Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

University of St.Gallen

Transparent Field Device Management for Complex Cyber-Physical Industrial Automation Systems

Semester Thesis

Jonas Brütsch, ETH Zürich

brujonas@ethz.ch

Interaction- and Communication-based Systems Research Group
Institute of Computer Science
University of St.Gallen

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Iori Mizutani and Prof. Dr. Simon Mayer, University of St.Gallen
Prof. Dr. Roger Wattenhofer, ETH Zürich

January 3, 2021

Acknowledgments

At this point, I would like to thank my supervisor, Iori Mizutani, for supporting and guiding me throughout the thesis. Not only was he able to support me with his knowledge in the field of industrial automation, but he also taught me programming-specific and general working practices. I am very grateful for his support especially in these difficult times.

I also want to thank Professor Roger Wattenhofer and Professor Simon Mayer for giving me the opportunity to work on this exciting topic. Thanks to their friendly and uncomplicated way of communicating, it was pleasant to do this work in such an unusual setting.

For making such a collaboration possible, I would also like to thank the student administration of the D-ITET, especially Doris Doebeli, who always very kindly supports and advises students in exactly such situations.

Abstract

As industrial automation systems become more and more complex with the adaption of Industry 4.0, new requirements arise for the increasingly interconnected systems. A central requirement for such systems is the accessibility of device information. However, due to abstraction and the use of heterogeneous field devices, this device information is often hidden in the system and not accessible from an operational point of view.

To address this problem, this paper presents a mechanism that enables transparent field device management for heterogeneous devices. Furthermore, the implementation of a simple proof-of-concept automation system is discussed, which applies the developed mechanism and enables further research.

Acronyms

FSM Finite-state machine. 11

IAS Industrial Automation Systems. 1, 2, 4, 6, 19

IIoT Industrial Internet of Things. 1

IoT Internet of Things. 1

MCU Microcontroller unit. 8, 11, 14, 15

PLC Programmable logic controller. 6

PoC Proof of concept. 2, 4, 13, 16

UUID Universally Unique Identifier. 5

Contents

Chapter 1

Introduction

1.1 Motivation

The modern world is slowly turning into a networked world where not only people but also physical objects are connected. Especially with the rise of the Internet of Things (IoT), new technologies and their potential applications have emerged. This trend is also widely seen in the industrial sector, where the Fourth Industrial Revolution, also called Industry 4.0, arises. One of the key enabler for this digital transformation is the Industrial Internet of Things (IIoT), which applies the principles of the IoT to industrial applications [1].

In industry, the automation of processes has been a core topic for decades, therefore the design and operation of Industrial Automation Systems (IAS) are well researched and understood. Still, one of the main challenges of IAS is the heterogeneity of field devices (e.g., data formats, protocols, and communication interfaces), which makes automation systems more and more complex. To increase uniformity and consistency of automation system management, many modern factories are built based on a hierarchical model as often referred to as "Automation Pyramid" (standardized as IEC 62264 [2], see Fig. 1.1).

When these traditional IAS are enhanced to a more interconnected system by applying the IIoT principles, transparency becomes a central requirement for the system. The problem of heterogeneity and the modularisation of the classical model, however, make it difficult to provide necessary explanations about the automation system at higher levels. If, for example, a manufacturing process stops working due to the failure of a single field device, it is difficult for the factory operator to locate the fault as most manufacturing steps can only be accessed in an abstracted form at the supervision level. In addition, the adaption of Industry 4.0 often requires a redesign of the manufacturing front and replacements of outdated appliances, which leads to a more complex system architecture that further reduces transparency. Many state-of-the-art interconnecting technologies typically assume an IP-ready middlebox for edge peripheral devices to access sensor data and drive actuators. Unfortunately, edge devices are not always equipped with a dedicated middlebox, and the use of a generic middlebox would

make the implementation details between peripherals and middleboxes hidden from the application services.

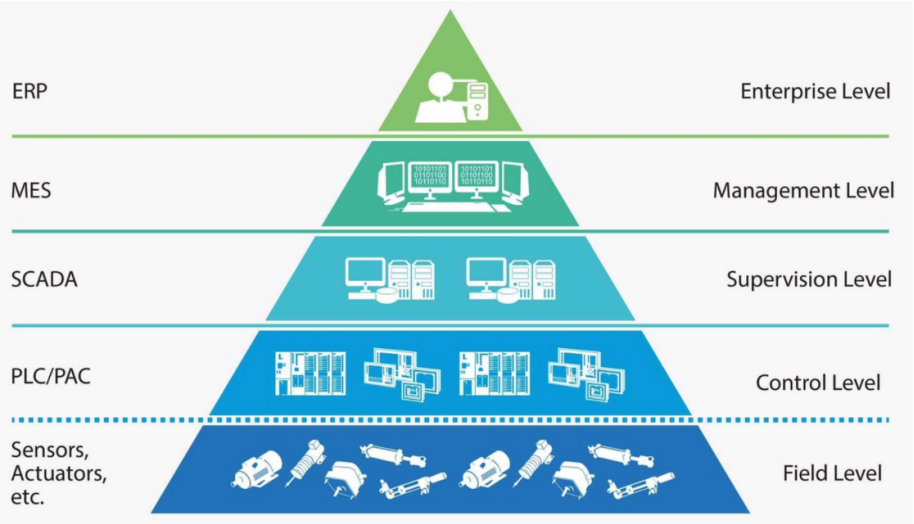

Figure 1.1: Automation Pyramid[1]

1.2 Objective

The objective of this project is to create a Proof of concept (PoC) manufacturing automation system which can be used as a base-system for future research in designing novel IAS architectures. To investigate on the problem of transparency, the focus of this thesis is on designing a mechanism to represent device accessibility information in an open data format. Specifically, a registration and interaction pattern should be designed to enable the configuration and operation of field devices. The proposed mechanism should, furthermore, absorb the heterogeneity problem of physical device integration (Section 1.1) by supporting different types of interfaces and protocols.

[1] M.I.A.C. Automation: http://www.miac-automation.com/mes-oee-track-and-trace

1.3 Outline

To facilitate reading, the structure of the report is briefly explained in the following paragraph.

The main body of this report is divided into two chapters. The first, Chapter 2, introduces the abstract principles and mechanisms created to enable a transparent and flexible field device management which supports heterogeneous devices. The second, Chapter 3, describes all implementation-specific details of the PoC manufacturing automation system. Both chapters are subdivided into three sections, each representing one layer of the entire system.
The last chapter concludes the insights gained and briefly summarises the contribution made.

Chapter 2

Field Device Management and Control

Similar to traditional IAS introduced in Section 1.1, the PoC manufacturing system is based on a 3-layer architecture. Fig. 2.1 illustrates how these layers, the Gateway, the Controllers and several field devices are combined to form a manufacturing automation system. In the context of this report, a manufacturing automation system refers to a simple implementation of a general IAS. The central task of an IAS is to execute specific Actions on field devices in order to carry out a manufacturing process. In the following sections, control messages and a global feedback mechanism are presented, which enable a transparent control of simple and also more complex field devices.

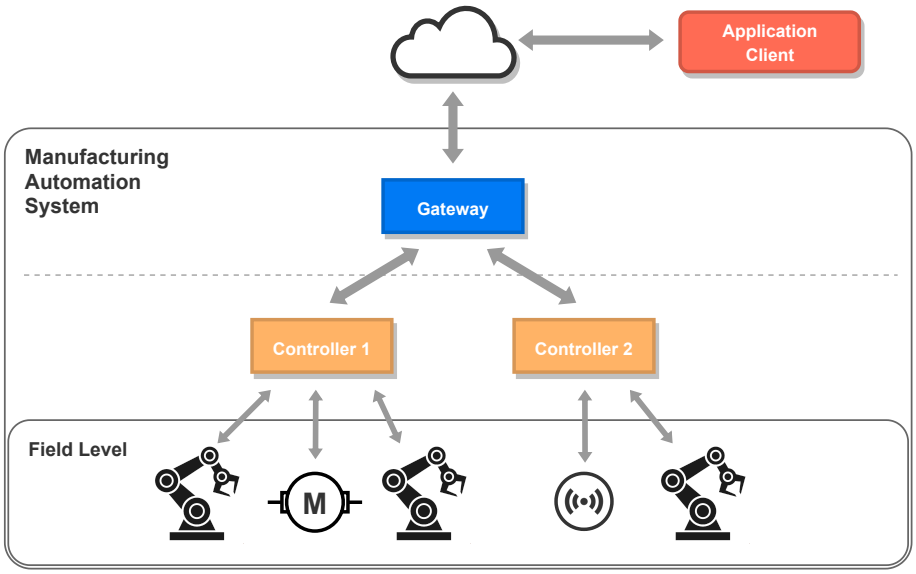

Figure 2.1: Overview of the automation system

2.1 Field Devices

A field device is a single technical appliance which can influence the surrounding environment and/or has some sort of sensing capabilities. At the field level, several field devices work together to perform manufacturing processes and provide the system with relevant information for the operation of the process, such as sensor measurements. The functional complexity of a field device can be as simple as a light sensor or more sophisticated like a robotic arm.
We hereby introduce a term *Action* to refer to generic operations on field devices such as controlling an actuator and reading out sensors. An Action of a field device can be, for example, a single reading of a certain sensor value or a specific movement of a robotic arm. To be able to manage and control all the different types of devices, a uniform device description must be established, which is further discussed in the following section.

2.1.1 Device Profile

A *Device Profile* represents a virtual instance of a physical device. Such a profile contains necessary meta information for the data access and interactions with the corresponding device. The meta information stored in the Device Profile includes:

- Field Device Identifier
- Profile State (see Section 2.3.2)
- Driver Identifier
- Driver Specific Configuration

Two different types of identifiers are used for the device identification. To identify the device on a local scope, a shorter ID, later referred to as Profile ID, is used. For a global scope, the longer Universally Unique Identifier (UUID) was employed to globally identify a unique Device Profile [3]. In this context, the *local* scope refers to the application within a Controller and its directly attached Field Devices. On the Gateway, the Profile ID is only used for Device Profile management and will not be accessible for external clients. The *global* scope on the other hand refers to the external application. The association between these two identifiers is done on the Gateway so that the lower layers only process the more compact Profile ID. This provides an efficient handling of profiles as the required network bandwidth and memory space decreases.

To manage all devices of the system, the state of each device in use has to be carefully monitored. For this reason, each Device Profile is assigned a *Profile State* to indicate whether the corresponding device is currently available or not.

All possible Profile States of a Device Profile are described in Section 2.3.2.
Due to the heterogeneity of the field devices, Device Profiles must support a variety of metadata formats. This is achieved by using the driver-based device handling scheme described in Section 2.2.2.

2.2 Field Device Controller

The purpose of the Controller, similar to a Programmable logic controller (PLC) in a traditional IAS, is to make all field devices accessible to the Gateway and to perform simple control operations. More precisely, the Controller has to translate incoming requests from the Gateway and execute the Actions on the corresponding devices. Additionally, each Action on a device should trigger a feedback to the Gateway to return relevant information and maintain a consistent global state. To ensure transparency, operations on the Controller should be kept as simple as possible and Actions should not be abstracted whenever possible. For example, if the Controller translates a complex manoeuvre of a robotic arm into several atomic Actions, some of the information about the current state of the robotic arm could be lost due to the abstraction on the Controller. Unlike PLCs, the Controller should be able to configure and dynamically initialise new devices without having to update the firmware of the controller hardware. By making this function accessible to external clients through the Gateway, a transparent operation of the field level is possible as the configuration settings of the field devices are available on the client side and not hidden within the Controllers firmware.

2.2.1 Device Management

To keep the structure of the Controller as simple as possible, all Device Profiles are managed and stored on the Gateway. Controllers only cache the driver specific configuration settings in order to reduce redundant data exchange in the network.

2.2.2 Dynamic Device Configuration

One of the key features of the proposed system is the dynamic device configuration. To be able to dynamically install and switch physical devices, a driver-based configuration scheme was applied. Such a driver-based system has the advantage, that once a corresponding driver is installed on the Controller, the firmware of the Controller has not to be modified and all configurations can be done on a higher level. This requires, however, a dedicated driver for the field device in use.

Driver Based Controller Firmware

A driver needs to implement the following two main tasks, the initialisation of a device on the controller hardware and the execution of an Action on this device. When a new request is received from the Gateway to the Controller, the Controller parses the request and forwards it to the associated driver. In order to identify each driver, each driver must have a unique driver name. The functionality of such a driver can be as simple as a single digital I/O pin controller or a more complex, device specific, controller. To preserve transparency over the entire system, no essential information should be lost when using these drivers. Therefore, the level of abstraction of the drivers has to be carefully designed.

Registration Messages

To initialize and configure a new field device on the Controller, the Gateway sends a *Registration Message*. This Registration message contains a unique Profile ID, a unique driver name and all driver-specific configuration settings. On controller-side, the configuration settings get cached using the Profile ID as an index to speed up lookups. After the corresponding driver has initialized the device, the Controller sends an ACK response to acknowledge the successful registration. During the initialisation of a device, the Gateway has to ensure that no other request is sent to the Controller in order to guarantee an undisturbed initialisation.

2.3 Device Management Gateway

The highest component in the system, the Device Management Gateway, acts as an entry-point for external service applications. Besides exposing the automation system to application clients, the Gateway is also responsible for managing and monitoring all Controllers and registered field devices.

2.3.1 Device Operation

The following sections present the basic procedure to perform Actions on a device and further discuss a mechanism to handle complex devices and their applications.

Action Messages

After a device has been successfully registered, the Gateway can send *Action Messages* to execute specific commands or read sensor values on a field device.

Action messages, similar to Registration messages, contain a unique Profile ID, a unique driver name and driver-specific Action fields. To handle events and asynchronous Actions, an additional field is included into the message structure. This will be discussed in the following sections. Once an Action is completed, the Controller sends a DATA message to the Gateway to confirm successful execution and return relevant information.

Blocking vs. Non-Blocking Actions

Depending on the field device, some Actions can be executed almost instantaneously (e.g. digital reading), while others that are processed asynchronously are prolonged by delays (e.g., moving actions of robots). In the following we refer to the instantaneous Actions as *non-blocking* and the asynchronous Actions as *blocking* Actions. The procedure of such blocking and non-blocking Action requests are shown in Fig. 2.2. In a blocking Action all devices connected to the same Controller are blocked until the Action is completed. Note that this problem only occurs when the Controller is implemented on a constrained device such as a single-core Microcontroller unit (MCU). To avoid such blocked phases, an additional field is included in the Action message to indicate asynchronous Actions. After an asynchronous Action has been started, the Controller sends an ACK message to the Gateway to confirm receipt of the Action. The Gateway may now send further requests to other devices connected to the same Controller. Upon completion of the asynchronous Action, a DATA message is sent to the Gateway for confirmation. To detect the completion of the asynchronous Action on the Controller, a simple event handling mechanism is implemented.

Event Handling

In this context, we define an *Event* as an unsolicited or asynchronous reaction of a field device. This can be, for example, the feedback of a pressed button or that of a robotic arm after it has completed a movement. For each registered Device Profile the Controller stores whether an Event of the corresponding device is expected or not. When a Profile expects an Event, the Event handling function of its driver is called to check if an Event has actually occurred. If it has, then the Controller returns a DATA message to the Gateway to report the Event. Any driver that supports Events must have an Event handling function implemented.

Fig. 2.2 shows the flow of a device Registration (grey dots), a simple blocking Action (red dots) and a non-blocking Action with the Event handling mechanism described in this section (green dots).

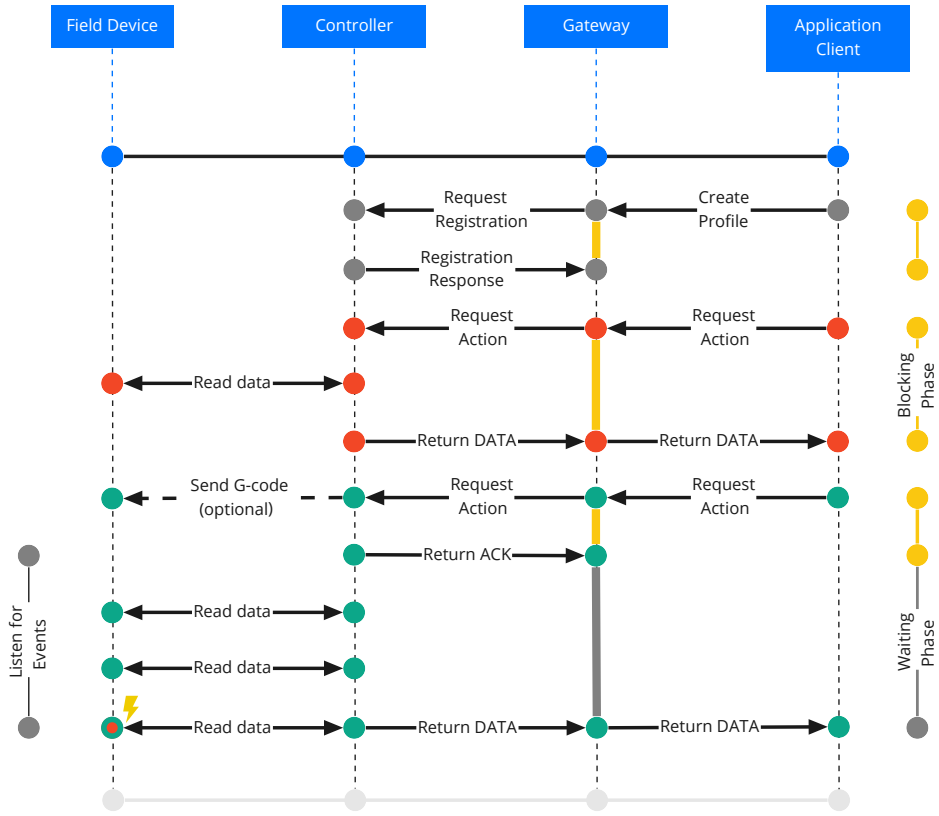

Figure 2.2: Sequence of blocking vs. non-blocking Actions.

Sensor Polling

Some service applications may want to execute a certain Action periodically, such as reading out sensor values. To reduce repetitive Action messages from the client application, a *polling*-based request scheme can be applied. In such a polling scheme, one component of the system has to manage the initiation of a polling request. Configuring the reading intervals on the Controller would result in less network traffic, but transparency would be lost. If, on the other hand, polling requests are initiated on the client side, all information would be available on the client, but the network overhead would be much higher. Since both measures are important for a functioning system, choosing the Gateway for initiating the polling requests seems to be the most suitable approach.

Fig. 2.3 illustrates how such a polling scheme can be configured. After the profile has been successfully registered on the Gateway, a client can send a configuration request to define the reading interval that will be used for polling the sensor values. The Gateway then sends Action requests to the Controller at the specified intervals to retrieve the current sensor values. The received values are stored on the Gateway to be available for potential client requests.

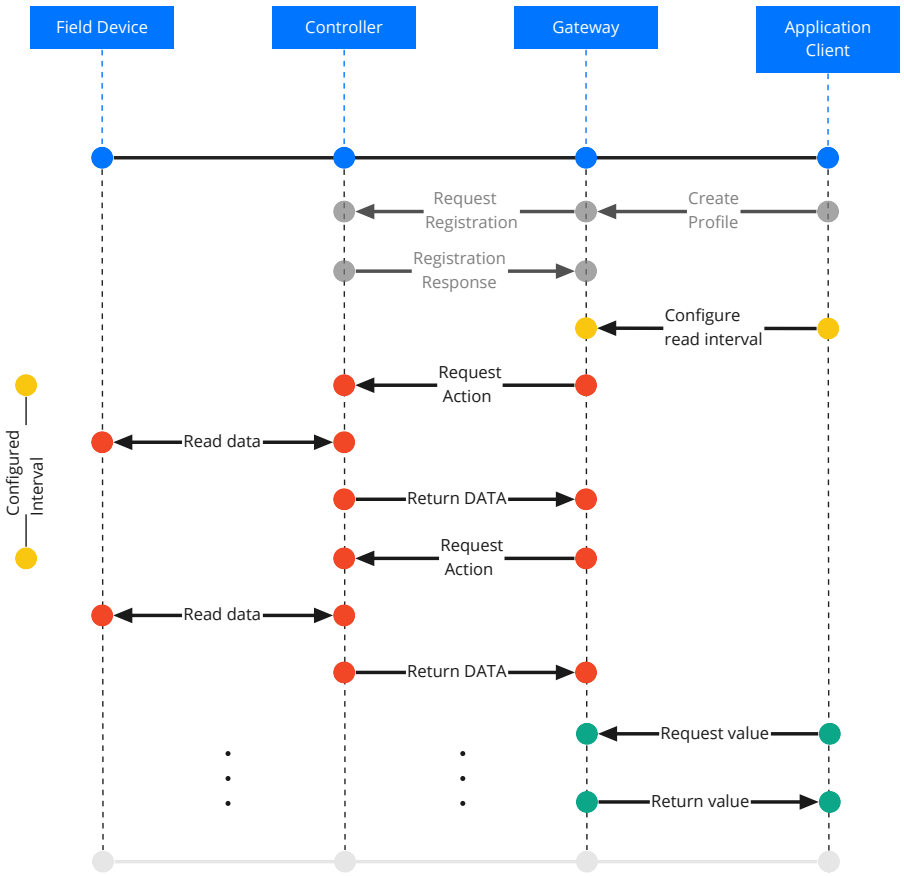

Figure 2.3: Configuration and flow of a polling scheme.

2.3.2 Profile Management

To ensure that no Controller or field device is congested by receiving multiple requests, the Gateway must carefully monitor the Profile States of all registered devices. A Device Profile can be in one of the four distinct states listed in

Table 2.1.

Profile State	Description
UNREG	The profile is created but not yet registered on a Controller.
IDLE	The profile is registered and available for Actions.
BLOCKING	An action request has been sent to the device, but no response has been received yet. All devices connected to the same Controller are blocked.
WAITING	The device is currently waiting for an Event. Only requests to this specific device are blocked.

Table 2.1: Description of all profile states.

The life cycle of a Device Profile follows the Finite-state machine (FSM) depicted in Fig. 2.4. After the Gateway receives a Registration request for a new device from an application client, it forwards this request as a Registration message to the corresponding Controller and awaits for a registration acknowledgement. If the Controller is based on a constrained device such as a single-core MCU, all devices connected to the same Controller are blocked while the device is being registered. For simplicity, this intermediate step is not included in Fig. 2.4. To handle the asynchronous Actions discussed in Section 2.3.1, the additional state WAITING is employed. Since the Profile States are handled on the Gateway, the transitions of the FSM are labelled from the Gateway's point of view.

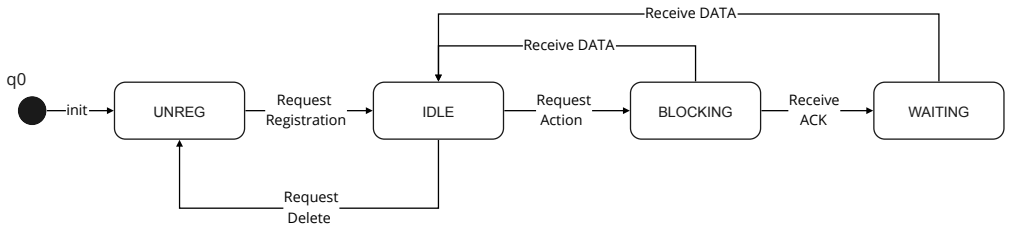

Figure 2.4: Device Profile FSM.

2.3.3 Service Exposure and Discovery

Finally, available Actions of all the configured field devices shall be advertised to the outside world through the Gateway. Upon registration of field devices on a Controller, the supervising Gateway also creates an API for client applications, which compose an execution task by combining the exposed Actions. Such exposed Actions at the Gateway (*Services*) are not bounded to one particular architectural style like REST or RPC, and we leave this part open to maintain the transparency of the end-to-end communication by choosing the right API as the situation demands. One example of the Services is briefly introduced in Chapter 3 together with the Gateway implementation and demonstrates how such a Service can also become discoverable to service applications.

Chapter 3

Implementation

To apply the principles and mechanisms introduced in Chapter 2, a simple PoC manufacturing automation system, let's call it hereon *Miniature Factory*, was built. This chapter describes the physical setup of the Miniature Factory and goes into some important implementation-specific details. Fig. 3.1 shows the setup of the Miniature Factory.

Figure 3.1: Setup of the Miniature Factory.

3.1 Field Level

The Miniature Factory consists of several field devices to mimic a manufacturing line. The robotic arm, uArm Swift Pro from UFactory [1], is the key device among them. Each uArm has an integrated MCU which handles incoming commands through UART on the USB serial interface and controls the robots' movements. UFactory provides a proprietary protocol based on the G-Code (standardized in ISO 6983 [4]) to command the uArm. Since many actions of the uArm robots are asynchronous, these robots are suited for testing more complex control procedures. To replicate a simple production line, the sensors and actuators listed in Table 3.1 are used in combination with the robotic arms.

As field devices can be dynamically configured and installed in the system, other sensors and actuators can be integrated to the Miniature Factory, if they work with interfaces supported by the uArm Controller and have a dedicated driver implemented in the firmware.

Field Device	**Interface**	**Description**
uArm Conveyor Belt	Motor port [2]	Conveyor belt (UFACTORY) to move wooden cubes between two uArms.
uArm Slider	Motor port	Automatic slider rail (UFACTORY) to move a uArm along the rail.
Ultrasonic sensor	Digital port	Ultrasonic Ranger v2.0 (Grove) used for detecting objects on the conveyor belt.
Tube sensor	Digital port	Line Finder v1.1 (Grove) used to sense if a wooden cube is available.
Color sensor	I2C port	I2C Color Sensor (Grove) used to detect the colours of the cubes.

Table 3.1: Description of the Field Devices used in the Miniature Factory.

[1] https://www.ufactory.cc/pages/uarm
[2] Proprietary port of the uArm Controller specifically for the uArm slider and belt conveyor.

3.2 Field Device Controller

3.2.1 uArm Controller

To be able to integrate different types of field devices into the system, multiple uArm Controllers were used. The uArm Controller is an ATmega2560-based Controller board which is specifically designed to control uArm and other peripherals merchandised by UFactory, including some Grove accessories. Various interfaces such as UART, IIC, digital ports and proprietary motor ports are available on the Controller. For direct control, a joystick, some buttons and other devices are also integrated on the Controller. A more detailed description of the uArm Controller is given in Appendix A.2.

3.2.2 Controller Firmware

Since the Controller is based on an Arduino Mega 2560, the firmware can be written in C++, supplemented with some Arduino-specific functionalities. Several handler functions are implemented to forward incoming Registration and Action request to the responsible drivers. In addition, a simple Event listening mechanism is included to intercept possible Events from registered devices.

Driver Management

To apply the dynamic device configuration introduced in Section 2.2.2, we implemented a specific driver for all field device types listed in Section 3.1. Each driver provides an initialisation and an Action handling function. If a field device should also be capable of handling Events, like introduced in Section 2.3.1, the driver has to implement an additional function for Event handling. To simplify the process of adding new drivers, an automatic driver initialisation script was written that inserts several skeleton code snippets and adds a customised template driver. In addition to the field device drivers, an MCU-specific driver is implemented that enables control of the firmware version, monitoring of RAM usage and a soft reset of the Controller.

3.2.3 Line Protocol

The *Line Protocol* is a custom protocol developed for the communication between the uArm Controller and the Gateway. It uses Protocol Buffers [5] to boilerplate the message packets in a machine-readable `.proto` open data format for cross-platform implementations. With the Line Protocol, predefined messages can be serialised and transferred via a USB interface.

Protocol Buffers

Protocol buffers are a language-neutral and platform-neutral method to serialize structured data. With automatically generated code, the encoding and decoding of such messages can be seamlessly implemented in different languages. This feature was used in the PoC system to generate protocol handlers written in C++ and Python. More specifically, to generate the protocol handler on the Controller side, Nanopb [6] was employed. Nanopb is an implementation of Google's Protocol Buffers designed specifically for memory-constrained embedded systems. Since Protocol buffers are an efficient and fast way to serialize data, the latency of the line protocol can be held below 1ms, which is small enough for the intended purposes.

Message Structure

Two main message types are defined to implement the communication mechanism between the Gateway and the Controllers described in Chapter 2. Request messages, which are used to send control Actions to the Controller and Response messages to send feedback to the Gateway. A Request message can either contain a Registration message or an Action message. The first is used to initialize a new device on the Controller and the latter is used to send control messages to the corresponding device. To define the configuration and control settings, both messages include a driver-specific sub-message. For clarification, Fig. 3.2 shows the previously described message structure.

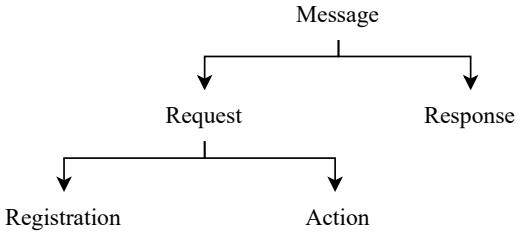

Figure 3.2: Tree diagram of the message structure.

The Response message contains a *Response Code* to distinguish between the different types of responses discussed in Section 2.3.1. Possible Response Codes and their meaning are listed in Table 3.2.

Response Code	Description
DEBUG	The message is for debugging purposes only.
ERROR	The message should raise an error exception on the Gateway to handle possible initialization failures.
ACK	The message is used to acknowledge a received Action or a completed Registration. Apart from a device identification, it contains no data.
DATA	The message is used to return raw data to the Controller and report successful execution.

Table 3.2: Description of all Response Codes.

The following code snippet shows how such Protocol Buffer messages are defined. Each field in a message has a specific type which (e.g. in the case of the Response Code) can be an enumeration type. To restrict a message to contain only one of multiple fields, the keyword `oneof` can be used. This enables efficient coding while supporting heterogeneous message structures.

```
// message sent from Controller to Gateway
message Response {
  ResponseCode code = 1;
  uint32 profile_id = 2;
  bytes payload = 3;
}

// message sent from Gateway to Controller
message Request {
  oneof request_type {
    Action action = 1;
    Registration registration = 2;
  }
}
```

Listing 3.1: Example structure of Protocol Buffer messages.

3.3 Gateway Implementation

To manage the Device Profiles and control the uArm Controller, a simple Gateway implementation was written in Python. The Line Protocol Handler was implemented using Protocol Buffer's code generation and a custom class was created for each driver to support the different field devices. For testing purposes, the Device Management Gateway was run on a regular computer connected to the uArm Controller via USB. Instead of a regular computer, the Gateway implementation can also be loaded onto a Linux box to create a stand-alone Manufacturing Automation System. To enable the operation of a large number of field devices, multiple uArm Controllers can be connected to the Gateway.

Chapter 4

Conclusion and Future Work

The mechanisms and patterns presented in this work enable transparent management and control of heterogeneous field devices. In addition, the dynamic driver initialisation makes it possible to configure new devices at runtime and thus simplifies rearrangements on the shopfloor. The functionality of these principles could be validated through the implementation of the Miniature Factory. In particular, the communication mechanism (e.g., the use of Protocol Buffers for Field Device management messages) achieved a light-weight communication scheme while maintaining transparent interaction between the service applications and hardware components. With the Miniature Factory, a basic system was built to support future research on IAS architectures and their integration into explainable systems.

4.1 Future Work

Since the diversity and number of the field devices tested in the Miniature Factory is limited, the completeness of the Device Profile still requires further evaluation. Specifically, the usability of the Device Profile must be thoroughly tested when it is exposed to external service applications. This work focused more on the mechanisms within the autonomous system. To complete the structure of the system, more research needs to be done on the interfaces to external service applications. Furthermore, it would be interesting to investigate on what kind of characteristics of the Device Profile and the generated API on the Gateway contribute to the transparency of the system. To further improve the management of field devices, a method needs to be developed to convey hardware requirements and setup descriptions along with the Device Profile. This would simplify the operation of an IAS. Following the trend of automation, it should be further investigated whether different transparency requirements arise when the operation of an IAS is controlled by humans or by machines. This also includes a consideration of the extend of automation, e.g. which parts can be executed automatically and which cannot.

Bibliography

[1] R. Rio, "What are iot, iiot and industry 4.0?" (accessed Jan. 02, 2021). [Online]. Available: https://www.arcweb.com/blog/what-are-iot-iiot-industry-40

[2] IEC, "62264-1: Enterprise-control system integration–part 1: Models and terminology," *IEC: Geneva, Switzerland*, 2013.

[3] P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace," Internet Requests for Comments, Network Working Group, RFC 4122, July 2005. [Online]. Available: https://tools.ietf.org/html/rfc4122

[4] ISO, "6983-1: Automation systems and integration — Numerical control of machines — Program format and definitions of address words — Part 1: Data format for positioning, line motion and contouring control systems," International Organization for Standardization, Geneva, CH, Standard, 2009.

[5] "Protocol buffers," Google, (accessed Dec. 23, 2020). [Online]. Available: https://developers.google.com/protocol-buffers

[6] P. Aimonen, "Nanopb – protocol buffers with small code size," (accessed Dec. 30, 2020). [Online]. Available: https://jpa.kapsi.fi/nanopb/

[7] *uArm Swift Pro – Developer Guide*, UFACTORY, (v1.0.6).

[8] *uArm Controller – User Manual*, UFACTORY, December 2018, (v1.0.1).

Appendix A

Hardware Component Description

A.1 uArm Swift Pro

The uArm Swift Pro robotic arm is illustrated in Fig. A.1 and further specifications are given in Fig. A.2. These figures and a more detailed description of the uArm Swift Pro can be found in the Developer Guide [7].

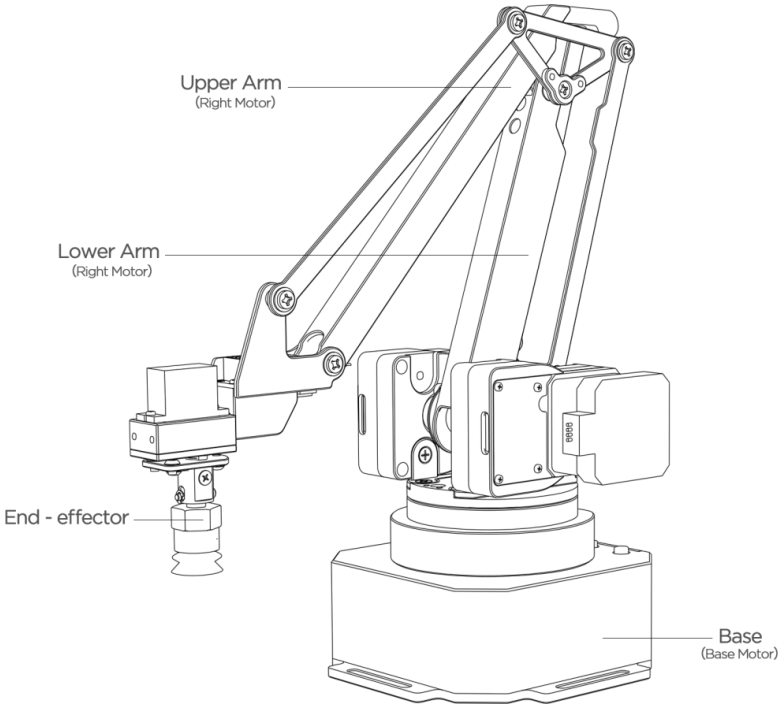

Figure A.1: Depiction of the uArm Swift Pro.

Specifications	
Weight	2.2kg
Degrees of Freedom	4
Repeatability	0.2mm
Max. Payload	500g
Working Range	50mm ~ 320mm
Max. Speed	100mm/s
Connector	Micro USB
Wireless	Bluetooth 4.0
Input Voltage	DC 12V
Power Adapter	Input:100 ~ 240V 50/60Hz; Output: 12V5A 60W
Operation Temperature & Humidity	0℃-35℃ 30%RH-80%RH noncondensing
Storage Temperature & Humidity	-20℃-60℃ 30%RH-80%RH noncondensing
Hardware	
Joint Type	Customized Gearbox + Stepper
Position Feedback	12 bit Encoder
Reducer	Customized ultra-thin Gearbox
Dimension(L*W*H)	150mm*140mm*281mm
Mother Board	Arduino MEGA 2560
Material	Aluminum
Baud Rate	115200bps
Extendable I/O Interface	I/O *27，IIC *1，5V*1，12V*1，Stepper*1

Figure A.2: Specifications of the uArm Swift Pro.

A.2 uArm Controller

Fig. A.3 and Fig. A.4 give a more detailed description of the uArm controller. All contents of this section are taken from the uArm Controller User Manual [8].

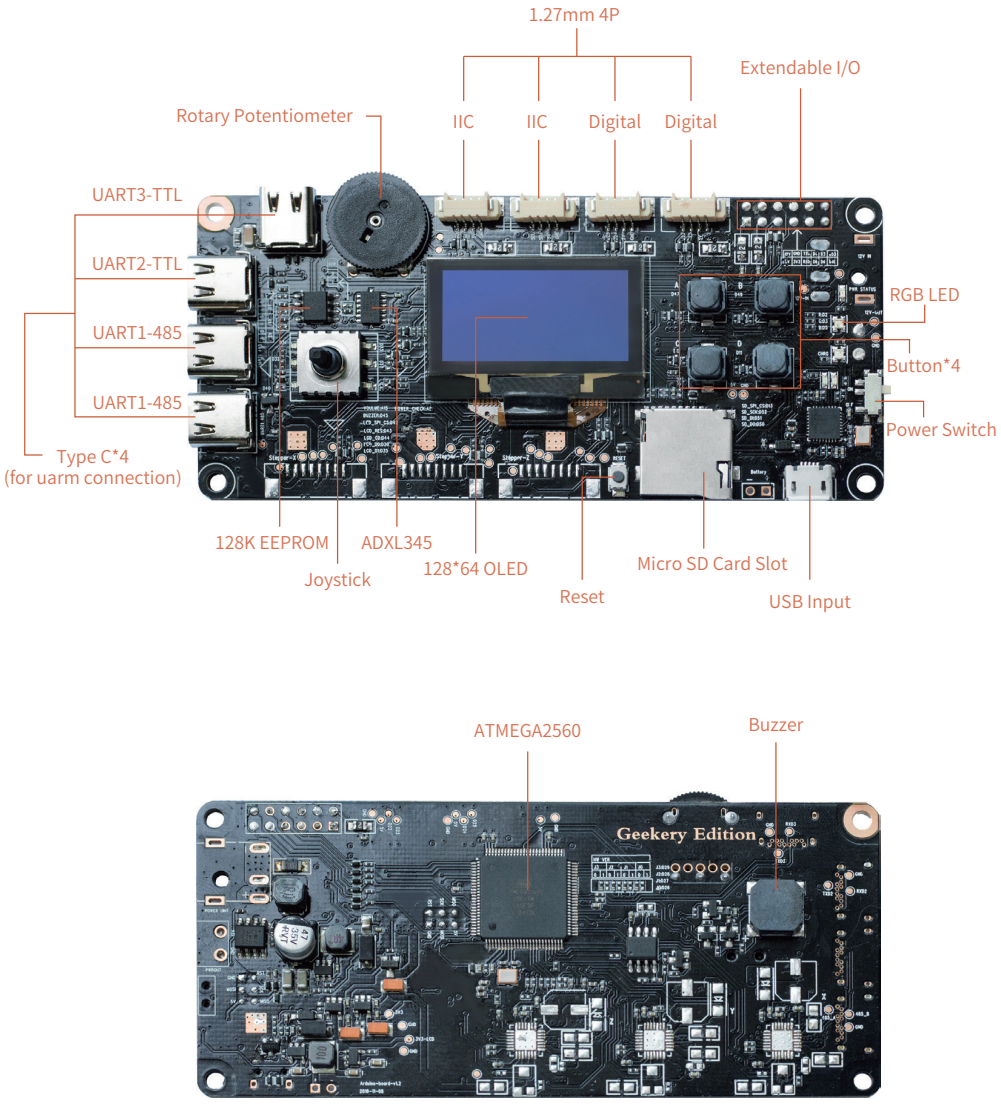

Figure A.3: Labeled illustration of the front and back of the uArm controller.

Specification	
Weight	0.15kg
Dimension(L*W*H)	150mm*132mm*281mm
Connection with PC	Micro USB
Input Voltage	USB 5V
Main Controller	ATMEGA2560 (Arduino compatible)
Display	128x64 OLED
Buttons	4
Rotary Potentionmeter	1
TypeC	4 (only for uarm connection)
RGB LED	1
Micro SD	1
4P Connector	2xDigital IOs / 2xIIC
Extendable I/O	6xdigital IOs
Operation Temperature & Humidity	0℃-35℃ 30%RH-80%RH noncondensing
Storage Temperature & Humidity	-20℃-60℃ 30%RH-80%RH noncondensing

Figure A.4: Detailed description of the uArm controller parameters.