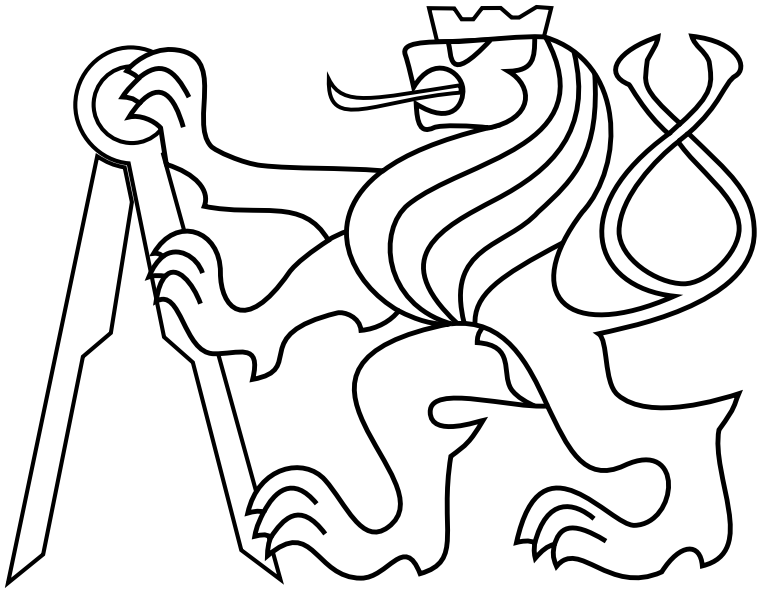
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

MASTER'S THESIS



Oxana Kovbasjuková

Distributed control on SmartWire Device Technology without a master node

Department of Control Engineering

Thesis supervisor: **Ing. Michal Sojka, Ph.D.**

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Kovbasjuková Oxana** Personal ID number: **434704**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

Branch of study: **Cybernetics and Robotics**

II. Master's thesis details

Master's thesis title in English:

Distributed control on SmartWire Device Technology without a master node

Master's thesis title in Czech:

Distribuované řízení na sběrnici SmartWire Device Technology bez master zařízení

Guidelines:

1) Get familiar with Smart Wire Device Technology (SWD), Smart Wire Protocol and device firmware implementation.
2) Design an extension of SWD technology that allows multiple devices to communicate with each other without the master device. Implement changes in device firmware according to the design.
3) Integrate master-less communication with so called Local Control Function (LCF), that allow users to program different device behaviour.
4) Implement loader of Local Control Function code into the device through the SWD bus.
5) Demonstrate the result with several connected devices implementing a simple use case with specific Local Control Functions. Document the results.

Bibliography / sources:

[1] Eaton, "SmartWire-DT: The System", users manual, 5th ed., 5/2015.
https://www.eaton.com/content/dam/eaton/products/industrialcontrols,drives,automation&sensors/smartwire-dt-intelligent-wiring-system/user-guides/smartwire-dt-system-manual-mn05006002z.pdf
[2] Eaton, "SmartWire-DT Protocol Structure", reference manual, 2010.

Name and workplace of master's thesis supervisor:

Ing. Michal Sojka, Ph.D., Embedded Systems, CIIRC

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **10.02.2020** Deadline for master's thesis submission: **14.08.2020**

Assignment valid until: **30.09.2021**

Ing. Michal Sojka, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to thank my supervisors Ing. Michal Sojka, Ph.D. for valuable feedback and pieces of advice, and Ing. Pavel Dědourek for his guidance and shared wisdom. I would like to also thank my parents for their support and my cats Kernel and Sisi for their meows of encouragement and for always reminding me of the bright side of life.

Abstract

SmartWire-DT is a communication technology developed by Eaton for interconnecting industrial devices. These devices are usually controlled by a programmable logic controller, which is often more expensive than the devices themselves. Decentralized decision making removes the need for the controller, which reduces the cost of the system. This thesis aims to implement support for the decentralization of the industrial devices connected by the SmartWire-DT. To this end, we modify the SmartWire-DT device firmware to allow receiving data from all devices rather than only from the master. We implement support for the execution of simple logic control programs on the devices themselves and develop a loader that allows reprogramming of devices on the fly. Our final demonstrator shows the functionality of the proposed solution. The application of the results achieved in this thesis reduces costs by hundreds of Euros for every SmartWire-DT bus due to the lack of the programmable logic controller. Besides that, the delay of the customer-specified data is at least two times smaller, depending on the bus setup even more.

Abstrakt

SmartWire-DT je komunikační technologie vyvíjená společností Eaton, která je zaměřená na propojení průmyslových zařízení. Zařízení jsou obvykle řízena řídící jednotkou, která je mnohdy dražší než samotná zařízení. Rozprostřením schopnosti rozhodování na jednotlivá zařízení ušetříme zákazníkům peníze. Cílem diplomové práce je podpora pro decentralizaci průmyslových zařízení, která jsou propojena sběrnicí SmartWire-DT. Za tímto účelem upravujeme firmware zařízení tak, aby bylo schopno přijímat data ze všech zařízení a nejen od řídící jednotky. Implementujeme podporu pro samostatné řízení v zařízeních samotných. Vyvýjíme také program na nahrávání změn aplikace uvnitř zařízení při jejich nasazení.Názorná ukázka na konci práce znázorňuje správnou funkcionalitu řešení. Produkt práce snižuje náklady o stovky Eur na každou sběrnici díky absenci řídící jednotky. Navíc mají data, která si zákazník specifikuje, minimálně dvakrát menší dopravní zpoždění.

Contents

List of Figures

1 Introduction

Automation and its aspects heavily influence the modern industry. The increased speed of production, reduction of labour-intensive work, and increased safety standards are just some of the perks of highly-automated manufacturing. Although the history of automation spans several decades, new concepts and their applications are being developed now more than ever. The current state-of-the-art technology in computers, robotics, and additive manufacturing enable engineers to come up with improvements that used to be beyond our imagination.

Devices are defined by their hardware (HW in short) and software (SW in short). From an HW point of view, most of the devices need to be connected by the cables that supply electrical energy but also cables for communication. Devices need to have implemented communication protocols in order to be able to communicate. SW also defines the behaviour od the device. SW controlling the HW is called firmware (FW in short). Devices can be divided between smart devices and simple devices. Simple devices do not make any decision on their own and rely on a smart device to decide what to do. Smart devices have a Central processing unit (**CPU** in short) and either control their behaviour or control a whole network of the devices. Devices that control the whole network of the devices are usually (micro)computers. Such devices for industrial use are Programmable Logic Controller (**PLC** in short).

Distributed Control System is an automated control system that consists of geographically distributed control elements. Distributed Control System (**DCS** in short) consists of a few levels defined by the standard. Sensors and actuators are on the lowest level. The second level consists of microcontrollers. The third level contains supervisory computers and monitoring production. The top-level comprises production control. Controllers on levels from two to the top-level often are PLCs. PLC are widely adopted as automation controllers for their reliability and possibility to diagnose faults. PLCs control devices on the lower levels by master/slave communication protocols. These protocols implement asymmetric data transfer where one device (master) controls one or more slave devices.

One of the communication technologies on the market is SmartWire Device Technology (**SmartWire-DT** in short) that allows communicating over 8 wires in one cable [2]. SmartWire-DT protocol is a master/slave type of protocol and it is used to connect Eaton devices. Figure 1 illustrates the use of the SmartWire-DT.

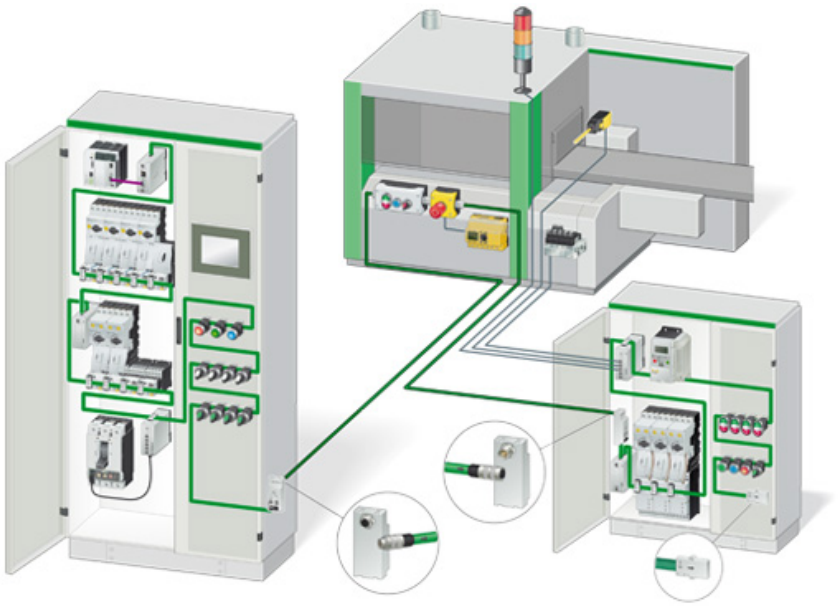


Figure 1: Example of industrial devices connected by industrial bus [1]

This master thesis concentrates on the two lowest levels of DCS. Specifically, the aim is the control of the smart actuators and smart sensors (**devices** for future references) in the factories and communication between said devices. This master thesis implements changes in firmware of devices and presents a more user-friendly way of programming behaviour of said devices.

It is expected that the results of this thesis bring benefit such as decreasing the time delay in data transmission, decreasing of the communication load and removing the requirement for the presence of the master device which results in cost reduction. We can remove the master device because we shift control to the lowest possible level of DCS. The designed and implemented solution is beneficial predominantly for the automation industry but can be valuable also in home technologies and others.

Thesis requirements are discussed in more detail in Section 3.

2 Background

2.1 SmartWire Device Technology

SmartWire-DT is an Eaton proprietary field bus system with integrated communication for industrial sensors, actuators, circuit breakers and input and output devices. SmartWire-DT cable has two variants: ribbon cable and round cable. Round cable is for long distances while ribbon cable connects local devices. The bus can reach up to 600 meters without repeaters. Ribbon cable is shown in Figure 2.

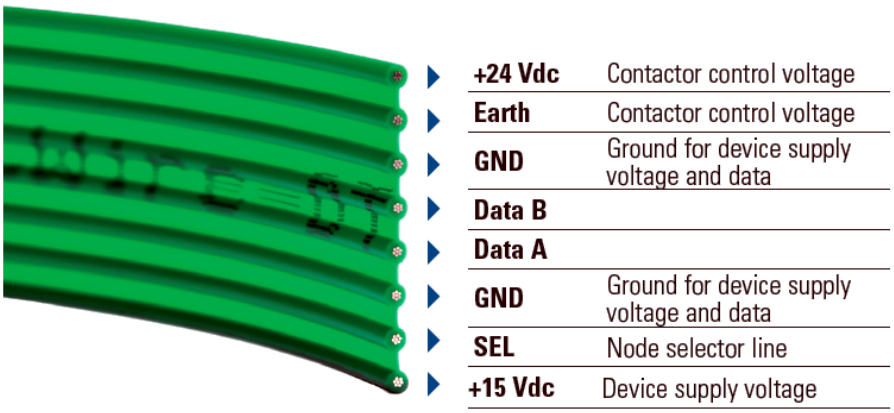


Figure 2: flat SmartWire-DT cable [2]

The communication protocol is based on a concept of master-slave communication and one master device (master in short) can be connected with up to 99 slave devices [5].

SmartWire-DT offers two types of data transmission:

- cyclic data transmission and
- acyclic data transmission.

Before communication starts, the bus needs to be initialized.

2.1.1 Initialization

The bus establishes a communication in few steps. The first step is master initializing the bus by obtaining the configuration (setup) of the devices on the bus and storing information about each of the slave devices. Master uses stored configuration to create a

data frame of the correct length and periodically sends the data frame. Each data frame consists of slots reserved for each slave device. Data are propagated on the bus via slots in the data frame. Size of the slot depends on the individual slave device. Therefore the length of the data frame depends on devices. Length of the data frame influences speed of the communication.

SmartWire-DT bus needs a master device to run the communication protocol. Every device is aware of their address from bus initialization. Master assigns addresses automatically via the Node selector line (see Figure 2) and starts sending the data frame cyclically.

2.1.2 Cyclic data transmission

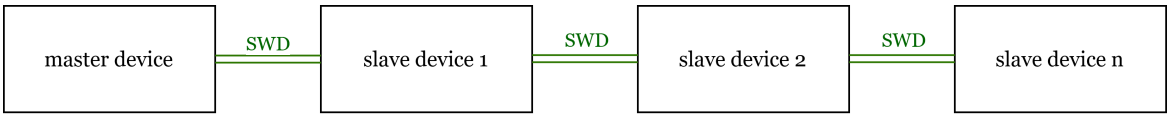
Typical applications use primarily cyclic data transmission. As described above, all the slave devices define the number of output bytes and input bytes they need. Master sends the data frame as fast as possible (see equation). Cycle time (period of cyclic data transmission) depends primarily on the length of the data frame which contains output and input bytes of the slave devices. The number of devices itself has only a small impact (see Figure 4). Symbol rate for the flat SmartWire-DT cable is 250 Kbaud. Cycle time for 25 devices with the length of the data frame 50 Bytes on the bus is 3.4 ms and 100 devices with totally 200 Bytes has cycle time 10 ms. One data frame can have up to 1000 Bytes.

SmartWire-DT cycle time can be calculated using the following formula, where each byte is calculated to comprise 10 bits since a start bit and a stop bit are required for each byte for synchronization purposes [3]:

$$t[ms] = \frac{1}{C}(n \cdot 10bit + 2bit \cdot n_{TN} + 30bit \cdot 10),$$

where n is number of input and output bytes of the devices, C is Baud rate [kBit/s] and n_{TN} is number of the devices.

Structure of I/O data bytes on the cyclic data frame on Figure 3:



...	outputs for slave 1	outputs for slave 2	...	outputs for slave n	inputs from slave 1	inputs from slave 2	...	inputs from slave n	CRC

Figure 3: Structure of a cyclic data frame

The dependency of the cycle time on the input and output data bates is presented on Figure 4 with the following characteristics:

1. 1 SmartWire-DT device with n data and speed 125 kBit/s
2. 99 SmartWire-DT devices with n data and speed 125 kBit/s
3. 1 SmartWire-DT devices with n data and speed 250 kBit/s
4. 99 SmartWire-DT devices with n data and speed 250 kBit/s

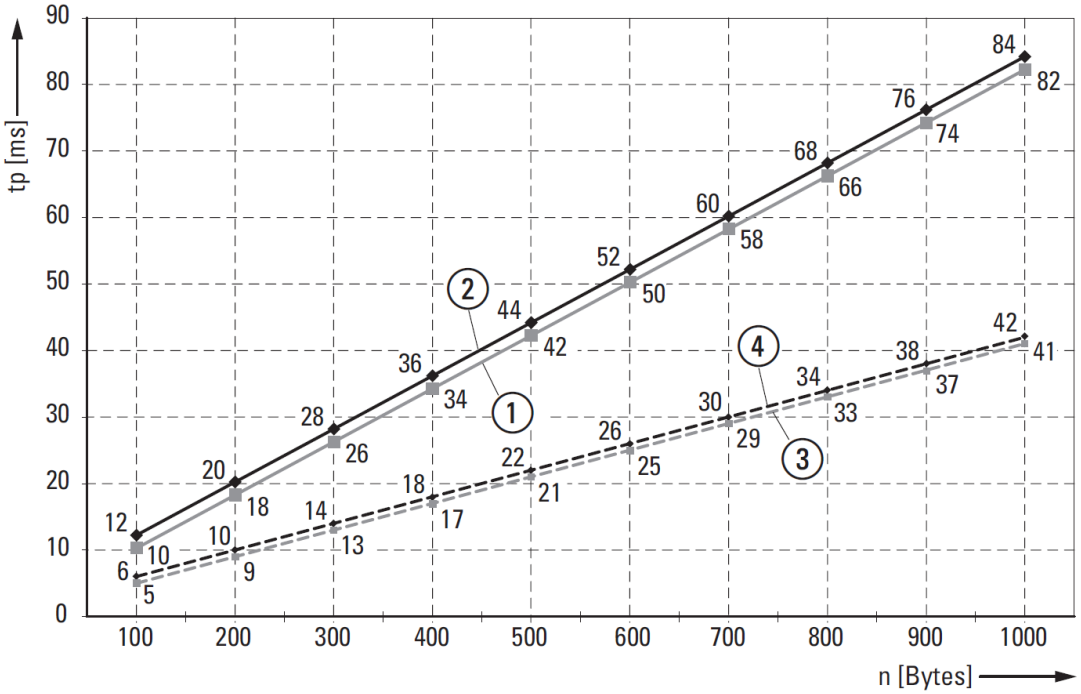


Figure 4: SmartWire-DT cycle time based on the amount of transmitted payload data bytes [3]

Every slave device contains three pieces of information. One is a state of the device, second is an output byte(s) of the device and last is an input byte(s) of the device. Slave extracts its output byte(s) from the data frame. Master sends outputs for every slave device on the bus. See Figure 3 for a structure of the cyclic data frame.

Every device (master and slaves as well) calculates a CRC32 over the data frame. At the end of the data frame, the master sends the calculated CRC32 and all devices compare it to the self calculated value. If the values do not match, the device can set a flag in the data frame and dis-acknowledge the whole data frame. Dis-acknowledged data frame will be ignored by all devices. If a device is disconnected (unplugged from the SmartWire-DT bus) during operation, the master fills the data frame with zeros at the right position and the bus can keep on working.

2.1.3 Acyclic data transmission

Acyclic data transmission is object orientated transmission for diagnosis of the bus system and configuration of single devices. A single acyclic data frame can be sent between two cyclic data frames. Each device can initiate an acyclic data transmission and each device can receive an acyclic data transmission. The device indicates its wish for acyclic data transmission by setting a bit flag in the cyclic data frame. After receiving the data frame with the flag the master grants the access right to the device.

Acyclic data frame does not propagate the inputs and outputs of the slave device. Structure of the data frame is defined by command at the beginning of the data frame and by the length of the acyclic data frame.

Services for acyclic transmission are either:

- Confirmed services (read and write) or
- Unconfirmed services (diagnosis).

Acyclic data transmission offers three modes of operation defined by the number of receivers:

Based on the number receivers, the acyclic datagram is either:

- Unicast datagram (one transmitter and one receiver),
- Multicast datagram (one transmitter and several receivers), and
- Broadcast datagram (one transmitter and all receivers).

The application of an SmartWire-DT device can provide several objects for the acyclic data exchange. The significance of these objects is device-specific or application-specific. Communication can use up to 256 objects per device. Each object can be associated with a maximum of 120 bytes of data [2].

2.1.4 SmartWire-DT Devices

Eaton manufactures industrial devices of a great variety. Some of the most used devices are analogue or digital I/O modules, contactors, circuit breakers, control relays, frequency drives, soft starters, electronic motor starters, push-buttons, selector switches, stack lights, pneumatic and hydraulic valves, sensors, etc.

The core of the SmartWire-DT devices is an application-specific integrated circuit (ASIC [6]). ASIC 1 and ASIC 2 are two versions of custom manufactured chips used in SmartWire-DT devices. The chip contains firmware which is executed on two processors. Versions of ASIC have different firmware with similar behaviour. Firmware (FW) of these devices is written in C language. Both processors have their RAM and very limited shared flash memory. Overall free memory is a limiting resource for the development of new features for SmartWire-DT devices.

Figure 5: Eaton industrial devices

2.1.4.1 Diagnostics

SmartWire-DT offers a variety of diagnostic information that provides details on the network and module status in the control system. SmartWire-DT contains two types of diagnostic information [2]:

- General network/module diagnostics, which is available for all SmartWire-DT modules.
- Individual, module-specific diagnostics.

Individual diagnostics allows to report for example error cause 'short-circuit/overload on at least one output' for the I/O modules.

2.1.4.2 Multi-core communication

The ASIC2 is based on dual ARM-M0+ microcontroller. The two cores are called Communication CPU (CCPU) and Application CPU (ACPU). The ACPU implements applications using the peripherals and selects the communication stack. The CCPU implements industrial communication standard protocol (SmartWire-DT, MODBUS RTU, ...). Cores communicate with each other only by using a communication interface. The interface is based on requests and callbacks and accesses mailbox that is stored in memory shared between the CPUs. Figure 6 depicts a communication between the CPUs.

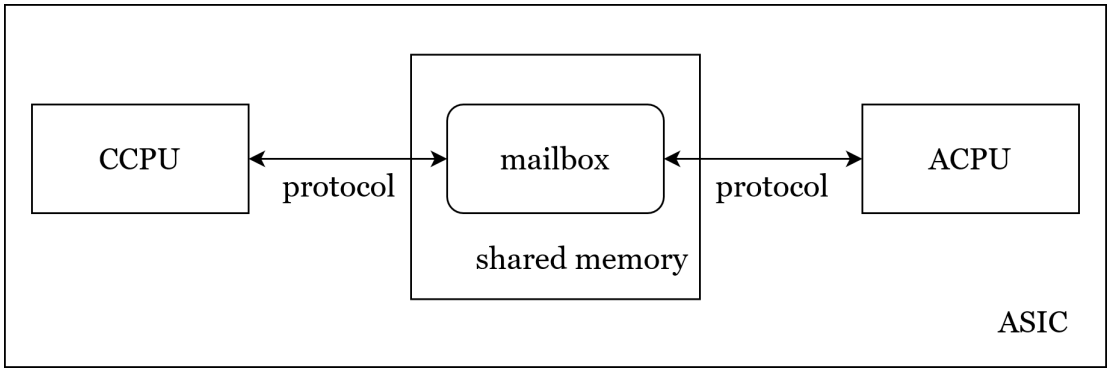


Figure 6: Data propagation between the cores

2.1.4.3 CFG of a device

Data load of the device is stored as a number called configuration object (CFG in short). CFG has a fixed length of 4 bytes and contains the description of the input and output data used by the device. Figure 7 shows an example of devices and their data load. Every device is aware of how many bytes it sends and receives based on the CFG. CFGs of all the devices are propagated in the initialization stage of the communication. The array of all CFGs is then stored in the master from where it can be obtained.

The SmartWire-DT device can have multiple profiles. Profiles have different CFGs so the same SmartWire-DT devices can have a different amount of I/O data.

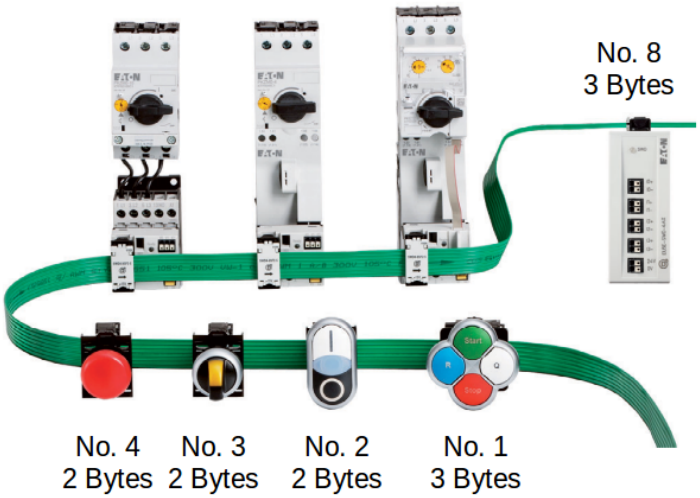


Figure 7: Data load of individual SmartWire-DT devices on the bus (defined by CFG)

2.1.5 Standalone Coordinator

A Standalone Coordinator (**SAC** in short) is as a gateway with no field bus or PLC, thus has no source for output data and no destination for input data [7]. SAC fills all the output data fields with zeros. SAC operates as a plug and play device and adds a clock to the SmartWire-DT communication. As a result, the cyclic data frame is established without a master device and SmartWire-DT communication fully works. SAC is unable to detect missing devices and communication on the bus will continue without it. When the device is replugged to the SmartWire-DT bus, the regain algorithm will reestablish the devices and bus communication.

SAC is used in the thesis to present master less functionality of the connected SmartWire-DT devices.

Figure 8: Standalone Coordinator

2.2 Compilers

A compiler translates a language into byte code. Compiler defined by Mogensen is split into several phases [8]. Phases usually operate in sequence, but some can be interleaved. Mogensen describes front-end phases used in this thesis as following:

Lexical analysis	Reading the input text and dividing it into tokens. Tokens represent symbols in the programming language.
Syntax analysis	Arranging list of tokens in a tree structure (syntax tree).
Type checking	Analysing the syntax tree to determine violation of the language's grammar.
Code generation	Translating the program to an intermediate code.

Table 1: Phases of compilation process [8]

Grammar

Grammar is a finite set of rules for a particular language as Chomsky [9] defined. Compilers use grammar to checks for a syntax violation. Grammar evaluation in this thesis is inspired by Melichar et al. [10].

Abstract syntax tree

Abstract syntax tree (**AST** in short) are defined in [8] as a tree whose leaves are tokens from the input text and when leaves are read from left to right, the result is the same as the input. AST is a representation of the source code, which allows the generation of the intermediate code.

3 Problem statement

This thesis consists of three main problems to solve. The first problem is that SmartWire-DT devices do not have access to all the data from the bus. All the data are periodically updated on the bus, but every device filters those data and keeps only the data that are destined to it. We would like to allow devices to access all the data and decide inside the application CPU whether to keep the specific data or not. Accessing the data from other devices allows simulating device to device (**D2D** in short) communication. This introduces another subproblem of how to define what data to access to prevent wasting of the memory.

The second problem is related to control of the SmartWire-DT devices. Every device has an application defining its default behaviour but device controls its peripherals based on data from the master device. To implement the behaviour of the device, we need to define language so that customers can implement it by themselves. Then we need to design and implement the compiler that takes the program in the specified language and returns byte code for the state machine inside the device (interpreter). The devices already have implemented state machine inside the firmware as described in [4].

The last problem is how to load changes in the program to the SmartWire-DT device. The resulting loader has to load byte code discussed in the previous paragraph and the D2D communication specifics from the first paragraph. For the clarity of the thesis we are introducing terminology **variable deifinitions** that refers to D2D communication specifics. Variable definitions identify data from the bus that we want to store.

3.1 Thesis requirements

This section summarizes the requirements for the work developed in this thesis.

- Data are propagated from the SmartWire-DT bus to the Communication Central processing unit (CCPU).
- Variables (data of interest) are stored as variables in the Application Central processing unit (ACPU)
- Definition of variables is modifiable.
- Compiler supports logic control.
- Compiler supports variables.
- Interpreter supports variables.
- Byte code inside the SmartWire-DT device is modifiable.

- Loader can upload new variable definitions into the SmartWire-DT device.
- Loader can upload new byte code into the SmartWire-DT device.
- Keep compatibility of the modified device with unmodified devices.
- All the features above are demonstrated in a demo use case.

3.2 Thesis tasks

This section lists individual sub-tasks for the work developed in this thesis. It expands on the thesis guidelines.

- Study firmware (FW) of the device and SmartWire-DT communication protocol.
- Find out the part of a device FW that handles the SmartWire-DT data.
- Identify function that filters the data.
- Modify the filter parameters so that data are not filtered out in the CCPU.
- Design variable definitions for storing the data of interest.
- Implement filter in ACPU for storing variables.
- Define the language for programming of the SmartWire-DT devices.
- Design and implement compiler for programming the behaviour of the SmartWire-DT device.
- Design a way to load byte code and variable definitions to the device.
- Implement a Python script for loading the byte code and variable definitions.
- Modify FW of the ACPU so it supports the loading of the LCF code and variable definitions.
- Identify possible applications for LCF with the usage of variables.
- Set up demonstrator with all the changes described above.

4 Device to device communication

Change in the processing of the communication data is one of the main parts of this thesis. Communication between devices does not follow a peer-to-peer method since it needs to uphold the current communication protocol. Hence device to device (**D2D** in short) communication needs to be implemented in a non-disruptive way.

The behaviour of the devices on the bus depends on decisions from the master device. The distributed decision taking allows for a faster reaction from devices and uses less data payload. If devices can communicate with each other, there will be less need for PLC, and the data do not need to travel to the PLC and then back to a different device.

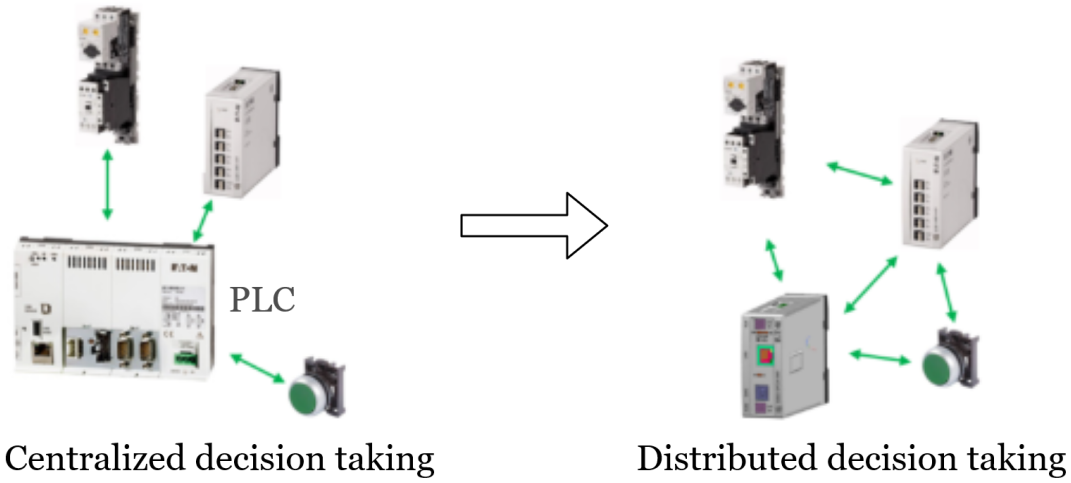


Figure 9: Decentralization of the decision making

We want to eliminate the need for the PLC completely. For example, a system with a push-button, an LED, or any arbitrary actuator (see Figure 10). The state of the push-button triggers the LED or any other actuator because the actuator communicates with the push-button directly, without intermediate PLC.

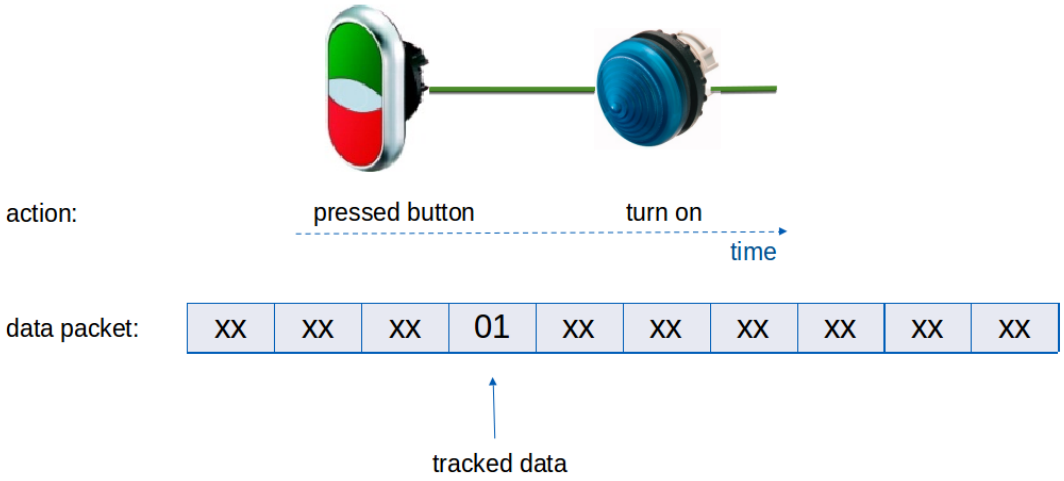


Figure 10: Example of device to device communication

The scenario is even more practical with LCF, as it includes the capability of logic control in each device. For example, signalling light can turn on when two starting push-buttons are triggered and the whole bus system would not need any PLC.

The main principle of D2D communication is to listen to all the data on the bus and storing relevant parts of the cyclic data frame. Enabling communication from device to device consists of a few changes in the SmartWire-DT devices. First is to modify the filtering of the data which enter the device. Second is adapting firmware to the new data flow. And the last one is the propagation of the outputs.

4.1 Removal of the data filtering

We are using the fact that all the communication goes through all of the connected devices already. Every device is aware of its ID number and hence knows which data slots are allocated to it. Firmware of the SmartWire-DT device contains parameters that are defining what data does the device receive. Setting higher value as the length of the collected data, the device receives the rest of the data from the bus and processes it all.

4.2 Data processing

Memory management is important in SmartWire-DT devices since the memory in the device is small and SmartWire-DT bus can connect of up to 100 devices, whose data are sent on the bus. Hence storing only the relevant data is enforced by variable definition. A variable is a storage for data that is defined by its ID and position in the cyclic data frame.

Every SmartWire-DT bus has to be configured first. Variable definitions are created in the process of configuration of the SmartWire-DT bus. More about variable definition in the following Section 5.1. Software SWD Assist [11] shown in Figure 11 manages the configuration of all the devices on the bus. SWD Assist contains a database of all available devices and all modifiable parameters. The application runs on the computer, which is connected to the bus. SWD Assist is used to define variables. The SmartWire-DT device then accesses the position of data to store and saves them to the variables.

From the production point of view, using SWD Assist is the simplest and most customer-friendly approach and therefore the final product requires a modified application. Changes in the application are out of scope of this master thesis and hence only static definition array inside of device FW is implemented.

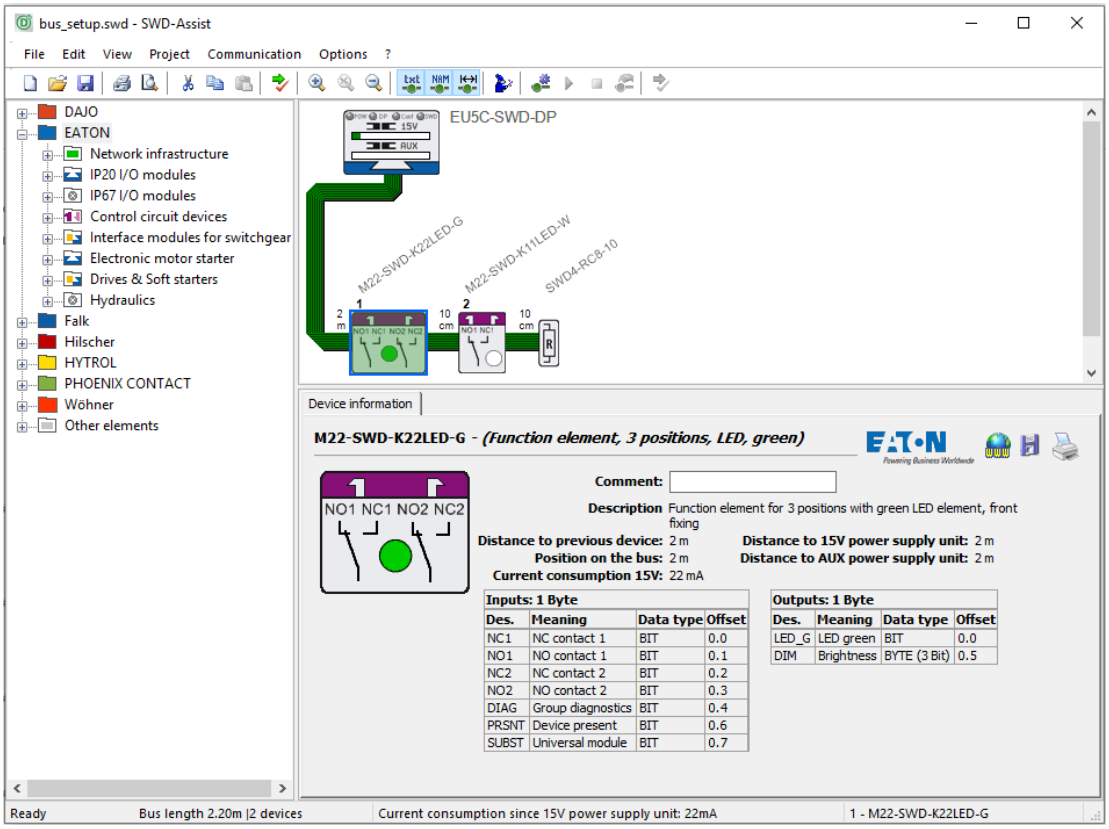


Figure 11: Configuration of the bus in SWD Assist

Positions of all the relevant data are stored in an array inside of the device. When the device receives a new cyclic data frame, the array is used as a filter for the data frame.

Variables are updated every time device receives a cyclic data. A snippet of a callback function is shown in the following code:

```
void swd_slave_cyc_data_rx_isr (uint8_t *data) {
  uint8_t data_byte;
  bool value;
  for(int i = 0; i < num_of_invariables; i++) {
    data_byte = data[invariable_data_offsets[i]+1];
    value = data_byte & invariable_bit_masks[i];
    lcf_io_reader_save_value_in_variable_register(i, value);
  }

  ...
  cyclic_payload_output_received = true;
}
```

4.3 Definition of the variables

Data that are collected by the slave device are stored as variables. Definition of the variables and their position on the SmartWire-DT bus has to be customer-friendly so that it can be configured lately in case of different needs. Variables are configured with a simple JSON file. Identification of the variable is by their position in the JSON file (first, second, ...) and their position on the SmartWire-DT bus can be extracted from the SWD Assist. Example of a **com_config.JSON**:

```
{"variables": {
    "invariables": {
        "amount": 3,
        "assignments": [
          {
            "byte_offset": 4,
            "bit": 1
          },
          {
            "byte_offset": 4,
            "bit": 3
          },
          {
            "byte_offset": 6,
            "bit": 0
          }
```

```
            ]
    },
    "outvariables": {
        "amount": 2
    }
}}
```

Difference between invariable and outvariable is explained later in section 5.1.

4.4 Propagation of outputs

Communication protocols used with PLCs do not support sending outputs in the direction from slave to master but strictly in the opposite direction [12]. Slave device connected with the master receives its output data from the master and send its input data back to the master. Since the goal is to implement the device to device (D2D) communication without the master, the slave device has to propagate its outputs on its own.

If the SmartWire-DT device controls its output data, the data have to be stored at a different data slot. To propagate outputs to the SmartWire-DT bus, they have to be copied to input data.

Increased number of inputs, physical or virtual, calls for a different profile of the device. The specific profile has its ID and CFG (object storing I/O information). Specifically, the CFG contains information about how many inputs and outputs does the device have.

Some devices have reserved bits inside their input data [13]. Reserved bits are not used and hence can be written to. For performance-oriented projects, where the increased payload is not desirable, reserved bits can play a big role. Propagating virtual inputs in reserved bits saves space when it comes to saving data from SmartWire-DT bus. Less data payload also means faster cycle time.

5 Local Control Function

Local Control Function (**LCF** in short) is a software platform allowing distributed control while still connected to the centralized system. Preserving the compatibility allows connecting one device with LCF features to a centralized system. It addresses the need to enclose simple control functions within the device and perform data preprocessing before sending the data. This reduces the load of the bus and the master device. The aim is to increase control efficiency while decreasing material cost. Control in this thesis indicates logic control and specific behaviour implemented in a single device.

LCF consists of three parts: code generator, compiler and interpreter. The code generator is a part of a graphic user interface for machine operators and it uses simple blocks for easy and intuitive behaviour programming. The code can be also written by hand, without the code generator, it just needs to follow defined syntax (structured text [14]) which is also used for PLC programming.

The resulting code is used as input for the compiler, which contains lexer, parser and binary converter. The compiler transforms generated code into byte code which is then written into a device whose behaviour is being programmed.

Last part of the LCF is the Interpreter [4] which runs inside the device. The interpreter is a part of application firmware and it is implemented as a stack-based virtual machine. The interpreter executes the byte code.

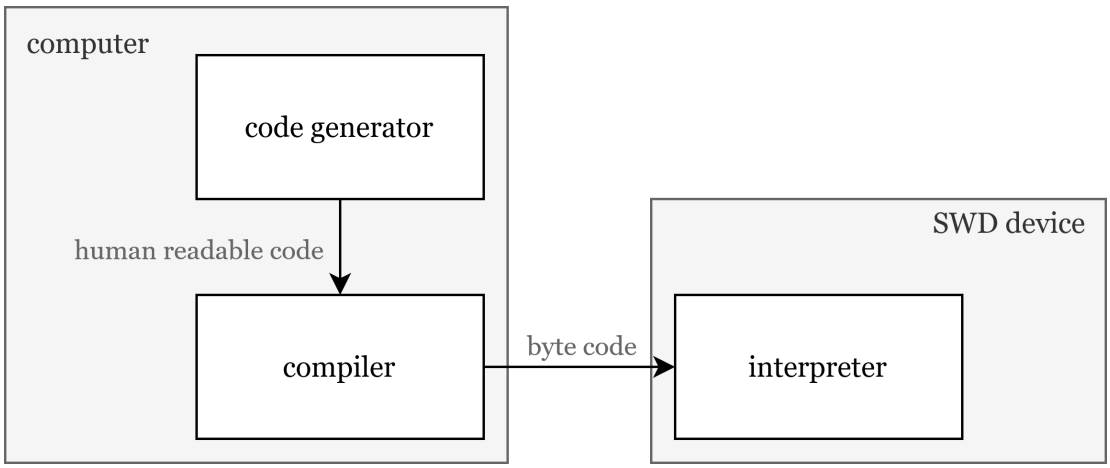


Figure 12: Local Control Function architecture

LCF controls the outputs of the device based on a logic function of input and output data. For example, pressing two buttons results in turning on the LED. Without the LCF, this behaviour would have to be programmed in the PLC connected to the device.

Devices enriched with D2D communication broaden the usability of LCF. Incorporating variables (from D2D communication) enables using all the data on the SmartWire-DT bus and act upon changes in the different slave device.

Use case of a combination of LCF and D2D is seen in Figure 13. Second slave (4D4D I/O module) is collecting the data from the SmartWire-DT bus. Hence the states of the push-buttons are available in the I/O module. If the sole important information consists of both buttons being pressed, it is redundant to propagate states of both push-buttons. Since the I/O module has LCF program inside, it can calculate **push-button 1** AND **push-button 2** and propagate one output instead of two bits. This information can be next processed as preferred to, e.g. store it in the cloud which would again benefit from LCF by smaller data payload.

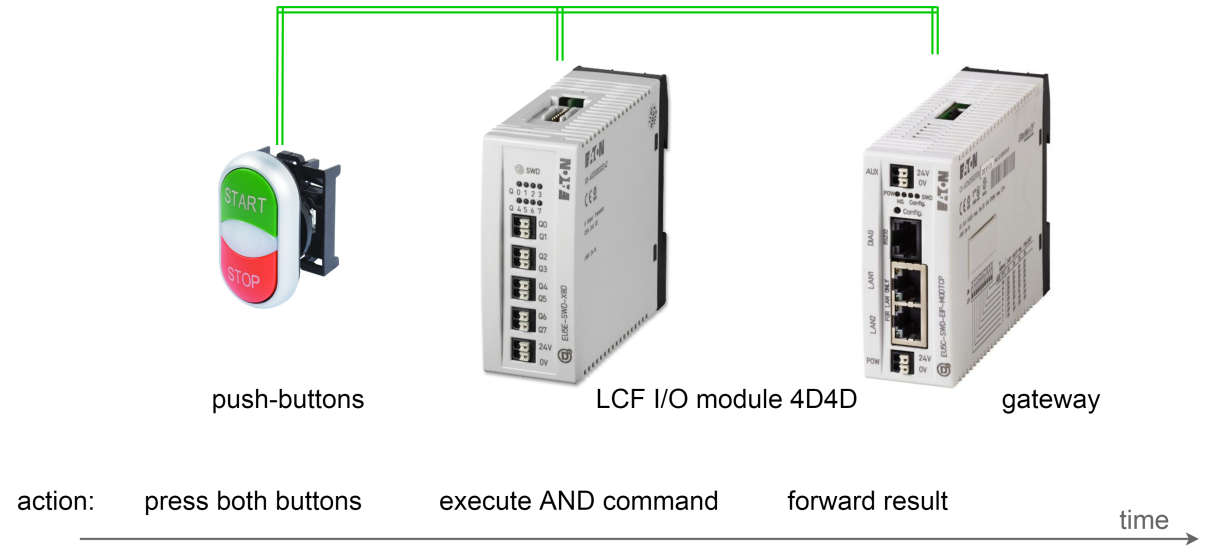


Figure 13: Use case for Local Control Function is following: The user pushes both push-buttons, which propagate its values to the SmartWire-DT bus. I/O module read the values from the bus and executes the AND command. I/O module then forward the result to the bus, from where it is forwarded to the master device and further (to the clouds).

5.1 Inputs, outputs and variables

Most of the SmartWire-DT devices have inputs coming from peripherals of the device and outputs transmitted to the peripherals the device. I/Os connected to the device peripherals are called physical input data and physical output data. I/O data are stored in registers that are accessed by ACPU. Physical input data are automatically communicated via SmartWire-DT bus. Physical output data and variables are saved as a virtual input in order to propagate them from the device into the bus. Virtual inputs are not connected to any periphery of the device but undergo the same processes as the physical inputs. Differences in the mapping of data between physical input data, virtual input data and physical output data are delineated in Figure 14.

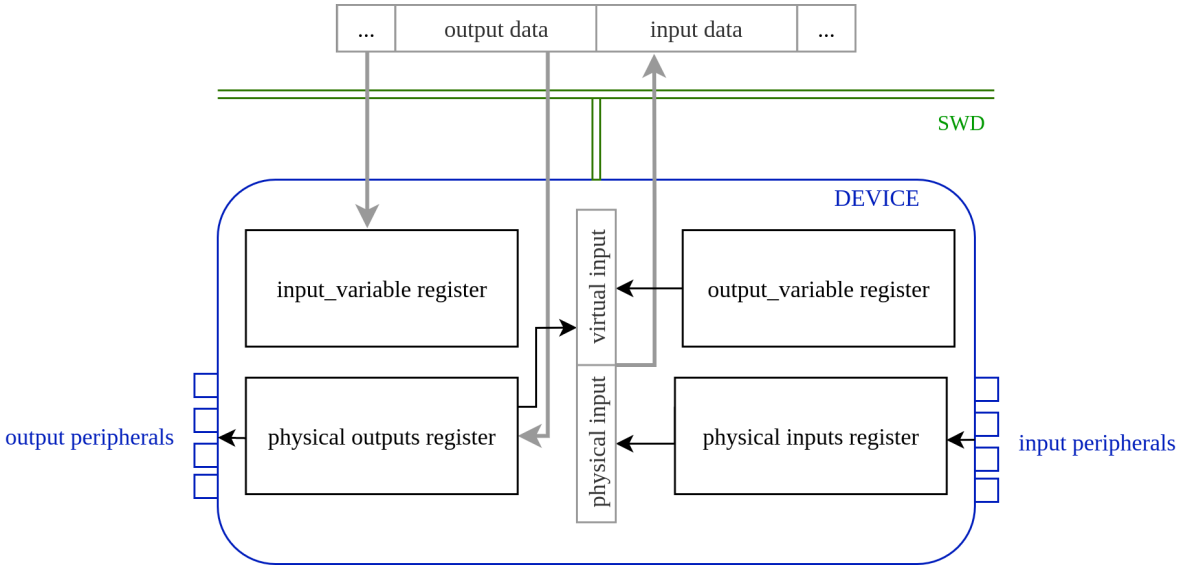


Figure 14: Mapping of the physical and virtual I/O data

Calculated output data and output variables are the additional payload of the device and require to expand the size of the data frames. Device propagates to the SmartWire-DT bus two kinds of data: physical inputs and virtual inputs. Virtual inputs are copied physical outputs and output variables, as we want to send them on the SmartWire-DT bus. All of those are packed in data slot for inputs of the device in the following order:

Physical inputs	Copy of physical outputs	Output variables

Table 2: Structure of data outgoing from the device

Difference between output variables and input variables is their use. LCF program reads the input variables and writes to the output variables. All input data (physical inputs and input variables) are used for calculating the output data (physical outputs or output variables) as shown in an example in Section 5.2.

LCF supports 4 physical inputs, 4 physical outputs and 16 variables. The number of physical I/Os is sufficient because it is the maximum number supported by ASIC devices. Without the variables, the LCF program accesses only physical I/O data and is limited to the device peripherals only. Including variables allows the device to use a peripheral of a different device. Example of the LCF program without variables is I/O module that has sensors connected to input peripherals and actuators connected to the output peripherals. LCF program controls the outputs only from the inputs. Example of the LCF program with variables is a smart motor starter on the SmartWire-DT bus with start and stop buttons. The LCF program can access the status of the push-buttons and control the outputs of the motor starter.

The difference in the data handling between physical input data, input variables, physical output data and output variables data are:

- Physical inputs (stored in physical input registers) are physically brought in the input peripherals of the device and then propagated to the bus.
- Physical outputs are calculated by the LCF from inputs and signals are sent to the output ports of the device. Copy of the outputs is stored as virtual inputs, which are propagated to the bus.
- Output variables are calculated by the LCF from inputs and stored as virtual inputs, which are propagated to the bus.
- Input variables are used for LCF calculations and stored as virtual inputs, which are propagated to the bus.

5.2 LCF Language

We defined the LCF Language with a limited number of symbols. Input code can be written by hand or with GUI for people with no experience in code writing. Code syntax abides by the CoDeSys syntax for PLC programming [15].

Input code is set of statements. Every statement has to assign value to the output or to the variable. Value on the right side can be combination of inputs or variables. Every statement has to end with semicolon symbol. Example of the input code:

```
%QX1 := %IX0 AND %IV1;
%QX3 := NOT (%IX2 OR %IX3);
%VX0 := (%IX0 AND %IX1) OR (%IX1 AND %IX3;);
```

Tokens %QX symbolise physical outputs, %QV output variables, %IX physical inputs and %IV input variables. The whole set of supported symbols is shown in Table 3.

5.3 LCF Compiler

The LCF Compiler designed and implemented in this thesis consists of four procedures as shown in Figure 15. Their function follows Mogensen [8] idea, only type checking and code generation run simultaneously.

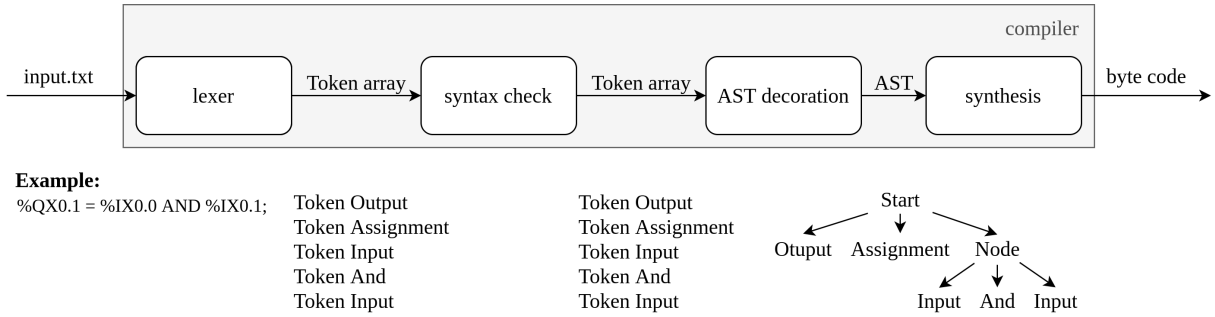


Figure 15: The LCF Compiler and its process flow

Individual phases of the LCF compilation process are described in more detail in the following sections.

5.3.1 LCF Lexer

The LCF Lexer creates an array of tokens from the input text. If there is an unknown token, it stops the execution and throws an error. The LCF Lexer groups up letters one by one until a white space is found. If the group of letters represents a token, a new token is added to the list. Special tokens like input, output and variable have additional value containing the address. Supported tokens are shown in the Table 3. The LCF Lexer reads a text and returns an array with objects Tokens. From this point of the thesis, a token is an object.

LCF Lexer detects undefined tokens and throws **Syntax violation exception**. Process of the LCF Compiler is stopped immediately after detecting an exception.

Token	Role	Value
%IX	input	0..3
%QX	output	0..3
%IV	input variable	0..15
%QV	output variable	0..15
AND	logic control	none
OR	logic control	none
XOR	logic control	none
NOT	logic control	none
SWITCH	control	none
:=	assignment	none
;	termination	none
(and)	parentheses	none

Table 3: Supported tokens in the LCF language

5.3.2 LCF Parser

The LCF Parser executes the rest of the LCF Compiler. The Parser reads the token array, checks the syntax according to defined grammar and creates the intermediate code called byte code in this thesis.

5.3.2.1 Grammar

This chapter describes the grammar that parser needs to parse input text. The LCF Parser uses grammar to checks for a syntax violation. Violation results in **Syntax violation exception** and process of the parser is stopped immediately after detecting an exception.

Grammar for the LCF supports tokens from Table 3 and parentheses. Expressions are evaluated from left to the right. Every line of LCF input file is one statement. The statement has to end with termination (e symbol). The statement is an assignment of an expression to an output. Uppercase names are referencing to a specific token while lowercase names are parts of more than one token. The correct syntax is ensured after the LCF Compiler successfully splits the whole LCF programs to the individual tokens.

The grammar used for syntax check of the LCF code:

```
program -> START stm_list END
stmt_list -> stmt stmt_list | e
stmt -> OUTPUT ASSIGNMENT expr | e
expr -> term term_tail
inv_expr -> NOT expr
term_tail -> and_op term term_tail | e
term -> factor factor_tail
factor_tail -> or_op factor factor_tail | e
factor -> OPENP expr CLOSEDP | INPUT | number
and_op -> AND
or_op -> OR | XOR
```

The LCF Compiler checks the syntax of a token list before the compiler creates syntax trees. Example of a syntax check is described in the next paragraph.

LCF Lexer tokenizes LCF program '%QX1 := %IX2 AND %IX3;' as an array of Tokens [START, OUTPUT, ASSIGNMENT, INPUT, AND, INPUT, TERMINATION, END]. The LCF Compiler starts the syntax check and reads the first token. The first token is START so the array is identified as a **program** and the rest of the array is forwarded to the function that splits the **stm_list** because of LCF grammar. Function **stm_list** forwards the array to a function **stmt**. Function **stmt** strips the statement of OUTPUT token and ASSIGNMENT token and forwards the rest of the array to a function **expr**. Then it is forwarded to a **term**, from there to **factor**, which strips the INPUT token. The rest is passed to **term_tail**, which passes the array to the **and_op**. Function **and_op** strips the AND token and the rest of the array is returned to the **term_tail**, another INPUT token is stripped and termination of a statement is stripped when the rest of the array returns to the **stmt**. The last token is END, which finished the function **program** and the correct syntax is assured. Whenever unexpected (not allowed by the LCF grammar) is found, the compiler raises the **Syntax violation exception**.

5.3.2.2 Syntax tree

The LCF Parser creates a syntax tree to translate the LCF program to byte code. Every token has one of three roles as shown in Table 3: I/Os, command or miscellaneous. Every command has defined a number of values it expects. Command NOT expects one value while the rest require two.

Every statement creates its own LCF AST with the statement as root and an output as the left child of the root. List of LCF ASTa is evaluated sequentially. Every node represents one command with I/Os paired to it. Middle child is a command and the other two are either specific input or output or another node.

From the definition of a statement, the first child on the left is the output and the middle child is an assignment command.

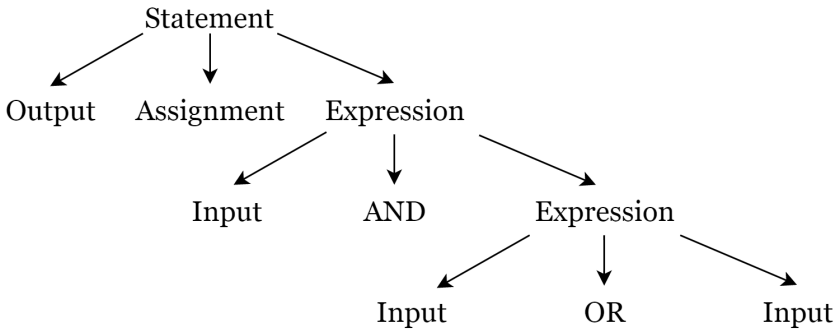


Figure 16: Example of a LCF syntax tree

Parsing can be achieved from two directions [16]. The first one is Top-Down while the later is Bottom-Up. The LCF Parser evaluates the LCF AST from bottom to the top. The lowest layer is the one with no more nodes as children. Commands in the lowest layer have to be evaluated before the layers above can be evaluated. Evaluated node returns a value and passes it up the tree so the rest of the LCF AST can be evaluated. The last command is then the assignment of the result value to the output variable.

The LCF Parser synthesizes and translates the LCF AST to the byte code simultaneously.The output of the LCF Compiler is intermediate code, which already by nature follows command order of a stack machine. Every token has a property called **code** that serves as an opcode. The LCF Parser obtains opcode for every token and creates an intermediate code. Some tokens (I/O tokens) have also address code as explained in the next chapter.

5.3.2.3 Intermediate code

This section describes how commands and I/O values are translated from the input file with the LCF program to intermediate code and how instructions are defined.

Intermediate code is an array of opcodes. Every opcode represents one instruction. Instruction is a function implemented inside the interpreter (in the device FW).

Every instruction has defined structure. Only variables (I/Os) and related instructions need to specify the address and address follows after the opcode. If the instruction has no address, only one opcode is expected.

Opcodes are used to pair Token from the compiler with instruction in the interpreter. All values are binary so we can execute the LCF program with instructions from the table. For example, token AND can be calculated as a MAX instruction and thus token AND and instruction max have the same opcode 0x04. More detail in Table 5.

Type	Variable	Adress
Input	IX0...3	0x00...0x03
Output	QX0...3	0x00...0x03
Input variable	IV0...15	0x00...0x15
Output variable	QV0...15	0x00...0x15

Table 4: Addresses of variables in hexadecimal code

Opcode follows the structure of stack because interpreter in the device is state-machine based on a stack. For example, reading input variables is executed as an instruction push followed by the address and writing output means instruction pop also followed by the address. The opcode for NOT is always preceded with pushing immediate value 1 in order to calculate NOT by executing instruction subtract.

Token	Instruction
IV (input variable)	PUSH
IX (physical input)	PUSH_P
QX (physical output)	POP_P
QV (output variable)	POP
AND	MAX
OR	MIN
NOT	SUB
XOR	COMPARE_NEQ
SWITCH	SWITCH

Table 5: Instructions representing tokens

Complete list of instructions used in LCF and their byte codes are shown in Table 6 with a short description.

Instruction	Code	Address	Description
PUSH	0x00	Table 4	Read the value from the virtual input and store it on the top of the stack.
PUSH_P	0x01	Table 4	Read the value from the physical input and store it on the top.
POP_P	0x02	Table 4	Write the value from the top of the stack to a physical output and remove it from the stack.
POP	0x03	Table 4	Write the value from the top of the stack to a virtual output and remove it from the stack.
MAX	0x04	none	Find maximal value from the two values on the top of the stack and put there the result.
MIN	0x05	none	Find minimal value from the two values on the top of the stack and put there the result.
SUB	0x06	none	Subtract the value on the second position from the top of the stack and put there the result.
COMPARE_NEQ	0x07	none	Compare whether are the two values on the top of the stack not equal and put there the result.
SWITCH	0x08	none	Check the last value of the switch and in case of the rising edge on the selected input, negate the last value. Push the resulting value on the top.

Table 6: Instruction set with short description

5.4 LCF Interpreter

The LCF Interpreter is a state machine running inside the device firmware. The LCF Interpreter accesses the LCF code in memory and follows the byte code command after the command. States of the interpreter are shown in Figure 17. The LCF Interpreter's architecture and processes are explained in detail in [4].

The LCF Interpreter was implemented as part of the LCF but as a different thesis.

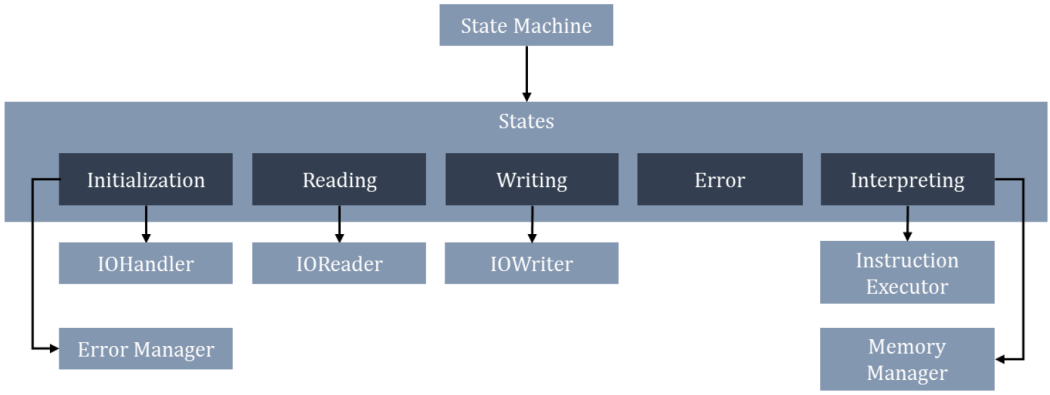


Figure 17: Component diagram of the LCF Interpreter and its dependencies [4]

6 LCF loader

This chapter motivates the need and describes the implementation of the loader between the computer and the slave device for applying changes in the device firmware (FW).

6.1 Aim

The need for a loader originates in the demand for a modifiable behaviour for the slave device. The behaviour of the device is determined by the LCF code, which is easy to change (as described in section 5.2). LCF code is stored in a device FW and needs to be modified inside of the FW.

The tunnel also presents an opportunity to change the definition of the variables. Variables definition carries the definition of the variables and their position on the SmartWire-DT bus. Customer can reevaluate desired variables and change the definition via the loader. Usual changes in device FW (updates/upgrades) are unavailable for the customer and safe user-friendly approach is needed. The loader should use only already connected cables,

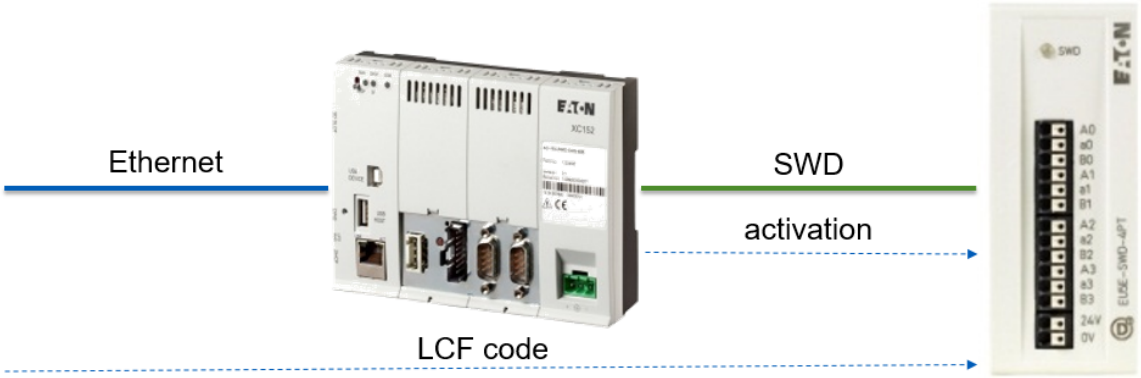


Figure 18: Loader

6.2 Architecture

The loader consists of two parts. The first part is a Python script **loader.py** and a second part is a receiver inside the slave FW. The loader can either load new LCF code, new variables definition or both. New LCF code is stored in a separate file in a format obtained from the LCF compiler (described in section 5.3). New variables definition is stored in a JSON file. It has to be structured like the example in section communication 4.3. The second part of the loader, the receiver inside the FW, is an extension for a callback function.

The receiver reads the data and stores then according to the type of data (LCF code or variables definition). After the data are updated, the LCF interpreter state machine is restarted. The architecture of the loader is shown on a Figure 19.

The loader utilises the acyclic data transmission (described in section 2.1.3) for sending the data do the slave device. Updating LCF code or variables definition via acyclic data transmission changes the behaviour of the device without the need to restart the device.

Unlike previous parts of the thesis, the loader depends on the master device to start the acyclic data transmission. Acyclic data transmission requires a communication server that the SAC does not perform. In order to send the new data (LCF code or the definition of the variables) to the device, a master device needs to be temporarily connected to the SmartWire-DT bus. SmartWire-DT system with LCF devices uses the Standalone Coordinator (described in section 2.1.5) to synchronize the communication. The master device can be plugged instead of the SAC for the loading process. The computer has to be connected by Ethernet cable to the master device that is on the same SmartWire-DT bus with the slave device. Both devices (SAC and temporary master) have to be connected at the beginning of the SmartWire-DT bus, hence they cannot be connected simultaneously.

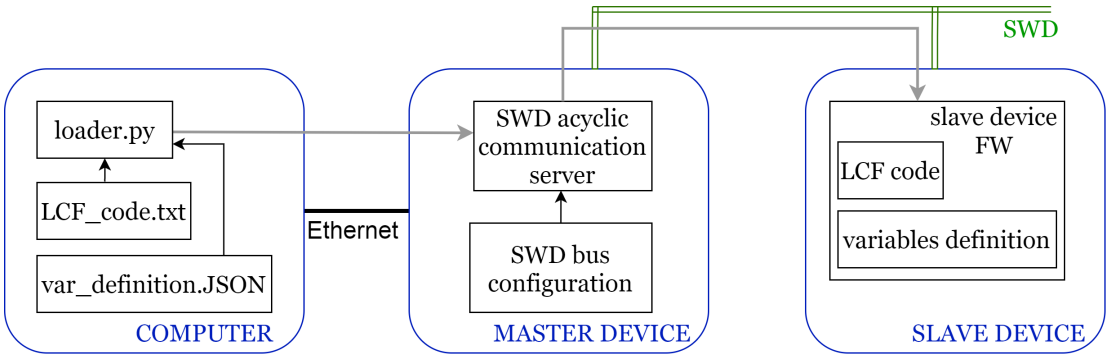


Figure 19: Loader architecture

The script **loader.py** takes following arguments:

- -host ... address of a master device,
- -address ... address of the slave device on the SmartWire-DT bus (1-99),
- -lcf_file ... name of the text file with the LCF code,
- -com_file ... name of the JSON file with the variable definitions and
- -action ... 1 for loading new LCF code, 2 for loading new variable definitions and 3 for both.

Execution of the loader performs the following sequence of actions:

1. Script **loader.py** on the computer reads the arguments, opens appropriate files and formats needed data. The structure of the data format is shown on Figure 7.
2. The script connects to the master device via Modbus-TCP protocol and writes into memory address registers that are indicated for acyclic data transmission specifications.
3. The master device starts the acyclic communication server and sends the acyclic data packet.
4. Slave device with a matching destination address reads the data packet and starts the acyclic callback function.
5. Callback function reads the ID of the data packet and determines what action is required since acyclic communication already has some functions implemented.
6. Callback function determines whether the data contain LCF code or variable definitions by the **action** argument.
7. Callback function rewrites the LCF code and restarts the interpreter state machine.

6.2.1 Memory address registers

SmartWire-DT acyclic data transmission has defined the structure of data inside the memory address registers [3]. The registers content is described in Figure 20. All values are written in hexadecimal number into registers starting at address 0x8000.

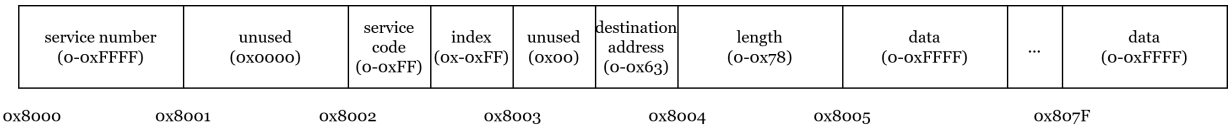


Figure 20: Registers and their content

The service number is an ID of the acyclic request and can be any value. Master copies the service number into response registers and erases it after the request was processed. Service number for LCF loader was chosen 0xDF. Service code describes the action required by the service.

Service code can have following values:

- 0x00 for no action,
- 0x03 for reading the data and
- 0x10 for writing the data.

The index is an arbitrary value defined by a developer. Callback function determines what command does the acyclic data contain based on the index, so it needs to equal to the one inside the device FW. Index for the loader was chosen to be 0x83, since it was available. The destination address is the address of the slave device. Possible values start at 0x01 since the master has the address 0x00. The maximum number of the devices on the SmartWire-DT bus is 99 and so is the upper bound for the destination address. Register with length contains the length of the data starting from the register at 0x8005. The maximum length of transmitted data can be 120. Registers for transmitted data contain the new LCF code.

Filled acyclic data packet looks like this:

0x00DF	0x0000	0x1083	0x00$address$	len	$data[0]$	...	$data[(len/2)-1]$

Table 7: Contents of registers from Figure 20 for LCF Loader

Since loader needs to distinguish between LCF code and variables definition, the beginning of the data is devoted as a header for the specific action (LCF/variables/both). To establish the required action, the first byte in data is **action** byte. The content of the following data varies based on **action** byte. Figure 21 shows register content for loading LCF code, Figure 22 shows register content for changing variable definitions and Figure **??** shows loading both at the same time.

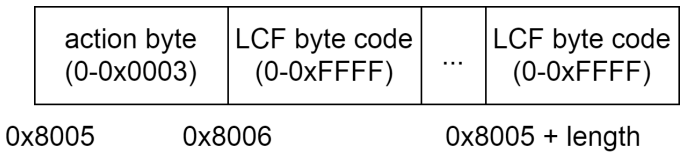


Figure 21: Structure of the data for loading LCF code

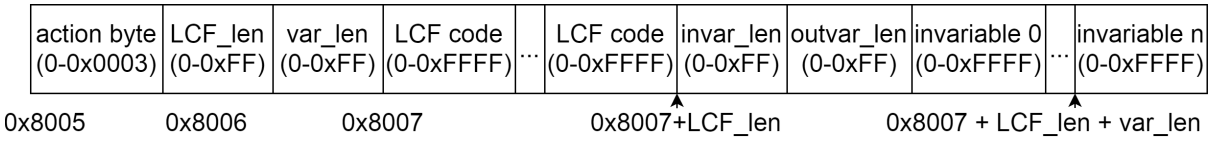


Figure 23: Structure of the data for loading both LCF code and variable definitions

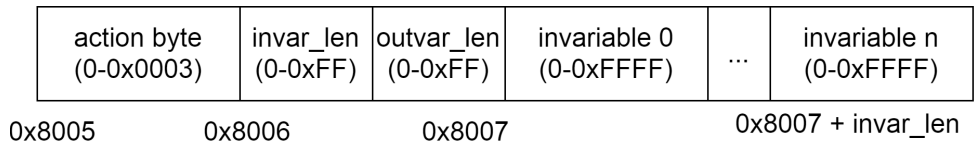


Figure 22: Structure of the data for loading variable definitions

6.2.2 Loader inside the device

The receiver inside the FW has to store data from the correct addresses based on the definitions above. Following code describes the algorithm of the receiver inside the FW.

```
#define SWD_ACYC_REQ_LOADER 0x83

#define SWD_ACYC_LOADER_ACT_LCF 0x01
#define SWD_ACYC_LOADER_ACT_VAR 0x02
#define SWD_ACYC_LOADER_ACT_BOTH 0x03
    ...

 switch(index) {

    ...

  case SWD_ACYC_REQ_LOADER:
    {
      printf("Acyclic data request for loader.\n");
      uint8_t action_byte = data[1];
      switch(action_byte){

      case SWD_ACYC_LOADER_ACT_LCF:
        {
          lcf_state_rewrite_lcf( (uint8_t*)data + 2, length - 2);
          refresh_LCF = true;
          break;
        }

      case SWD_ACYC_LOADER_ACT_VAR:
        {
          num_of_active_invariables = data[2];
          num_of_active_outvariables = data[3];

          uint8_t i = 0;
          uint8_t data_index = 4;
```

```
            for (i; i < num_of_active_invariables; i++)
            {
              uint8_t data_byte = data[data_index];
              invariable_data_offsets[i] = data_byte>>1;
              invariable_bit_masks[i] = data_byte & 0x0F;
              data_index = data_index + 1;
            }

            break;
          }

        case SWD_ACYC_LOADER_ACT_BOTH:
          {
            uint8_t lcf_len = data[2];
            uint8_t com_len = data[3];
            lcf_state_rewrite_lcf( (uint8_t*)data + 4, lcf_len);
            refresh_LCF = true;

            uint8_t data_index = lcf_len + 4 + 1;
            num_of_active_invariables = data[data_index];
            data_index += 1;
            num_of_active_outvariables = data[data_index];
            data_index += 1;

            uint8_t i = 0;

            for (i; i < num_of_active_invariables; i++)
            {
              uint8_t data_byte = data[data_index];
              invariable_data_offsets[i] = data_byte>>1;
              invariable_bit_masks[i] = data_byte & 0x0F;
              data_index = data_index + 1;
            }
            break;
          }
        }
      }

    ...
  }
```

7 Evaluation

This chapter describes three possible use cases for the product of the thesis.All of the use cases are implemented ad master-less systems that offer cost cut and modular solutions.

7.1 Signalling use case

Signalling devices like LEDs, stack lights and others can work without master by implementing their logic via LCF. LCF can either forward triggers straight from the SmartWire-DT bus to the output or use them for logic gates. For example, Figure 24 shows a system with three SWD devices: push-buttons, a robot and a stack lights. A push-buttons control the robot and simultaneously signalize the status of the system. Stack lights are collecting data from the SmartWire-DT bus as variables and then assigns them to the individual LEDs. The first variable stores the value of push-button "ON", second variable stores the value of the push-button "OFF" and third variable stores the diagnostic status of the robot. Variables are then assigned to the "RUN", "STOP" and "ERROR" LEDs respectively.

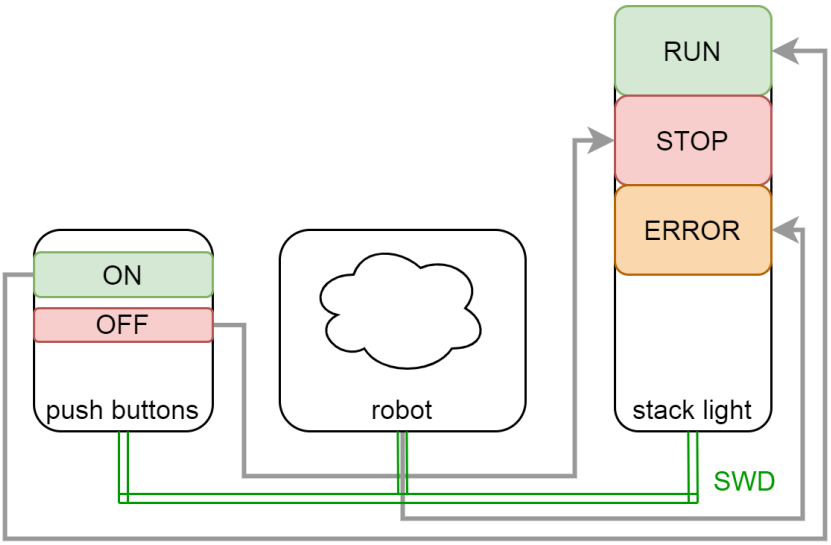


Figure 24: Use case for controlling signal devices with push-buttons

7.2 Error detection use case

LCF can be used for detecting the faults in the bus. As discussed in section 2.1.4.1, SmartWire-DT offers a variety of diagnostic information stored in flag bit **DIAG** . Another useful status information is flag bit **PRSNT** that holds the information whether is the device connected or not.

Both of these bits can be used for detecting whether the result of LCF code is valid or not. Some applications in the industry require two separate push-buttons for greater safety. Figure 25 shows a diagram of the LCF program that stores the input data of the push-buttons but also stores the diagnostic bits and check them with AND logic gate. When one device is compromised or missing, the result will be invalid.

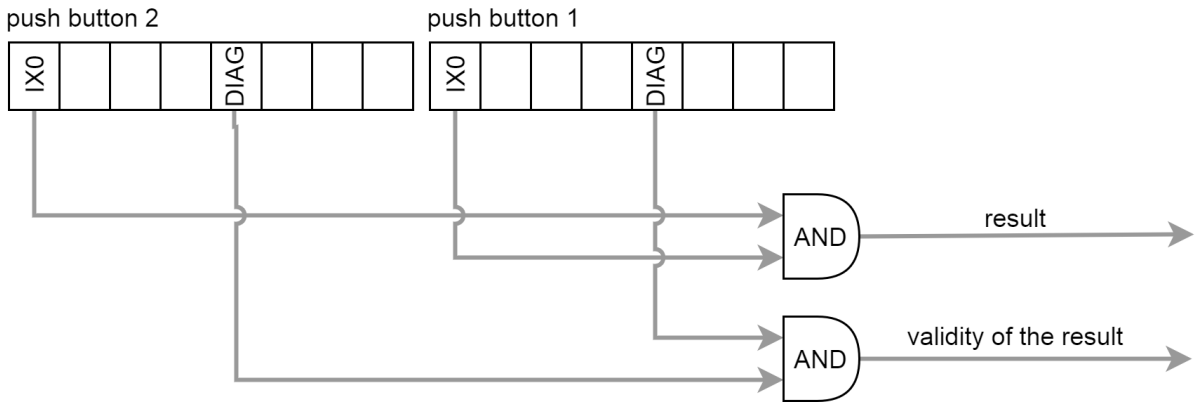


Figure 25: Use of a diagnostic bit in the LCF

7.3 Demo use case

To show the functionality of the thesis results, we designed and assembled a demonstration use case.

Use-case system consists of the following components:

- Standalone Coordinator,
- push-buttons,
- 4D4D I/O module with 4 inputs and 4 outputs,
- DIL contactor module,
- LED 1 (running),
- LED 2 (DIL status) and
- a motor.

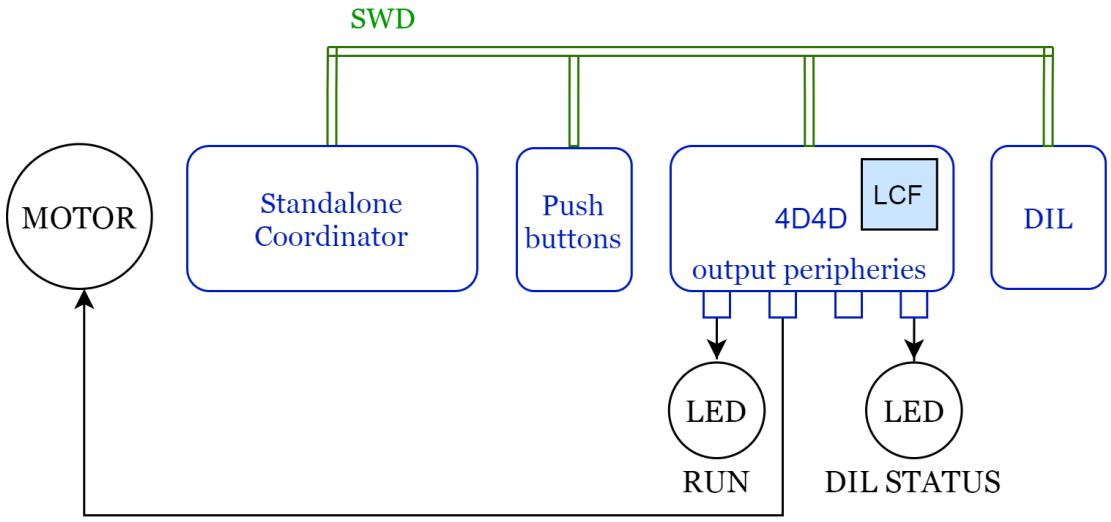


Figure 26: Architecture of assembled use case

Figure 26 shows the architecture of the use case. SmartWire-DT devices are blue while devices connected to the outputs are black. Only I/O module has LCF inside. LEDs are connected to the output of the I/O module because they do not have the ASIC2 (discussed in section 2.1.4) and hence do not support the LCF.

Standalone Coordinator synchronizes the communication, which allows devices in the system to communicate without a master device.

Push-button device has two push-buttons that fulfil the role of Start and Stop. Push-buttons behaves like a switch for two reasons. The first reason is increased safety and the second one is to confirm the functionality of an FW switch where it is not possible to add

HW switch. Status of the push-buttons represent the state **RUNNING** (true) into the SmartWire-DT bus.

The DIL contactor reads its inputs and propagates to the SmartWire-DT bus its state value **DIL STATUS**.

I/O module collects from the SmartWire-DT bus information to determine status **RUNNING** and **DIL STATUS**. The first two outputs are assigned the same value, which is a switch from push-buttons representing state **RUNNING**. When the state is true, the LED 1 is on and the motor is started. When the state is true, the LED 1 is turned off and the motor stops. The LCF assigns to the last output the state **DIL STATUS** from the SmartWire-DT bus. The LED 2 is on when the state is true and vice versa.

The motor is not an SmartWire-DT device and has to be connected to the input peripheries of the I/O module.

Cyclic data frame is described in Figure 27 and was verified by SmartWire-DT Monitor [17]. Output values are all zero because the behaviour of the SmartWire-DT devices is not defined by the master device. In order to propagate outputs back to the SmartWire-DT bus (as discussed in section 4.4) copies of the outputs are stored as virtual inputs (highlighted blue).

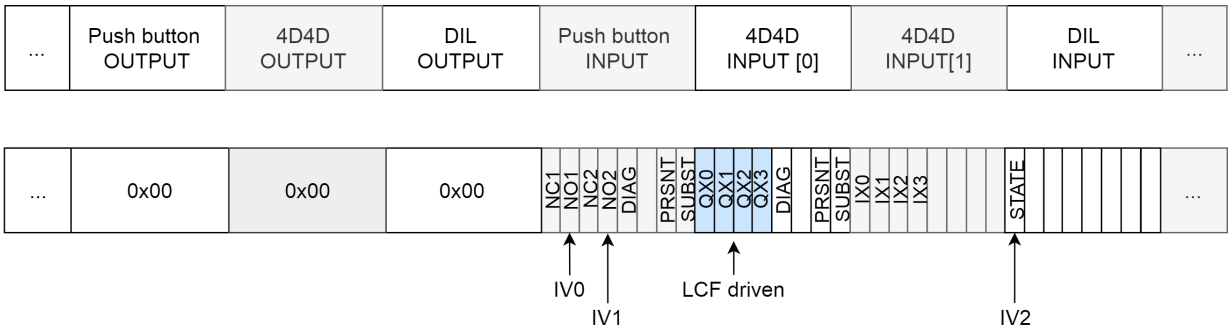


Figure 27: Cyclic data frame of the use case

I/O module stores following data as variables:

1. **IV0** is status of the push-button ON (NO2 from Figure 27),
2. **IV1** is status of the push-button OFF (NO1 from Figure 27) and
3. **IV2** is state of the DIL contactor.

LCF code in I/O module defines outputs as:

```
%QX0 := (%IV0 SWITCH %IV1);
%QX1 := (%IV0 SWITCH %IV1);
%QX3 := %IV2;
```

LCF bytecode from the LCF code is following:

```
0x12 0x06 0x00 0x01 0x00 0x00 0x08 0x02 0x00
0x00 0x01 0x00 0x00 0x08 0x02 0x01
0x00 0x02 0x02 0x03
```

The assembled demo use case is shown on the Figure 28. SmartWire-DT devices are located on the bottom rail, the motor is connected through the back and is positioned on the left side of the board. LEDs are above the SmartWire-DT devices. Extra push-buttons are there because the board was reused for this purpose and are not connected.

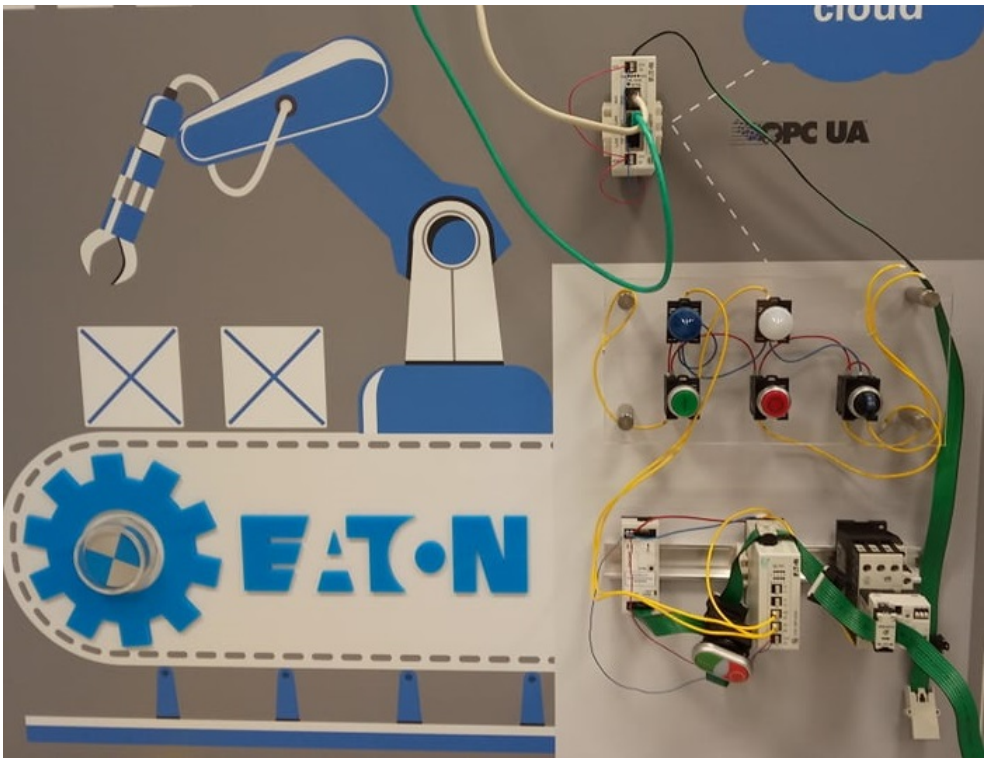


Figure 28: Photo of the use case setup

7.4 Assessment of results

Demo use case from the previous section 7.3 demonstrates that device to device communication works since LCF I/O module collects the data from the SmartWire-DT bus and uses them to calculate its outputs. The functionality of the Local Control Function is proved by LED devices that are connected to the LCF 4D4D I/O module and hence are controlled by the LCF. Everything in this paragraph is demonstrated without the master device.

The LCF Loader is demonstrated on the video on the attached CD. Change in the variable definition is demonstrated by swapping the push-buttons. At first, the green push-button started the motor and the red push-button stopped it. After loading new variable definitions, the red push-button starts the motor and the green push-button stops the motor. Loading of different LCF byte code is demonstrated on the second output of the LCF I/O module (and LED connected to it). At first the LED is **ON** when the DIL contactor status is true and after loading new LCF byte code it is the opposite (the LED is **ON** when the DIL contactor status is false) because of added control gate NOT. The LCF Loader is demonstrated with MODTCP Gateway [3] as a master device.

Compatibility with devices with unmodified firmware is proved by the push-buttons and DIL contactor connected to the SmartWire-DT bus. None of those has ASIC2, let alone modified ACPU.

Variables in demo use case are values of push-button **ON** and push-button **OFF**. If we program the use case with PLC, variables would be propagated to the SmartWire-DT bus in one cycle, then the data would be processed inside of the PLC and it would take a second cycle for the data to reach the I/O module. In contrast, the LCF I/O module reads the variables in the same cycle, in which the push-buttons propagated its values into the SmartWire-DT bus. Hence we reduced the data transfer delay from two cycles into one cycle.

Change in data transfer delay is depending on the application. A vital factor is a number of additional variables. If we add too many variables (how much is too many depends on the specific devices and applications), the delay can be even bigger than with master device.

The results developed in this thesis can be split into two parts. The first part, which is written in Python, consists of new code and accounts for a few thousands of lines. The second part is the firmware of the device written in C. This code was accessed, modified, and adapted, which required a deep understanding of the existing large codebase with over one hundred thousand lines.

7.5 Future work

The course of the following work could be the following points:

- Implement new functions for the LCF (e.g. timers, if branching, ...).
- The LCF interpreter handles inputs and outputs in a device-specific manner. Implement interface or redesign I/O handling, so it is easier to adapt for new devices.
- Store the LCF code and variables definition in the FLASH memory, so that it stays updated after restarting the device.
- Optimize memory usage of the LCF interpreter.
- Identify memory and speed limitations and measure the influence of the LCF on communication speed.

8 Conclusion

The focus of this thesis was to decentralize control in industrial devices. The main emphasis was cutting the cost of the system by getting rid of PLCs and similar expensive devices. Result of this thesis proves that SmartWire-DT devices can function without the PLC by being programmed into self-sufficient agents without any external control signals.

The filtering procedure of incoming SmartWire-DT data was modified to allow the firmware of the SmartWire-DT device to have access to all data from the SmartWire-DT bus. Every device is aware of its position on the bus and filtered out data that were not destined to it. Filter configuration was modified to include all the data from the bus. Another change in the FW was implemented so that the data were forwarded from communication CPU to the application CPU (ACPU). ACPU then stores only the data of interest due to space limitations. Data are assigned to variables that are defined by their IDs and position in the data frame.

As a result of this thesis, the LCF implements logic control in the sense of logic gates. The language for the LCF was defined. The LCF compiler, that translates the program into byte code, was implemented from scratch. This required to define a set of commands had to be defined and all tokens were assigned an opcode. We designed the LCF so that a customer can reprogram the behaviour of the device by themself. To assure the correct functionality, the parser checks the syntax. The syntax of the LCF code is checked by using the abstract syntax trees and grammar. The LCF interpreter was modified to support more functions and to use the variables collected from the SmartWire-DT bus.

We have implemented the LCF loader for LCF devices. The loader consists of two parts. The first one is a Python script on the computer and the second part is the callback function inside the device. The loader uses acyclic SmartWire-DT communication and for that, it needs a master device. The loader has three modes: load LCF code, load variable definitions or load both. The loader can send up to 120 bytes of data at once, which is sufficient for both the LCF code and the variable definitions for most use cases.

All the implemented components were evaluated and possible use cases for a system with such devices were indicated. Use cases are not limited to industrial use and bring benefits for a lot of situations including households or hobbies.

References

[1] Eaton Corporation, *SWD presentation.* , 2015.

[2] Eaton Corporation, "Manual SmartWire-DT The System." `http://www.eaton.com/ecm/groups/public/@pub/@electrical/documents/content/mn05006002z_en.pdf`, 2015. Accessed: 15.09.2019.

[3] Eaton Corporation, "Manual SmartWire-DT Gateway EU5C-SWD-EIP-MODTCP." `https://www.eaton.com/content/dam/eaton/products/industrialcontrols-drives-automation-sensors/smartwire-dt-intelligent-wiring-system/user-guides/smartwire-dt-gateways-ethernet-mn120003z.pdf`, 2013. Accessed: 20.04.2020.

[4] M. S. A., "Local control function interpreter," BS.c. Thesis, Prague College, Prague, CZ, Sept. 2019.

[5] Eaton Corporation, *Manual SmartWire-DT Protocol Structure.* , 2010.

[6] MAZ Brandenburg GmbH, *EATON SWD ASIC2 Specification.* , 2017.

[7] Eaton Corporation, *Standalone manual.* , 2020.

[8] T. Mogensen, *Introduction to Compiler Design.* Cham: Springer International Publishing AG, 2nd 2017 ed., 2017.

[9] N. Chomsky and G. A. Miller, "Finite state languages," *Information and Control*, vol. 1, no. 2, pp. 91 – 112, 1958.

[10] B. Melichar, J. Janoušek, L. Vagner, and České vysoké učení technické v Praze. Fakulta informačních technologií, *Parsing and translation.* Praha: Česká technika - nakladatelství ČVUT, 1st ed., 2013.

[11] Eaton Corporation, "SW product: SWD Assist." `http://www.eaton.eu/Europe/Electrical/ApplicationSolutions/SmartWireDT/SWD-Assist/index.htm?wtredirect=www.eaton.eu/swdassist`, 2015. Accessed: 09.02.2020.

[12] UNITRONICS, "What is the definition of PLC?." `https://unitronicsplc.com/what-is-plc-programmable-logic-controller/`.

[13] Eaton Industries GmbH, *Manual SmartWire-DT SWD module IP20.*

[14] PLC Academy; Peter, "Structured text tutorial for PLC." `https://www.plcacademy.com/structured-text-tutorial/`, 2015. Accessed: 10.01.2020.

[15] 3S-Smart Software Solutions GmbH, "CODESYS control v3 manual." `https://www.codesys.com/`, 2019. Accessed: 20.01.2019.

[16] D. Grune and C. J. H. Jacobs, *Parsing Techniques: A Practical Guide*. Springer, 2007.

[17] Eaton Industries GmbH, *Manual SmartWire-DT Bus monitor: Quick Start Guide.* , 2020.

Appendix A CD Content

In Table 8 are listed names of all root directories on CD.

Directory name	Description
thesis	Masters's thesis in pdf format.
thesis_sources	Latex source codes.
compiler	Compiler source codes.
loader	Script for loading LCF code and communication configuration into the slave device.
demo	Demo video showing functionality of the implemented changes.

Table 8: CD Content

Appendix B List of abbreviations

In Table 9 are listed abbreviations used in this thesis.

Abbreviation	Meaning
SmartWire-DT	SmartWire Device Technology
SW	Software
HW	Hardware
FW	Firmware
DCS	Distributed Control System
PLC	Programmable Logic Controller
I/O	Input and Output
CRC	Cyclic Redundancy Check
ASIC	Application-specific integrated circuit
LCF	Local Control Function
SAC	Stanalone Coordinator
D2D	Device to device
CFG	Configuration number
AST	Abstract Syntax Tree
CPU	Central processing unit
ACPU	Application Central processing unit
CCPU	Communication Central processing unit
EU5E-SWD-4D4D	SWD I/O module with 4 digital inputs and 4 digital outputs
EU5C-SWD-EIP-MODTCP	Gateway for SmartWire-DT

Table 9: Lists of abbreviations