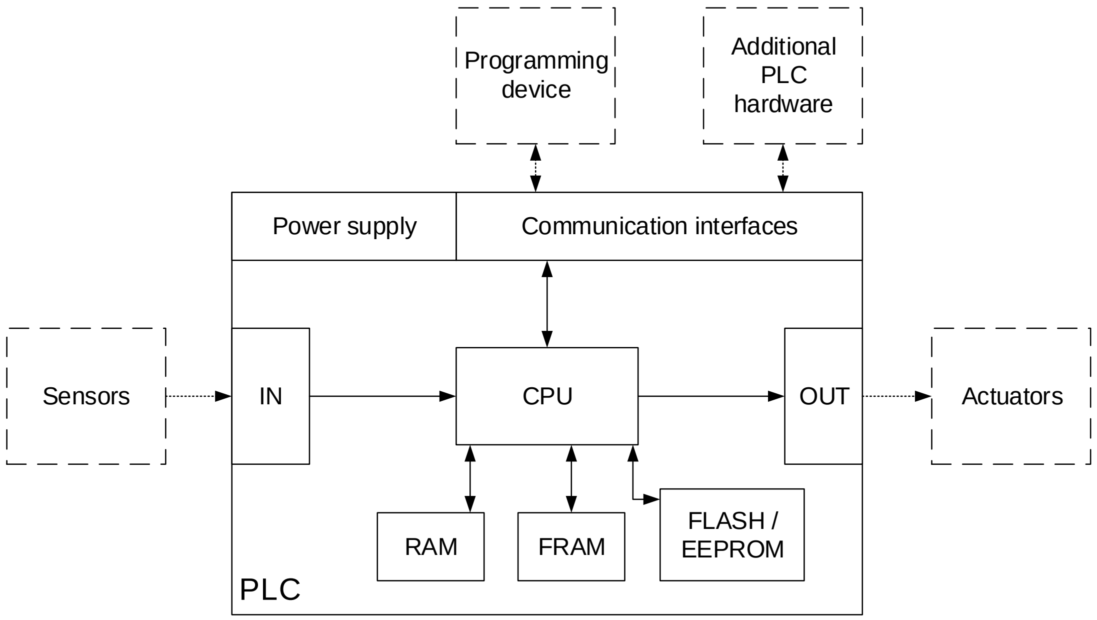


Figure 2.3: PLC Hardware Structure [4]

A PLC consists of the following elements all being typically conducted by a power supply of 24V DC: CPU, RAM, FRAM, FLASH/EEPROM, I/O- and communication interfaces (cf., Figure 2.3). The CPU contains a microprocessor and executes application code. Depending on input signals, the CPU updates outgoing values. The memory units contain different information. While currently running applications are stored in a volatile RAM, the operating system and boot applications are stored in non-volatile EEPROM or FLASH storage. FRAM can persistently store certain user application values, which will not be deleted if power supply gets disconnected. It also offers the same speed as SRAM [5]. Input and output interfaces receive signals from sensors and send signals to actuators. For example, an input device can be a photo-transistor of a light-barrier, sending a discrete signal whether it is interrupted (no voltage) or not (certain voltage, typically 24V). The input section (IN) decodes discrete signals into digital signals, being either zero or one (cf., Figure 2.3). A PLC may have analogue inputs as well, e.g. to process the signal of a color sensor, which can be a voltage level representing a certain color. To be more precise, such a sensor sends a voltage proportional to some color.

The communication interfaces are used for communication with programming devices or other additional hardware, e.g., another PLC or control system. Communication interfaces are also used to integrate the PLC into a cyber-physical supervisory system as a PLC typically is part of an automation system [3].

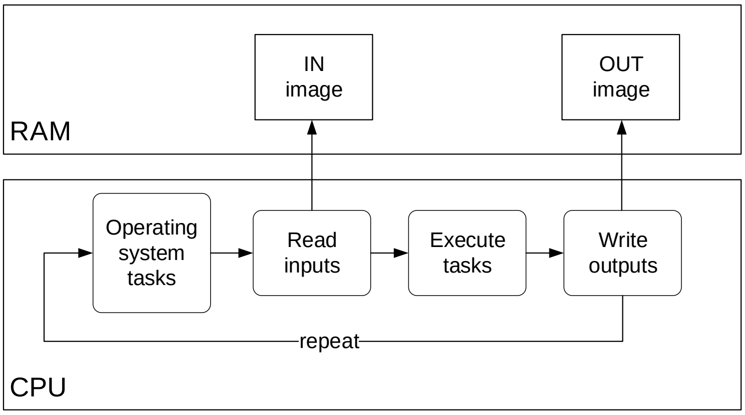


Figure 2.4: PLC Code Execution Cycle (based on [3])

A PLC executes code different to standard PCs: application code is executed in cycles (cf., Figure 2.4). At each start of a cycle, computing time is reserved for different tasks to ensure real time execution, e.g., to preserve operating system tasks. Input signals, being applied to the input interface, are copied to a specific part in RAM. Depending on the signals, a PLC executes application code sequentially. The whole application is executed line by line and output signals may be updated, depending on the application. At the end of every cycle, changed values of output variables are written to the OUT image (stored in RAM), which is directly pushed to the outgoing interface. After a cycle is finished, the CPU repeats the steps within the next cycle. One cycle typically takes a few milliseconds.

Within a cycle, multiple tasks may be executed. Tasks can be executed in three different modes: *cyclic*, *time-cyclic* and *event-driven* [6]. *Cyclic* tasks do have the lowest priority and are most commonly used. A task in this mode is executed every CPU cycle. The cycle time states how long one cycle takes. This depends on the application and can vary from cycle to cycle. In contrast, *time-cyclic* tasks have a mean prioritization and are executed periodically, e.g., every minute. If a time-cyclic task is triggered, standard cyclic tasks are temporarily displaced (cf., Figure 2.5).

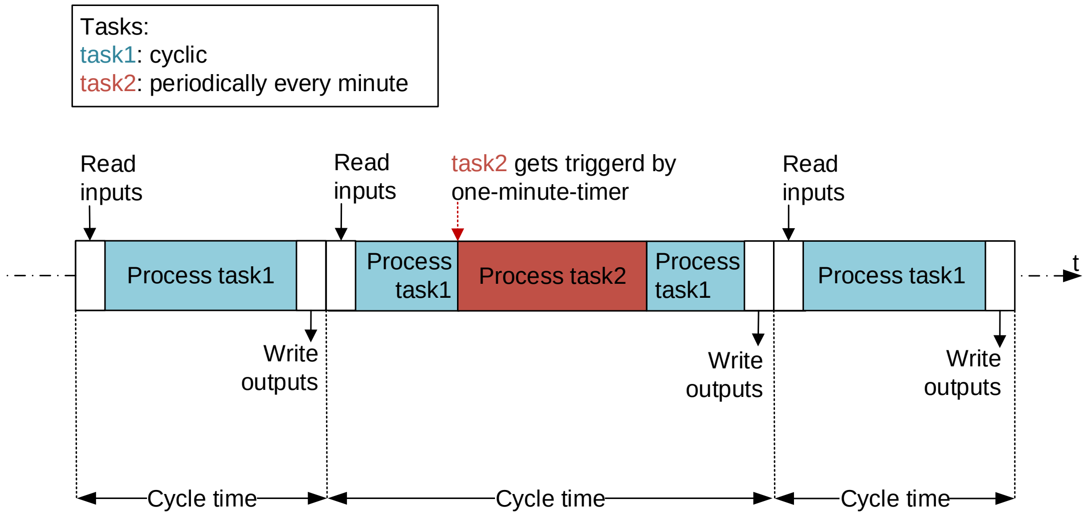


Figure 2.5: PLC Task Displacement

When a higher prior task has been finished, the CPU continues with processing the prior displaced task. This behaviour results in a longer cycle time for the cycle in which the displacement happened. *Event-driven* tasks have the highest priority and are triggered by an interrupt. If an interrupt occurs, a lower prior task is displaced by the event-based task which has been triggered by the interrupt. Which input signals are correctly detected in a cycle depends on the execution duration of all tasks that are executed in this cycle (cf., Figure 2.6).

In order to control the factory simulation, a *Berghof EtherCAT Compact Controller ECC2250* PLC is used [7]. It consists of a 800Mhz Arm Cortex-A9 single core CPU, 256MB RAM, 256MB flash storage, and 100kB FRAM at its side. The PLC has multiple communication interfaces: Ethernet, EtherCAT, CAN, RS232 and supports protocols such as EtherCAT Master, CANopen Master, Modbus RTU, Modbus TCP, SNMP, OPC UA, Ethernet/IP and Profinet. Furthermore, the PLC offers 16 digital inputs, 16 digital outputs, 12 analogue inputs and six analogue outputs. As the amount of digital in- and outputs is not enough for the number of sensor and actuators of the factory simulation,

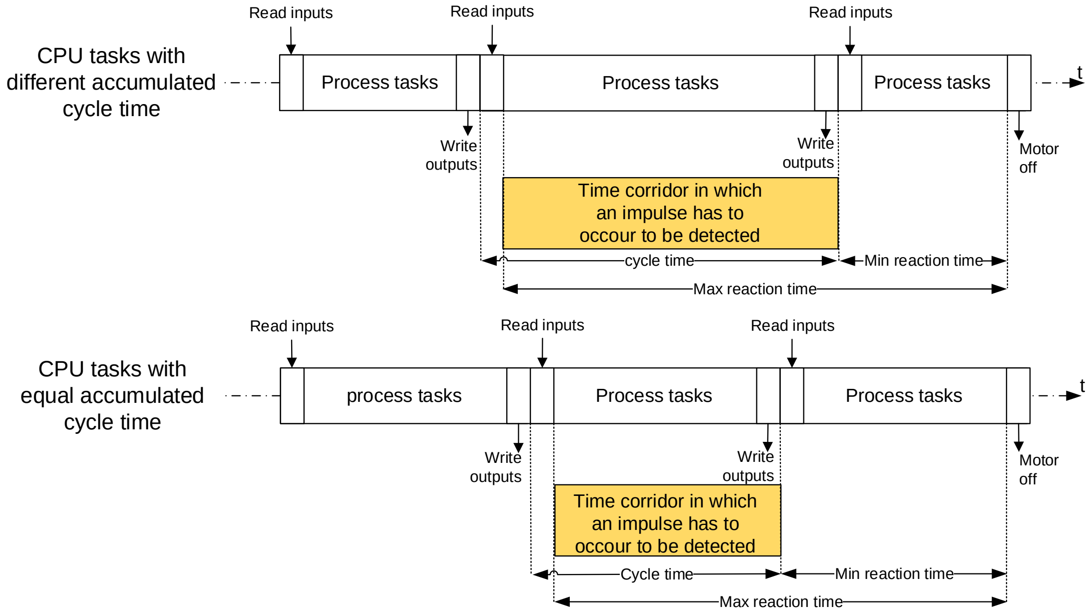


Figure 2.6: PLC Task Execution

the PLC is extended by a *Berghof EtherCAT Compact I/O ECC-DIO 16/16 I/O-extension*. It comprises 16 in- and output pins which can be used by the PLC alongside its own I/O-pins. Thereby, the I/O-extension is connected to the PLC via EtherCAT.

2.1.3 Encoder Board

The encoder motors that are used at the stations HR and VG have a maximum speed of 214 rotations per minute, and send three impulses per rotation of the motor shaft (10,7 rotations per second). Following, the PLC must not have a longer cycle time than 93ms to correctly capture all impulses. Since the PLC also has to be able to capture the moment between two impulses, the maximum cycle time is 46,5ms. Additionally, the factory simulation application runs as a cyclic task, there is no guarantee that a cycle has been finished within 46,5ms.

Initial tests during implementation revealed a potential undersampling of encoder motor signals, e.g., resulting in inaccuracies of VGs movements. According to the Nyquist

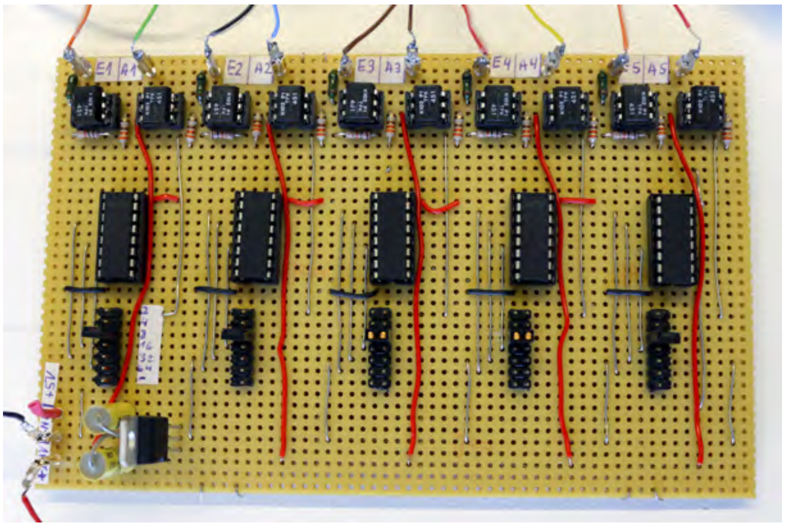


Figure 2.7: Encoder Board – Components

Shannon sampling theorem, undersampling means the sampling of a signal having a sample rate that is below the signals Nyquist rate [8]. The Nyquist rate is defined as twice the upper cutoff frequency of a signal. In order to avoid undersampling, an encoder board was developed in order to reduce the amount of impulses sent by the encoder motors. By using this hardware, the PLC is able to detect more impulses correctly. The board comprises the integrated circuit (IC) 4040, a frequency divider to divide the amount of impulses sent to the PLC by a factor of four. Thus, the maximum cycle time without missing an impulse rises to 186ms. By using the encoder board, there is still no guarantee that the PLC computes every task within this upper bound, but massively shrinks the amount of not detected impulses, as further tests revealed.

A reduction of the amount of impulses sent by the encoder motors to the PLC results in a loss of precision. However, this drawback can easily been deferred to, as there is no need for a higher precision. For example, driving the tower of HR left by one encoder motor impulse what equates to one third of a rotation of the motor shaft, results in a distance of 0.5mm. By using the encoder board, the maximum resolution is divided by four, resulting in 2mm maximum precision — enough for the purpose of the factory simulation. Further tests showed, that a higher impulse division leads to a non acceptable loss in precision.

2.2 PLC Programming Languages

PLC programming languages are standardized by the International Electrotechnical Commission (IEC). It published its third version of the standard IEC 61131-3 in 2012. This document is a guideline for PLC-programming and consists of requirements towards PLC systems and concepts of PLC programming [9]. This includes the PLC software model and programming languages, which are introduced in the following.

2.2.1 Software Model

The software model of IEC 61131-3 defines the whole setting of a PLC as a *configuration*. A configuration includes all resources, tasks, programs and corresponding data [10]. Figure 2.8 shows the components of the IEC software model. If there are multiple PLCs within one control system, each PLC will have its own configuration. The access of a PLC to programs or data of other PLCs is realized by *access paths*, that define which software parts are accessible. A PLC can consist of multiple processing units which are called *resources* in the IEC 61131-3 standard [9]. A resource executes tasks and handles access to physical I/O-pins of the PLC. A *task* has a priority, it is executed either periodically, cyclic or event-driven, and runs one or more program instances. A *program* may include calls of other programs, functions or function blocks. During the execution of a task, all parts of the assigned program are processed once.

Programs (PRGs), *function blocks* (FBs) and *functions* (FUNs) are *program organization units* (POUs) which are implemented in one of the five IEC programming languages [3]. A PLC project consists of several POU components. A POU can be compiled by the compiler independently from other program parts. The three component types differ in their features and use cases [9]. All physical addresses of I/O-pins have to be declared in a program, resource or configuration. There is only one name space for the names of all POUs in a project. Consequently, the name of a POU must be unique. In contrast to other programming languages, POUs do not have any kind of sub routines.

PLC projects consist of multiple POUs which can be bundled into a reusable library. Therefore, the hardware independence of written POUs is important. A POU consists of

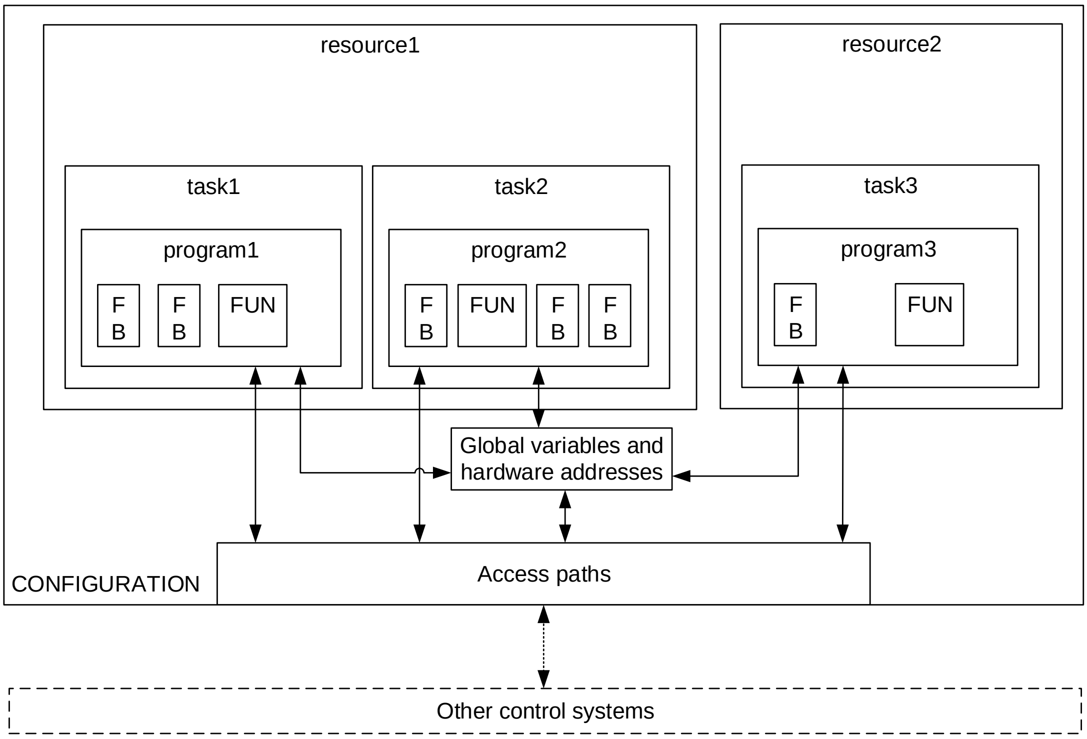


Figure 2.8: IEC 61131-3 Software Model [3]

a declaration and a instruction part, framed by a POU type and name (cf., Figure 2.9). If a function is declared, the data type of its return value has to be declared as well. The declaration part holds the interface variables and local variables. Each variable has a name, a data type and a optional initialization value. In addition, its properties are set, such as battery buffered or I/O pin mapping. The body of a POU holds instructions being implemented in one of the five programming languages.

There are different types of variables with different access rights which may be used in programs, function blocks or functions (cf., Table 2.2.1). VAR are local variables, which can only be accessed inside the POU, in which they are declared. VAR_TEMP are only locally accessible too, but, in contrast to other variable types, their values are reset after the POU call has been finished. All other variables are static, they survive a program call. VAR_INPUT variables are input parameters which will be set "call-by-value" when the POU is called. They can not be written by the called

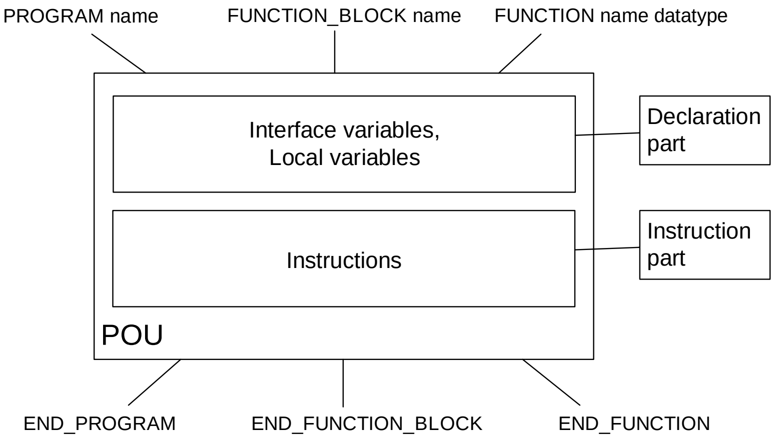


Figure 2.9: Program Organization Unit – Structure [9]

	Access rights		Allowed to use in		
Variable Type	External	Internal	PRG	FB	FUN
VAR	-	RW	x	x	x
VAR_TEMP	-	RW	x	x	-
VAR_INPUT	W	R	x	x	x
VAR_OUTPUT	R	RW	x	x	x
VAR_IN_OUT	RW	RW	x	x	x
VAR_EXTERNAL	RW	RW	x	x	-
VAR_GLOBAL	RW	RW	x	-	-
VAR_ACCESS	RW	RW	x	-	-

Table 2.1: Variable Types, Access Rights and their Use [9]

POU. In contrast, VAR_IN_OUT variables store a pointer, so they implement the "call-by-reference" principle. VAR_OUTPUT variables are only readable available by the calling POU in a "return-by-value" manner. VAR_EXTERNAL are needed to access VAR_GLOBAL of other POUs. Therefore, a variable must have the same declaration in both POUs. This implies that VAR_GLOBAL variables will be accessible by any POU if they have declared an appropriate VAR_EXTERNAL. If a communication between different resources or configurations is necessary, VAR_ACCESS variables are used to define access paths realizing the required communication.

A PRG is a main program that has access to peripheral equipment, global variables and access paths. In contrast to function blocks or functions, it is allowed to address physical

PLC-addresses for I/O activities inside a PRG. Furthermore, a PRG can be assigned to a task.

FBs can not be assigned to a task. They can be instanced. A function block being declared in a POU is visible inside the POU and can be made visible for other POUs when it is declared as a VAR_GLOBAL. Following, other POUs can access the function block by declaring it in VAR_EXTERNAL. Equally to PRGs, FBs can hold input and output variables. PRGs and FBs can produce different results dependent on their local variable values (having the same input signals in multiple cycles), because they are not deleted after a POU call.

In contrast, FUNs do not hold static variables and produce the same result every cycle while having the same input signals. In order to achieve this behaviour, values of function variables are deleted after every function call. This implies that functions do not trigger side effects. The purpose of functions is to extend the set of operations of a PLC. The name of a function is globally valid in a project — thus, it can be called by any POU inside the project. In addition, functions can have multiple input and output variables, but only one return value.

2.2.2 Textual Languages

Two of the five IEC 61131-3 PLC programming languages are textual languages: *instruction list* (IL) and *structured text* (ST). IL is related to assembler and embedded programming, as it consists of simple instructions in each line. IL is often used as an intermediate language onto the other IEC languages are mapped. ST, on the other hand, resembles high-level programming languages with its good readable syntax. In the following, IL and ST are further introduced.

Instruction List (IL)

IL is a line-oriented programming language [9]. Each line consists of one instruction being constituted of an operator and one or more operands. Optionally, each line can have one jump label at its beginning, which can be used to jump to this instruction.

Operands are constants, variables or input parameters of a function. The operator is an IL-operator or a function name. Comments `(* ... *)` are allowed at any position where blanks are allowed. ";" is not allowed at any position. In addition, there is no column formatting. In IL, there is an accumulator — the "current result" — which is not a memory area with fixed length, as it is in other assembler languages, but with dynamic length. The IL-compiler adjusts the accumulator size, dependent on the operands data type. Figure 2.1 shows an example program written in IL, where two variables are multiplied and it is checked whether the result is positive or negative. At first, the variables are declared: in line 2, `op1`, `op2` are declared as integer and are initialized with the value 10. `res` is also an integer, but is initialized with 0 in line 3. `resPostive` is declared as a boolean in line 4 and if it is true, it states that `res` is positive. After the declaration, the actual execution code follows. `j1` and `j2` are jump labels. The second column holds operators while the third column consists of operands. `op1` is loaded into the accumulator (LD), multiplied (MUL) with `op2` and stored (ST) in `res`. GT means "greater than", compares the current result with zero and will store `true` in the current result if it is greater than zero, otherwise `false`. JMPC is a conditional jump that will be triggered if the accumulator is `true`. If `res` is positive, the program pointer jumps to `j2`. Here, `true` is loaded into the accumulator and stored in `resPositive`. If `res` is negative, `false` will be stored in `resPositive`.

```
1   VAR
2     op1, op2: INT := 10;
3     res: INT := 0;
4     resPositive: BOOL := FALSE;
5   END_VAR
6   ...
7   j1: LD     op1
8       MUL    op2
9       ST     res
10      GT     0
11      JMPC   j2
12      LD     FALSE
13      ST     resPositive
14      JMP    j3
15  j2: LD     TRUE
16      ST     resPositive
```

Listing 2.1: IL Code Example

Structured Text (ST)

Structured text is a textual language declared by the IEC 61131-3 standard. In contrast to IL, it is a high-level programming language with a corresponding syntax that allows compact statements, structured clear code and constructs to control the program flow, such as IF conditions, CASE constructs and loops [9]. This programming comfort results in a possible drawback of loosing performance, because ST code is on a higher abstraction level than IL code. ST code consists of instructions that can span multiple lines. Common to other high-level languages, each instruction is separated from each other by a semicolon. Single- and multiline comments can be made in the same way as in IL by using "(*" at the beginning of the comment and "*)" at its end. Unlike IL, ST has no jump labels. Listing 2.2 shows the same program as listing 2.1, but written in ST without declaration part (same as in IL).

```
res := op1 * op2;
IF res > 0 THEN
    resPositive := TRUE;
ELSE
    resPositive := FALSE;
END_IF
```

Listing 2.2: ST Code Example

2.2.3 Graphical Languages

Beside the two presented textual languages, there are three graphical languages: *function block diagram* (FBD), *ladder diagram* (LD) and *sequential function chart* (SFC).

Function Block Diagram (FBD)

FBD is originated in signal processing [9]. The declaration part, where variables are declared, is similar to textual languages: it is separated from the instruction part and can be edited textually or graphically. The instruction part consists of networks which are built up by rectangles and connections between them. Inputs of the rectangles can be variables or constants. Furthermore, there are graphical elements to control

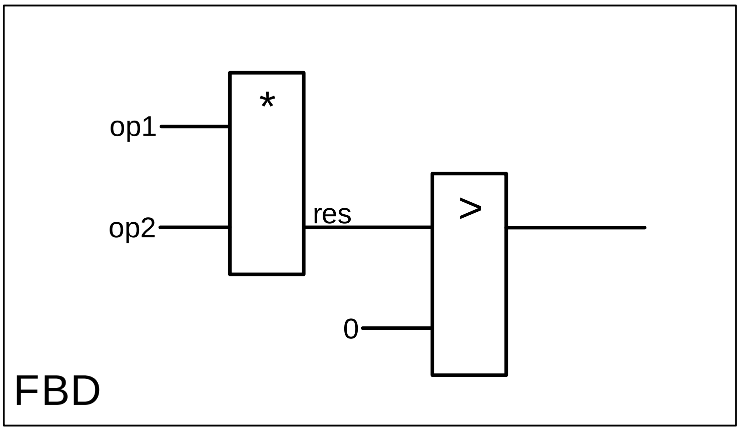


Figure 2.10: FBD Code Example

the execution flow, e.g., jump labels. Listing 2.10 shows an example, written in FBD, implementing the same logic as the example Listings 2.1 and 2.2.

Ladder Diagram (LD)

LD is originated in electromagnetic relay systems and focuses on boolean algebra [9]. Therefore, this programming language is inappropriate to use for simulating a factory and will not be introduced here.

Sequential Function Chart (SFC)

SFC is the fifth programming language defined in standard IEC 61131-3. It is used for structuring complex applications by encapsulating the application in clear, separated steps. Similar to FBD and LD, SFC consists of one or more networks. Each network has certain *steps* which are separated from each other by *transitions*. Each step can be implemented in one of the five programming languages. If a step is written in SFC, it represents a sub-network. Thus, a hierarchical structure can be realized. If a step is active, the underlying code will be executed cyclically, until the following transition condition is fulfilled. A transition condition is a boolean expression and can be implemented in ST, FBD or LD [9]. In addition to the executable code of a step, there may be *actions* belonging to it. Entry actions are executed in the first cycle only, after a step has become active. In contrast, exit actions are executed once, after the following transition condition has become `true`. Standard actions of a step are executed every cycle, as long as the step is active.

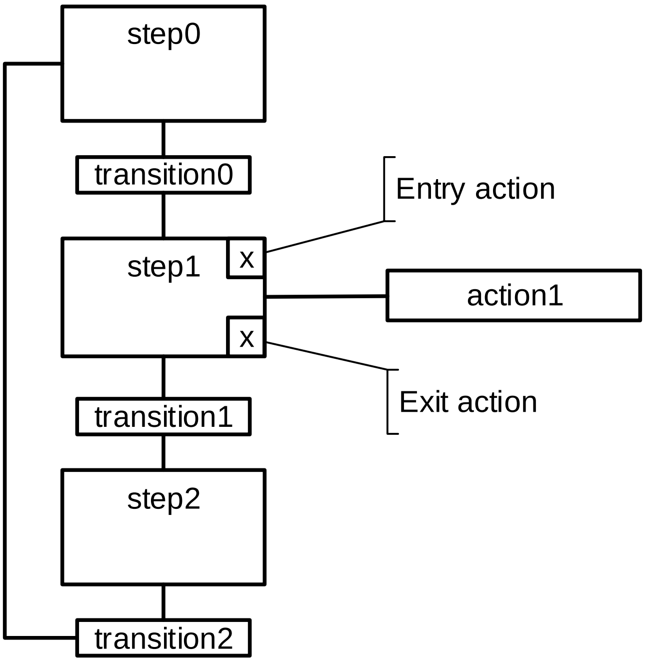


Figure 2.11: SFC Network Elements and their Composition

Figure 2.11 shows the basic structure of a SFC network. If the POU is called, it starts its execution at the start step which is `step0` in the example. `step0` is executed cyclically until `transition0` gets `true`. Next, the entry action of `step1` is executed once. Afterwards, `step1` and its associated `action1` are executed every CPU cycle, until `transition1` gets fulfilled. In this case, the exit action of `step1` is executed once, just before the processor starts the execution of `step2`. `step2` will be finished when `transition2` becomes `true`. Then, the CPU continues with the execution of `step0`.

To influence the control flow, there are alternative and parallel branching, as illustrated in 2.12. Alternative branches require an own entry and exit transition. If `transition0` is fulfilled, `alternative1` will be executed. If `transition1` is fulfilled, `alternative2`

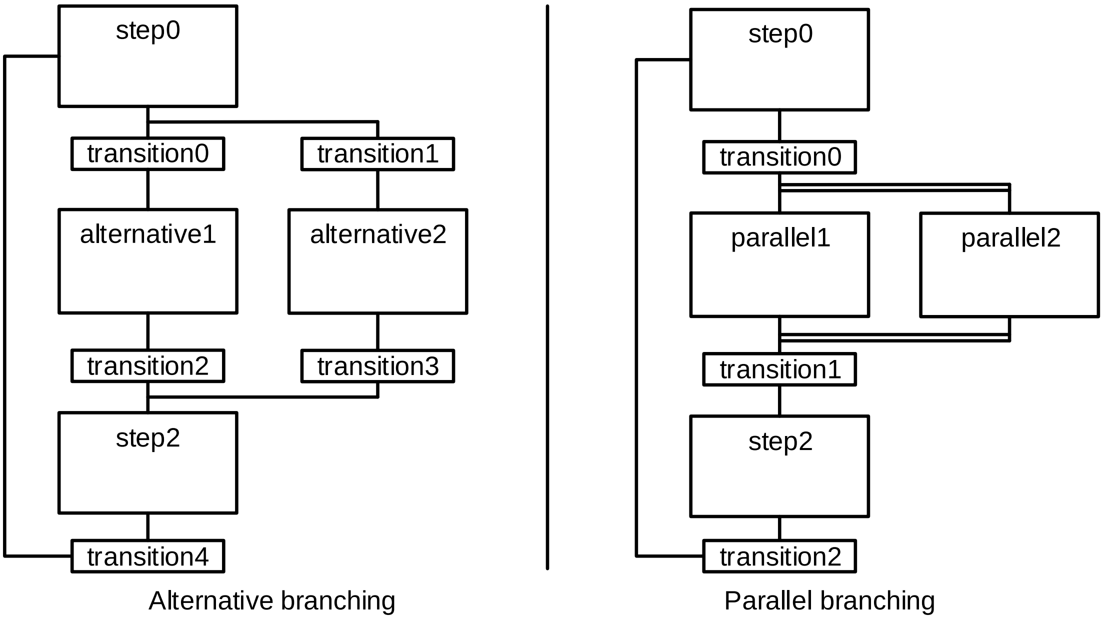
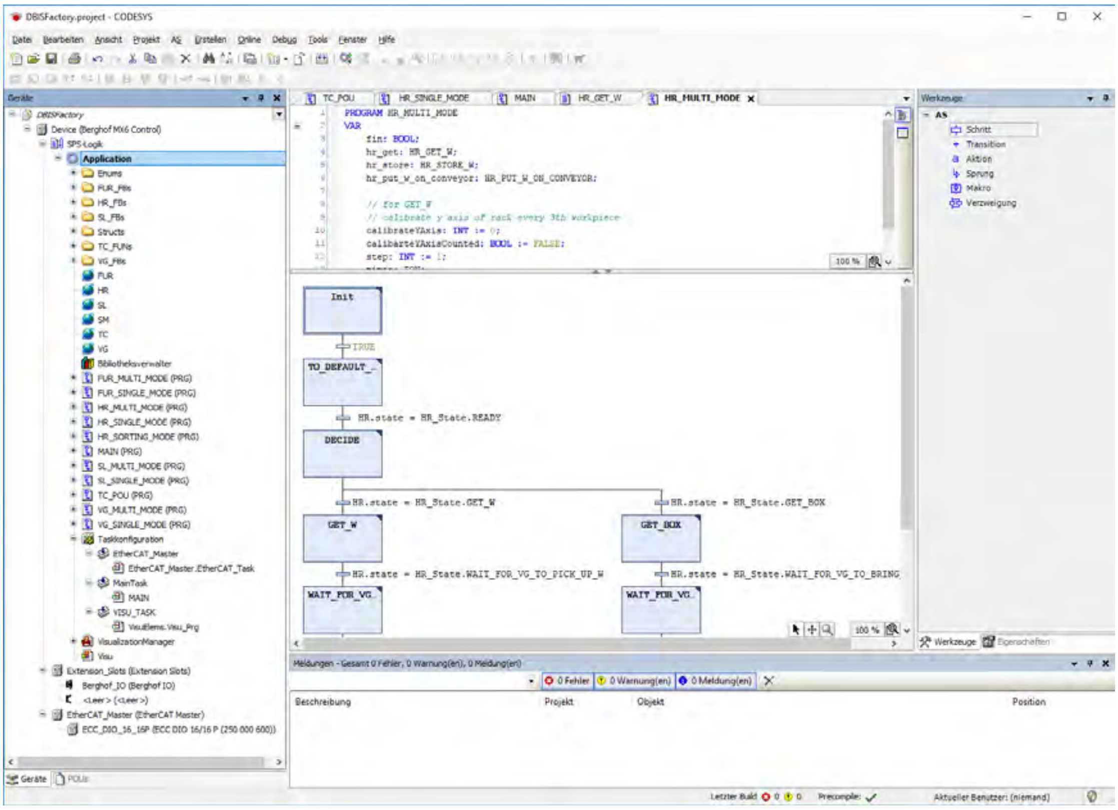


Figure 2.12: SFC Networks – Alternative and Parallel Branching

will be executed. As opposed to this, when `transition0` is fulfilled, parallel branches `parallel1` and `parallel2` are simultaneously started.

2.3 Codesys Integrated Development Environment

The Codesys IDE is a software programming development environment for industrial programming of control and automation technology [11] (cf., Figure 2.13). It supports IEC 61131-3 compliant PLCss. The Codesys IDE includes editors and compilers for all IEC 61131-3 programming languages. It fully supports object-oriented programming, e.g., inheritance, beside functional programming. Furthermore, there are edit and online modes: during edit mode, the implementation is written, and during online mode written code can be executed and debugged on a connected PLC.



```
PROGRAM HR_MULTI_MODE
VAR
    fin: BOOL;
    hr_get: HR_GET_W;
    hr_store: HR_STORE_W;
    hr_put_w_on_conveyor: HR_PUT_W_ON_CONVEYOR;

    // for GET_W
    // calibrate y axis of rack every 3th workpiece
    calibrateYAxis: INT := 0;
    calibarteYAxisCounted: BOOL := FALSE;
    step: INT := 1;
```

Figure 2.13: Codesys IDE – Screenshot

During online mode, the application has been loaded onto the PLC and is running. The IDE shows live changes of variable values as well as a supervised control flow.

For debugging, breakpoints can be set. Code can be changed and deployed without stopping the application.

In edit mode, a range of tools are provided by Codesys to fluently create textual or graphical POUs. Manufacturer dependent hardware can be easily integrated into a Codesys project with the help of configuration files. In a project, an I/O-pin layout can be defined. All Codesys tools can be extended by packages being loaded from the Codesys Store, for example, a SVN plugin or an integration of the OPC UA protocol [12]. Moreover, boot applications can be created. These applications can be pushed onto the PLC, so every time it boots up, it automatically starts to run the boot application.

3 Concept

This Chapter presents the concept of the factory simulation. First, use cases are described, which have to be supported by the factory simulation. Then, system requirements and the software architecture are declared. The chapter concludes with a description of technical processes which shall be supported by the factory simulation.

3.1 Use Cases

The main use case of the factory simulation is to simulate the processing of *workpieces*. A workpiece is stored in the high rack at the beginning of each process. It has certain attributes, e.g. color, position, state, and a *format* that defines how to process a workpiece. Choosable options are to burn it, to mill it, to check the color, to sort it depending on its color, to store it back into the high rack or to eject it at the eject station. There are three modes in which workpieces are processed: *single mode*, *multi mode* and *sorting mode*.

In *single mode*, only one workpiece is processed by the Fischertechnik factory at a stroke. A workpiece is forwarded from station to station while it passes through the factory. If a station has finished processing the workpiece, it will wait until each station has finished processing before starting to process a new one. As this is not an efficient way to process workpieces, a new mode is introduced, solving this lack of efficiency by parallelization: the multi mode.

In *multi mode*, multiple workpieces are processed in parallel. Wait states of all stations are minimized. After a station has finished processing a workpiece, it is immediately checked whether the next workpiece can be processed. However, it is not possible to eliminate all wait states, because a decent amount of dependencies between the stations exist. Thereby, the workload of each station heavily depends on the order of workpieces and their formats. The performance advantages of multi mode compared to single mode are analyzed in section 5.2.

A further use case is to sort workpieces in the high rack. This use case is realized by the *sorting mode*. Using this mode, workpieces can be sorted by a set of simple swap operations. Thereby, each swap operation switches the positions of two workpieces in high rack. This may be used to bring boxes, holding a workpiece, closer to the conveyor during idle time of HR while empty boxes are brought farther afield in order to minimize processing time.

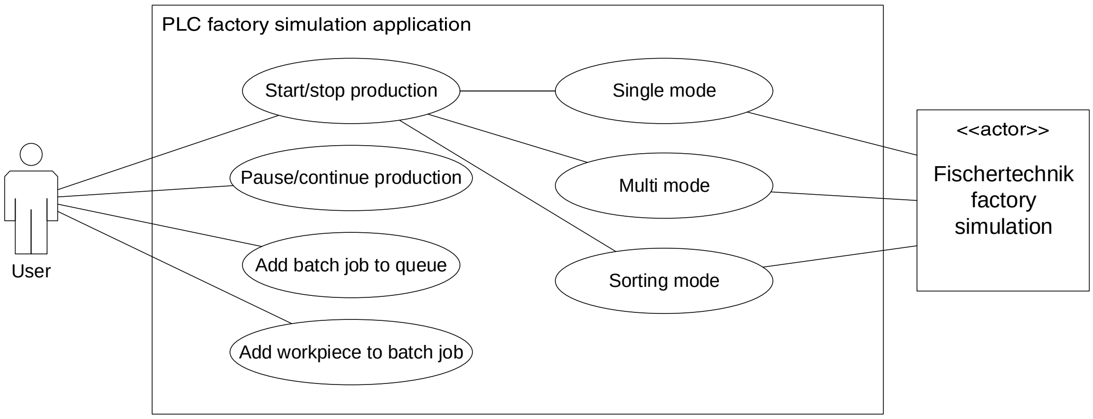

PLC factory simulation application
Start/stop production
Pause/continue production
Add batch job to queue
Add workpiece to batch job
Single mode
Multi mode
Sorting mode
User
<<actor>>
Fischertechnik factory simulation

Figure 3.1: Factory Simulation – Use Case Diagram

Figure 3.1 shows further use cases, being directly connected to the user of the factory simulation application. A user of the application is able to start or stop the workpiece production. The production mode of the factory depends on the workpieces and batch jobs the user has loaded into the systems queue. More details about the data model can be found in section 3.3. At each point in time, the user is able to pause the running factory and to continue the production simulation when paused. In addition to these use cases, it shall be possible to add workpieces to the process queue during production.

This includes the possibility to access data structures from outside the PLC, e.g., via with the OPC UA communication protocol [13].

3.2 Requirements

In this section, requirements towards the factory simulation are presented. They are split into functional requirements and marginal conditions. First, functional requirements are depicted, many of them can be assigned to the stations of the factory.

First, at HR it must be possible to *store* (FR1), to *take out* of store (FR2), and to *sort* (FR3) workpieces. This implies that functions exist to *pick up* (FR4) a workpiece, to *put down* (FR5) a workpiece, to *move the tower* (FR6) and to *move the conveyor belt* (FR7).

VG has to able to *pick up* (FR8) a workpiece at HR and SL, and to *put it down* (FR9) at HR, at FUR, and at the eject point. This implies the need of functions which *move the VG tower* (FR10) and *control the compressor and valve* (FR11) of the gripper.

The FUR station needs functions for *firing* (FR12) and *milling* (FR13) workpieces. For realization, functions are required to *move* (FR14) the wagon, vacuum gripper, rotary desk, and pneumatic slider. The latter ejects a workpiece to SL.

At station SL, the conveyor belt must be movable in order to *transport* (FR15) workpieces through the color checker and to the sorting positions. To *check the color* (FR16) of a workpiece, an *algorithm* to detect the colors by using the color sensor has to be implemented. The pneumatic sliders have to be used to *eject* (FR17) the workpieces to the sorting positions. All these functions must be supported by software components.

Another requirement is that data structures have to be designed in a way that *CRUD operations* (FR18) on them are possible via network access, e.g., by using the OPC UA protocol. This implies that there have to be functions to create, read, update, and delete certain data structures.

Alongside functional requirements, there are marginal conditions (cf., Table 3.2).

Marginal condition	Description
Robustness	The factory simulation application must not react flawed to any user input or sensor signals. To accomplish this, all input signals and user inputs should be validated.
Performance	Due to the fact that the simulation is fed by fast altering sensor values, it must be fast enough, to process them in order to adjust outgoing signals. Besides, the maximum precision of actuator movements depends on the performance of the application.
Security	It has to be made sure that unauthorized access to the system is not possible. To secure input signals, being sent by the user through network, from man in the middle attacks, encryption and certificates will be necessary.
Privacy	As there are no personal information used in the system, the development will not focus on privacy.
Maintainability	Changes should be easily realizable at any time. If something gets broken, it must be possible to adjust the corresponding control flow.
Software portability	Although the application is written in IEC 61131-3 programming languages, and should imply easy portability, it depends on the manufacturer implementation of the standard, whether the application can be easily ported to another target PLC. Given the PLC of Berghof and the IDE Codesys, portability between devices of Berghof should be possible.
Availability	Being a simulation, the application must not be up and running at any time. Nevertheless, if multiple people work on the factory simulation, there always has to be a operational application, and maintenance tasks should be done quickly, so everyone can continue his work.
Reliability	The system has to withstand unusual scenarios, such as a temporary blackout. As the application directly interacts with real-world actuators, which can cause damage to the Fischertechnik factory, it has to be made sure that, at any point in time, the application can be stopped by the user.

Table 3.1: Factory Simulation – Marginal Conditions

3.3 Software Architecture

In the following, the software architecture is presented. It is derived from defined uses cases, requirements and marginal conditions. The software architecture includes the systems *POU composition* with its *system operations*, *technical processes* and *data model*.

The usage of all five programming languages would cause confusing code. As mentioned in [9], SFC is suitable for processes that are similar to state machines, being processed step by step. Since the factory simulation requires such processes, SFC is chosen to structure the POU composition and their steps (cf., Section 2.2.3). To implement a certain step or transition, ST is used, because it is the IEC 61131-3 programming language that is closest to object oriented programming languages. Complex boolean expressions as well as dynamic flow control are enabler for clear and short code, being realized by ST (cf., Section 2.2.2).

3.3.1 Program Organization Units

As mentioned in section 2.2.1, POUs are splitted into PRGs, FBs and FUNs. The factory simulation uses PRGs to structure the control flow and, thus, the PRGs are implemented in SFC. Inside the SFC steps, being implemented in ST, FBs and FUNs are called. Figure 3.2 shows all PRGs and their call relation. In each cycle, the PLC starts the code execution by calling the `MAIN` PRG. This PRG coordinates the PRG-calls of other PRGs, based on the information of `TC`. The PRG calls depend on the `ExecutionMode` of the current `BatchJob`, being processed at this point in time. For example, if there is a `BatchJob` running in single mode, `MAIN` will call the single mode PRGs of each station.

Hence, every POU is responsible for a station in a certain mode. Some PRGs are supported by FBs and FUNs. In case of HR PRGs, there are nine FBs supporting them (cf., Figure 3.3). Each FB implements a certain functionality and may depend on further FBs. For example, `HR_PUT_W_ON_CONVEYOR` moves the cantilever of the high rack tower in such a way that the box it holds is put down onto the conveyor belt. For the movements of the tower, it uses `HR_MOVE`. This FB implements the tower

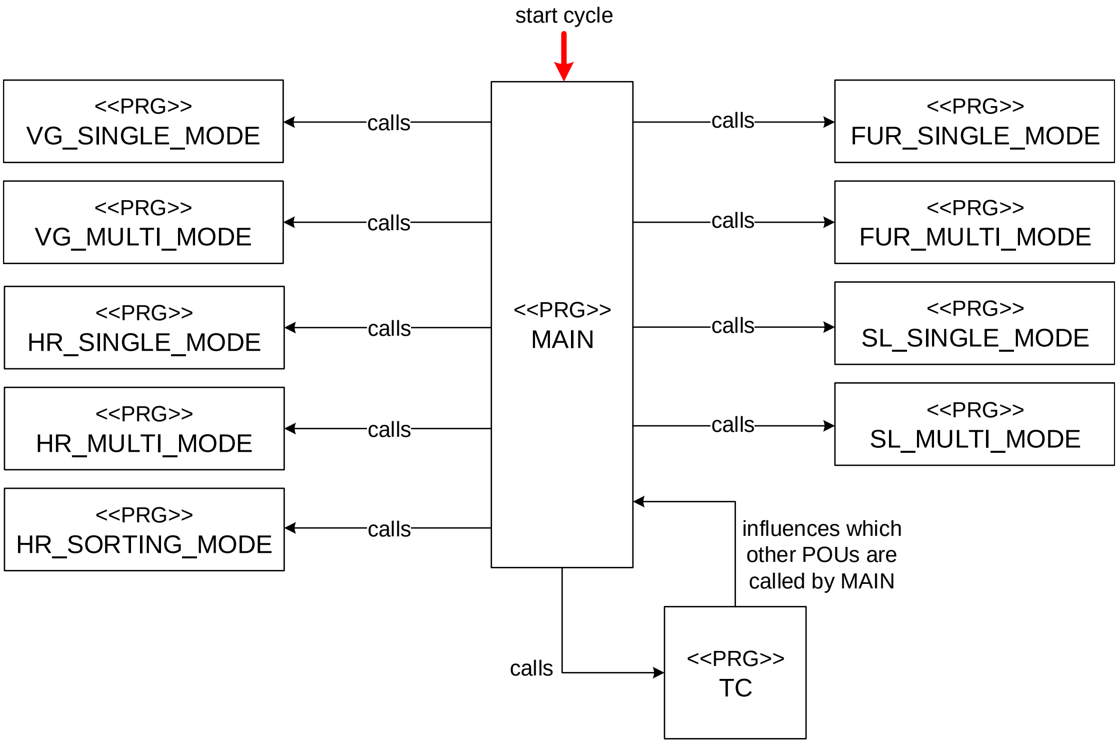


Figure 3.2: Factory Simulation – PRGs Overview

movements in all directions. While HR_MOVE gets the distance it shall move as an input, HR_MOVE_TO_POS gets a certain position where to go. HR_TAKE_W_FROM_CONVEYOR does the same steps as HR_PUT_W_ON_CONVEYOR, but visa versa.

HR_TAKE_W and HR_PUT_W realize the same actions as HR_PUT_W_ON_CONVEYOR and HR_TAKE_W_FROM_CONVEYOR, but at the high rack. HR_GET_W takes a workpiece out of the high rack and brings it to the conveyor belt. By contrast, HR_STORE_W stores a workpiece in high rack by picking it up at the conveyor, moving it to the rack and putting it down there. In order to accomplish this, it makes use of HR_TAKE_W_FROM_CONVEYOR, HR_MOVE, HR_MOVE_TO_POS and HR_PUT_W. HR_SWITCH_WS is used by HR_SORTING_MODE only, as it implements the functionality of swapping the positions of two workpieces inside the high rack.

In Figure 3.4, the PRGs of VG, FUR and SL are shown, alongside their supporting FBs. While FUR_SINGLE_MODE and SL_SINGLE_MODE do not use any FBs or FUNs,

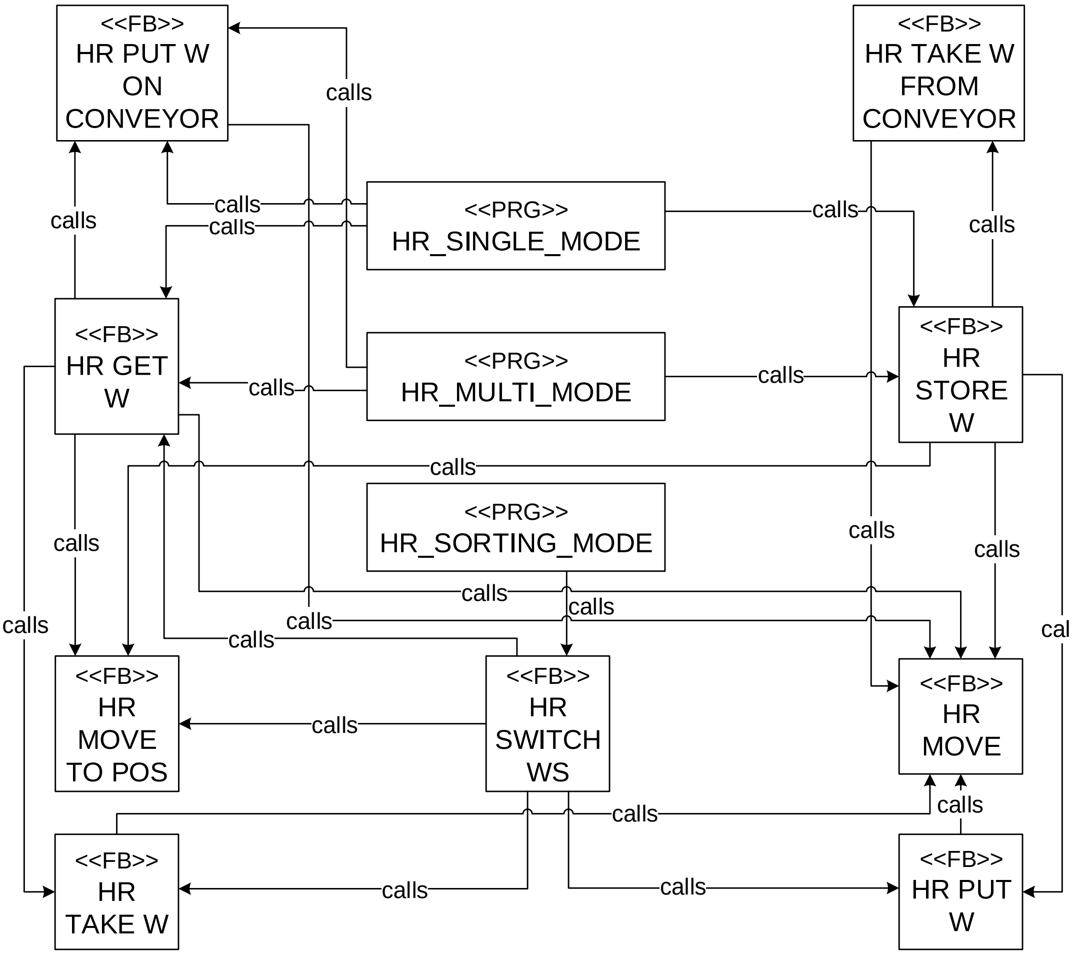


Figure 3.3: HR POUs and their Call Relations

FUR_MULTI_MODE uses FUR_BURN, implementing the burning of a workpiece in multi mode. SL_MULTI_MODE uses the SL_FB, implementing the functionality of SL in multi mode. VG_SINGLE_MODE and VG_MULTI_MODE call VG_MOVE and VG_MOVE_TO_POS in order to move the VG tower. Equally to movement FBs of HR, VG_MOVE realizes movement by a certain distance while VG_MOVE_TO_POS realizes movement to certain positions, e.g., to the first position of the eject station.

Figure 3.5 shows functions used by MAIN to manipulate data structures, to init the batch job queue and fill it with batch jobs and workpieces. newBatchJob, newFormat and newWorkpiece act as a constructor, as it creates an instance of the considered struct.

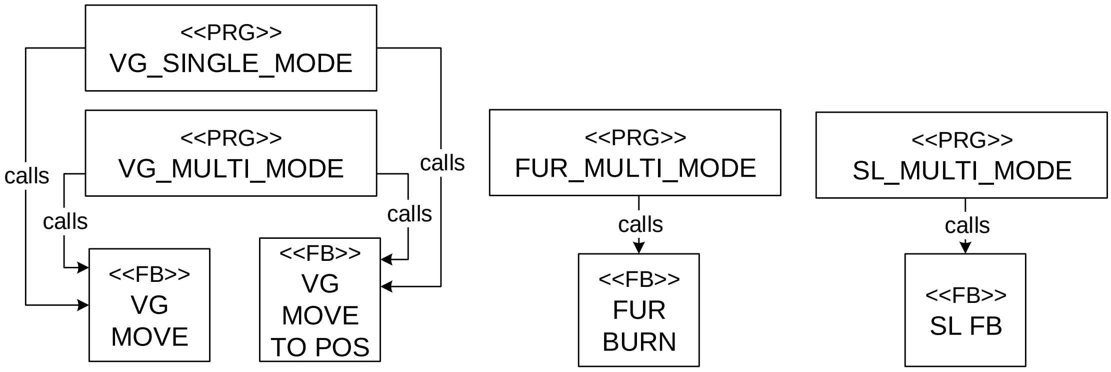


Figure 3.4: POUs with Call Relations to VG, FUR and SL

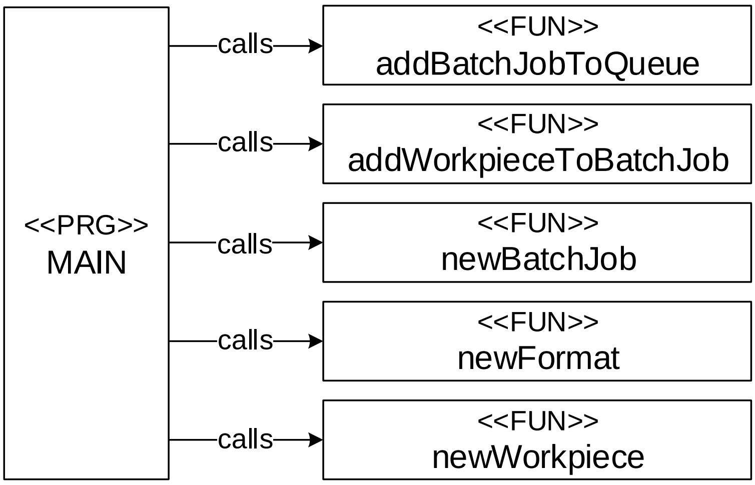


Figure 3.5: FUNs, used by MAIN

By using `addWorkpieceToBatchJob`, a workpiece can be added to the `workpieces` array of a `BatchJob` instance. `addBatchJobToQueue` adds a `BatchJob` instance to the `batchJobs` array of `addBatchJobToQueue`. If one of these arrays is full, the appropriate instance is not added.

3.3.2 Technical Processes

In this section, the process structure inside a component is presented, alongside with their relations. Therefore, we distinguish between single and multi mode. First, the task of the *task configurator* (TC) is explained, as well as its process model, and the process model of `MAIN`.

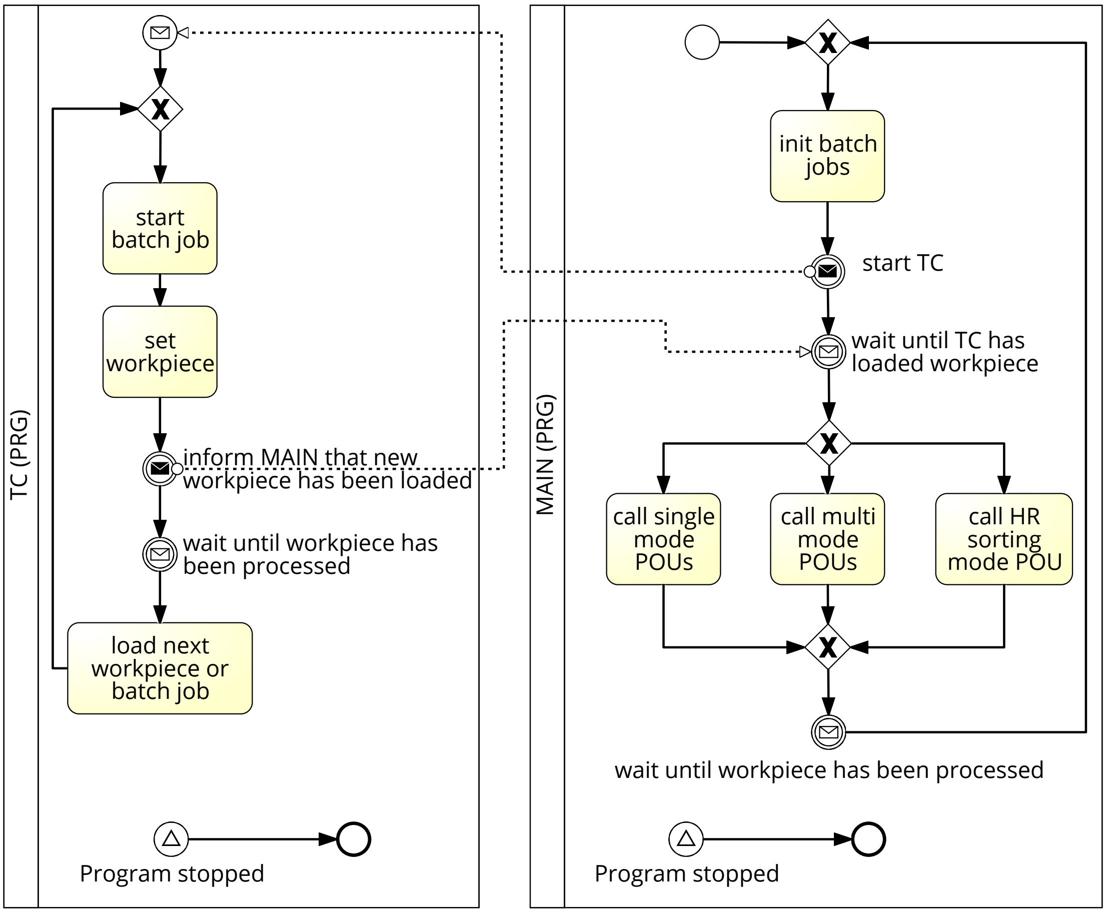

TC (PRG)
start batch job
set workpiece
inform MAIN that new workpiece has been loaded
wait until workpiece has been processed
load next workpiece or batch job
Program stopped
MAIN (PRG)
init batch jobs
start TC
wait until TC has loaded workpiece
call single mode POUs
call multi mode POUs
call HR sorting mode POU
wait until workpiece has been processed
Program stopped

Figure 3.6: TC and MAIN – Process Model

TC is the software component which maintains the batch job queue. It loads the next waiting batch job when the current batch job has been finished. During a running batch job, TC also loads the next workpiece. If a workpiece has been loaded, HR is allowed to start processing the workpiece. As shown in Figure 3.6, the execution starts with calling the MAIN PRG. It initializes the batch jobs and loads them into the batch job queue. Afterwards, MAIN calls TC and waits until TC has loaded a workpiece to HR. Dependent on the production mode of the batch job to which the current workpiece belongs to, MAIN calls the POUs of the production mode. TC waits until all stations have finished processing the workpiece. Afterwards, TC loads the next workpiece of the current batch job to HR or, if it were the last workpiece of the current batch job, TC would load the first workpiece of the next batch job in queue.

The process, describing the single mode, begins with calling the single mode PRGs by MAIN when a workpiece has to be processed in single mode (cf., Figure 3.7). HR brings its actuators to default position, so they are calibrated. Next, a workpiece is taken out of high rack, is brought to HRs conveyor and VG is informed, that the workpiece is ready to pick it up. During these actions, VG moved its actuators to default position and then drove to HR to wait for the workpiece. When the message arrives that the workpiece is ready for picking it up, VG takes the workpiece and informs HR that it has taken the workpiece. If the eject option is set in the format of the workpiece and no other option is set, VG drops the workpiece at the eject point and informs TC that it can load the next workpiece to HR, as processing the current workpiece has already been finished here. Otherwise, the workpiece is dropped at FUR by VG. VG moves its tower back to default position for calibration reasons while FUR is starting processing the workpiece. If the option of burning the workpiece is set in its format, the workpiece gets burned in the furnace. Afterwards, the workpiece is moved to the saw and is milled, if the option is set. Thereafter, the workpiece gets ejected to SL. FUR informs SL about this action. Then, SL starts processing the workpiece. If no color check is demanded, the workpiece will be ejected directly at eject point one. Otherwise, the workpieces color is checked inside the color checker. Depending on the detected color, the workpiece gets ejected to position one (white), position two (red), position three (blue), or the "trash" (no color detected). Subsequently, VG is informed that it can pick up the workpiece. Therefore, VG moves its tower to the appropriate eject position and picks up the workpiece. If ejecting the workpiece at the eject station is required, VG executes the action and informs TC, that all stations finished processing the workpiece after that. Otherwise, it drops the workpiece into a box at HRs conveyor and notifies HR about this step. Finally, HR stores

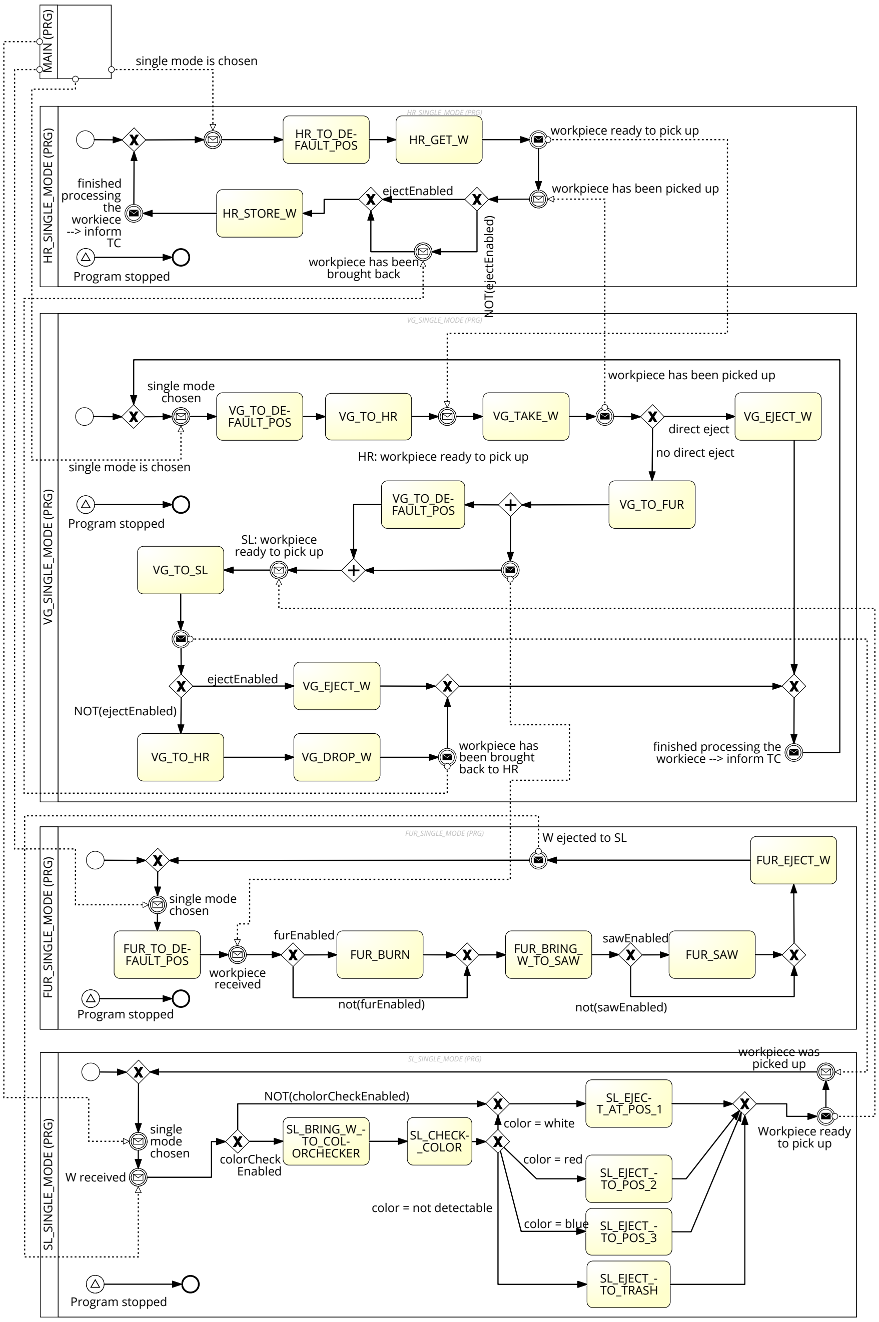


Figure 3.7: Single Mode – Process Model

the workpiece in the high rack. Now, TC is notified that all stations finished processing the workpiece.

In multi mode, the PRGs are structured differently (cf., Figure 3.8). If multi mode is chosen, HR, VG and FUR bring their actuators to their default positions. The HR multi mode PRG decides on whether it takes a workpiece out of the high rack, or stores a workpiece in the high rack. The decision making depends on a set of variables and states (cf., Section 4.3.3). When a workpiece is taken out of high rack, it is waited with storing the empty box until VG has picked up the workpiece. If a workpiece is moved to HR, it chooses the other path: it brings a box to its conveyor, waits until VG has dropped a workpiece into the box, and stores the box in high rack.

VGs multi mode behaviour highly depends on the `DECIDE` step. Inside this step, the next action of VG is determined. After this action has been finished, the next step will be decided. The decision making is addressed in Section 4.3.4. The following actions can be executed:

- `TO_DEFAULT_POS`
- `FROM_HR_TO_FUR`
- `TO_SL_TAKE_W`
- `TO_HR_PUT_W`
- `TO_HR_TAKE_W`
- `TO_EJECT`

`TO_DEFAULT_POS` moves back the actuators to their default position.
`FROM_HR_TO_FUR` moves a workpiece from HR to FUR and drops it there.
`TO_SL_TAKE_W` drives VGs tower to SL and let him pick up a workpiece. `TO_HR_OUT_W` moves the tower of VG to HR in order to drop the workpiece onto the conveyor of HR. `TO_HR_TAKE_W` moves the tower of VG to HR in order to pick up a workpiece at the conveyor of HR. In `TO_EJECT`, the VG tower is driven to the eject station to drop the workpiece at one of the three eject positions.

The FUR multi mode PRG coordinates its three machines by calling them in parallel. Each parallel step implements its appropriate functionality. The `SL_MULTI_MODE` PRG

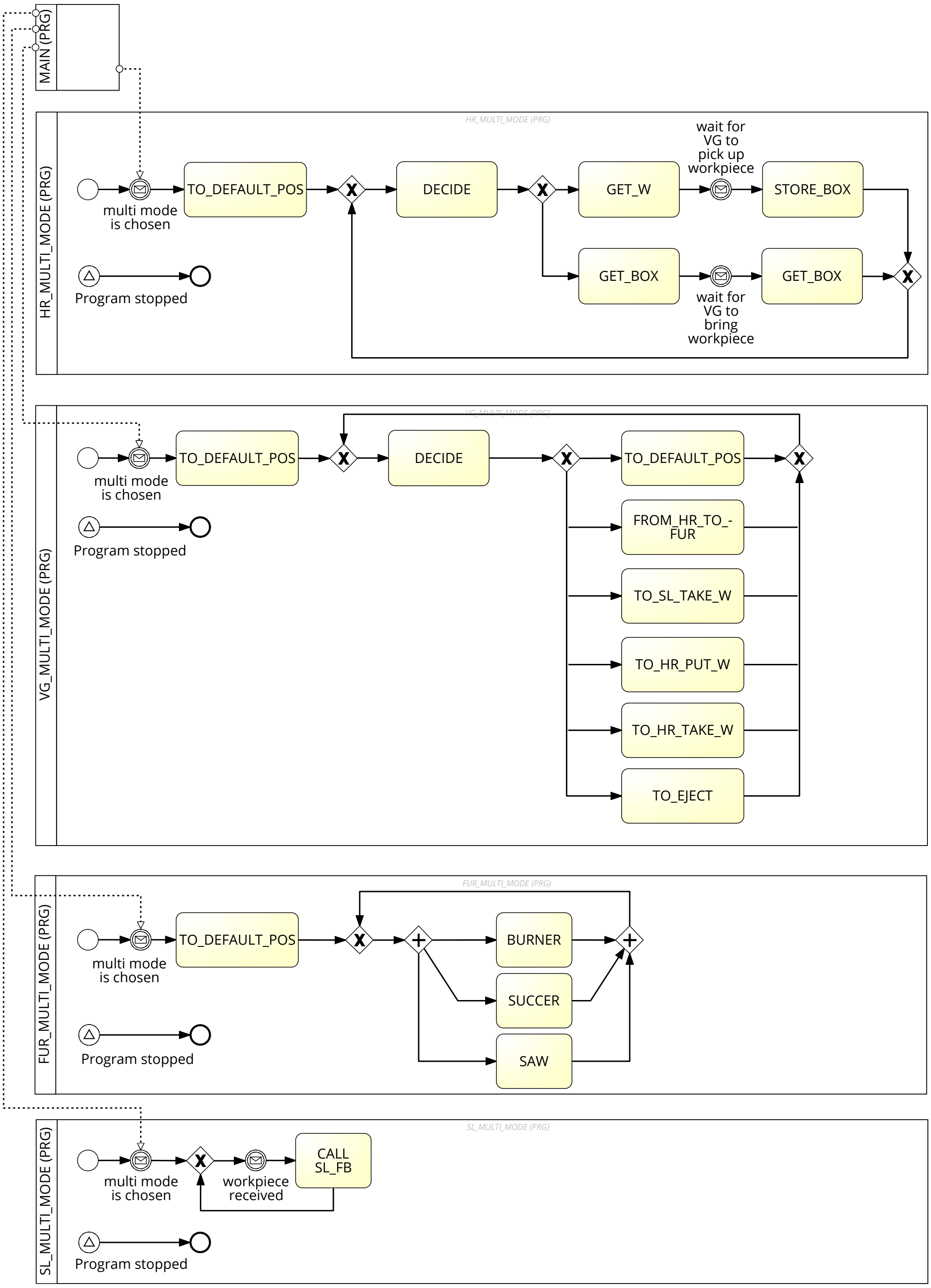


Figure 3.8: Multi Mode – Process Model

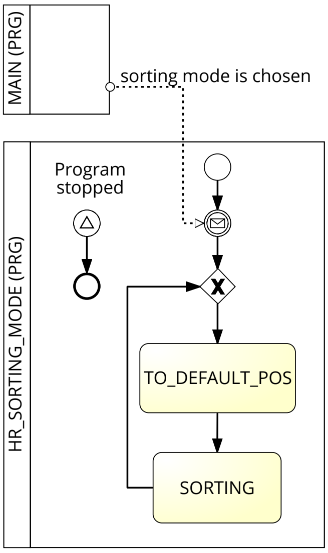


Figure 3.9: Sorting Mode – Process Model

just calls the SL_FB when it has received a workpiece. The SL_FB implements in about the same functionality as the SL single mode PRG does.

In sorting mode, only the HR PRG is necessary. As shown in figure 3.9, it has a very simple structure: after sorting mode has been chosen by MAIN, HRs tower is driven to its default position. Then, the sorting is executed. The implementation of the sorting is explained in Section 4.3.3.

3.3.3 Data Model

In the following, the data structures are explained. As shown in 3.10, there is the struct `BatchJobQueue`. It implements a ring buffer, holding the batch jobs which are going to be executed. Therefore, the integer variable `nextFreePos` states the next free position in the array `batchJobs` while `amountBatchJobs` counts the batch jobs being stored in the ring buffer. The boolean variable `full`, which states whether the queue is full or not. These variables are the basis to implement a waiting queue.

The `BatchJob` struct includes the array `workpieces`, holding the workpieces that are processed in this batch job. However, `workpieces` is not a ring buffer, but an array with fixed length: it can hold up to nine workpieces, because the high rack has nine positions where workpieces can be stored. Furthermore, the `state` variable is a `BatchJobState` enum and holds information about the current status of the batch job, such as "waiting for execution" or "started and currently running". More details about all possible enum values can be found in appendix A.4. Integer variable `batchSize` of the `BatchJob` struct counts the amount of workpieces being stored in `workpieces`. Integer variable `nextfreeWPos` holds the index of the next free position in `workpieces`, if `workpieces` is not full. Integer variable `nextNotProcessedIdx` points to the workpiece position in `workpieces` which will be processed next. Integer variable `amountFinishedW` counts the amount of workpieces of this batch job which have already been processed. Variable `mode` (an enum of `ExecutionMode`) defines the execution mode, i.e., single, multi or sorting mode. In addition, a `BatchJob` contains certain timestamp variables in order to track execution.

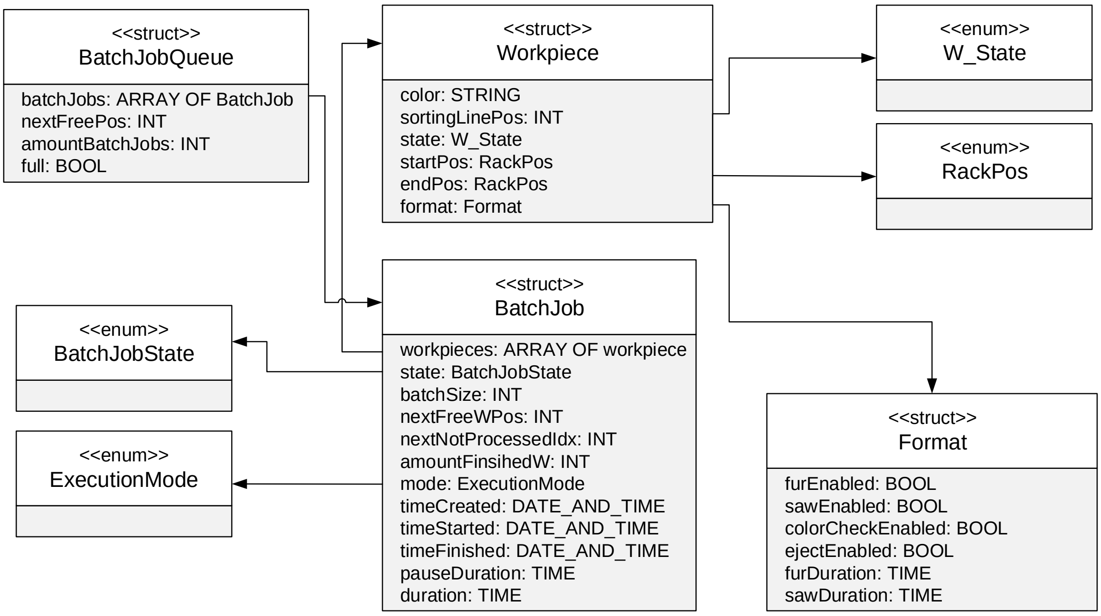


Figure 3.10: Data Structures – Class Model

A `workpiece` struct includes the workpieces `color`, its `startPos` in high rack, its `endPos` in rack (if not ejected) and its position in the sorting line (`sortingLinePos`). `state` is an enum of `W_State`, giving information about the current position of a workpiece, e.g., in the furnace. Each `Workpiece` points to a `format` struct, which holds the execution options of this workpiece. Consequently, it can be indicated whether the workpiece is burned, milled, color-checked or ejected. Moreover, for burning and milling, a duration can be defined with `furDuration` and `sawDuration`.

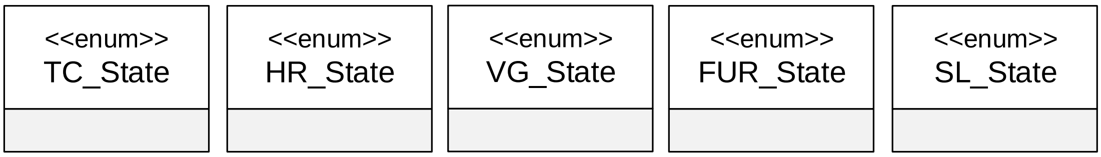


Figure 3.11: Enums – Overview

Beside these structs and enums, further enums exist (cf., Figure 3.11). They are used to show the current station states. This information may be used to visualize the factory simulation. Enum `TC_State` is used to control the task configuration. These enums are not used by the above shown structs, but by the POUs.

4

Proof-of-Concept Implementation

The implementation of the factory simulation is based on the above introduced concept (cf., Chapter 3). In the following, important parts of the implementation are explained, including the Codesys project setup, hardware setup, implemented components, component functions, component composition, and dependencies between components.

4.1 Setup

The setup description describes software and hardware setup. The software setup outlines the Codesys project structure and its components, while the hardware setup deeply explains the hardware composition.

4.1.1 Software Setup

In order to realize the factory simulation, the Codesys Integrated Development Environment V3.5 SP10 Patch 5 is used, because it supports the Berghof PLC. The application software is organized in a Codesys project. In order to create a Codesys project for a Berghof PLC, the Codesys target package of the PLC has to be installed. It is a Codesys plugin integrating the Berghof PLC. In order to get access the I/O pins of the PLC, an so called software side "extension slot" has to be added to the project inside Codesys. The Berghof I/O software component is assigned to this software part. To be able to communicate with the PLC I/O extension hardware, the EtherCAT master software module has to be attached to the project. Thereafter, the Berghof ECC IO 16/16P I/O extension software module must be assigned to the EtherCAT master in its settings. In the device communication tab, a gateway has to be chosen and the IP address of the PLC must be assigned, in order to be able to establish a connection between the development machine and the PLC. Between all these software packages, dependencies and possible incompatibility have to be checked.

4.1.2 Hardware Setup

The hardware setup of the factory simulation includes a workstation, e.g., a laptop with Ethernet interface running the IDE Codesys, and an Ethernet-switch, connecting the workstation to the PLC (cf., Figure 4.1). The PLC is connected to the PLC-I/O-Extension by a CAT-5 cable and runs the EtherCAT protocol for communication. Furthermore, a 24V DC power supply is required to provide power to the PLC, PLC-Extension, and Fischertechnik factory simulation I/O-boards.

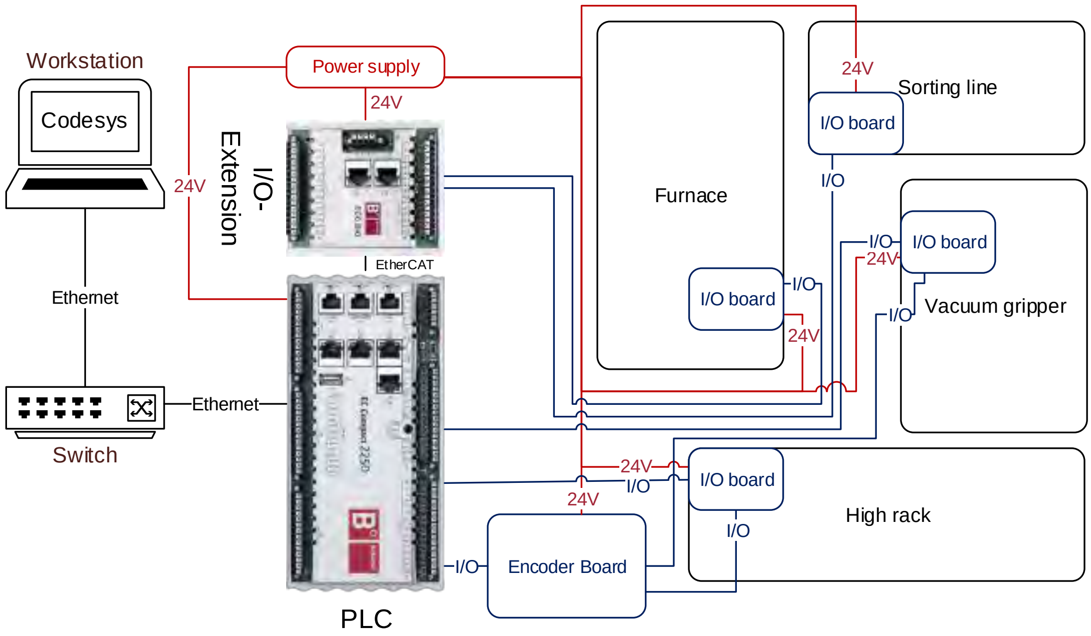


Figure 4.1: Factory Simulation Hardware Components

Each Fischertechnik station has a dedicated I/O-board, controlling the actuators of the station, depending on the signals coming from the PLC. While the I/O-pins of the PLC are directly wired to the I/O-Boards of HR and VG, the I/O-boards of FUR and SL are wired to the I/O-pins of the I/O-extension board. The complete wiring scheme can be found in appendix B. To organize the wiring, flat ribbon cable are used (cf., Figure 4.2).

Furthermore, an encoder board is connected between some input pins of the PLC and some output pins of the I/O-boards of HR and VG. The encoder board is necessary to encode signals of encoder motors in a certain way to make them readable for the PLC, as described in section 2.1.3. The board uses ICs of type 4040, which are frequency dividers to divide the frequency of the motor encoder signals by up to 1:128 (cf., Figure 4.3). By using jumpers, the frequency division can be adjusted. In our setup, the jumpers are placed in the third slot, so the frequency is divided by four.

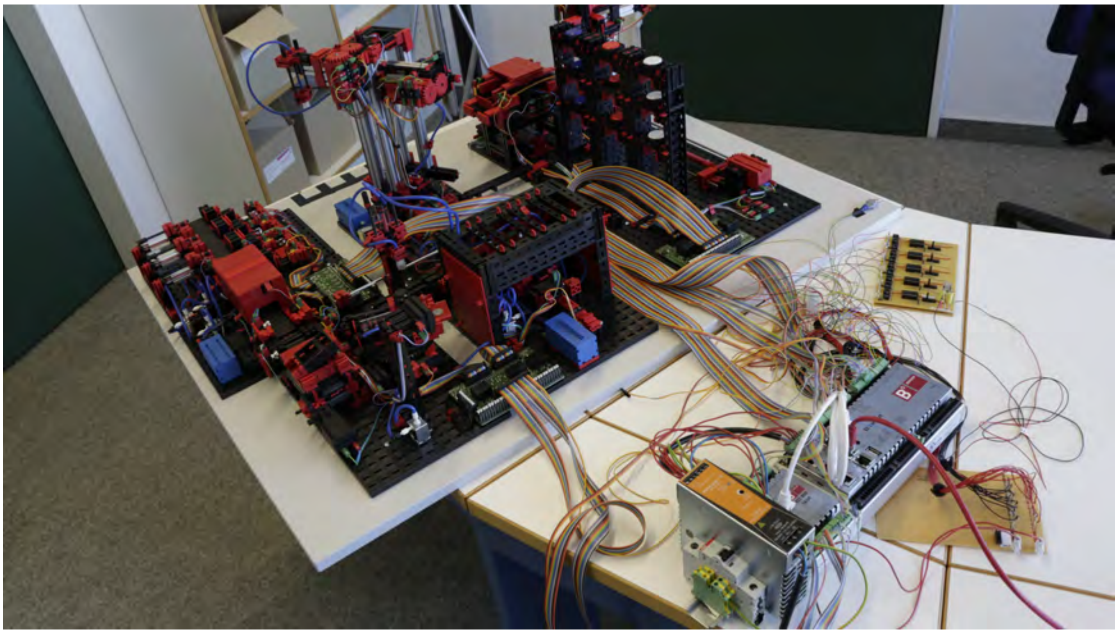
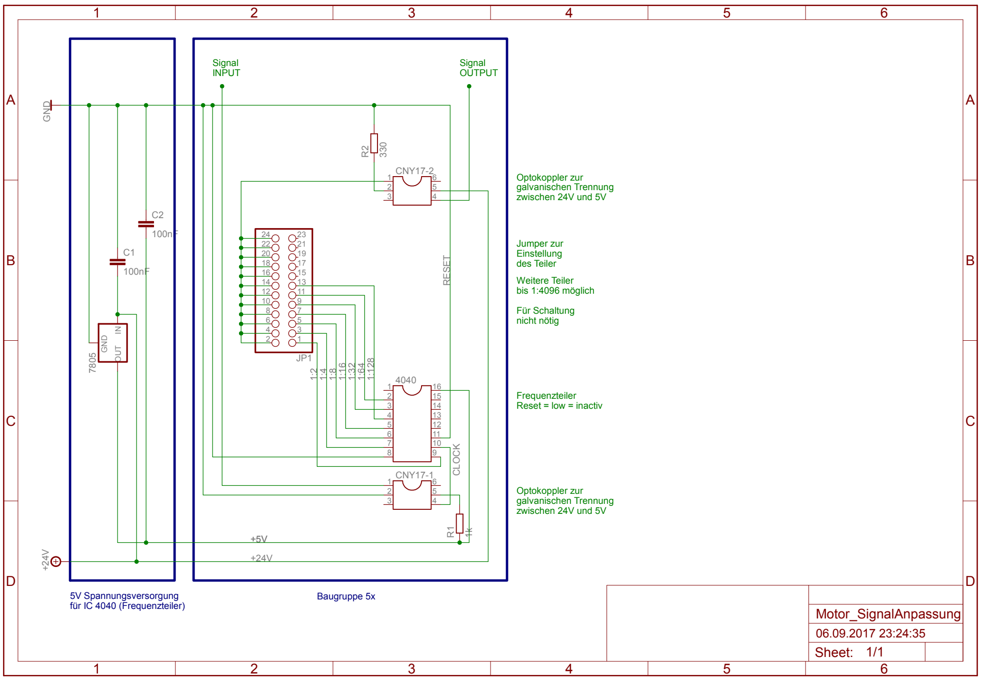


Figure 4.2: Fischertechnik Factory Simulation with PLC and Wiring

As the ICs require 5V voltage supply, the 24V voltage supply is transferred to 5V by using a voltage divider. Photocouplers are used to isolate the circuits galvanically. A encoder board consists of five modules for each of the five encoder motors. The input signal comes from the encoder motor and is used as the clock of the IC. The IC counts incoming clock signals and adjusts the outgoing pins 1-8 in the way that they represent



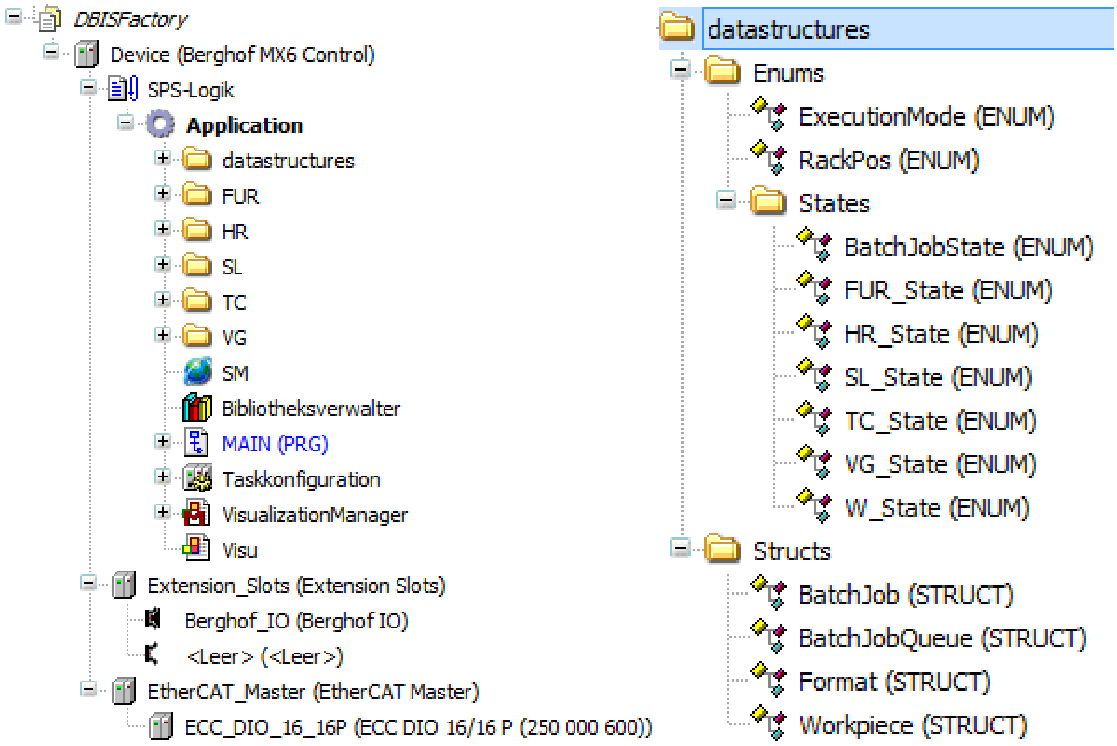
06.09.2017 23:27:04 d:\eagle\WTK1\Motor_SignalAnpassung.sch (Sheet: 1/1)

Figure 4.3: Encoder Board – Circuit Diagram

the amount of counted impulses as a binary value. As the input signal is a 24V signal, a photocoupler steps down the signal to 5V to be usable by the IC. Similarly, the output signal must be converted back, hence, an additional photocoupler is used to step up the 5V output signal to 24V.

4.2 Code Structure

The Codesys project is structured like a tree, starting with the PLC device as root node (cf., Figure 4.4). It is split into *PLC logic*, *extension slots*, and the *EtherCAT_Master*. The PLC logic consists of the actual application code. One extension slot is used by the Berghof_IO software module. It implements the access to input and output pins of the PLC. To be more precise, the addresses of the pins are defined here. For example, the address %IX1.0 points to the first digital input pin. With the help of the EtherCAT_Master module, communication to the PLC I/O extension is possible by a software module being assigned to its slot. As a result, the I/O pins of the extension can be used just as the actual PLC pins by addressing them by using their names which are defined by the software module ECC_DIO_16_P itself. The PLC, and extension slot, hardware pin address assignment is listed in appendix B.

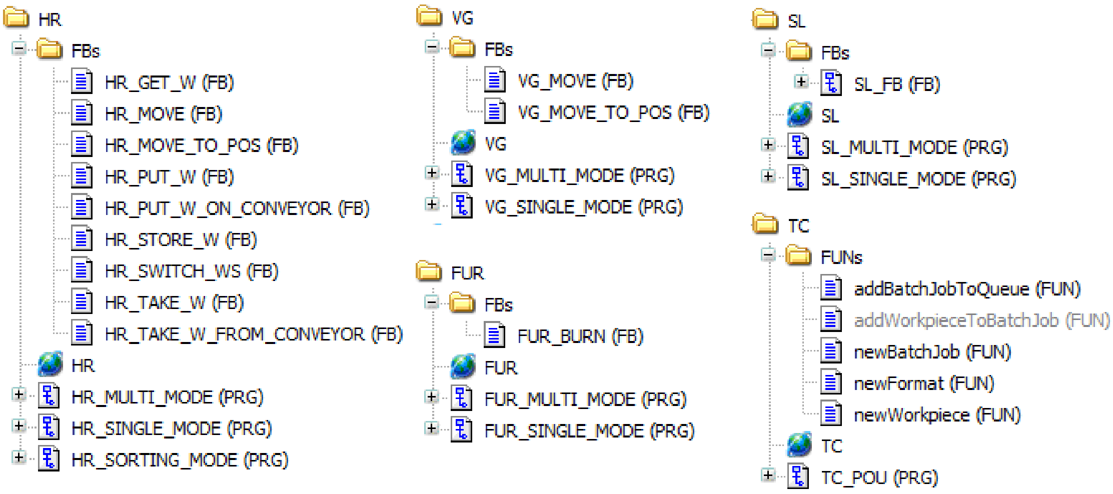


- DBISFactory
  - Device (Berghof MX6 Control)
    - SPS-Logik
      - Application
        - datastructures
        - FUR
        - HR
        - SL
        - TC
        - VG
        - SM
        - Bibliotheksverwalter
        - MAIN (PRG)
        - Taskkonfiguration
        - VisualizationManager
        - Visu
    - Extension_Slots (Extension Slots)
      - Berghof_IO (Berghof IO)
      - <Leer> (<Leer>)
    - EtherCAT_Master (EtherCAT Master)
      - ECC_DIO_16_16P (ECC DIO 16/16 P (250 000 600))

- datastructures
  - Enums
    - ExecutionMode (ENUM)
    - RackPos (ENUM)
    - States
      - BatchJobState (ENUM)
      - FUR_State (ENUM)
      - HR_State (ENUM)
      - SL_State (ENUM)
      - TC_State (ENUM)
      - VG_State (ENUM)
      - W_State (ENUM)
  - Structs
    - BatchJob (STRUCT)
    - BatchJobQueue (STRUCT)
    - Format (STRUCT)
    - Workpiece (STRUCT)

Figure 4.4: Codesys Project Structure and Data Structures

The application, being located in the PLCs logic directory, consists of written code, task configurations, a library manager, a visualization manager, and a visualization. The `datastructures` directory consists of self-defined structs and enums (cf., Figure 4.4). Furthermore, there is a global variable list `SM` necessary for single mode. In the application's root directory, the PRG `MAIN` is located. This PRG is called by the MainTask, as it is assigned to this task by the task configuration. Additionally, there is a task that handles EtherCAT communication, and another task handling visualization. All three tasks run in cyclic mode. There are directories for each station of the Fischertechnik factory, containing all POUs as well as one global variable list of the appropriate station. Furthermore, there is a directory for TC, consisting of TCs POUs and the global variable list `TC` (cf., Figure 4.5).



```
HR
  FBs
    HR_GET_W (FB)
    HR_MOVE (FB)
    HR_MOVE_TO_POS (FB)
    HR_PUT_W (FB)
    HR_PUT_W_ON_CONVEYOR (FB)
    HR_STORE_W (FB)
    HR_SWITCH_WS (FB)
    HR_TAKE_W (FB)
    HR_TAKE_W_FROM_CONVEYOR (FB)
  HR
  HR_MULTI_MODE (PRG)
  HR_SINGLE_MODE (PRG)
  HR_SORTING_MODE (PRG)

VG
  FBs
    VG_MOVE (FB)
    VG_MOVE_TO_POS (FB)
  VG
  VG_MULTI_MODE (PRG)
  VG_SINGLE_MODE (PRG)

FUR
  FBs
    FUR_BURN (FB)
  FUR
  FUR_MULTI_MODE (PRG)
  FUR_SINGLE_MODE (PRG)

SL
  FBs
    SL_FB (FB)
  SL
  SL_MULTI_MODE (PRG)
  SL_SINGLE_MODE (PRG)

TC
  FUNs
    addBatchJobToQueue (FUN)
    addWorkpieceToBatchJob (FUN)
    newBatchJob (FUN)
    newFormat (FUN)
    newWorkpiece (FUN)
  TC
  TC_POU (PRG)
```

Figure 4.5: Factory Stations and Task Configurator – File Structure

4.3 Components

In the following, the different software components are explained in detail. Functions and function blocks of a station are explained first. Afterwards their usage in different execution modes is illustrated.

4.3.1 MAIN Program POU

The `MAIN` POU is a PRG which is implemented in SFC. It contains steps, that are implemented in ST. The first executed step `Init` initializes some variables. Afterwards, `INIT_BATCH_JOBS` is called, because the transition between the steps is always `TRUE` (cf., Figure 4.6). `INIT_BATCH_JOBS` initializes the batch job queue by defining arbitrary batch jobs, workpieces, and formats. In every further CPU cycle, the five steps

TC_BLOCK, HR_BLOCK, VG_BLOCK, FUR_BLOCK and SL_BLOCK are executed in parallel (cf., Figure 4.6). These steps are executed in an infinite loop, because the following transition is always FALSE.

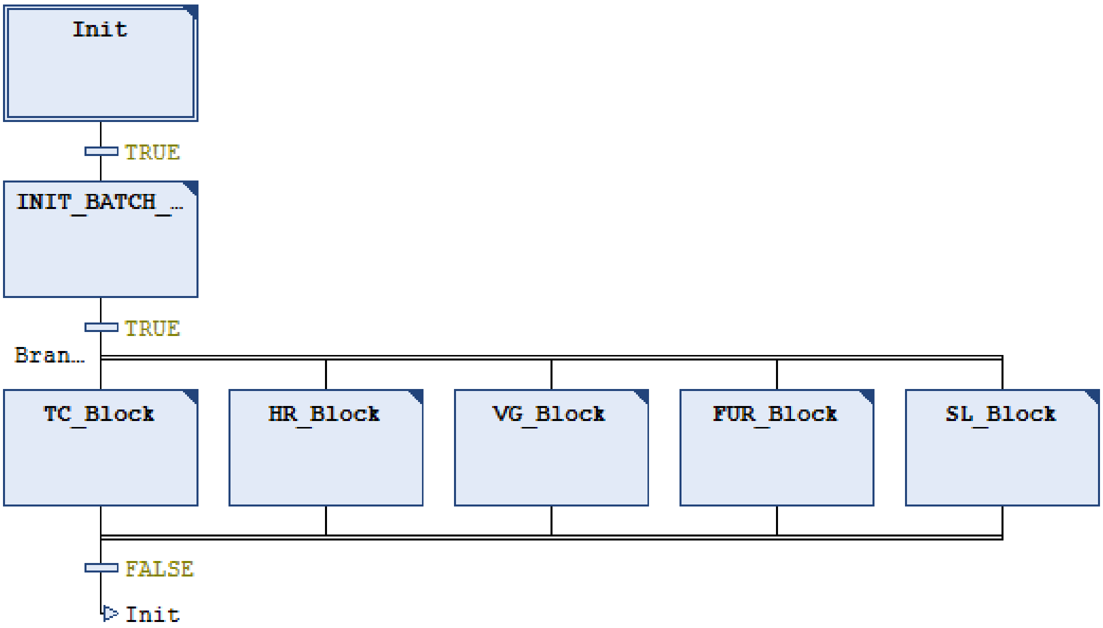


Figure 4.6: MAIN PRG – Graphical Implementation

TC_Block just consists of one code line, being the program call, i.e., TC_POU(); to invoke the TC_POU program at every CPU cycle. HR_BLOCK invokes the appropriate HR program that fits the execution mode of the current batch job, e.g., in single mode. Therefore, HR_BLOCK calls the HR_SINGLE_MODE program. Analog to this step, there are steps for VG, FUR and SL as well, realizing the same mechanism by invoking their appropriate programs.

4.3.2 Task Configuration PRG and GLV

The task configuration program (TC_POU) is implemented in SFC and consists of seven steps, being implemented in ST. The global variable list (GLV) TC has a reference to the enum `TC_State` (cf., Figure 4.7). Before the process sequence of `TC_POU` is explained, the GLV `TC` and enum `TC_State` are introduced.

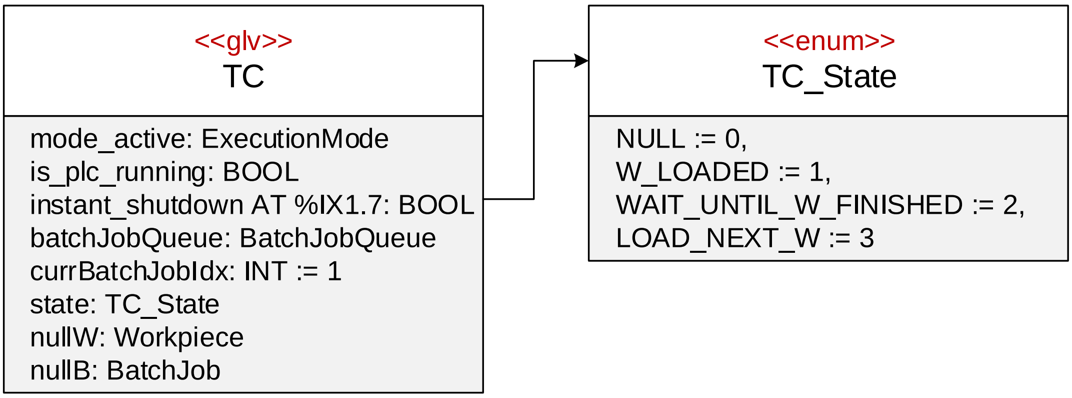

<<glv>>
TC
mode_active: ExecutionMode
is_plc_running: BOOL
instant_shutdown AT %IX1.7: BOOL
batchJobQueue: BatchJobQueue
currBatchJobIdx: INT := 1
state: TC_State
nullW: Workpiece
nullB: BatchJob
<<enum>>
TC_State
NULL := 0,
W_LOADED := 1,
WAIT_UNTIL_W_FINISHED := 2,
LOAD_NEXT_W := 3

Figure 4.7: Task Configurator – Global Variable List and State

The GLV `TC` pools variables that give a global overview over the factory. For example, the enum `mode_active` of type `ExecutionMode` states which `ExecutionMode` the factory is currently running in. The boolean variable `instant_shutdown` is used to pause the entire factory simulation by a hardware switch. If the switch is activated, all machines will stop their actions, because all outgoing signals of the PLC will be set to `FALSE`. Furthermore, `TC` provides the `batchJobQueue` which has been filled with batch jobs by `MAIN`. `nullW` and `nullB` are objects of type `Workpiece` and `BatchJob` in which no variable is initialized. As there is no NULL value for objects in the PLC programming languages, pointers of objects can be bend to `nullW` or `nullB` to realize kind of null objects. To be more precise, both structs — `Workpiece` and `BatchJob` — have a `state` enum variable. This variable can hold the value `NULL` to indicate that this object shall be treated as a NULL-object. TC comprises a `state` which gives information about which `TC` steps have been executed yet. The state is used to control the program flow from outside. For example, if the factory has just been finished with processing a workpiece in single mode, the next workpiece has to be loaded. Assuming that the workpiece was stored back to high rack, `HR_SINGLE_MODE` will alter the `state` of the GLV `TC` to the value `LOAD_NEXT_W`. As the `state` is used in the transitions of `TC_POU`, it will go on with the step `NEXT_W` (cf., Figure 4.8).

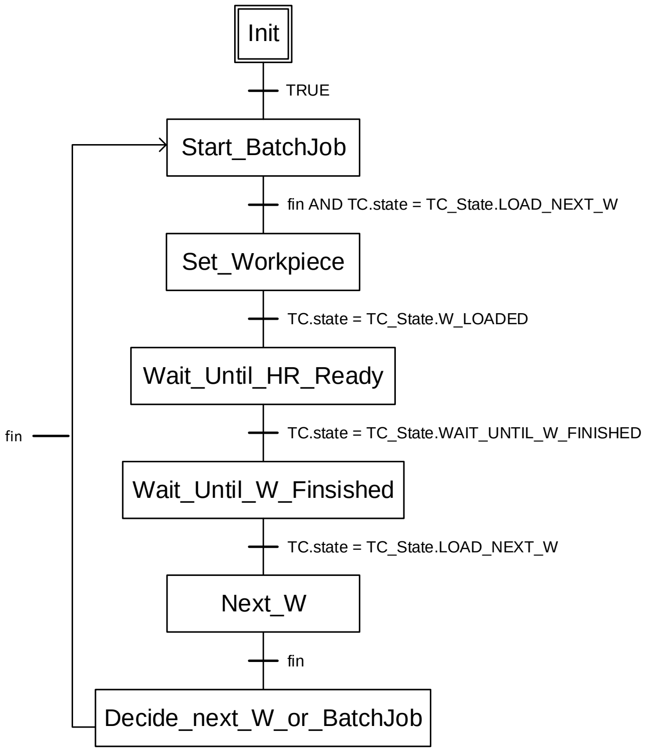

Init
TRUE
Start_BatchJob
fin AND TC.state = TC_State.LOAD_NEXT_W
Set_Workpiece
TC.state = TC_State.W_LOADED
Wait_Until_HR_Ready
TC.state = TC_State.WAIT_UNTIL_W_FINISHED
fin
Wait_Until_W_Finsished
TC.state = TC_State.LOAD_NEXT_W
Next_W
fin
Decide_next_W_or_BatchJob

Figure 4.8: TC_POU Process Sequence

The process sequence of `TC_POU` is implemented as follows. At first, the step `Init` initializes the enum `state` with the value `NULL`. Next, the step `Start_Batch_Job` starts the workpiece processing, by setting the `TC.batchJobQueue.batchJobs[TC.currBatchJobIdx].state` to `BatchJobState.STARTED`. Depending on the execution mode of the current batch job, the step `Set_Workpiece` bends the pointers of the stations GLVs to the appropriate workpiece and batch job which shall be processed. Afterwards, `TC_POU` waits until the workpiece has been processed. Step `Next_W` increments the variable `nextNotProcessedIdx` of the current batch job. Step `Decide_next_W_or_BatchJob` will do nothing if the current batch job holds a further workpiece. Otherwise, it will set the `TC.currBatchJobIdx` to the next batch job in queue.

4.3.3 High Rack Warehouse

The implementation of the high rack warehouse consists of multiple software components. There is the GLV `HR` that defines the variables, referencing the I/O pins of HR, and some program variables, such as `state` of type `HR_State` (cf., Figure 4.9). On the one hand, the `state` variable is used to monitor the stations actions, on the other hand, it is used to control the program flow. Many transitions between steps of HRs PRGs make use of `state`. Further important variables are `bIdx`, `wIdx` and `currW`. The variable `bIdx` is the index of the batch job in TCs batch job queue, which is currently processed by the simulation. `wIdx` is the index of the workpiece in the array `workpieces` of the current batch job. The variable `currW` directly points to the current workpiece object. The variable `any_motor_running` is `TRUE`, if at least one motor or compressor is active.

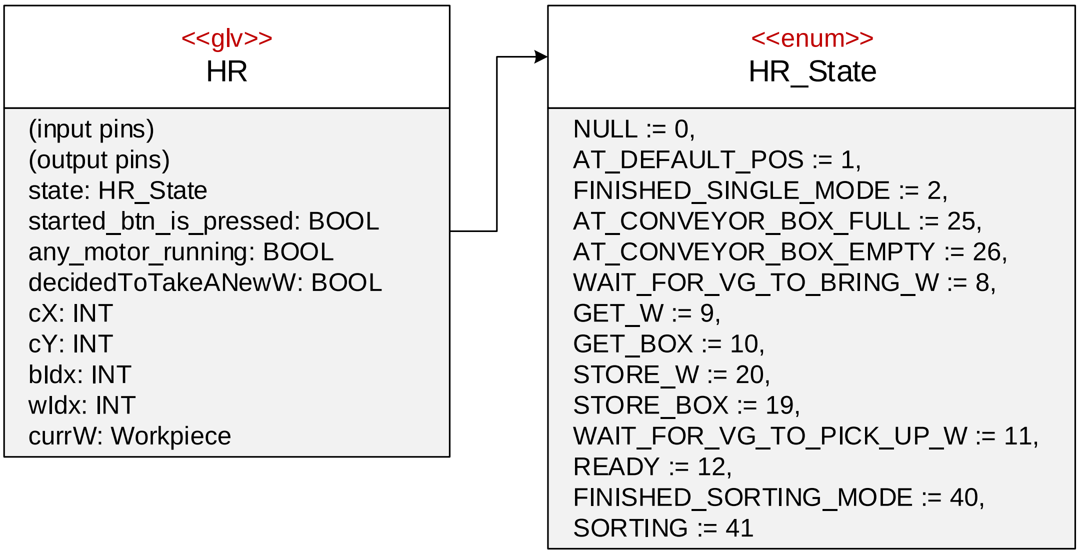


Figure 4.9: HR GLV and HR_State Enum

As already illustrated in section 3.3, the high rack concept includes multiple function blocks that are used by its programs or other function blocks. In the following, the implementation of these function blocks is explained.

`HR_MOVE` requires two input parameters, `x`, the distance HR will go right (negative value) or left (positive value) and `y`, the distance HR will go up (negative value) or down (positive value). Here, the distance is represented by the amount of impulses that are sent by the encoder motor. `HR.cX` and `HR.cY` are counter variables that save the current position of the high rack tower in the GLV. At the default position of the tower (tower presses against the two switches `sHorizontal` and `sVertical`), both counters are `0`. For example, the function block gets called with the inputs `x:=20` and `y:=30`. After the function block has been executed, the tower moved 20 impulses left and 30 impulses down. This is memorized by updating `HR.cX` and `HR.cY`. In case, x is negative, the tower is going to move right by setting `HR.mHorizontalRight` to `TRUE` (cf., Listing 4.1 line 12), as long as the local counter `cX` has not reached the value of `x` (cf., Listing 4.1 line 4). After the movements on both axis are finished, the local counters `cX` and `cY` are reset to `0`, while the global counters represent the new position.

```
 1  // X AXIS
 2  IF x < 0 THEN
 3          // drive right
 4          IF cX = x THEN
 5                  // if counter reached demanded position, we're done
 6                  HR.mHorizontalRight := FALSE;
 7                  HR.mHorizontalLeft := FALSE;
 8                  xFin := TRUE;
 9          ELSE
10                  xFin := FALSE;
11                  IF cX > x THEN
12                          HR.mHorizontalRight := TRUE;
13                          IF HR.encHorizontal AND NOT(xAlreadyCounted) THEN
14                                  // update local counter
15                                  cX := cX - 1;
16                                  // update global counter
17                                  HR.cX := HR.cX - 1;
18                                  xAlreadyCounted := TRUE;
19                          ELSIF NOT(HR.encHorizontal) THEN
20                                  xAlreadyCounted := FALSE;
21                          END_IF
22                  END_IF
23          END_IF
24  ELSE
25      ...
```

Listing 4.1: HR_MOVE Function Block – Code Snippet

The FB `HR_MOVE_TO_POS` is implemented similar to `HR_MOVE`, but gets a position, where to go, as an input, represented by `pos` and `upOrDown`, which are both enums of type `RackPos`. This enum represents certain positions at this station, such as for example `RACK_11` or `CONVEYOR`. Additionally, there are the two enum values `UP` and

`DOWN`, representing the upper or lower position. A call of the function block looks as follows:

```
VAR
    hr_move_to_pos: HR_MOVE_TO_POS;
END_VAR
...
hr_move_to_pos(pos := RackPos.CONVEYOR, upOrDown := RackPos.DOWN);
```

Listing 4.2: HR_MOVE_TO_POS call

This function block call moves the high rack tower to the conveyor to the lower position where it can pick up a box.

All further function blocks of HR are structured the same way as `HR_PUT_W` (cf., Listing 4.3). There is a `CASE` construct which chooses one step, depending on the value of `step`. If a step is finished, the `step` gets incremented and the next step will be executed in the next cycle. If the last step has been executed, the `fin` flag is set to `TRUE` and the `step` is reset to `1`.

```
CASE step OF
        1:      // forward
                fin := FALSE;
                IF NOT(HR.sCFront) THEN
                        HR.mCForward := TRUE;
                ELSE
                        HR.mCForward := FALSE;
                        step := step + 1;
                END_IF
        2:      // down
                hr_move(x := 0 , y := 16);
                stepFin := hr_move.fin;
                IF stepFin THEN
                        step := step + 1;
                        stepFin := FALSE;
                END_IF
        3:      // back
                IF NOT(HR.sCBack) THEN
                        HR.mCBackward := TRUE;
                ELSE
                        HR.mCBackward := FALSE;
                        step := step + 1;
                END_IF
        4:      // finished
                fin := TRUE;
                step := 1;
END_CASE
```

Listing 4.3: HR_PUT_W Function Block – Implementation

The FB HR_PUT_W drives the cantilever of the tower forward, until the switch HR.sCFront signalizes TRUE. Thereafter, the tower is moved down by 16 impulses. After this step, the cantilever is driven back until the switch HR.sCBack signalizes TRUE.

When using function blocks, having the explained structure, it is important to continue inside the calling program, once fin is TRUE. Otherwise, the function block is executed again.

Other function blocks of HR are not further explained, as they all follow the same structure. In the following, the implementations of the different mode PRGs of HR are explained. HR_SINGLE_MODE consists of ten steps, which are sequentially executed, starting with step Init (cf., Figure 4.10). After the default position is reached, the step HR_CALIBRATE_Y_AXIS is executed. This step resets the y-axis counter and position of the tower and is needed, because—although the encoder board increases the consistency of encoder motor movements—they are still too inaccurate after a couple of movements, because the PLC shows detection delays for each impulse. The function blocks are implemented in the way that after a certain amount of impulses, a movement is finished and the motor is turned off by setting the output bit to zero. In step HR_SET_TC_STATE, the PRG sets the TC.state to LOAD_NEXT_W and waits until TC_POU has set the pointers bIdx, wIdx and currW of the GLV HR. The pointers are loaded once TC.state holds the value TC_State.W_LOADED. Now, HR can start to take out a workpiece from rack, represented by HR_GETS_W. Afterwards, HR_SINGLE_MODE is waiting until VG picks up the workpiece and then either waits until VG brings it back or, if ejectEnabled is TRUE, directly starts to store the box back in the rack.

The multi mode of HR is implemented as specified in section 3.3.2. Thus, the SFC steps look like the BPMN process activities. The most complex step of this POU is DECIDE (cf., appendix A.1). In this step, it is decided which of the two paths will be taken and when it will be taken. Being in multi mode, this depends on many variables and states. For example, if a batch job has been processed completely and a new one is loaded, it depends on the mode of the previous batch job.

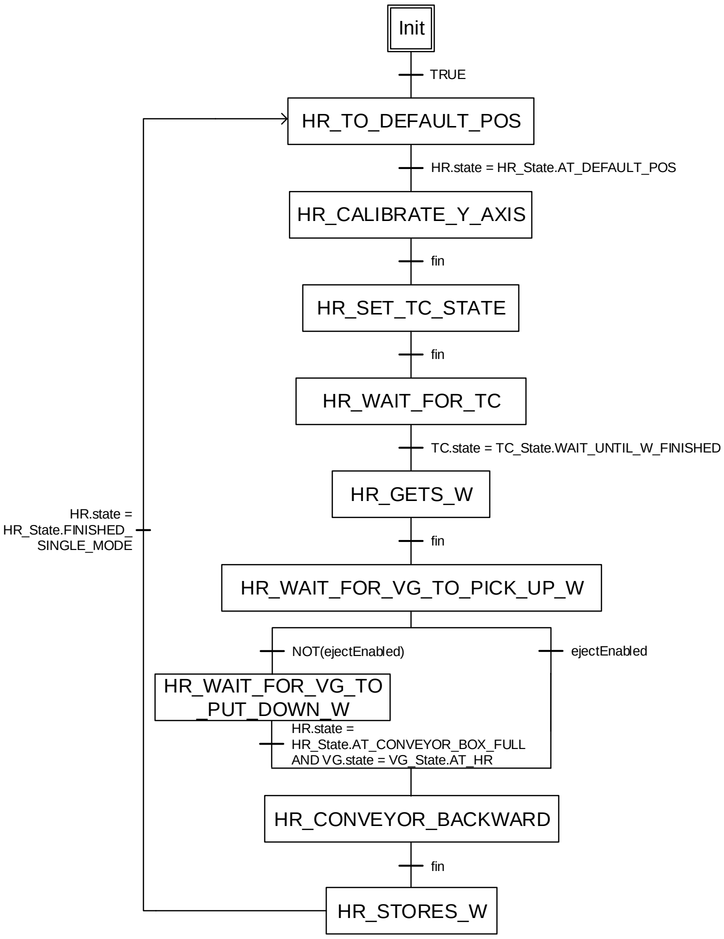
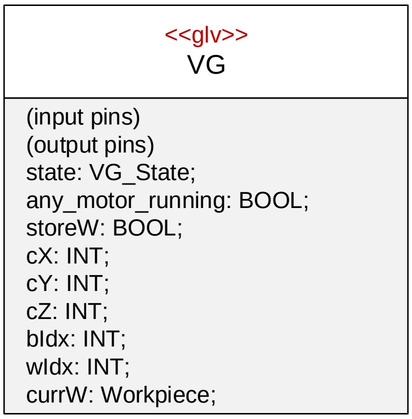

Init
TRUE
HR_TO_DEFAULT_POS
HR.state = HR_State.AT_DEFAULT_POS
HR_CALIBRATE_Y_AXIS
fin
HR_SET_TC_STATE
fin
HR_WAIT_FOR_TC
TC.state = TC_State.WAIT_UNTIL_W_FINISHED
HR_GETS_W
HR.state = HR_State.FINISHED_SINGLE_MODE
fin
HR_WAIT_FOR_VG_TO_PICK_UP_W
NOT(ejectEnabled)
ejectEnabled
HR_WAIT_FOR_VG_TO _PUT_DOWN_W
HR.state = HR_State.AT_CONVEYOR_BOX_FULL AND VG.state = VG_State.AT_HR
HR_CONVEYOR_BACKWARD
fin
HR_STORES_W

Figure 4.10: HR_SINGLE_MODE SFC Process Sequence

In sorting mode, HR uses the function block `HR_SWITCH_WS` in step `SORTING` to switch the position of two workpieces. This mode is implemented as defined in section 3.3.2. The implementation can be found in appendix A.1.

4.3.4 Vacuum Gripper

The vacuum gripper follows the same code structure as HR. The GLV `VG` defines variables to access input and output pins (cf., Figure 4.11). In addition, it holds variables, such as `bIdx`, `wIdx`, and `currW`. These variables have the same purpose as in HRs implementation. Similar to HR, VG has a reference `state` and an enum of type `VG_State` being used to control and monitor the station. VG has two function blocks: `VG_MOVE` and `VG_MOVE_TO_POS`. Still similar to HR, they implement the movements of the tower. Their implementation resembles the implementation of `HR_MOVE` and `HR_MOVE_TO_POS`.



<<glv>>
VG

(input pins)
(output pins)
state: VG_State;
any_motor_running: BOOL;
storeW: BOOL;
cX: INT;
cY: INT;
cZ: INT;
bIdx: INT;
wIdx: INT;
currW: Workpiece;

Figure 4.11: VG – Global Variable List

VGs implementation of the single mode is not linear as the implementation of HRs single mode. Instead, there are different possible execution paths (cf., Figure 4.12). `VG_SINGLE_MODE` always starts by bringing its tower to the default position, when the POU gets called for the first time. Next, the tower is moved to the high rack in step `VG_TO_HR`. Once it has reached the position (`VG.state = VG_State.AT_HR`), and a workpiece is ready to pick up (`HR.state = HR_State.AT_CONVEYOR_BOX_FULL`), `VG_SINGLE_MODE` continues with `VG_TAKE_W` and picks up the workpiece. At the end of this step, it has to be decided whether to bring the workpiece to the furnace or to bring it to the eject station. If only `ejectEnabled` is set and no other flags are set in `SM.currW.format`, `VG_SINGLE_MODE` will continue with `VG_DIRECTLY_EJECT_W` and then jumps back to `VG_TO_HR` (cf., Listing 4.4). Otherwise, the workpiece is brought to FUR in `VG_TO_FU`. Then, the tower is moved back to its default position in `BACK_TO_DEFAULT`, and waits until a workpiece is available at SL, in order to pick it up there in step `VG_TO_SL`.

```
IF SM.currW.format.ejectEnabled AND NOT(SM.currW.format.furEnabled)
    AND NOT(SM.currW.format.sawEnabled) AND NOT(SM.currW.format.colorCheckEnabled) THEN
        // directly eject workpiece
        VG.state := VG_State.FROM_HR_TO_EJECT;
ELSE
        // bring workpiece to FUR
        VG.state := VG_State.FROM_HR_TO_FUR;
END_IF
```

Listing 4.4: VG_TAKE_W Step – Decision Logic

At this point, it has to be decided, whether the workpiece is moved back to HR, or whether it is moved to the eject station. If it is moved to HR, the steps VG_FROM_SL_TO_HR, VG_DROP_W, and VG_WAIT_UNTIL_BOX_EMPTY are executed sequentially. Otherwise, the workpiece is ejected at the eject station by the sequential steps VG_YUp, VG_TO_EJECT_POINT, VG_EJECT and VG_EJECT_W. Independent from which path is chosen, the PRG finally jumps back to the step VG_TO_HR.

In contrast to VG_SINGLE_MODE, the implementation of VGs multi mode is constructed differently. First, VGs tower is driven to its default position by the step TO_DEFAULT_POS when the PRG is called for the first time. Afterwards, the DECIDE step chooses one step out of the six possible ones: TO_DEFAULT_POS, FROM_HR_TO_FUR, TO_SL_TAKE_W, TO_HR_PUT_W, TO_HR_TAKE_W and TO_EJECT (cf., Figure 4.13). After the chosen step has been executed, DECIDE is called again, to determine the step being executed next. Thereby, VG.state indicates, which step has been chosen by DECIDE. For example, if FROM_HR_TO_FUR is chosen, DECIDE sets VG.state to VG_State.TO_FUR_PUT_W. As the transition between DECIDE and FROM_HR_TO_FUR is VG.state = VG_State.TO_FUR_PUT_W, it fires as the condition has become TRUE. This implies that FROM_HR_TO_FUR is executed next. Once the step has been finished, VG.state is set to VG_State.TO_FUR_PUT_W_FINISHED and the transition, being located after the step, fires. This leads to a jump back to DECIDE (cf., Figure 4.13).

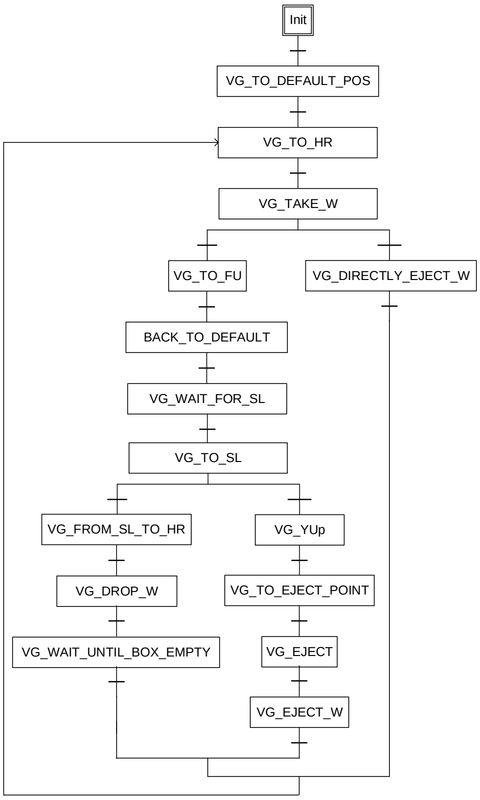


Figure 4.12: VG Single Mode – SFC Process Sequence

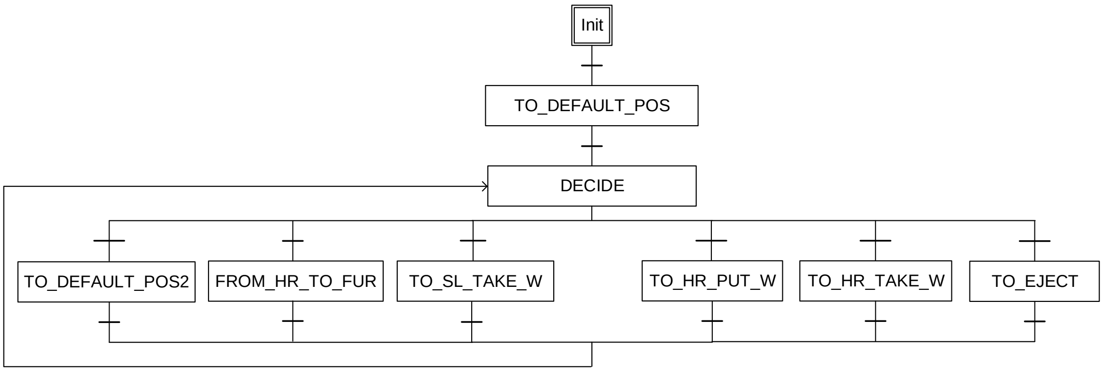
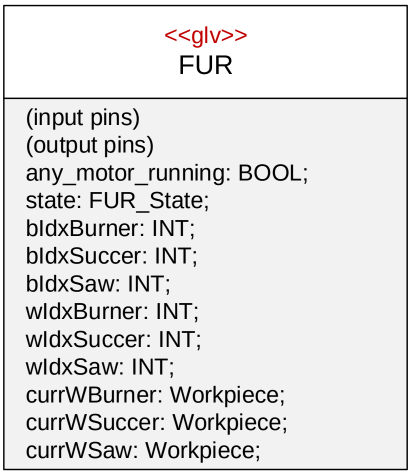


Figure 4.13: VG Multi Mode – SFC Process Sequence

4.3.5 Furnace

The furnace station has the GLV FUR, similar to HR and VG. It consists of I/O-variables and program variables (cf., Figure 4.14), e.g., bIdx, wIdx and currW for every machine, being used in multi mode. In addition, state shows the current FUR_State. Similar to other station, single and multi mode PRGs exist, namely FUR_SINGLE_MODE, and FUR_MULTI_MODE.



<<glv>>
FUR

(input pins)
(output pins)
any_motor_running: BOOL;
state: FUR_State;
bIdxBurner: INT;
bIdxSuccer: INT;
bIdxSaw: INT;
wIdxBurner: INT;
wIdxSuccer: INT;
wIdxSaw: INT;
currWBurner: Workpiece;
currWSuccer: Workpiece;
currWSaw: Workpiece;

Figure 4.14: FUR – Global Variable List

FUR_SINGLE_MODE is a PRG, written in SFC, that has a linear process layout with options for skipping a machine, e.g. skip burning the workpiece. At the beginning of the POU, all machines are moved to their default position and the station waits until VG has laid down a workpiece onto the wagon. Afterwards, the step FUR_BURN realizes the actions of burning a workpiece when SM.currW.format.furEnabled is set. While a workpiece is in furnace, the vacuum succer is driven towards the wagon in order to pick up the workpiece after burning. The next five steps, starting with FUR_SUCCER_DOWN and ending with FUR_SUCCER_UP2, bring the workpiece to the rotating desk, where the saw is mounted. FUR_SAW will mill the workpiece, if sawEnabled is set. Otherwise, the execution continues with FUR_TO_CONVEYOR. After this step has been finished, FUR_SINGLE_MODE jumps back to Init (cf., Figure 4.15).

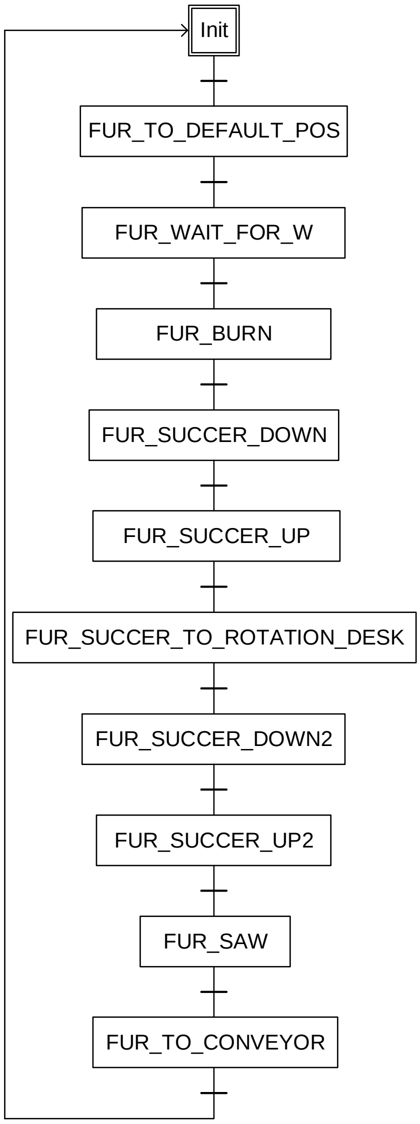


Figure 4.15: FUR Single Mode – SFC Process Sequence

In FUR_MULTI_MODE, the actions of furnace, succer, and saw are parallelized. Furthermore, the actions that are needed to simulate the burning of a workpiece are outsourced into the function block FUR_BURN.

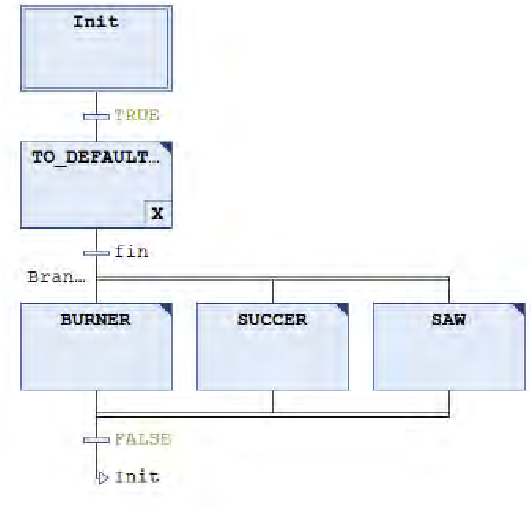


Figure 4.16: FUR Multi Mode PRG – Graphical Implementation

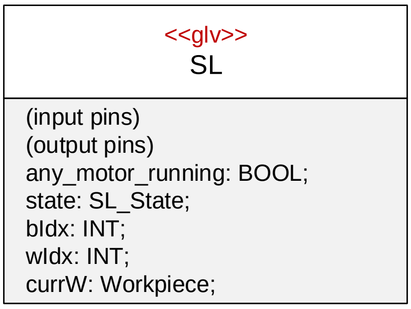
The implementation of the step BURNER is exemplary explained (cf., Listing 4.5). Every workpiece has a state that indicates the current status of a workpiece, e.g., it current position. As long as FUR.currWBurner.state is W_State.NULL, the BURNER does nothing. FUR.currWBurner.state is changed by VG once it has dropped the workpiece onto the wagon. Then, the workpiece gets burned when FUR.currWBurner.format.furEnabled is set. When finished, the pointers of the succer are set to the current workpiece of the burner. Afterwards, the pointers of burner are reset. This implies, that BURNER waits again, until FUR.currWBurner is changed to a value not equal to W_State.NULL.

```
IF FUR.currWBurner.state <> W_State.NULL THEN
    // fur burner got an w
    IF FUR.currWBurner.format.furEnabled AND NOT(finBurner) THEN
        // burn
        fur_burn(w := FUR.currWBurner);
        finBurner := fur_burn.fin;
    ELSE
        IF FUR.currWSuccer.state = W_State.NULL THEN
            // bend pointers of fur succer to current workpiece
            FUR.bIdxSuccer := FUR.bIdxBurner;
            FUR.wIdxSuccer := FUR.wIdxBurner;
            FUR.currWSuccer := TC.batchJobQueue.batchJobs[FUR.bIdxSuccer].workpieces[FUR.wIdxSuccer];
            // reset own pointers
            FUR.bIdxBurner := 0;
            FUR.wIdxBurner := 0;
            FUR.currWBurner := TC.nullW;
            finBurner := FALSE;
        END_IF
    END_IF
END_IF
```

Listing 4.5: FUR_MULTI_MODE PRG – BURNER Step

4.3.6 Sorting Line

The sorting line has the GLV `SL`. It consists of I/O-pin variables and the program variables `state`, `bIdx`, `wIdx`, and `currW`. There are two PRGs, namely `SL_SINGLE_MODE` and `SL_MULTI_MODE`. Both are implemented similarly: they call a `SL_FB` when they have received a workpiece. The actual process is implemented in `SL_FB`. It waits until a workpiece activates the light barrier located at the beginning of the conveyor belt. Afterwards, the workpiece gets driven trough the color checker. While going through the color checker, the color sensor constantly measures the color. Thereby, the minimum of the analog signals is taken, as it is unique for every color. Then, the workpiece is driven to one of the three ejecting pneumatic sliders, depending on the determined color, in order to push the workpiece to one of the three places. E.g., if color *white* is detected, it is ejected to the first place.



<<glv>>
SL

(input pins)
(output pins)
any_motor_running: BOOL;
state: SL_State;
bIdx: INT;
wIdx: INT;
currW: Workpiece;

Figure 4.17: SL – Global Variable List

4.3.7 User Interface

The user interface implements a button "GO!", and a LED "PLC running" (cf., Figure 4.18). Without pressing the button, the factory simulation does not start. Once the button has been pressed, the factory simulation is running and can not be stopped by pressing the button again. The LED is lighted, when the program is executed by the PLC. A sophisticated GUI can be implemented outside the PLC application. For example, the GUI can be implemented as web application retrieving data from the PLC application via OPC UA.

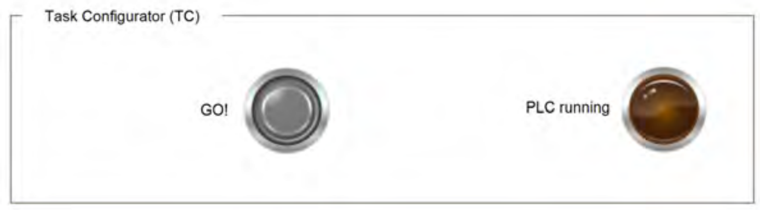


Figure 4.18: Graphical User Interface – Codesys Representation

4.4 Integration

In the following, the interaction between the components is illustrated.

Every PLC CPU cycle starts with executing `MAIN`. Depending on the `ExecutionMode` of the `BatchJob` being located in `TC.batchJobQueue` at the index `TC.currBatchJobIdx`, the appropriate PRGs of the stations are called (still in the same CPU cycle). If the batch job is executed in single mode, the `SM.currW` pointer is bend to the workpiece that is located in the `workpieces` array of the current batch job. Now, every single mode PRG has access to the workpiece which is going to be processed.

If the batch job is executed in multi mode, `TC_POU` bends the pointers `HR.bIdx`, `HR.wIdx` and `HR.currW` to the current workpiece. Here, `HR.bIdx` is set to `TC.currBatchJobIdx`, `HR.wIdx` is set to `TC.batchJobQueue.batchJobs[TC.currBatchJobIdx].nextNotProcessedIdx` and `HR.currW` is set to `TC.batchJobQueue.batchJobs[Hr.bIdx].workpieces[Hr.wIdx`. Every time, a station has given a workpiece to the next station, it also bends the stations pointers to the current workpiece it has gotten. For example, if VG takes a workpiece from HR, `VG.bIdx` is set to `HR.bIdx`, `VG.wIdx` is set to `HR.wIdx` and `VG.currW` is set to `HR.currW`.

Afterwards, the pointers of HR are reset. If a batch job has been finished, and the next batch job has to be executed in a different mode, it has to be ensured that pointers to workpieces and batch jobs are not set until all stations have finished processing the last workpiece.

5

Test and Evaluation

During development, the factory simulation has been tested and the requirements has been evaluated. However, testing of PLC applications is different to classical software unit and system tests, as the system depends on real world sensors and actuators. Furthermore, there are many dependencies between software components which may not be mocked in a way that they represent the real behaviour.

5.1 System Test

A system test consists of test cases, testing the interaction between components of the system and the systems behaviour in general [14]. Every test case of the factory simulation system test has access to 16 different formats. As a format consists of four boolean variables which can be either `TRUE` or `FALSE`, there are 16 different possible combinations. The duration variables are set to five seconds in every format (cf., Listing C.1).

The first test case covers the single mode (cf., Listing C.2), the second test case covers the multi mode (cf., Listing C.3) and the third testcase covers the sorting mode (cf., Listing C.4). The three test cases consist of 16 workpieces that are separated in two batch jobs with eight workpieces each. They all passed, because every workpiece has been processed as expected by the factory simulation.

The fourth test case covers the transitions between batch jobs, having the same or different execution modes. It consists of ten batch jobs covering all nine possible transitions (cf., Listing C.5). For every execution mode, there are three possible transitions (cf., Figure 5.1).

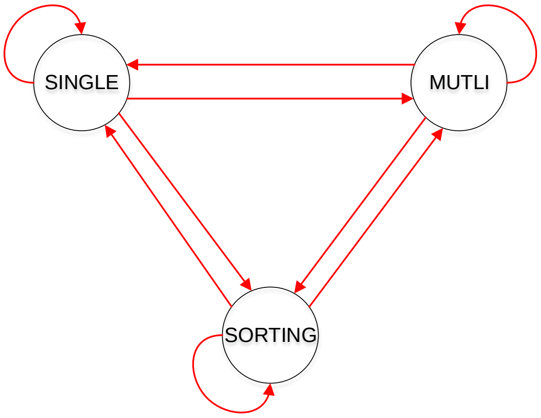


Figure 5.1: Batch Job Execution Mode Transitions

The single mode test case has been processed within 21.20 minutes, whereas the multi mode test case has been processed within 14.42 minutes. Thus, the multi mode reduced the processing time by about 33 precent in the manner described.

5.2 Process Evaluation

The implementation is collated with the functional requirements and marginal conditions, defined in section 3.2.

At HR, it is possible to store (FR1) and to take out (FR2) workpieces, realized by the function blocks `HR_GET_W` and `HR_STORE_W`. Sorting (FR3) workpieces is realized by `HR_SORTING_MODE`. Therefore, functions exist to *pick up* (FR4) a workpiece (`HR_TAKE_W` and `HR_TAKE_W_FROM_CONVEYOR`), to *put down* (FR5) a workpiece (`HR_PUT_W` and `HR_PUT_W_ON_CONVEYOR`) and to *move the tower* (FR6) (`HR_MOVE` and `HR_MOVE_TO_POS`). To *move the conveyor belt* (FR7), there is no function block, as this is realized directly inside the PRGs.

At VG, *picking up* a workpiece (FR8) at HR is realized by the steps `VG_TAKE_W` in `VG_SINGLE_MODE` and `TO_HR_TAKE_W` in `VG_MULTI_MODE`. *Picking up* a workpiece at SL is implemented in the steps `VG_TO_SL` (single mode) and `TO_SL_TAKE_W` (multi mode). To *put down* (FR9) a workpiece at HR, the step `VG_DROP_W` is used in single mode and the step `TO_HR_PUT_W` is used in multi mode. A workpiece is dropped at FUR by the step `VG_TO_FU` in single mode and `FROM_HR_TO_FUR` in multi mode. *Ejecting* a workpiece at the eject station is realized in single mode by the steps `VG_DIRECTLY_EJECT_W` if directly ejected, otherwise by `VG_TO_EJECT_POINT` and `VG_EJECT`. In multi mode the step `TO_EJECT` implements this. All these steps make use of the function blocks `VG_MOVE` or `VG_MOVE_TO_POS` to realize tower movements (FR10). The valve and compressor do not have specific function blocks controlling them (FR11). Instead, they are directly controlled by certain step in the different execution mode programs.

At FUR, *firing* (FR12) a workpiece is realized by the step `FUR_BURN` in single mode and by an eponymous function block in multi mode. Workpieces get *milled* (FR13) in the step `FUR_SAW` in single mode and `SAW` in multi mode. Movements of machines (FR14) are realized inside the some steps of the PRGs.

At SL, there all steps that move the conveyor in order to *transport* the workpiece can be found in the function block `SL_FB` (FR15, FR16, FR17). The algorithm to *detect the color* of a workpiece is taking the minimum of all measured values and compare it to stored thresholds.

A further requirement describes the possibility to execute *CRUD operations* (FR18) on the internal data structures. The *create* operation is realized by functions `newBatchJob`, `newFormat`, and `newWorkpiece`. Operations *read* and *update* are supported by the pointers of the GLVs, pointing to the objects. Thus, the objects can be accessed and manipulated. Deleting an object is possible by setting pointing to a workpiece or batch job to values `nullW` or `nullW`.

In general, all functional requirements are met. The marginal conditions evaluate as follows. The *robustness* of the system is ensured by system tests. However, there is still a great variety of untested input signal scenarios. The *performance* of the application

Functional Requirement	Realized by	Type
FR1	HR_GET_W	FB
FR2	HR_STORE_W	FB
FR3	HR_SORTING_MODE	PRG
FR4	HR_TAKE_W; HR_TAKE_W_FROM_CONVEYOR	FBs
FR5	HR_PUT_W; HR_PUT_W_ON_CONVEYOR	FBs
FR6	HR_MOVE; HR_MOVE_TO_POS	FBs
FR7	Different statements in multiple steps	SFC steps
FR8	VG_TAKE_W; TO_HR_TAKE_W; VG_TO_SL; TO_SL_TAKE_W	SFC steps
FR9	VG_DROP_W; TO_HR_PUT_W; VG_TO_FU; FROM_HR_TO_FUR; VG_DIRECTLY_EJECT_W; VG_EJECT VG_TO_EJECT_POINT; VG_EJECT	SFC steps
FR10	VG_MOVE; VG_MOVE_TO_POS	FBs
FR11	Lots of statements in steps of certain PRGs	Statements
FR12	FUR_BURN	SFC Step & FB
FR13	FUR_SAW; SAW	SFC steps
FR14	FUR_SINGLE_MODE, FUR_MULTI_MODE	PRGs
FR15	SL_FB	FB
FR16	SL_FB	FB
FR17	SL_FB	FB
FR18	newBatchJob; newFormat; newWorkpiece	FUNs

Table 5.1: Functional Requirements Evaluation

is fast enough to realize the factory simulation. In terms of *security*, it is possible to access the PLC, if access to the local area network is given. However, the PLC can be secured via username and password. The application can be altered at any time inside Codesys and reloaded to the PLC in a few seconds, which implies great *maintainability*. Nevertheless, changes should always be tested. There was no focus on *software portability* during implementation, as this constraint is defined by Codesys. The application should be always up and running and should not to be restarted at any time. This is possible when adding new batch jobs via network. This functionality is not yet implemented. The system has to be restarted, if new batch jobs shall be executed. The system is not as *reliable*: it interacts with real world actuators, and, thus, must always be supervised—actuators may create system failures or physical damage.

6
Related Work

Beside this thesis, lots of scientific approaches exist, covering some aspects of the industrial internet of things (IIoT) trend. Akin the factory simulation being described in this thesis, Wien University runs projects dealing with the trend, but with another focus. They built a miniature Industry 4.0 out of Lego to demonstrate their work on a physical showcase and to help them demonstrating certain projects. One of this projects is the project: "Life Cycle Support of Instance-spanning Constraints in flexible Process-Aware Information Systems" (CRISP) which covers the topic process compliance [15]. Another smart factory showcase exists at THW Dresden [16]. It consists of an industry 4.0 factory model, implementing partly automated production processes.

The factory simulation shall be integrated into a manufacturing execution system once. The project "Adventure" of Wien University implements such a platform. Interconnected and integrated BPM platforms become important to cover business processes and their execution [17].

As the factory simulation implements production processes, BPM platforms, supporting production processes become important. A BPM platform to model and execute process models within one platform is the Clavii BPM platform [18]. It focuses on end-users being process model developers as well as executors. A supporting approach is to create different process views, dependent on the range of tasks the user of a BPM platform has [19]. The CaPI platform, implementing the context-aware adoption of process models [20], could make use of the factory simulation context and processes to control the process flow of the factory.

7 Summary and Outlook

In this thesis, a factory simulation has been created. It is based on a Fischertechnik construction kit, and is controlled by a PLC application implementing multiple technical processes. Thereby, sensors and actuators of the Fischertechnik factory can be controlled by the PLC application. The implemented processes are supported by developed data structures, storing states of different stations and workpieces. They can be used to visualize the factories behaviour. In addition, the hard- and software system has been tested on system level.

The implementation supports different production modes: the single mode processes one workpiece at a time. In addition, the multi mode is able to handle multiple workpieces at a time. Furthermore, the sorting mode has been introduced to provide a basis for sorting algorithm implementations on higher software layers.

Future advancements of the factory simulation can comprise the integration of OPC UA to provide data to higher software levels, e.g. for visualization, or external batch job control. The PLC IDE Codesys already provides software components for an OPC UA server [13]. Another desirable extension is data logging. Therefore, IEEE extensible event streams (XES) can be used [21]. Moreover, process mining can be realized to learn from process changes [22] and to discover reference models [23] — based on the data provided by the OPC UA server [24]. Augmented reality technology, such as Microsofts Holo Lens could be integrated, to simulate a supporting system for factory workers having different spectrum of tasks [25, 26].

Bibliography

[1] Baheti, R., Gill, H.: Cyber-Physical Systems. The Impact of Control Technology **12** (2011) 161–166

[2] Dummermuth, E.: Programmable Logic Controller (1976) US Patent 3,942,158.

[3] von Aspern, J.: SPS-Grundlagen: Aufbau, Programmierung (IEC 61131, S7), Simulation, Internet, Sicherheit. Hüthig (2009)

[4] Bolton, W.: Programmable Logic Controllers. Elsevier Science (2015)

[5] Bondurant, D.: Ferroelectronic Ram Memory Family for Critical Data Storage. Ferroelectrics **112** (1990) 273–282

[6] Loth, M.: Grundwissen SPS-Technik. Technische Hochschule Mittelhessen (2010) https://homepages.thm.de/ mlth53/wp-content/uploads/2010/04/sps.pdf. Accessed: 17.11.2017.

[7] Berghof Automation GmbH: EtherCAT Compact Controller ECC2250 Datenblatt. (2017) https://www.berghof-automation.com/fileadmin/user_upload/images/Themen/04_Navigation/06_Systemloesungen/03_SPS_Steuerung/02_Kompakte_Steuerung/01_ECC-2000_Serie/04_ECC2250/Berghof-A_TD_ECC2250_11-16.pdf. Accessed: 17.11.2017.

[8] Kargl, F.: Internet Access at Home. Advanced Concepts of Computer Networks (2016) 20–26

[9] John, K.H., Tiegelkamp, M.: SPS-Programmierung mit IEC 61131-3: Konzepte und Programmiersprachen, Anforderungen an Programmiersysteme, Entscheidungshilfen. VDI-Buch. Springer (2009)

[10] Erickson, K.T.: Programmable Logic Controllers: Hardware, Software Architecture. (2010) https://www.isa.org/standards-publications/isa-publications/intech-magazine/2010/december/automation-basics-programmable-logic-controllers-hardware-software-architecture/. Accessed: 17.11.2017.

[11] 3S-Smart Software Solutions GmbH: CODESYS – industrielle IEC 61131-3 SPS-Programmierung/Automatisierung. (2017) https://de.codesys.com/produkte/codesys-engineering/development-system.html.

[12] Lange, J., Iwanitz, F., Burke, T.J.: OPC: Von Data Access bis Unified Architecture. VDE (2010)

[13] Wagner, R.: Webinar CODESYS OPC UA Server (D). 3S-Smart Software Solutions GmbH (2016) https://www.youtube.com/watch?v=-htrsSi8VzI. Accessed: 17.11.2017.

[14] Wirtz, G.: Systemtest. Fakultät für Wirtschaftsinformatik und angewandte Informatik der Universität Bamberg (2016) http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Hauptaktivitaten-der-Systementwicklung/Software-Implementierung/Testen-von-Software/Systemtest. Accessed: 17.11.2017.

[15] Fdhila, W., Gall, M., Rinderle-Ma, S., Mangler, J., Indiono, C.: Classification and Formalization of instance-spanning Constraints in process-driven Applications. In: International Conference on Business Process Management, Springer (2016) 348–364

[16] Jäpel, N.: Industrie 4.0 Modellfabrik. HTW Dresden - Hochschule für Technik und Wirtschaft Dresden (2017) https://www.htw-dresden.de/de/fakultaet-informatikmathematik/forschung/arbeitsgruppen/smart-production-systems/forschungsprojekte/industrie-40-modellfabrik.html. Accessed: 17.11.2017.

[17] Mangler, J., Rinderle-Ma, S.: CPEE - Cloud Process Exection Engine. In: Int'l Conference on Business Process Management. CEUR-WS.org (2014)

[18] Kammerer, K., Kolb, J., Andrews, K., Bueringer, S., Meyer, B., Reichert, M.: User-centric Process Modeling and Enactment: The Clavii BPM Platform. In: Proceedings of the BPM Demo Session 2015 (BPMD 2015), co-located with the 13th International Conference on Business Process Management (BPM 2015). Number 1418 in CEUR Workshop Proceedings, CEUR-WS.org (2015) 135–139

[19] Kolb, J., Kammerer, K., Reichert, M.: Updatable Process Views for User-centered Adaption of Large Process Models. In: 10th Int'l Conference on Service Oriented Computing (ICSOC'12). Number 7636 in LNCS, Springer (2012) 484–498

[20] Kammerer, K., Mundbrod, N., Reichert, M.: Demonstrating Context-aware Process Injection with the CaPI Tool. In: Demo Track of the 15th International Conference on Business Process Management (BPM 2017). (2017)

[21] : IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams. IEEE Std 1849-2016 (2016) 1–50

[22] Guenther, C.W., Rinderle-Ma, S., Reichert, M., van der Aalst, W.M., Recker, J.: Using Process Mining to Learn from Process Changes in Evolutionary Systems. Int'l Journal of Business Process Integration and Management, Special Issue on Business Process Flexibility **3** (2008) 61–78

[23] Li, C., Reichert, M., Wombacher, A. In: Discovering Reference Models by Mining Process Variants Using a Heuristic Approach. Springer Berlin Heidelberg, Berlin, Heidelberg (2009) 344–362

[24] van der Aalst, W.M.: Process Mining: Data Science in Action. Springer (2016)

[25] Henderson, S.J., Feiner, S.K.: Augmented Reality for Maintenance and Repair (ARMAR). Technical report, Columbia Univ New York Dept of Computer Science (2007)

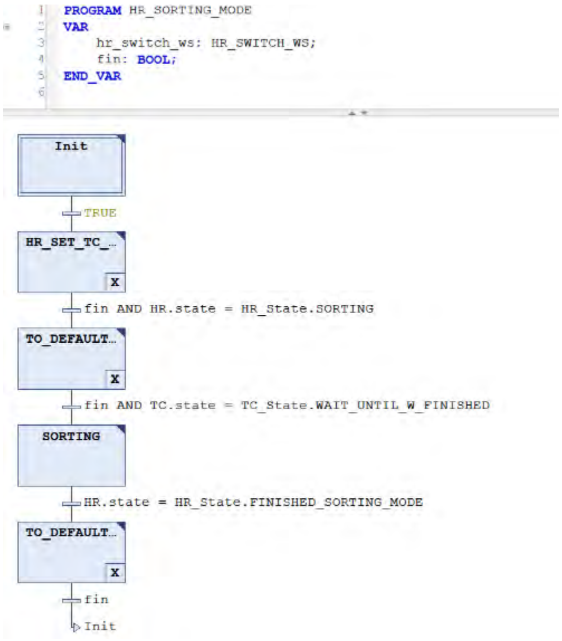
[26] Friedrich, W., Jahn, D., Schmidt, L.: ARVIKA-Augmented Reality for Development, Production and Service. In: ISMAR. Volume 2002. (2002) 3–4

A

Source Code

```
// if sl positions free or if not free if workpiece will not be ejected
// AND VG is not storing
// AND HR does not already decided to take a new workpiece
// THEN
IF ((SL.lbRed AND SL.lbWhite AND SL.lbBlue) OR (SL.currW.format.ejectEnabled AND
    NOT(SL.currW.state = W_State.NULL))) AND NOT(VG.storeW) AND NOT(decidedToTakeANewW) THEN
        HR.bIdx := 0;
        HR.wIdx := 0;
        HR.currW := TC.nullW;
        HR.decidedToTakeANewW := TRUE;
        VG.storeW := FALSE;
ELSIF HR.decidedToTakeANewW THEN
        // get new w out of rack
        IF TC.state = TC_state.W_LOADED THEN
                // set pointers to make sure that they are set
                Hr.bIdx := TC.currBatchJobIdx;
                Hr.wIdx := TC.batchJobQueue.batchJobs[TC.currBatchJobIdx].nextNotProcessedIdx;
                HR.currW := TC.batchJobQueue.batchJobs[Hr.bIdx].workpieces[Hr.wIdx];
                // workpiece is loaded, start production
                HR.state := HR_State.GET_W;
                // signalise that production starts
                TC.state := TC_State.WAIT_UNTIL_W_FINISHED;
        ELSE
                // no workpiece loaded, load it now
                TC.state := TC_State.LOAD_NEXT_W;
        END_IF
ELSE
        // get empty box out of rack
        // set workpiece to get the right box from rack
        IF NOT(SL.lbWhite) OR NOT(SL.lbRed) OR NOT(SL.lbBlue) THEN
                HR.bIdx := SL.bIdx;
                HR.wIdx := SL.wIdx;
                HR.currW := SL.currW;
        ELSE
                HR.bIdx := VG.bIdx;
                HR.wIdx := VG.wIdx;
                HR.currW := VG.currW;
        END_IF
        HR.state := HR_State.GET_BOX;
        // reset flag
        VG.storeW := FALSE;
END_IF
```

Listing A.1: HR_MULTI_MODE – DECIDE Step



```
PROGRAM HR_SORTING_MODE
VAR
    hr_switch_ws: HR_SWITCH_WS;
    fin: BOOL;
END_VAR
```

Init
TRUE
HR_SET_TC_...
X
fin AND HR.state = HR_State.SORTING
TO_DEFAULT...
X
fin AND TC.state = TC_State.WAIT_UNTIL_W_FINISHED
SORTING
HR.state = HR_State.FINISHED_SORTING_MODE
TO_DEFAULT...
X
fin
Init

Figure A.1: HR_SORTING_MODE – SFC Steps

```
IF NOT(fin) THEN
        hr_switch_ws(w1 := TC.batchJobQueue.batchJobs[TC.currBatchJobIdx].
            workpieces[TC.batchJobQueue.batchJobs[TC.currBatchJobIdx].nextNotProcessedIdx],
            w2 := TC.batchJobQueue.batchJobs[TC.currBatchJobIdx].workpieces[TC.batchJobQueue.
            batchJobs[TC.currBatchJobIdx].nextNotProcessedIdx+1]);
        fin := hr_switch_ws.fin;
ELSE
        // sorting finished
        fin := FALSE;
        HR.state := HR_State.FINISHED_SORTING_MODE;
END_IF
```

Listing A.2: HR_SORTING_MODE – SORTING Step

```
IF VG.state = VG_State.AT_HR THEN
        // this can happen if the batch job before was a single mode batch job.
        // Then VG is already AT HR and is not in ready state
        VG.state := VG_State.TO_HR_TAKE_W;
ELSIF VG.state = VG_State.READY AND (HR.state = HR_State.GET_W AND HR.currW.state <> W_State.NULL) THEN
        // go to HR and take w
        VG.state := VG_State.TO_HR_TAKE_W;
ELSIF VG.state = VG_State.READY AND (NOT(SL.lbWhite) OR NOT(SL.lbRed) OR NOT(SL.lbBlue))
    AND SL.currW.state <> W_State.NULL AND (NOT(HR.decidedToTakeANewW) OR HR.currW.state = W_State.NULL) THEN
        // go to SL and take w
        // bend VG pointers to workpiece being picked up
        VG.bIdx := SL.bIdx;
        VG.wIdx := SL.wIdx;
        VG.currW := TC.batchJobQueue.batchJobs[VG.bIdx].workpieces[VG.wIdx];
        VG.state := VG_State.TO_SL_TAKE_W;
ELSIF VG.state = VG_State.TO_SL_TAKE_W_FINISHED THEN
        // reset SL pointers
        SL.bIdx := 0;
        SL.wIdx := 0;
        SL.currW := TC.nullW;
        // at SL, w taken, decide if going to EJECT or HR
        IF VG.currW.format.ejectEnabled THEN
                // go to eject point and eject w
                VG.state := VG_State.TO_EJECT;
        ELSE
                // go to HR and put down w
                VG.state := VG_State.TO_HR_PUT_W;
        END_IF
ELSIF VG.state = VG_State.TO_HR_TAKE_W_FINISHED THEN
        // at HR, w taken, decide if going to FUR or EJECT
        IF VG.currW.format.ejectEnabled AND NOT(VG.currW.format.colorCheckEnabled
            OR VG.currW.format.furEnabled OR VG.currW.format.sawEnabled) THEN
                // go to eject point and eject w
                VG.state := VG_State.TO_EJECT;
        ELSE
                // go to FUR and put down w
                // but only if FUR-succer is at its default pos
                IF FUR.sSuckerPosRa THEN
                        VG.state := VG_State.TO_FUR_PUT_W;
                END_IF
        END_IF
ELSIF VG.state = VG_State.TO_EJECT_FINISHED OR VG.state = VG_State.TO_FUR_PUT_W_FINISHED
    OR VG.state = VG_State.TO_HR_PUT_W_FINISHED THEN
        // eject or putting w to FUR finished, back to default position for calibrating axes
        // likewise if VG put down w at HR
        VG.state := VG_State.TO_DEFAULT_POS;
ELSIF VG.state = VG_State.AT_DEFAULT_POS THEN
        // state = ready if VG is at default pos or if VG put down a w at HR --> VG is ready for next w
        VG.state := VG_state.READY;
        // reset own pointers
        VG.bIdx := 0;
        VG.wIdx := 0;
        VG.currW := TC.nullW;
END_IF
```

Listing A.3: VG_MULTI_MODE – DECIDE Step

```
(*
        States in which a BatchJob can be
*)
TYPE BatchJobState :
(
        // NULL always expresses that the object has not been initialized yet
        NULL := 0,

        // BatchJob has been initialized and is waiting to be executed
        WAITING := 1,

        // BatchJob has been started and is being executed now
        STARTED := 2,

        // BatchJob has been paused during execution
        PAUSED := 3,

        // BatchJob has been aborted. This may happen at any time
        ABORTED := 4,

        // BatchJob has been finished
        FINISHED := 5
);
END_TYPE
```

Listing A.4: BatchJobState Enum – Implementation

B

PLC I/O Pin Mapping

furnace (FUR)			Ext.	
var name	type	pin	I	O
sRaPosSucker	IN	IX300.0	1	
sRaPosConveyorBelt	IN	IX300.1	2	
lbConveyorBelt	IN	IX300.2	3	
sRaPosSaw	IN	IX300.3	4	
sSuckerPosRa	IN	IX300.4	5	
sFurnanceSliderInside	IN	IX300.5	6	
sFurnanceSliderOutside	IN	IX300.6	7	
sSuckerPosFF	IN	IX300.7	8	
lbFurnance	IN	IX301.0	9	
mRaClockwise	OUT	QX40.0		1
mRaCClockwise	OUT	QX40.1		2
mConveyorBeltForward	OUT	QX40.2		3
mSaw	OUT	QX40.3		4
mFurnanceSliderIn	OUT	QX40.4		5
mFurnanceSliderOut	OUT	QX40.5		6
mSuckertoFurnance	OUT	QX40.6		7
mSuckertoRa	OUT	QX40.7		8
lightFurnance	OUT	-		
compressor	OUT	QX41.0		9
valveVacuum	OUT	QX41.1		10
valveLowering	OUT	QX41.2		11
valveFurnanceDoor	OUT	QX41.3		12
valveSlider	OUT	QX41.4		13

Table B.1: PLC I/O Mapping of Furnace

high rack (HR)			PLC	
var name	type	pin	I	O
sHorizontal	IN	IX0.0	1	
lbIn	IN	IX0.1	2	
lbOut	IN	IX0.2	3	
sVertical	IN	IX0.3	4	
sCFront	IN	IX0.4	5	
sCBack	IN	IX0.5	6	
mCbForward	OUT	QX0.0		1
mCbBackward	OUT	QX0.1		2
mHorizontalLeft	OUT	QX0.2		3
mHorizontalRight	OUT	QX0.3		4
mVerticalDown	OUT	QX0.4		5
mVerticalUp	OUT	QX0.5		6
mCForward	OUT	QX0.6		7
mCBackward	OUT	QX0.7		8
vacuum gripper (VG)			PLC	
var name	type	pin	I	O
sVertical	IN	IX0.6	7	
sHorizontal	IN	IX0.7	8	
sRotation	IN	IX1.0	9	
encVertical	IN	IX1.3	12	
encHorizontal	IN	IX1.4	13	
encRotation	IN	IX1.5	14	
mVerticalUp	OUT	QX1.0		9
mVerticalDown	OUT	QX1.1		10
mHorizontalBackward	OUT	QX1.2		11
mHorizontalForward	OUT	QX1.3		12
mRotationClockwise	OUT	QX1.4		13
mRotationCClockwise	OUT	QX1.5		14
compressor	OUT	QX1.6		15
valve	OUT	QX1.7		16

Table B.2: PLC I/O Mapping of High Rack and Vacuum Gripper

sorting line (SL)			Ext.	
var name	type	pin	I	O
IN	IX301.1	10		
IN	IX301.2	11		
IN	IX301.3	12		
IN	ID37	-		
IN	IX301.5	14		
IN	IX301.6	15		
IN	IX301.7	16		
OUT	QX41.5		14	
OUT	QX41.6		15	
OUT	QX41.7		16	
OUT	QX41.7		16	
OUT	QX41.7		16	
START/STOP switch			PLC	
var name	type	pin	I	O
hwSwitch	IN	IX1.7	16	

Table B.3: PLC I/O Mapping of Sorting Line and Switch

C

Source Code of Test Cases

```
(*
        Defining formats
        newFormat(furEnabled,sawEnabled,colorCheckEnabled,ejectEnabled,furDuration,sawDuration)
*)
format1  := newFormat(FALSE, FALSE, FALSE, FALSE, T#5S, T#5S);
format2  := newFormat(FALSE, FALSE, FALSE, TRUE,  T#5S, T#5S);
format3  := newFormat(FALSE, FALSE, TRUE,  FALSE, T#5S, T#5S);
format4  := newFormat(FALSE, FALSE, TRUE,  TRUE,  T#5S, T#5S);

format5  := newFormat(FALSE, TRUE, FALSE, FALSE, T#5S, T#5S);
format6  := newFormat(FALSE, TRUE, FALSE, TRUE,  T#5S, T#5S);
format7  := newFormat(FALSE, TRUE, TRUE,  FALSE, T#5S, T#5S);
format8  := newFormat(FALSE, TRUE, TRUE,  TRUE,  T#5S, T#5S);

format9  := newFormat(TRUE, FALSE, FALSE, FALSE, T#5S, T#5S);
format10:= newFormat(TRUE, FALSE, FALSE, TRUE,  T#5S, T#5S);
format11:= newFormat(TRUE, FALSE, TRUE,  FALSE, T#5S, T#5S);
format12:= newFormat(TRUE, FALSE, TRUE,  TRUE,  T#5S, T#5S);

format13:= newFormat(TRUE, TRUE, FALSE, FALSE, T#5S, T#5S);
format14:= newFormat(TRUE, TRUE, FALSE, TRUE,  T#5S, T#5S);
format15:= newFormat(TRUE, TRUE, TRUE,  FALSE, T#5S, T#5S);
format16:= newFormat(TRUE, TRUE, TRUE,  TRUE,  T#5S, T#5S);
```

Listing C.1: Formats of Workpieces the Testcases depend on

```
(*
	Testcase 1: Test all different possible formats in single mode
*)
workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_11, RackPos.RACK_11, RackPos.RACK_11, format1);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_12, RackPos.RACK_12, RackPos.RACK_12, format2);
workpieces[3] := newWorkpiece('unknown', W_State.AT_RACK_13, RackPos.RACK_13, RackPos.RACK_13, format3);
workpieces[4] := newWorkpiece('unknown', W_State.AT_RACK_21, RackPos.RACK_21, RackPos.RACK_21, format4);
workpieces[5] := newWorkpiece('unknown', W_State.AT_RACK_22, RackPos.RACK_22, RackPos.RACK_22, format5);
workpieces[6] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format6);
workpieces[7] := newWorkpiece('unknown', W_State.AT_RACK_31, RackPos.RACK_31, RackPos.RACK_31, format7);
workpieces[8] := newWorkpiece('unknown', W_State.AT_RACK_32, RackPos.RACK_32, RackPos.RACK_32, format8);
addBatchJobToQueue(newbatchJob(8, workpieces, ExecutionMode.single));

workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_11, RackPos.RACK_11, RackPos.RACK_11, format9);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_12, RackPos.RACK_12, RackPos.RACK_12, format10);
workpieces[3] := newWorkpiece('unknown', W_State.AT_RACK_13, RackPos.RACK_13, RackPos.RACK_13, format11);
workpieces[4] := newWorkpiece('unknown', W_State.AT_RACK_21, RackPos.RACK_21, RackPos.RACK_21, format12);
workpieces[5] := newWorkpiece('unknown', W_State.AT_RACK_22, RackPos.RACK_22, RackPos.RACK_22, format13);
workpieces[6] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format14);
workpieces[7] := newWorkpiece('unknown', W_State.AT_RACK_31, RackPos.RACK_31, RackPos.RACK_31, format15);
workpieces[8] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format16);
addBatchJobToQueue(newbatchJob(8, workpieces, ExecutionMode.single));
```

Listing C.2: Single Mode Testcase – Implementation

```
(*
	Testcase 2: Test all different possible formats in multi mode
*)
workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_11, RackPos.RACK_11, RackPos.RACK_11, format1);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_12, RackPos.RACK_12, RackPos.RACK_12, format2);
workpieces[3] := newWorkpiece('unknown', W_State.AT_RACK_13, RackPos.RACK_13, RackPos.RACK_13, format3);
workpieces[4] := newWorkpiece('unknown', W_State.AT_RACK_21, RackPos.RACK_21, RackPos.RACK_21, format4);
workpieces[5] := newWorkpiece('unknown', W_State.AT_RACK_22, RackPos.RACK_22, RackPos.RACK_22, format5);
workpieces[6] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format6);
workpieces[7] := newWorkpiece('unknown', W_State.AT_RACK_31, RackPos.RACK_31, RackPos.RACK_31, format7);
workpieces[8] := newWorkpiece('unknown', W_State.AT_RACK_32, RackPos.RACK_32, RackPos.RACK_32, format8);
addBatchJobToQueue(newbatchJob(8, workpieces, ExecutionMode.multi));

workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_11, RackPos.RACK_11, RackPos.RACK_11, format9);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_12, RackPos.RACK_12, RackPos.RACK_12, format10);
workpieces[3] := newWorkpiece('unknown', W_State.AT_RACK_13, RackPos.RACK_13, RackPos.RACK_13, format11);
workpieces[4] := newWorkpiece('unknown', W_State.AT_RACK_21, RackPos.RACK_21, RackPos.RACK_21, format12);
workpieces[5] := newWorkpiece('unknown', W_State.AT_RACK_22, RackPos.RACK_22, RackPos.RACK_22, format13);
workpieces[6] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format14);
workpieces[7] := newWorkpiece('unknown', W_State.AT_RACK_31, RackPos.RACK_31, RackPos.RACK_31, format15);
workpieces[8] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format16);
addBatchJobToQueue(newbatchJob(8, workpieces, ExecutionMode.multi));
```

Listing C.3: Multi Mode Testcase – Implementation

```
(*
        Testcase 3: Test different positions in sorting mode
*)
workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_11, RackPos.RACK_11, RackPos.RACK_11, format1);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_12, RackPos.RACK_12, RackPos.RACK_12, format2);
workpieces[3] := newWorkpiece('unknown', W_State.AT_RACK_13, RackPos.RACK_13, RackPos.RACK_13, format3);
workpieces[4] := newWorkpiece('unknown', W_State.AT_RACK_21, RackPos.RACK_21, RackPos.RACK_21, format4);
workpieces[5] := newWorkpiece('unknown', W_State.AT_RACK_22, RackPos.RACK_22, RackPos.RACK_22, format5);
workpieces[6] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format6);
workpieces[7] := newWorkpiece('unknown', W_State.AT_RACK_31, RackPos.RACK_31, RackPos.RACK_31, format7);
workpieces[8] := newWorkpiece('unknown', W_State.AT_RACK_32, RackPos.RACK_32, RackPos.RACK_32, format8);
addBatchJobToQueue(newbatchJob(8, workpieces, ExecutionMode.sorting));

workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_11, RackPos.RACK_11, RackPos.RACK_11, format9);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_12, RackPos.RACK_12, RackPos.RACK_12, format10);
workpieces[3] := newWorkpiece('unknown', W_State.AT_RACK_13, RackPos.RACK_13, RackPos.RACK_13, format11);
workpieces[4] := newWorkpiece('unknown', W_State.AT_RACK_21, RackPos.RACK_21, RackPos.RACK_21, format12);
workpieces[5] := newWorkpiece('unknown', W_State.AT_RACK_22, RackPos.RACK_22, RackPos.RACK_22, format13);
workpieces[6] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format14);
workpieces[7] := newWorkpiece('unknown', W_State.AT_RACK_31, RackPos.RACK_31, RackPos.RACK_31, format15);
workpieces[8] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format16);
addBatchJobToQueue(newbatchJob(8, workpieces, ExecutionMode.sorting));
```

Listing C.4: Sorting Mode Testcase – Implementation

```
(*
        Testcase 4: Test all transitions between different execution modes
        Test sequence: multi -> multi -> single -> single -> sorting -> sorting
        -> multi -> sorting -> single -> multi
        ---> covers all transitions
        As a test format, format15 is used
*)
workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format15);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format15);
addBatchJobToQueue(newbatchJob(2, workpieces, ExecutionMode.multi));

workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format15);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format15);
addBatchJobToQueue(newbatchJob(2, workpieces, ExecutionMode.multi));

workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format15);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format15);
addBatchJobToQueue(newbatchJob(2, workpieces, ExecutionMode.single));

workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format15);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format15);
addBatchJobToQueue(newbatchJob(2, workpieces, ExecutionMode.single));

workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format15);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format15);
addBatchJobToQueue(newbatchJob(2, workpieces, ExecutionMode.sorting));

workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format15);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format15);
addBatchJobToQueue(newbatchJob(2, workpieces, ExecutionMode.sorting));

workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format15);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format15);
addBatchJobToQueue(newbatchJob(2, workpieces, ExecutionMode.multi));

workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format15);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format15);
addBatchJobToQueue(newbatchJob(2, workpieces, ExecutionMode.sorting));

workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format15);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format15);
addBatchJobToQueue(newbatchJob(2, workpieces, ExecutionMode.single));

workpieces[1] := newWorkpiece('unknown', W_State.AT_RACK_23, RackPos.RACK_23, RackPos.RACK_23, format15);
workpieces[2] := newWorkpiece('unknown', W_State.AT_RACK_33, RackPos.RACK_33, RackPos.RACK_33, format15);
addBatchJobToQueue(newbatchJob(2, workpieces, ExecutionMode.multi));
```

Listing C.5: Testcase for testing Transitions between different Execution Modes – Implementation

List of Figures

List of Tables

Listings

Name: Manuel Göster Student ID: 886800

Declaration

I hereby declare that I have developed and written the enclosed Bachelor Thesis by myself and have not used sources or means without declaration in the text. This Bachelor Thesis has never been used in the same or in a similar version to achieve an academic grading or was published elsewhere.

Ulm, ...

Manuel Göster