DIPLOMARBEIT

Embedded Execution Environment for Modular Firmware Structures

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs

unter der Leitung von

Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Markus Vincze
Dipl.-Ing. Dr. techn. Alois Zoitl

E376
Institut für Automatisierungs- und Regelungstechnik

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Martin Melik-Merkumians
Matr.-Nr.: Matr.-Nr.: 0125853
Possingergasse 39–51/17/14, A–1160 Wien

Wien, im März 2009

Martin Melik-Merkumians

Abstract

The rising pressure of competition forces the industry to more flexible production. The ability of effective lot-size one and small batch production is becoming an important requisite for success on the todays competitive markets. However, classical manufacturing plants are not suited for the new requirements of todays markets. Decentralized systems helped to save money for cabling of the automation devices, but changed nothing on the monolithic structure of classic industrial process and measurement systems. Such structures are not suited for fast reconfigurability.

The emerging standard *IEC 61499 – Function blocks* introduces a distributed execution model, suited for fast reconfigurability. Unfortunately it is not widely used, as there are few automation devices that support IEC 61499 natively. Although there are runtime environments for IEC 61499, none of them is suited for use in cheap smart sensors and actuators, as memory size and computing power are too low on such devices. But direct integration of smart sensors and actuators would vastly improve and simplify the possibilities of designing and handling distributed control systems for e.g., manufacturing plants or building automation systems.

This diploma thesis targets this problem, by developing an embedded execution environment for modular firmware structures with direct integration in IEC 61499 networks suitable for embedded systems with limited memory. First the requirements of a firmware execution environment will be identified by an use case analysis of the programming phase and the operational phase of a typical automation device. Based on the findings of the use case analysis the requirements for the execution environment are identified, which are the base for a first conceptual design for such an execution environment.

The findings of the conceptual design are then used to modify the existing IEC 61499 runtime environment FORTE. The modification decreased the code size of FORTE for about 56% to 172076 bytes. Therefore it is now possible to run FORTE on cheap micro controllers, with limited ROM and RAM, commonly used in smart sensors and smart actuators. Thereby the granularity of a distributed system can be further reduced and therefore the available computing power can be better utilized, and the reuse and modularity of system parts can be increased.

Kurzfassung

Der steigende Wettbewerbsdruck zwingt die Industrie zu flexibleren Produktionsanlagen. Die Fähigkeit zu effizienter Losgröße Eins- und Kleinserien-Produktion sind wichtige Voraussetzungen für den Erfolg in den wettbewerbsintensiven Märkten. Jedoch sind klassische Produktionsanlagen nicht mehr den Anforderungen heutiger Märkte gewachsen. Dezentrale Systeme haben dazu beigetragen Geld für die Verkabelung von Automatisierungsgeräten zu sparen. Dies änderte jedoch nichts an der monolithischen Struktur der klassischen Automatisierungssysteme, welche nicht für schnelle Rekonfigurierbarkeit geeignet sind.

Der sich gerade etablierende Standard *IEC 61499 - Function blocks* führt ein verteiltes Ausführungmodell ein, welches für schnelle Rekonfigurierbarkeit geeignet ist. Leider ist der Standard noch nicht weit verbreitet, da es nur wenige Geräte gibt, die die Modelle der IEC 61499 unterstützen. Zwar gibt es Laufzeitumgebungen für IEC 61499, jedoch ist keine für den Einsatz in billigen intelligenten Sensoren und Aktuatoren, aufgrund der begrenzten Speichergröße und Rechenleistung dieser Geräte, geeignet. Die direkte Integration von intelligente Sensoren und Aktoren würde die Entwicklung und Handhabung verteilter Steuerungssysteme von z.B. Fertigungsanlagen oder Gebäudeautomationsanlagen erheblich verbessern und vereinfachen.

Das Ziel dieser Diplomarbeit ist es dieses Problem durch die Entwicklung einer Ausführungsumgebung für modulare Firmware-Strukturen für Systeme mit beschränkten Speicherressourcen und Rechenleistung zu lösen, die eine direkte Integration in IEC 61499-Systeme ermöglicht. Zuerst werden die Anforderungen an die Ausführungsumgebung mit Hilfe einer Use Case Analyse der Planungs- und der Betriebsphase eines typischen Automatisierungsgeräts ermittelt. Basierend auf den Ergebnissen der Use Case Analyse und der ermittelten Randbedingungen des Ausführungsumfeldes, wird ein erstes Grobdesign für eine solche Ausführungsumgebung entwickelt.

Die Ergebnisse der Entwürfe werden dann verwendet, um die bestehende IEC 61499-Laufzeitumgebung FORTE zu verändern. Die durchgeführte Modifikation der FORTE verringert die Codegröße der FORTE um rund 56% auf 172076 bytes. Dadurch ist die FORTE auf billigen Mikrocontroller mit begrenztem ROM- und RAM, die typischerweise für intelligente Sensoren und intelligente Aktuatoren verwendet werden, lauffähig. Durch die dadurch erreichte Verringerung der Granularität von verteilten Systemen, kann die zur Verfügung stehende Rechenleistung besser genutzt werden. Des Weiteren kann dadurch die Modularität und die Wiederverwendbarkeit von Systemteilen erhöht werden.

Acknowledgment

I want to thank my advisors Ao. Univ.-Prof. Dipl.-Ing. Dr.techn. Markus Vincze for supervising my diploma-thesis, and Dipl.-Ing Dr.techn. Alois Zoitl for his helpful suggestions and discussion, his patience while correcting my diploma-thesis, and his ongoing support the whole time during this work. I also want to thank Dipl-Ing. Reinhard Hametner and Dipl.-Ing. Ingo Hegny for their helpful hints, cheering words and the many coffee breaks we shared.

My thanks also go to my friends and fellow students René Paris, Dipl.-Ing. Maria Klonner and Irina Barnay for their support and friendship.

Special thanks goes to my girlfriend Anita, who supported me always during my studies and endured me when I had to learn for my exams.

Above all, I want to thank my parents Dipl.-Ing. (FH) Edwart Melik-Merkumians and Christa Melik-Merkumians who have supported me throughout my entire life. They have always encouraged me to do my best no matter what it was, and enabled my to study. Without them I wouldn't be where I am today.

I dedicate my diploma-thesis to my mother Christa Melik-Merkumians, who passed away this year, which was much too soon. We all love and miss you.

Martin Melik-Merkumians

Contents

List of Figures

List of Tables

1 Introduction

As eastern countries urge their way into the markets, the pressure of competition is growing steady on the western industry. Due to the low-labor-costs in those countries the western industry cannot compete with them in the low-cost mass production market. Therefore the western companies have to evolve and strive for the market segment of high quality small batch production.

But to be successful in the small batch and lot-size one production segment the manufacturing facilities have to be highly adaptive. Unfortunately most of the used manufacturing facilities are designed as monolithic systems, even as the latest trend to decentralized systems changed nothing on the principle monolithic structure. If a systems has to be changed the monolithic software, running in a programmable logic controller, has to be reprogrammed almost from scratch. The industry recognized the problem and a new standard has been developed.

The new standard *IEC 61499 – Function blocks* introduces a distributed execution model, where each device can be programmed separate and offer its services to the system. Therefore to design a system it is only necessary to connect the offered device service in such a way, that the behavior of the complete system fits the desired behavior. However there a few small systems which can be directly integrated in a IEC 61499 system, as there are no embedded systems native to the execution model of IEC 61499 and the available runtime environments for IEC 61499 are far to big to be run on an embedded system with limited memory. But if the firmware of devices like smart sensors and smart actuators could be implemented in a way that the device is compliant to IEC 61499 huge benefits in system integration could be generated.

Therefore the goal of this diploma thesis is to identify the needs of a embedded system firmware and then to develop an IEC 61499 runtime environment, which suits the needs of firmware development and the limited resources of an embedded system.

1.1 Conceptual Formulation

The goal of this work is to enable firmware development and execution in IEC 61499. Based on the 4DIAC[1] runtime environment (FORTE[2]), an embedded execution environment for modular firmware structures shall be developed. Therefore it is necessary to analyze the life-cycle of a typical embedded automation device. The main problem in the development of the embedded execution device will be the limited ROM size of typical embedded system, which is why the main interest will be to keep the code size of the embedded execution environment as small as possible.

1.2 Solution Statement

The first step will be to analyze a typical embedded automation device, such as a smart sensor or a smart actuator. The analysis will comprise a requirements analysis for embedded devices and a use case analysis of the programming phase and the operational phase of the embedded device. Based on the findings of the analysis a conceptual design of an embedded execution environment for modular firmware structures will be developed. This concept will be the base for the modifications of the existing IEC 61499 runtime environment FORTE, which will be the base for the embedded execution environment.

[1] Framework for Distributed Automation and Control
[2] Available at http://www.fordiac.org/

2 State of the Art

Software engineering, despite of being a young engineering discipline, had and still has a large impact on many other engineering disciplines. Through the use of programmable machines we gained much flexibility as one machine was now capable to perform many different tasks. The most famous programmable machine is the PC, which can be used as a typewriter, a calculator, an entertainment center, and much more. Programmable machines also found their way into the field of IPMCS. Programmable logic controllers (PLC) revolutionized the way of building and programming manufacturing facilities, as the new technology allowed to build much more complex systems as it has been possible with relay-based logic hardware. But also new challenges as real-time capabilities, software design for IPMCS and interoperability between different manufacturers arose. The IEC 61131 was developed to deal with those new challenges. Due to the high competition on todays markets new problems arise, such as flexible production lines and "lot-size one" production. To achieve this flexibility distributed IPMCS (dIMPCS) were devised and IEC 61499 was established to provide a common platform for IPMCS manufacturers.

2.1 Software Design

Software design often seems to be the first step of software development, but software design is not an end in itself. First the problem that the software shall solve must be identified. After the determination of the purpose and the specification of the software the software design process begins. It's main purpose is to solve the problems imposed by the specification and the planning of the software architecture as well as the low-level components and algorithm implementation as both are influencing each other. Most modern software design tools are using the Unified Modeling Language (UML). By the use of these tools developers try to manage the complexity imposed by the task of software design by splitting the problem in different parts (e.g., systems, subsystems, modules, classes) and different views (e.g., structural view, behavioral view, functional view). Design patters are another important and potent tool, as they describe a general solution for common design problems.

Through the use of design patterns a robust design can be developed as the advantages and disadvantages of the patterns are well known and the patterns are tested and proven to work. Yet another upcoming software model is the component software paradigm (also known as Component-based software engineering (CBSE)). The goal of CBSE is the separation of concerns in respect of the functionality in a software systems. Software components shall be able to provide a service to the rest of the software system without depending on the services or functions of other components. The benefit of such a design is, that those components can be easily reused and exchanged with newer versions of that component.

2.1.1 UML

The Unified Modeling Language (UML) is a standard developed by the Object Management Group (OMG). The initial version 1.1 was released in November 1997, since February 2009 the current version is 2.2.

UML defines many different diagrams as the structure diagram (also known as class diagram), the deployment diagram, state charts, activity charts, use case diagrams, and sequence charts. The core diagrams of UML are the structure diagram, the state chart, and the sequence diagram. The other diagrams model additional aspects of the system.[Dou99]

The goal of UML is to allow the user to define a model of the system. A model is a coherent set of abstractions that represents the model to be designed. As the model consists of semantics and the view of the user on those semantics an important part of the user is the definition of the semantics of the system under development[Dou02]. The three primary aspects of the semantics are:

- Structural aspect
- Behavioral aspect
- Functional aspect

The structural aspect deals with the entities that make up the system. If a "snapshot" from the system is taken at runtime, the set of objects and their relations represents the current state or condition of the whole system, therefore the set of classes and their relations specify all possible sets of objects and object relations at runtime. The difference between objects and classes is, that classes exist only at design-time, as classes are a specification, and objects only exist at run-time being instances of those classes. At larger scale subsystems and components (both basically big objects and classes) form larger-scale

abstractions for complex systems. Without those abstractions it would be impossible to handle today's comprehensive systems.[Dou99]

The behavioral aspect defines how the structural elements work and interact in the running system. This can be modeled for individual structural elements (e.g., objects, classes, subsystems, components) and for assemblies of elements to achieve large-scale behaviors. State charts and activity diagrams are used for individual structural elements to specify actions and their permitted sequencing. The behavior of assemblies of structural elements, also called collaborations, are modeled by sequence and collaboration diagrams.[Dou02]

The function aspect refers to the required behavior without regard of the implementation of that behavior. To model the behavior UML provides the use case diagram. Use cases describe the interaction between one or more actors. Actors can be humans but also other parts of the system (e.g., an automated teller machine requests the account balance of the user). As a use case is coarse description of the interaction the detailed requirements of the use case are modeled with state charts and interaction diagrams.[Dou02]

The goal of the design process is to create a complete, consistent and accurate application model that can be verified via analysis or execution and could be used to generate the source code, thus greatly reducing coding effort and maintain the consistency between the UML model and the source code.[Dou02] Detailed information on UML diagrams can be found in [OMG09a], [OMG09b], [ISO05], [Öst01], [Dou99] and [Dou02].

2.1.2 Design Patterns

Design patterns are generalized solutions to common design problems. They are not precoded chunks of software ready to be used by the programmer. A design pattern is the abstractions of a solution of a given design problem. As an example outside of the world of software design, the solution of crossing a ravine could be a bridge without specifying what kind of bridge, maximum bearing load, or what materials are to be used to build the bridge. There are several design patterns to one problem, as there is also more than one way to cross a ravine.
As design patterns are descriptions of abstract solutions, the authors of design patterns are commonly using following elements to structure a pattern:

- Name
- Problem
- Solution

- Consequences

The name of the pattern should be a catchword that describes the problem to solve and the solution in one or two words. The problem section gives a detailed explanation of the design problem, which the pattern will solve. The problem section will usually also state a number of conditions that must be met, so that the pattern is applicable. The solution section describes the elements of the design pattern and their interactions, responsibility, and relations to each other, so that the problem will be solved. The consequences section deals with the advantages and disadvantages that comes with the use of the particular design pattern.[GHJV96] As design is all about optimization, design patterns are also about optimization. Typical design parameters that are reviewed in the consequences section are:

- Worst case performance
- Average case performance
- Predictability
- Scheduability
- Memory usage
- Reuseability
- Portability
- Maintainability
- Extendability
- Development time/effort
- Safety
- Reliablilty
- Securtiy

As it is impossible to achieve all of this simultaneously there must be an order of importance so the final design is "optimal enough", because as we optimize certain aspects of the design, we also deoptimize other aspects[Dou02].

2.1.3 Component Software

The component software approach is a new paradigm for software reuse. Instead of reusing individual functions or classes or class hierarchies the component software approach strives to reuse so-called "components". Simply put a component is a system-independent software that provide certain services to the system. Components are characterized by three properties:

- It is a unit of independent deployment
- It is a unit of third-party compositions
- It has no (externally) observable state

Those three points have several implications. First for a component to be deployed independent it has to be separated from its environment and other components. As a component is a unit of deployment it can never be deployed partially. In this context a third-party is a component user that can not be expected to know the construction details of the component. Second for a component to be composable with third-party components, it needs a well defined interface for interaction with its environment and its implementation has to be encapsulated. Third and finally a component should not have any (externally) observable state. It is necessary that a component cannot be distinguished from copies of itself, except for attributes that are not contributing to the components functionality (e.g., serial numbers used for accounting).[Szy02] Components itself can be programmed in functional or object-oriented languages and although there should be no observable state at component level, it is allowed that the objects, that make up the component, are allowed to have a state as long as it is assured that the component itself has no externally observable state.
Based on the points above, it is possible to combine components to a system and replace or add components to a existing system very fast. This is a major advantage compared to classes. As a result of this, a new market of software component providers has evolved and system designers are able to buy standard software components and then customize it to their needs through parameters. It is also possible to buy components of different quality level for example a cheaper and slower one or a expensive and faster one. The advantage for system designers is that they don't have to develop every new system from scratch, as well tested and documented components are available. Thus system designers can reduce the risk on new developments. The system designers also gain another degree of freedom as they can decide which components will be bought and which components will be developed in-house.

2.2 IEC 61499

IEC 61499 – Function blocks is a new family of standards. Its main purpose is to introduce a function block (FB)-oriented programming model for distributed IPMCS (dIPMCS). The IEC 61499 standard family consists of three parts:

IEC 61499-1 (2005): Function blocks – Part 1: Architecture
This part describes the general architecture and all general model behind this standard.

IEC 61499-2 (2004): Function blocks – Part 2: Software tool requirements
Part 2 gives rules and concepts for software tool developers implementing an engineering tool for IEC 61499. Most of the information given is rather general. The most important definition of this part is an exchange data format for the software models defined in IEC 61499-1. This is a main requirement for vendor independent software libraries.

IEC 61499-4 (2005): Function blocks – Part 4: Rules for compliance profiles
IEC 61499 leaves several points open to implementation. How these items are solved should be described in related compliance profiles. IEC 61499-4 defines the structure of such compliance profiles.

The FB paradigm is a well established concept for software encapsulation and defining reusable components. The innovation of IEC 61499 is in the new event-driven execution model. A FB only starts the execution of an algorithm if the corresponding event occurs. Events are similar to boolean variables as their value can be 0 or 1, but unlike boolean variables events can only hold the value 1 for instantaneous moments. FBs are representations of components which can be implemented in the form of software but also in the form of hardware.[Vya07] This brings object-oriented (OO) programming languages to mind, as one of the main characteristics of OO languages (OOL) is encapsulation. But the FB paradigm of IEC 61499 is not a OO concept, due to the lack of a main characteristics of OOLs: inheritance.[Lew01] Therefore IEC 61499 is only an object-based programming language.

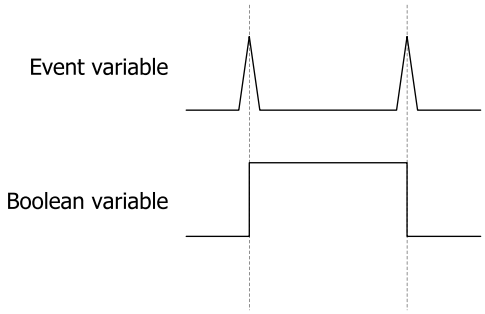


Figure 2.1: Boolean vs. Event variable based on [Vya07]

2.2.1 Function Blocks

The standard defines three different types of FBs:

- Basic FBs
- Composite FBs
- Service Interface FBs (SIFBs)

Each type sports the same interface as shown in Figure 2.2. The event interface is at the top of the FB, which is usually referred as the "head", where the event inputs are on the left-hand side and the event outputs are on the right-hand side. The data interface, usually called the "body", is located below the event interface and again the the inputs are on the left-hand side and the outputs are on the right-hand side. The FB type name is placed in the middle of the FB. The instance name above the FB is a user-defined name for this specific FB. As the position of data and event in- and outputs suggests the data and event flow is from left to right. If we again compare FB type name and instance name with elements of OOLs, then the FB type name would be the class name and the instance name would be the name of the instanced object of this class.
The "head" represents the execution control and the "body" represents the algorithms, functions, and internal data of the FB.

Basic Function Blocks

The main component of the Basic FB is a state machine which controls the actions taken by the FB if an input event occurs. This state machine is called Execution Control Chart (ECC) and is based on the Sequential Function Charts of IEC 61131-3.[Zoi09] The EEC consists of three elements:

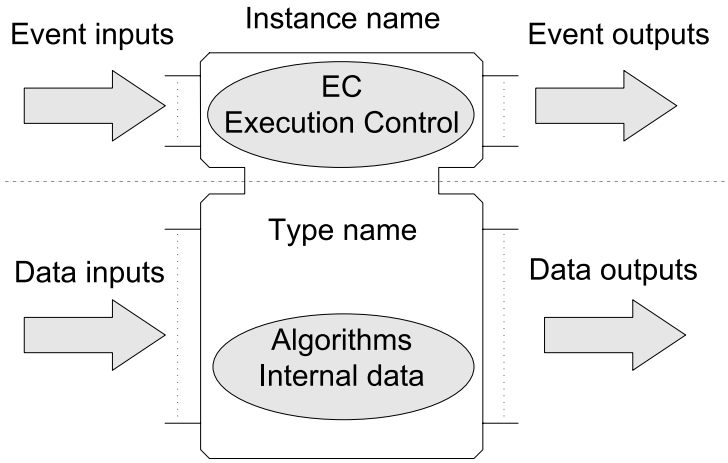


Figure 2.2: Properties of a IEC 61499 function block based on [IEC05a]

- EEC states
- EEC action
- EEC transitions and guards conditions

EEC actions typically consists of an algorithm to be executed and an output event to be sent, but it is also allowed that an action only includes an algorithm or an output event to be sent. Each EEC action is affiliated with an EEC state and the different states are connected through EEC transitions. The transitions are typically guarded by boolean logic expressions. The first transition of the actual state which guarding condition is true, will be taken. Upon state entry the associated ECC action will be performed. The algorithms can be programmed in any language, but the algorithms are only allowed to access data inputs, data outputs, and internal variables of the FB. Figure 2.3 shows an example ECC.

Composite Function Blocks

Composite FBs encapsulate FB networks (FBNs) and therefore helps to reduce complexity in the FBN like a subroutine in programming languages. The contained FBN is connected to the outer FBN through the event/data inputs and outputs of the Composite FB (see Figure 2.4). The Composite FB has no own ECC but the combined ECCs of the contained FBs can be seen as the ECC of the Composite FB.

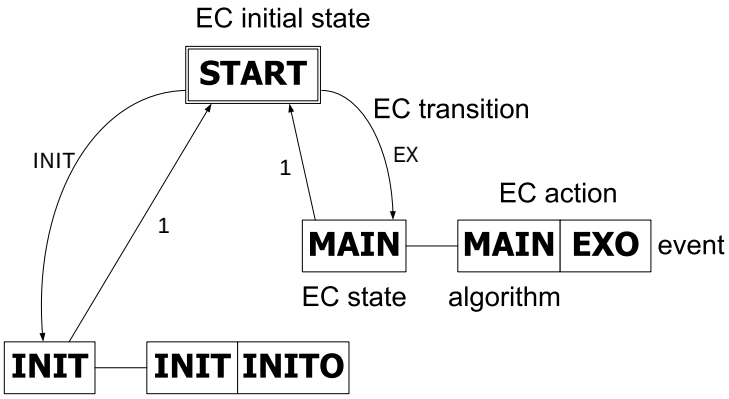


Figure 2.3: ECC example based on [IEC05a]

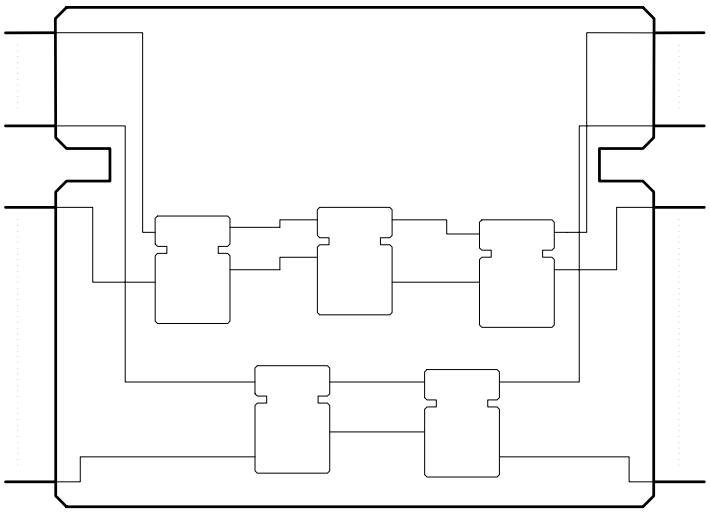


Figure 2.4: Composite FB based on [IEC05a]

Service Interface Function Blocks

SIFBs are used as interfaces to provide functionality that is beyond the scope of IEC 61499. Controlling the device hardware (e.g., I/O interface, communication interface), or including libraries that contain functions needed by the control system are typical application areas of SIFBs. The encapsulated functionality is described through service primitives in the form of time sequence diagrams. There are two general types of SIFBs, the responder and the requester type. The responder type is hardware-triggered, which means that the SIFB can send output events caused by the actions the the resource or the hardware (e.g., interrupts or traps) without prior activation through an input event. The requester type is application-triggered and will therefore wait until an input event arrives and triggering it's service an an output event if necessary.[Zoi09]

2.2.2 Execution Model

A major feature of IEC 61499 is the ability to reconfigure the IPMCS, thus following management commands are implemented to provide the ability to reconfigure a system:

- Create - creates FB types, resource types, data types, FB instances and resource instances
- Delete - deletes created types and instances
- Start, Stop, Kill - changes the state of a managed FB
- Read - reads from data inputs and data outputs
- Write - writes parameters to data inputs
- Query - gets information on available types, instanced FBs and resources, connections, and the status of an FB

As FBs can be created or deleted during runtime, it must be possible to start and stop an FB to change the system in an orderly fashion. The standard describes the operational behavior of a FB in form of a state machine [1], as shown in Figure 2.5. When a FB is created it enters the IDLE state. There the FB initializes and all its variables (data inputs, data outputs and internal variables) get their initial value. From the IDLE State the FB can only enter

[1] Not to be confused with the EC state machine

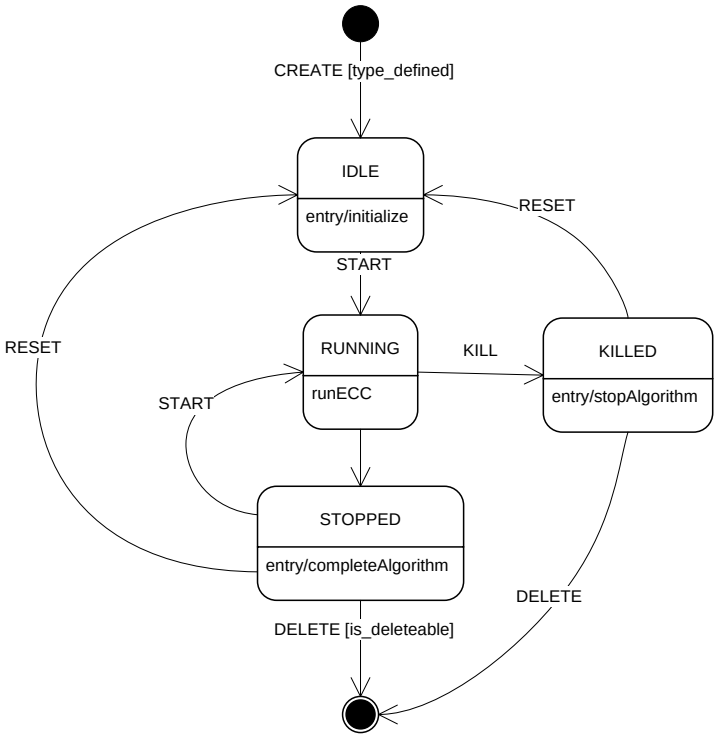


Figure 2.5: Operational State Machine of a IEC 61499 FB based on [IEC05a]

the RUNNING state when the "start" command occurs. In this state the FB will process incoming events and its internal actions will be executed. From there the FB can enter the STOPPED state of the "stop" command occurs or the KILLED state if the "kill" command is sent. If the FB enters the STOPPED state no further incoming events are accepted, but all running algorithms will finish. Otherwise if the FB enters the KILLED state also no further incoming events are accepted, but all running algorithms are aborted and therefore the internal state could be corrupted. A stopped FB can be brought back to the RUNNING state by the "start" command, can be deleted by the "delete" command or can be reseted by the "reset" command and therefore enter the IDLE state again. A FB in KILLED state can only be deleted or reseted and so enter the IDLE state again.

2.2.3 Distribution Model

Before discussing distribution in IEC 61499, lets consider the elements of the IEC 61499 environment (see Figure 2.6). The devices are physical units that

are connected to the controlled process. They consist of a communication interface, a process interface an device management components and can or can not contain resources. The communication interface provides the communication services for the application parts residing in the device. A resource is a functional element that has independent control of its operation and that can contain applications or application parts. Within a device a resource can be created, deleted, configured, etc. without interfering with other resources in the device and their contained application or application parts.[Zoi09] The process interface provides services for accessing the actuators and sensors needed to control the process. The application is a logical unit, built up by FBs. The standard states that a FB is an atomic unit of distribution[IEC05a], which means that it is not possible to distribute a FB. The IEC 61499 distribution model allows distribution by allocating FB instances to different resources in one or more devices. Due to the distribution of the FBs of an application the functionality of it shall not be affected, though the timing and the reliability of the communications function will affect the timing and the reliability of the distributed application.

2.3 Summary

Software engineering is of great importance for all kinds of business, also for the manufacturing industry. Through the use of PLCs all modern manufacturing facilities have a huge amount of processing power in their production lines. The IEC 61131-3 greatly improved reuse of PLC software and interoperability, but failed to tap in the full potential of this processing power, as every IEC 61131 control system is an isolated application. The IEC 61499 harness this potential of distributed processing power, but to use it to its fullest potential we have to honor the basic principals of software design. Design tools like UML, design patterns and design paradigms like component-oriented software help to devise a custom software system through standardized processes, thus decreasing development time and increasing software quality and therefore product quality.
Development of IPMCS device firmware with IEC 61499 FBs would be such a standardized process and can be implemented in the component-based software paradigm. The device itself would be represented by a FB and offers its services to the system through the FBs interface. The FB and the device could be reused in many different IEC 61499 systems without reprogramming parts of the firmware and without the need to adjust the interface for use in a IEC 61499 system.

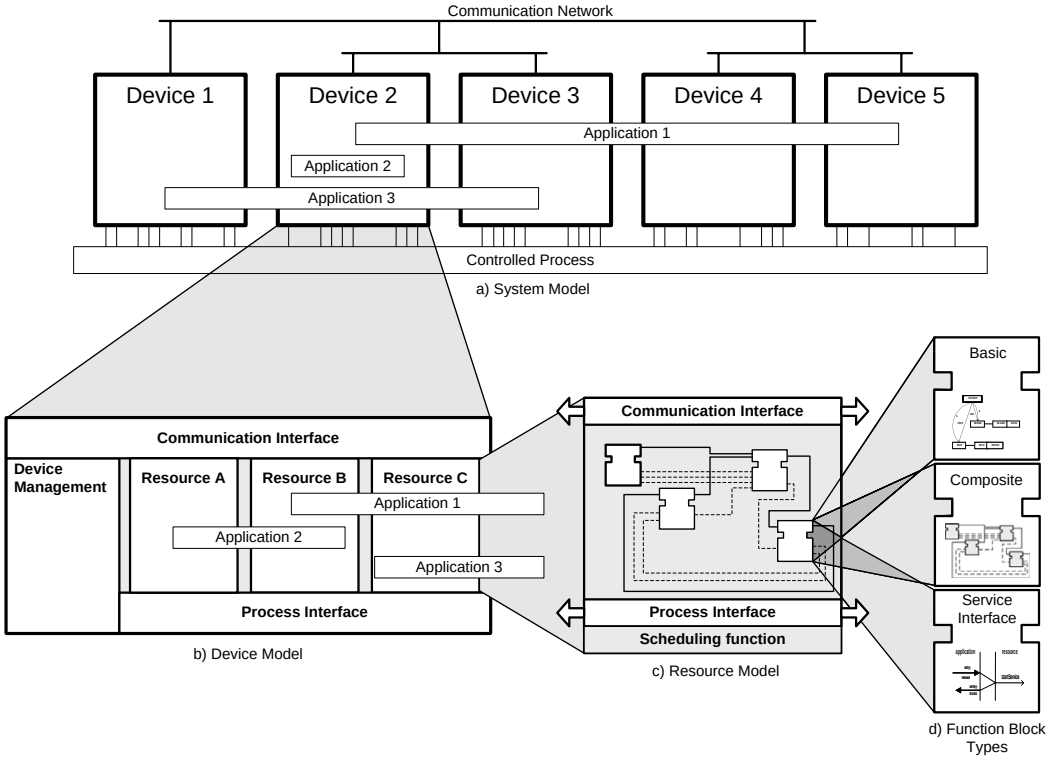


Figure 2.6: Distribution Model of IEC 61499 [Zoi07]

3 Concept

To formulate a concept for an embedded execution environment for modular firmware structures, it is essential to examine the life-cycle of an IPMCS and its components, but its of equal importance to consider the boundary conditions. The firmware design process normally has to heed the problems of low memory space for the firmware and low execution power of the device. Therefore the main task of a firmware designer is to overcome those two fundamental and often contrary problems and so the concept must take this problems into account.
A use case analysis and a requirements analysis will be performed based on a sample application, which will be defined in the next chapter. In the last section of this chapter a conceptual design will be developed based on the findings of the use case analysis.

3.1 Sample Application

The sample application is a simplified version of a temperature control for a single room which could be used in a similar (but of course more advanced) design in a building automation system. It consists of a heater, a temperature sensor, a HMI, and a heater control. The different parts of the temperature control are connected via some kind of network, field bus or wireless connection as shown in Figure 3.1. The correct spreading of the different system parts is not important (e.g., the heater control could also be a part of the sensor device), it is only important that the sample system is distributed. Figure 3.1 shows the physical mapping of the different FBs to the different devices in the distributed system.

The program of the application, represented as an IEC 61499 FB network (FBN), is shown in Figure 3.2. The FBN consists of several different FBs, where some represent physical devices and others don't have a physical representation. The `START`-FB of type `E_RESTART` and the `HYST`-FB of type `HYST_MW_LOW_ON` are such FBs without concrete physical representations. They only provide control functionality to the system.

The purpose of the `START`-FB is to start the FBN at start-up of the system

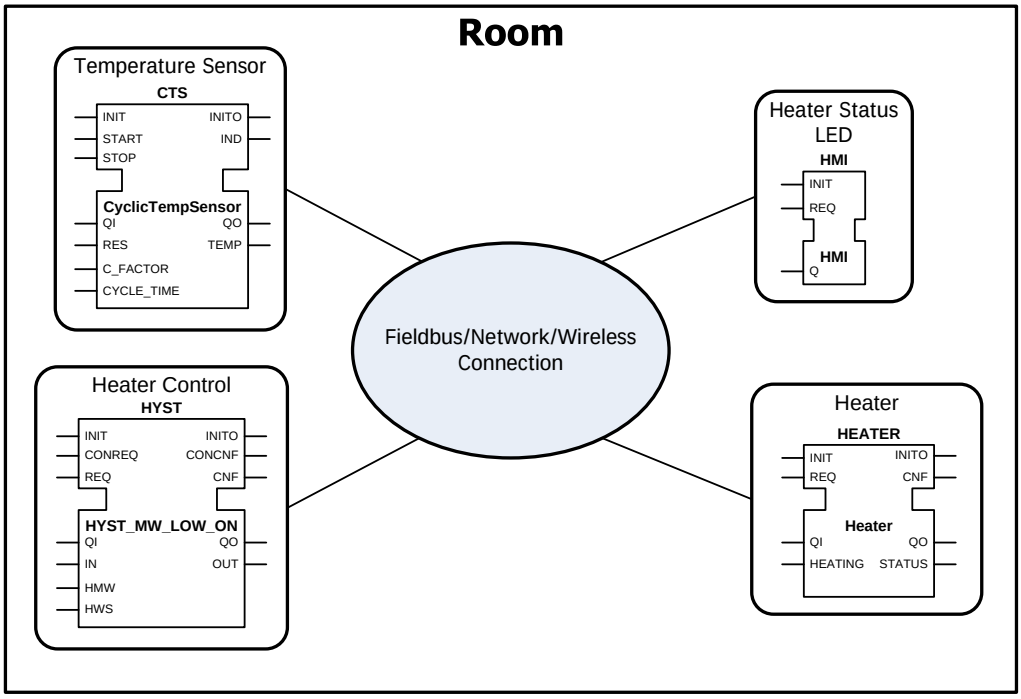


Figure 3.1: Sample Application — Temperature Control Devices

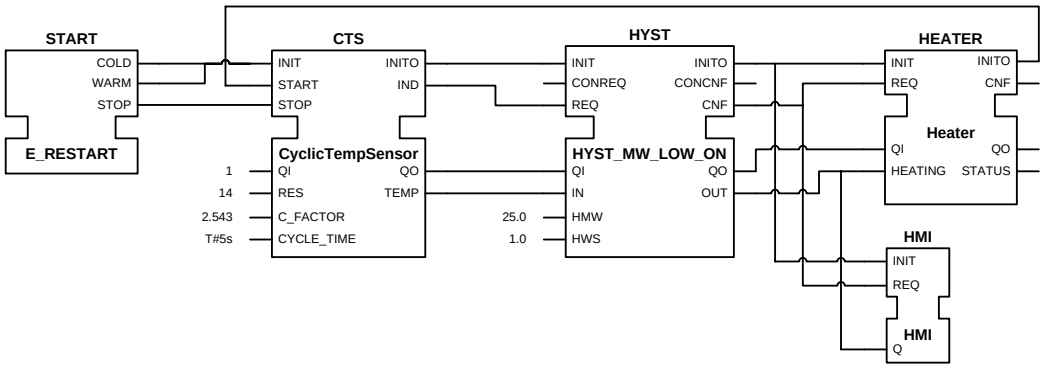


Figure 3.2: Sample Application — Temperature Controller

and to stop the FBN on the shut down of the system. The `HYST` implements a symmetrical hysteresis function where the data output `OUT` is high when the data input `IN` is below the lower switching threshold and vice versa. The threshold levels can be adjusted by the parameters `HMV` (hysteresis mean value) and `HWS` (hysteresis window size), where the `HMS` is the midpoint of the sym-

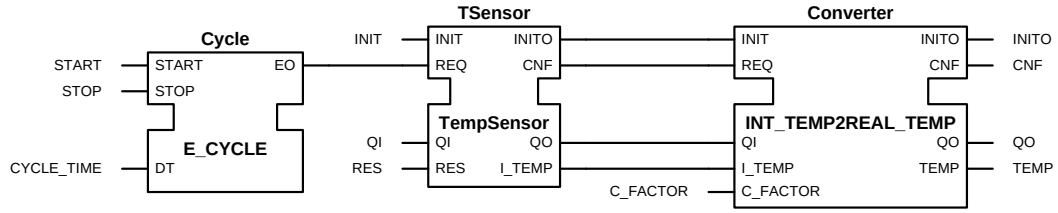


Figure 3.3: Composite FB — CyclicTempSensor

metrical hysteresis function and the `HWS` gives the distance from the `HMV` to the upper and lower switching threshold.

The FBs `CTS` of type `CyclicTempSensor`, `HMI` of type `HMI` and `HEATER` of type `Heater` represent physical entities. The `HMI`-FB represents a LED and via the boolean data input `Q` the LED can be switched on (`Q` is high) or off (`Q` is low). The heater device is controlled and represented by the `HEATER`-FB. Similar to the `HMI`-FB, the `HEATING` data input controls if the heater device is heating (`HEATING` is high) or not (`HEATING` is low), but additionally the `HEATER`-FB can submit its status via the `STATUS` data output.

The last FB `CTS` is a composite FB and the inner FBN is shown in Figure 3.3. This FB is special because it implements the minimal necessary firmware of the temperature sensor. The firmware can be parametrized through the data inputs `RES`, `C_FACTOR` and `CYCLE_TIME`, where `RES` gives the resolution of the temperature sensor in bits, `C_FACTOR` gives the conversion factor from the binary temperature value to the temperature value in degree Celsius and `CYLCE_TIME` defines the interval between two temperature measurements. In this application the `CYCLE_TIME` also defines the frequency of the control loop.

The firmware FB of the temperature sensor consists of three FBs as shown in Figure 3.3. The FB `Cycle` of type `E_CYCLE` generates the event that initiates the temperature measurement after the preset time interval. The hardware of the temperature sensor is accessed by the FB `TSensor` of type `TempSensor`. The conversion from the integer representation of the temperature value to a representation in degree Celsius is done by the FB `Conversion` of type `INT_TEMP2REAL_TEMP`.
The communication between the FB is handled by communication-FBs which are not shown in Figure 3.2, as they would only complicate the application and no additional insight could be gained.

3.2 Use Case Analysis

The purpose of the following use cases is to determine what capabilities the execution environment has to provide for convenient use and implementation of a device firmware. As written above the `CTS`-FB is a typical firmware FB and so most of the use cases revolve around this FB. For sake of simplicity the communication FBs have been neglected, but as the application is distributed there have to be means that allow FBs to communicate with each other. For simplicity in this application it is assumed that the communication partners can be addressed through IDs. The temperature sensor identifies itself with its sensor-ID and the rest of the application is identified by a temperature control-ID. The use cases cover most of the life-cycle of a typical automation device from the programming phase, via the start-up to the end of the products life. The use cases are described in the suggested form of [Öst01]. Each use case consists of a name for identification, a brief description, a list of actors who have an active role in the use case, a trigger for the use case, preconditions that have to be met, incoming informations necessary for the use case, the result of the actions taken in the use case and postcondition that is guaranteed after the steps are executed. At last the workflow sequence is described in the form of an enumeration.

3.2.1 Firmware and Application Development Phase Use Cases

To ensure convenient means for firmware and application programming, it is essential to analyze which IEC 61499 language elements (e.g., Basic FBs, SIFBs, Composite FBs) must be provided. The less elements has to be supported the smaller the execution environment can be, but software maintainability decreases and complexity increases.

Use Case: Programming the Firmware for the Temperature Sensor

Name:	Programming the Firmware for the Temperature Sensor
Brief description:	A firmware for the temperature sensor shall be programmed
Actor:	Firmware programmer, Product development department
Trigger:	New temperature sensor shall be developed
Preconditions:	Target hardware chosen, a single firmware FB shall represent the device, execution environment is already ported to target hardware
Incoming informations:	Target hardware, temperature sensor resolution, conversion factor
Result:	Firmware is programmed for the new device
Postcondition:	Temperature sensor firmware is working properly
Workflow:	1. Product development department issues task to code new temperature sensor firmware to firmware programmer 2. Firmware programmer codes hardware independent FB `INT_TEMP2REAL_TEMP` as seen in Figure 3.3 3. Firmware programmer codes hardware specific FB `TempSensor` as seen in Figure 3.3 4. Firmware programmer encapsulates the FBN in a composite FB as seen in Figure 3.2

The precondition that a single FB shall represent the device defines the necessity to support composite FBs. To be compliant with the IEC 61499 standard, hardware has to be handled with SIFB, therefore the execution environment must support them too. For software maintenance and for a FB to be easy to use, its always preferable to represent physical value in a native representation, but as sensors can only deliver analog or digital values it is necessary to convert this values into their native representation. Therefore in this case it is necessary to convert the temperature value from the delivered

integer representation to a representation in degree Celsius. For that kind of task, the standard provides basic FBs and therefore the execution environment has to support them.

Use Case: Port Existing Temperature Sensor Firmware to a New Target Hardware

Name:	Port Exisitng Temperature Sensor Firmware to a New Target Hardware
Brief description:	The firmware of the temperature sensor shall be ported to a new target hardware
Actor:	Firmware programmer, Product Development
Trigger:	New temperature sensor hardware shall be used
Preconditions:	Target hardware chosen, `E_CYCLE`-FB and `INT_TEMP2REAL_TEMP`-FB available, `TempSensor`-FB not implemented for new hardware, execution environment is already ported to target hardware
Incoming informations:	Target hardware, temperature sensor resolution, conversion factor
Result:	Firmware is ported to new hardware
Postcondition:	Temperature sensor firmware is working properly
Workflow:	1. Product Development issues task to port firmware to new target hardware 2. Firmware programmer codes hardware specific FB `TempSensor` as seen in Figure 3.3 3. Firmware programmer builds the FBN as a Composite FB as seen in Figure 3.3

This use cases confirms that the taken approach of firmware modularization is appropriate for a fast and easy port of the firmware to a new hardware platform.

Use Case: Program Temperature Control Application with the Temperature Sensor FB

Name:	Use Case: Program Temperature Control Application with the Temperature Sensor FB
Brief description:	The temperature sensor FB shall be used in a temperature control application
Actor:	Application programmer, Product Development
Trigger:	Temperature sensor shall be used in a temperature control application
Preconditions:	Target hardware chosen, all necessary FB available
Incoming informations:	Target hardware, temperature sensor resolution, conversion factor
Result:	A temperature control application is created
Postcondition:	The application is working properly
Workflow:	1. Product Development issues task to develop temperature control application with the temperature sensor 2. Application programmer uses existing FBs to create a FBN that implements the function of a temperature control (see Figure 3.2)

This use case confirms that the taken approach of encapsulation simplifies and accelerates the process of application design, as the application programmer must not deal with the inner workings of the temperature sensor-FB.

3.2.2 Operational Phase Use Cases

The operational phase use cases deal with the typical day-to-day work of the maintenance personal. As maintainability is a great issue, both in time and costs, the firmware execution environment must provide capabilities to simplify the necessary maintenance tasks. Therefore it is of great importance to examine this tasks carefully.

Use Case: Install Temperature Sensor

Name:	Install temperature sensor
Brief description:	Company technician affixes and configures the sensor
Actor:	Property management, Company technician
Trigger:	Property management gives order to install sensor
Preconditions:	None
Incoming informations:	Sensor location, sensor-ID, temperature control-ID
Result:	Sensor is placed and configured
Postcondition:	Sensor is on line
Workflow:	1. Property management issues company technician order to install sensor. 2. Property management imparts location, sensor-ID and temperature control-ID 3. Company technician configures sensor-ID and temperature control-ID of the sensor 4. Company technician affixes the sensor on the given location 5. Company technician brings the sensor on line

The first use case defines the necessary steps to install a temperature sensor in a room. The company technician configures the temperature sensor with the given configuration parameters provided by the property management. After that the company technician mounts the temperature sensor at the designated location and starts the temperature sensor. This shows the company technician has to write the parameters to the device's data input and then start the execution of the device. Therefore this use case identifies the needs for a `WRITE PARAMETER` and a `START DEVICE` command.

Use Case: Remove Temperature Sensor

Name:	Remove temperature sensor
Brief description:	The temperature sensor will be deactivated and removed
Actor:	Company technician, Property management
Trigger:	Property management gives order to remove temperature sensor
Preconditions:	Temperature sensor with given sensor-ID is installed at given location
Incoming informations:	Sensor-ID and sensor location
Result:	Temperature Sensor is deactivated and removed
Postcondition:	Sensor is off line
Workflow:	1. Property management issues company technician order to remove sensor 2. Property management imparts location, sensor-ID and temperature control-ID 3. Company technician checks sensor-ID an temperature control-ID of the sensor at the given location 4. Company technician stops temperature sensor 5. Company technician removes temperature sensor

The analysis of this use case, where a temperature sensor gets deactivated and unmounted, identifies the need for a `STOP DEVICE` command, so that an executing device can be shut down in a controlled manner.

Use Case: Reconfigure the Sensor-ID

Name:	Reconfigure the sensor-ID of a temperature sensor
Brief description:	Company technician reconfigures sensor-ID of a temperature sensor
Actor:	Property management, company technician
Trigger:	Property management gives company technician order to reconfigure sensor-ID
Preconditions:	Sensor with old sensor-ID is installed on given location and configured with given temperature control-ID
Incoming informations:	New sensor-ID, old sensor-ID, temperature control-ID and sensor location
Result:	Temperature sensor is configured with new sensor-ID
Postcondition:	Sensor continues to work properly with new sensor-ID
Workflow:	1. Property management issues company technician order to reconfigure sensor-ID 2. Property management imparts location, old sensor-ID, new sensor-ID, temperature control-ID, and sensor location 3. Company technician checks sensor-ID and temperature control-ID of the temperature sensor at the given location 4. Company technician stops device execution 5. Company technician reconfigures temperature sensor with new sensor-ID 6. Company technician restarts device execution

To perform the workfow of this use case, the following commands must be supported by the firmware execution environment: `STOP DEVICE`, `READ PARAMETER`, `WRITE PARAMETER` and `RESTART DEVICE`.

Use Case: Reconfigure Temperature Control-ID

Name:	Reconfigure temperature control-ID of the temperature sensor
Brief description:	Company technician reconfigures temperature control-ID of a temperature sensor
Actor:	Property management, company technician
Trigger:	Property management gives company technician order to reconfigure temperature control-ID
Preconditions:	Sensor with old temperature control-ID is installed on given location and configured with given sensor-ID
Incoming informations:	New temperature control-ID, old temperature control-ID, sensor-ID, and sensor location
Result:	Temperature sensor is configured with new temperature control-ID
Postcondition:	Sensor continues to work properly with new temperature control-ID
Workflow:	1. Property management issues company technician order to reconfigure temperature control-ID 2. Property management imparts location, sensor-ID, new temperature control-ID, old temperature control-ID, and sensor location 3. Company technician checks sensor-ID and temperature control-ID of the temperature sensor at the given location 4. Company technician stops device execution 5. Company technician reconfigures temperature sensor with new temperature control-ID 6. Company technician restarts device execution

This use case is analog to the use case above. Therefore the same commands `STOP DEVICE`, `READ PARAMETER`, `WRITE PARAMETER` and `RESTART DEVICE` are needed to execute the workflow of this use case.

Use Case: Replace Defective Temperature Sensor

Name:	Replace defective temperature sensor
Brief description:	The building automation system (BAS) reports a defective temperature sensor that must be replaced
Actor:	BAS, company technician
Trigger:	BAS reports defective temperature sensor
Preconditions:	Configuration and error status can be read-out
Incoming informations:	Sensor-ID, sensor location
Result:	Defect temperature sensor is replaced with working and correct configured temperature sensor
Postcondition:	New temperature sensor is working properly
Workflow:	1. BAS reports defective temperature sensor with sensor-ID and sensor location 2. Company technician queries error condition 3. Company technician queries configuration of the defective temperature sensor 4. Company technician configures new temperature sensor with queried configuration 5. Company technician stops defective temperature sensor 6. Company technician replaces defective temperature sensor with the new temperature sensor 7. Company technician starts new temperature sensor

This maintenance use case identifies the need for a `STOP DEVICE`, `READ PARAMETER`, `WRITE PARAMETER`, `READ DATA OUTPUT` and `RESTART DEVICE` com-

mands.

Use Case: Read Parameter List

Name:	Read parameter list
Brief description:	Company technician wants to query parameter names and types to configure a temperature sensor
Actor:	Company technician, temperature sensor
Trigger:	Company technician needs to configure a temperature sensor
Preconditions:	No manual of the temperature sensor available
Incoming informations:	None
Result:	Company technician knows parameter names and types
Postcondition:	None
Workflow:	1. Company technician queries parameter names and types of a temperature sensor 2. Temperature sensors returns parameter names and types

As the goal of this use case is to determine the names and types of the the available data inputs, it identifies two new commands: `QUERY DATA INPUT NAMES` and `QUERY DATA INPUT TYPE`. In contrast to the `READ PARAMETER` command, which returns the value of a single data input, those commands return the data input name (e.g., `RES` or `CYCLE_TIME` from the sample application) or the data input type of a single data input (e.g., `CYCLE_TIME` is of type `TIME`).

Use Case: Inclusion of a new Temperature Sensor into a Building Automation System

Name:	Inclusion of a new sensor into the BAS
Brief description:	A new sensor is integrated into the BAS
Actor:	Temperature Sensor, BAS, Company technician
Trigger:	New temperature sensor is installed
Preconditions:	The new temperature sensor is compatible with the BAS
Incoming informations:	None
Result:	The new temperature sensor is known to the BAS and can be configured by the BAS
Postcondition:	Temperature sensor is working properly in the BAS
Workflow:	1. New temperature sensor is installed by a company technician 2. The company technician declares a new temperature control application in the BAS and enters sensor-ID and temperature control-ID 3. The BAS queries the devices by the given IDs for parameter input names, parameter input types, data output names, data output types and working condition 4. The BAS generates a user interface for the application 5. The sensor is integrated into the BAS

For practical inclusion of the sample application into a building automation system (BAS), this use case identifies following commands: `QUERY DATA INPUT NAMES`, `QUERY DATA INPUT TYPES`, `QUERY DATA OUTPUT NAMES`, `QUERY DATA OUTPUT TYPES`, `QUERY CONDITION`.

3.3 Requirements Analysis

Now that the use case analysis is finished, it is time to define the requirements of the firmware execution environment. The first requirements have been identified through the operational phase use cases of Chapter 3.2.2. All those use cases have in common, that they identify the different management commands which the execution environment must support for a convenient use of the devices in the day-to-day tasks of startup and maintenance of systems and devices. The needed management commands are

- `START DEVICE`
- `RESTART DEVICE`
- `STOP DEVICE`
- `READ PARAMETER`
- `WRITE PARAMETER`
- `READ DATA OUTPUT`
- `QUERY CONDITION`
- `QUERY DATA INPUT NAMES`
- `QUERY DATA INPUT TYPES`
- `QUERY DATA OUTPUT NAMES`
- `QUERY DATA OUTPUT TYPES`

The programming phase use cases of Chapter 3.2.1 identified the required language constructs to enable firmware modularization, code reuse and encapsulation and confirmed the accuracy of demand for those capabilities. The needed language constructs of IEC 61499 are

- Basic FB
- Composite FB
- SIFB

The last set of requirements can be generated by the boundary conditions of firmware development. As mentioned in Chapter 1.1 memory size for firmware tends to be smalls and the computing power of smart actuators and smart sensors is low, both a consequence of the economic need for low device prices. Therefore the firmware execution environment must be optimized for embedded systems. Furthermore the embedded execution environment shall be compatible with IEC 61499 compliant devices and systems. At last the execution environment itself shall be easy to port and therefore the basic functions must be hardware independent, as hardware life-cycles are shortening and software often have to outlive several hardware revisions. Therewith the final requirements are

- The execution environment shall be optimized for code size and low computing power (embedded systems)
- Compatibility to IEC 61499 compliant devices and systems
- The basic functions of execution environment must be hardware independent
- Hardware specific functions of the execution environment must be easy to replace

3.4 Requirements Matching with IEC 61499

The required management commands for the convenient use of the firmware devices have been identified in Chapter 3.3. However IEC 61499 defines its own set of management commands, and as IEC 61499 shall be the target platform for the firmware execution environment the required and the defined management commands must be matched. The standard also defines so called "Device configurability classes" in IEC 61499-4 Annex B[IEC05b]. Devices are divided into one of three configurability classes, namely:

- **Class 0:** Simple devices
- **Class 1:** Simple programmable devices
- **Class 2:** User-reprogrammable devices

Table 3.1 gives an overview which configurability class has to support which management commands, where a dot means that the corresponding command

CMD	Object	Required	Class 0	Class 1	Class 2
CREATE	type_declaration				•
	fb_type_declaration				•
	fb_instance_definition			•	•
	connection_definition	•	•	•	•
	access_path_declaration			•	•
DELETE	data_type_name				•
	fb_type_name				•
	fb_instance_reference			•	•
	connection_definition			•	•
	access_path_name			•	•
START	fb_instance_reference	•	•	•	•
	application_name	•	•	•	•
STOP	fb_instance_reference	•	•	•	•
	application_name	•	•	•	•
KILL	fb_instance_reference			•	•
QUERY	all_data_types		•	•	•
	all_fb_types		•	•	•
	data_type_name				•
	fb_type_name				•
	fb_instance_reference			•	•
	connection_start_point			•	•
	application_name			•	•
	access_path_name	•			•
READ	access_path_name	•	•	•	•
WRITE	access_path_data	•	•	•	•

Table 3.1: IEC 61499 — Device Configuarbility classes based on [IEC05b]

must be supported. The column "Required" indicates the management commands have been identified by the requirements analysis. Further information on configurability classes can be found in [IEC05b]. The most of the required management commands are also needed for the Class 0 device with only a minor deviation in the "Query" section, as the use cases of Chapter 3.2.2 identified the need for easy parameter access, which is achieved easiest through a query of the so called access paths. An access path defines the access to a FB input or output through a defined name, which read or modified through the `READ` and `WRITE` management commands. According to the standard it is also possible to set a parameter by the `CREATE` command, and as the execution environment must be compliant to the standard, that command is also included.

3.5 Conceptual Design

Based on the findings of Chapter 3.3 a first conceptual design shall be developed now. To simplify the design process the execution environment will be split in several parts where each of them shall have a specific task. This will lead to a modular design, which is also desirable, as it will enable to separate the hardware specific parts from the hardware independent functions of the execution environment. As shown in Figure 3.4 the modules of the execution environment are

- Function Block Management Layer
- Application Execution Layer
- Hardware Abstraction Layer
- Device Specific Hardware Layer

which will be examined in the following chapters.

3.5.1 Function Block Management Layer

The Function Block Management Layer (FBML) is the controlling instance of the Application Execution Layer (AEL). Through the management commands, identified in Chapter 3.3 it is possible to control the execution state of the AEL (start, restart and stop). Furthermore the management commands can read the data inputs and data outputs, write the data inputs, query data names

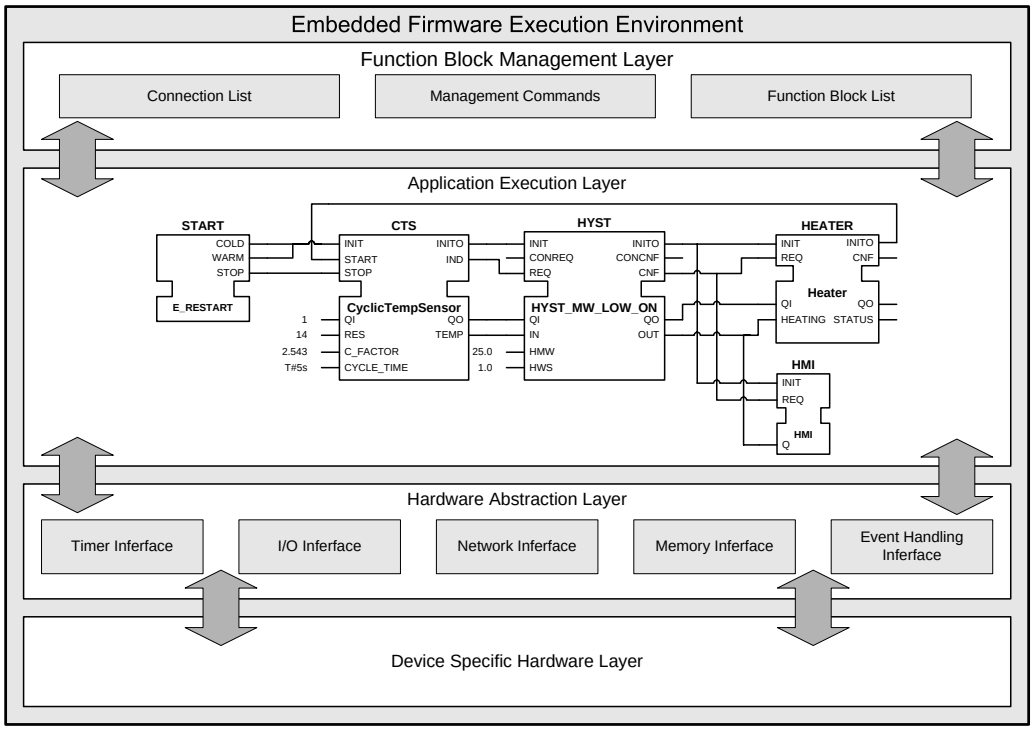


Figure 3.4: Execution Environment Conceptual Design

and types, and query the condition of every FB of the application in the AEL. To provide this functionality the FBML consists of the three elements

- Management Commands
- Function Block List
- Connection List

As any kind of application can be run in the AEL the FMBL must keep a list of all instanced FBs of the application to access their informations. Through that the FBML can identify and access individual FBs and can also query all FBs in the list for specific properties. The Connection List keeps track of the connections between the FBs, whereby it is possible to determine which data inputs are parameter inputs, as they will not be connected to another FB. The advantage of keeping a list of connections instead of keeping a list of parameters is, that thereby possible connection errors (and thereby application errors) can be detected by analyzing the connection list. The only penalty of this design is, that an automated user interface generator of a BAS has to query the lists of connections and data input names and then to match them to determine the parameter inputs, instead of just querying the list of parameters.

3.5.2 Application Execution Layer

The Application Execution Layer (AEL) has to emulate an IEC 61499 environment. It must be able to execute all kinds of FBs in the right manner. Basic FBs are controlled by their execution control charts (EECs), Composite FBs have to relay their event and data inputs to their inner FBN and to pass the results of their inner FBN to their data and event outputs. SIFBs don't follow such rules as their implementation is not defined by IEC 61499[1] and therefore can implement any kind of program sequence. The execution sequence of all FBs must also be according to IEC 61499. If an event arrives at an FB only the associated data inputs are read and the corresponding algorithm is executed. Data outputs are only written if the corresponding output event is sent. The AEL must also ensure that the data types defined in IEC 61499[2] behave hardware independent and according to the standard.

[1] As shown in Chapter 2.2.1 their execution behavior is only defined by time sequence diagrams

[2] Part of the data types are taken from IEC 61131-3

3.5.3 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) offers hardware services to the AEL, such as I/O interfaces, timers, memory management, event handling and network interfaces as shown in Figure 3.4. The HAL separates the specific implementation of the hardware services from the interface, wherefore the specific implementation can be replaced without changing the interface or FBs of the AEL. Therewith hardware independence of the upper layers can be achieved and the execution environment can be ported to a new target hardware by replacing the Device Specific Hardware Layer.

3.5.4 Device Specific Hardware Layer

As the HAL offers the abstract hardware services to the AEL, the Device Specific Hardware Layer (DSHL) offers its concrete hardware services to the abstract interface of the HAL. Therefore the DSHL implements the services offered by the HAL for a concrete target hardware and for every target hardware a new DSHL has to be programmed. As mentioned above, this two layered design allows to implement all high level function (FBML and AEL functions) hardware independent and enables to easily port the firmware execution environment to a new target hardware or a new revision of an embedded system (e.g. new external I/O interface chip).

3.6 Summary

To generate the requirements for an embedded execution environment for modular firmware structures, first a typical sample application has been defined. Based on this sample application a use case analysis has been performed in Chapter 3.2, and thereafter a requirements analysis. The analysis produced the necessary IEC 61499 language constructs and a set of generic management commands, that have to be supported for convenient use of such firmware devices in the day-to-day business. But as the execution environment must be compliant to IEC 61499 the generic management commands have been matched to the commands defined in the standard. At last a conceptual design has been produced based on the findings of the requirements analysis of Chapter 3.3.

4 Implementation

Based on the findings of Chapter 3 the already existing IEC 61499 runtime environment FORTE shall be modified to fit the needs of an embedded firmware execution environment and design flaws shall be identified and corrected. Therefore the first step is to analyze FORTE and give an overview of FORTE and its key components. Thereafter the parts of FORTE that have been identified for change are explained in detail in their original implementation. After the explanation of a FORTE part the revised implementation is shown, and its advantages are illustrated.

4.1 Overview

The 4DIAC runtime environment (FORTE) is an execution environment for IEC 61499 FBNs and is implemented in C++. It consists of three parts (see Figure 4.1)

- Core
- Architecture
- FB Library

The "Core" implements the hardware independent functions necessary to emulate a native IEC 61499 environment compliant to the standard. The "Core" enfolds the IEC 61499 data types, FB event handling, function block management, function block interface specifications, device and resource handling, and connection handling for the FBs.

The "Architecture" is split in two parts. The first part specifies the hardware service interfaces in a hardware independent way. The second part implements the functionality of those interfaces for each target hardware FORTE has been ported to.

The "FB Library" compasses the implementations of specific FBs. Those FBs must inherit an adequate interface specification for its type from the "Core". The library contains standard implementations of often needed FBs,

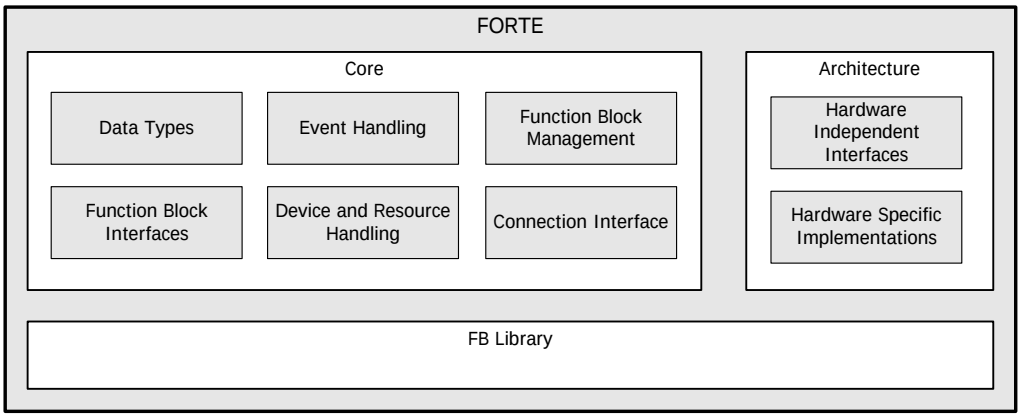


Figure 4.1: FORTE Overview

but the user of FORTE can also implement additional FBs and use them in the execution environment.

The "Core" and "Architecture" components will be explained in a bit more detail in the next chapters, as they are important for the functionality of the runtime environment.

4.1.1 Core

As shown in Figure 4.1 the core consists of the packages

- Data Types
- Event Handling
- Function Block Management
- Function Block Interfaces
- Device and Resource Handling
- Connection Interface

The "Data Types" package defines and implements the data types defined in IEC 61499. Those data types are used for implementing the interface of FBs, internal variables of the FBs, and to define the data types of the connection endpoints. Further details on the "Data Types"-package will be given in Chapter 4.3.

As IEC 61499 is a event-based execution system the rules for event handling have to be modeled in the execution environment. The "Event Handling" package implements the event eradication rules of IEC 61499. It ensures that the events are processed in the order of arrival.

The "Function Block Management" package is responsible for the creation and deletion of FBs and connections, starting and stopping FBs and IEC 61499 applications, reading and writing of data inputs and data outputs, queries for information, and the strict compliance of the operational state machine of IEC 61499 shown in Chapter 2.2.2.

The C++ interface for the different kinds of FBs are defined in the "Function Block Interface" package. It provides the base classes from which the concrete FBs in the FB library must inherit to be used in FORTE. The base classes provide the basic functionality of its FB kind, such as event processing and FB interface definitions.

In IEC 61499 each FBN is nested within a resource or device, and resources are nested in devices according to the distribution model of IEC 61499 (see Chapter 2.2.3). The "Device and Resource Handling" package provides the C++ constructs to implement the behavior and the interface of the IEC 61499 devices and resources.

The last package of the "Core"-component is the "Connections" package. It implements the event and data connections between the FBs and provides services to the "Function Block Management" package for connection and disconnection of those connections.

4.1.2 Architecture

The "Architecture" component consists of a hardware independent hardware service interface specification for the timer, serial interface, and network interface. For the sake of completeness there should be an interface specification for threads and synchronizing objects, but such base classes have been neglected for performance issues.

To implement the functions for a specific hardware, the interface of the given hardware service interface (e.g., the timer) is inherited from the given base class (e.g., the timer base class). Therefore the implementation of a hardware service is separated from its interface and can be reused for every new target hardware. Another benefit of this design is, that FORTE can be easily ported to a new target hardware, by implementing a new set of the interface classes for the new hardware.

4.2 Optimization Guidelines and Targets

Optimization is a process to enhance a system or process in a defined sense. Therefore it is necessary to define a optimization criterion, which indicates if the steps done in the optimization process contribute to the optimization goal. There are several kinds of optimization goals, like execution speed optimization, code readability optimization, code maintainability optimization, compile time optimization, and several more. The goal of this optimization is to reduce the code size of FORTE, so that it becomes a possible solution for firmware execution, as firmware code space tends to be small. FORTE is implemented in C++, which the optimization guidelines must take in account.

The code size reduction an be achieved through several programming and design techniques[1]. One of them is to eliminate as much virtual functions as possible, as each virtual function needs additional code space for the so called virtual function table. The table gives the information which specific function must be executed for the calling class. The execution speed is also improved through the elimination of virtual functions as a virtual function call usually takes three times more time as a simple function call.

The second technique is to design flat class hierarchies, as each level of inheritance adds the constructors and (commonly virtual) destructors of the base classes. Often the purpose of such class hierarchies is to further reuse and maintainability by separating the interface from its implementation, but at the cost of code size (and usually execution speed). If its not possible to flatten the class hierarchy without seriously impeding maintainability and reuse, this technique should be avoided.

A commonly used method for code size reduction, is to re-implement the used container classes (e.g., `list`, `map`, `string`), instead of using the provided pre-implemented ones from a library, such as the Standard Template Library (STL) of C++. The STL elements are general-purpose implementations with no special demand in mind, therefore those implementations are not optimized in any special way[Str99]. The advantage of using STL elements is, that they are well tested and proven to work. Own implementations of the container classes take the demand for small code size in account.

With regard of this techniques, the analysis of FORTE showed that the "Data Types" and "Function Block Management" packages have to be optimized to achieve the optimization goal. The STL container classes `string`, `list`, `map`, and `set`, which are used in FORTE, shall be re-implemented or, if possible

[1] The following points are not a complete collection of such techniques, but the most commonly used in the optimization of FORTE

and applicable, removed.

4.3 Data Types

The "Data Types" package is an essential package of FORTE, as the classes of the "Data Types" package are used in the data interface of all FBs and the connections between the FBs. Therefore it is important that the "Data Types" classes are maintainable, easy to understand and use, and efficient. A boundary constraint of the design of the "Data Types" class hierarchy is, that the hierarchy shall reflect the data types hierarchy defined in [IEC03]. Based on the analysis of the "Data Types" package of the next chapter, a revised design will be devised.

4.3.1 Original Design

The original class hierarchy design (shown in Figure 4.2) is heavily oriented on the data type hierarchy of [IEC03]. It contains a fair amount of abstract classes, which are:

- `CIEC_ANY`
- `CIEC_ANY_DERIVED`
- `CIEC_ANY_ELEMENTARY`
- `CIEC_ANY_DATE`
- `CIEC_ANY_STRING`
- `CIEC_ANY_BIT`
- `CIEC_ANY_MAGNITUDE`
- `CIEC_ANY_NUM`
- `CIEC_ANY_REAL`
- `CIEC_ANY_INT`

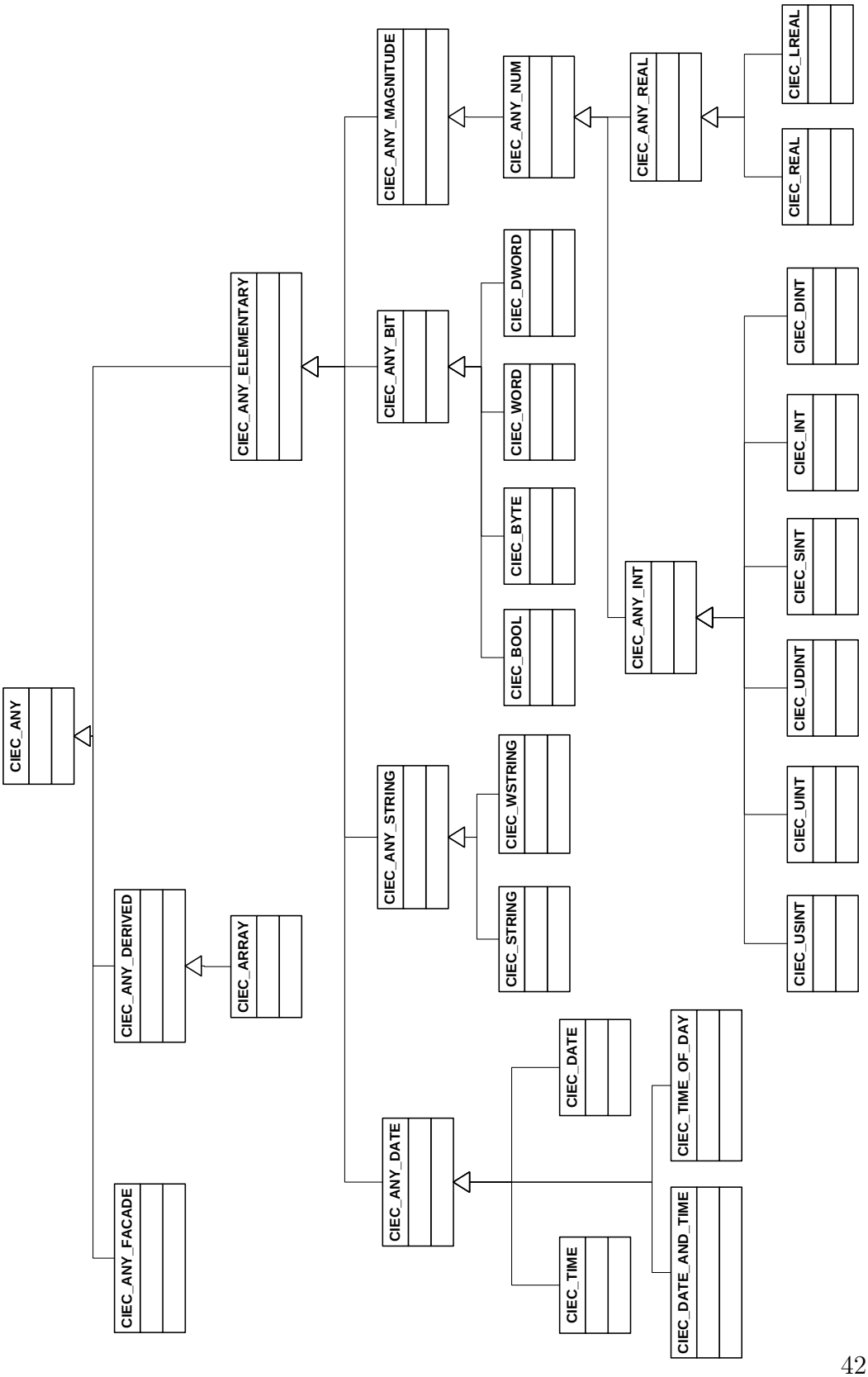


Figure 4.2: FORTE Data Type Class Diagram

The common interface for all data type classes is provided by `CIEC_ANY`. The purpose of the other classes is to divide the concrete data types into categories according to IEC 61131-3. The concrete class `CIEC_ANY_FACADE` implements a data type object that can assume the role of any concrete data type and the class `CIEC_ARRAY` implements an array for all concrete data types. The other concrete classes

- `CIEC_TIME`
- `CIEC_DATE`
- `CIEC_DATE_AND_TIME`
- `CIEC_TIME_OF_DAY`
- `CIEC_STRING`
- `CIEC_WSTRING`
- `CIEC_BOOL`
- `CIEC_BYTE`
- `CIEC_WORD`
- `CIEC_DWORD`
- `CIEC_USINT`
- `CIEC_UINT`
- `CIEC_UDINT`
- `CIEC_SINT`
- `CIEC_INT`
- `CIEC_DINT`
- `CIEC_REAL`
- `CIEC_LREAL`

implement the concrete data types. Each concrete class encapsulates the necessary C++ variable to hold the value (e.g., `CIEC_INT` contains a C++ int), and has to implement the interface the base class `CIEC_ANY` specifies.

The data type interface is specified through following function:

- `virtual void setValue(CIEC_ANY* pa_poValue)`
- `virtual void getValue(CIEC_ANY* pa_poValue)`
- `virtual CIEC_ANY* clone(void)`
- `virtual const EDataTypeID getDataTypeID()`
- `virtual bool fromString(const string &pa_rsValue)`
- `virtual bool toString(string &pa_rsValue)`
- `static void isCastable(EDataTypeID pa_eSource, EDataTypeID pa_eDestination, bool &pa_rbUp, bool &pa_rbDown, bool &pa_rbCross)`
- `bool castFrom(CIEC_ANY* pa_poValue, bool &pa_rbWarnIfOutOfRange)`
- `virtual void* getDataPtr(void)`

Both the functions `void setValue(CIEC_ANY* pa_poValue)` and `void getValue(CIEC_ANY* pa_poValue)` implement the assignment operator with interchanged left-hand side and right-hand side operators. While `void setValue(CIEC_ANY* pa_poValue)` implements the assignment operator as `Destination = Source`, `void getValue(CIEC_ANY* pa_poValue)` implements it as `Source = Destination`. As both functions implement the same functionality and the C++ convention for assignment operators is `Destination = Source`, the `void getValue(CIEC_ANY* pa_poValue)` function can be removed.

The function `CIEC_ANY* clone(void)` is used to create a copy of a data type object and returns a pointer to the copy. This function and by the "Connections" package for the connection creation.

With `const EDataTypeID getDataTypeID()` the type of a data type object can be determined at run-time.

The functions `bool fromString(const string &pa_rsValue)` and `bool toString(string &pa_rsValue)` allows to transform a data object to and from a string. As every data type has other transformation rules the functions need to be virtual.

Through the function `static void isCastable(EDataTypeID pa_eSource, EDataTypeID pa_eDestination, bool &pa_rbUp, bool &pa_rbDown, bool &pa_rbCross)` it is possible to determine if and in which way a source data type object can be cast into a destination data type object. The function `bool`

`castFrom(CIEC_ANY* pa_poValue, bool &pa_rbWarnIfOutOfRange)` implements the cast itself.

The last function `void* getDataPtr(void)` returns a pointer to the contained variable of the data type object holding the value.

The functions

- `virtual int serialize(TBYTE* pa_pcBytes, int pa_nStreamSize)`
- `virtual int serializeTag(TBYTE* pa_pcBytes)`
- `virtual int serializeValue(TBYTE* pa_pcBytes)`
- `virtual int deserialize(const TBYTE* pa_pcBytes, int pa_nStreamSize)`
- `virtual int deserializeTag(TBYTE pa_cByte)`
- `virtual int deserializeValue(const TBYTE* pa_pcBytes, int pa_-nStreamSize)`
- `static void serializeshort(TBYTE* pa_pcBytes, TUINT16 pa_nValue)`
- `static void deserializeshort(const TBYTE *pa_pcBytes, TUINT16 *pa_nValue)`
- `static void serializelong(TBYTE* pa_pcBytes, TUINT32 pa_nValue)`
- `static void deserializelong(const TBYTE* pa_pcBytes, TUINT32 *pa_nValue)`
- `static bool isNull(TBYTE* pa_pcBytes)`
- `static void serializeNull(TBYTE* pa_pcBytes)`

are also part of the `CIEC_ANY` class, but not part of the data type interface. They form a serialization interface which prepares the data to be sent over a communication interface. The data is transformed into a platform independent data representation according to the *IEC 61499 Compliance Profile for Feasibility Demonstrations*, which is specified in [HOL]. Including an interface that is not part of the role of a class is bad design, therefore the communication interface must be removed from the class and be re-implemented in its own class hierarchy.

4.3.2 Revised Design

Based on the analysis of Chapter 4.3.1 the design of the "Data Type" package shall be improved. First the interface of `CIEC_ANY` will be modified, as the communication interface will be removed. The functions `void getValue(CIEC_ANY* pa_poValue)` and `bool castFrom(CIEC_ANY* pa_poValue, bool &pa_rbWarnIfOutOfRange)` will also be removed, as `void getValue(CIEC_ANY* pa_poValue)` implements the same functionality as `void setValue(CIEC_ANY* pa_poValue)`, and `bool castFrom(CIEC_ANY* pa_poValue, bool &pa_rbWarnIfOutOfRange)` shall be implemented by cast-operators, as it is common usage in C++. As the interface of `CIEC_ANY` is now streamlined, the next steps are taken to reduce the number of virtual functions. As `void getValue(CIEC_ANY* pa_poValue)` has been removed the remaining virtual functions are:

- `virtual void setValue(CIEC_ANY* pa_poValue)`
- `virtual CIEC_ANY* clone(void)`
- `virtual const EDataTypeID getDataTypeID()`
- `virtual bool fromString(const string &pa_rsValue)`
- `virtual bool toString(string &pa_rsValue)`
- `virtual void* getDataPtr(void)`

Its not possible to change the implementation of `virtual CIEC_ANY* clone(void)` as every object has to return an object of its own type, and the object calling the function can only be determined at runtime.

The function `virtual const EDataTypeID getDataTypeID()` returns an enumerations value which represents the type of the object, which can also only be determined at runtime. But unlike the function `virtual CIEC_ANY* clone(void)` the operation is always the same, as the function returns the value of the member variable holding the type information. Therefore `virtual const EDataTypeID getDataTypeID()` does not have to be virtual.

Most of the concrete implementations of the function `virtual void setValue(CIEC_ANY* pa_poValue)` are exactly alike as the member variable, which is holding the actual value, of the source data type object is assigned to the member variable of the destination object by the assignment operator. Due to the design, that every concrete data type class holds a member variable for holding the actual value, the function `virtual void setValue(CIEC_ANY*`

pa_poValue) needs to be virtual. Therefore to solve this problem, the design for holding the value must be changed.

The first step is to identify the data types where the virtual void setValue(CIEC_ANY* pa_poValue) can be implemented in the exactly same way. The identified data types are

- CIEC_ANY_BOOL
- CIEC_ANY_BYTE
- CIEC_ANY_WORD
- CIEC_ANY_DWORD
- CIEC_ANY_SINT
- CIEC_ANY_INT
- CIEC_ANY_DINT
- CIEC_ANY_USINT
- CIEC_ANY_UINT
- CIEC_ANY_UDINT
- CIEC_ANY_REAL
- CIEC_ANY_LREAL
- CIEC_ANY_DATE
- CIEC_ANY_TIME
- CIEC_ANY_TIME_AND_DATE
- CIEC_ANY_TIME_OF_DAY

and will be further called "simple data types". The other data type classes need special implementations for the virtual void setValue(CIEC_ANY* pa_poValue) function.

The member variables and the virtual void setValue(CIEC_ANY* pa_poValue) function for simple data types will now be centralized in the CIEC_ANY class. The simple data types are collected in the union structure shown in Figure 4.3. A union can only hold one of the specified types at a time, but

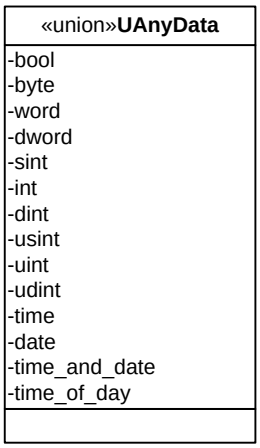


Figure 4.3: FORTE Union UANYData

as each concrete data object only needs to hold one type at a time this poses no problem. A minor disadvantage is, that the `union` always has to reserve memory space for the biggest type in the `union`, but this is only a small price for the gained advantages. Now the `virtual void setValue(CIEC_ANY* pa_poValue)` function can be implemented by an assignment between the source and destination `union`, which means the function has not to be virtual anymore for simple data types.

The other classes

- `CIEC_ARRAY`
- `CIEC_STRING`
- `CIEC_WSTRING`

(called "complex data types") need special treatment, as they further need special implementations for the `virtual void setValue(CIEC_ANY* pa_poValue)` function, and would therefore demand that the function remains virtual, which would be a acceptable design. But even as the main optimization goal is to optimize the code size, the execution speed aspect must be considered. With a small design change it is possible to increase the execution speed for simple data types. Therefore a new virtual function `virtual void setValueComplex(CIEC_ANY* pa_poValue)` is defined, which will handle the assignment of the complex data types. Simultaneously the function `void setValue(CIEC_ANY* pa_poValue)` is declared as non-virtual. The functions now chooses on the basis of the enumeration value, that holds the data type, if the

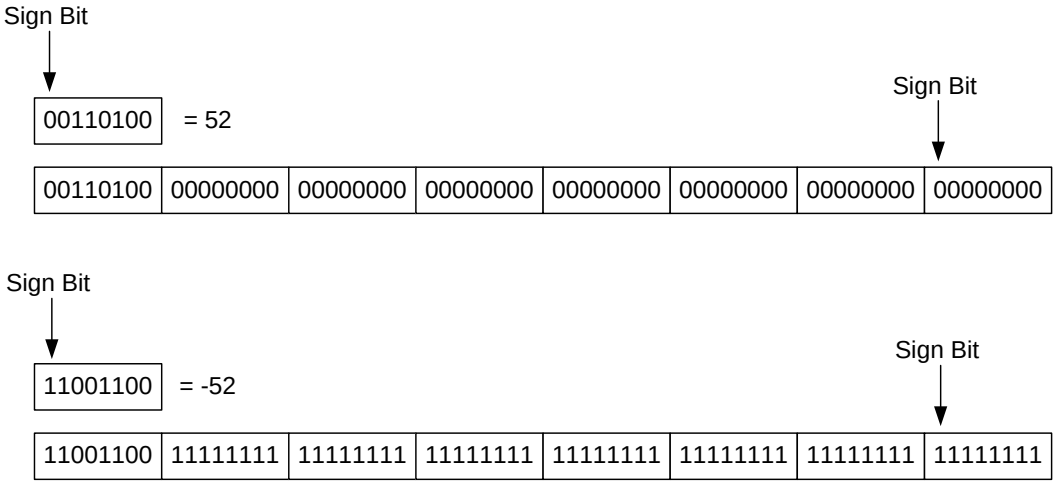


Figure 4.4: Little Endian Binary Data Representation

union assignment operator will be executed or if `virtual void setValue-Complex(CIEC_ANY* pa_poValue)` will be called. As normal function calls are approximately three times faster then virtual function calls, the execution speed for simple data types is improved more, than the execution speed for complex data types is decreased, as it now has to call an additional function before the assignment.

Another look on the `union` structure shows, that it contains three groups of variables: signed, unsigned, and other types. As the signed variables and unsigned variables in each case share the same data representation (two's complement for signed data types, magnitude for unsigned data types). This fact can be used to implement an automatic up-cast in Little Endian systems. If the value is stored in the variable with the greatest number range (e.g., a short int is stored in a double int) the sign bit and the value of the variable nevertheless remains the same, as shown in Figure 4.4. The positive sample value also shows the case of an unsigned value, which also keeps the correct value if stored in the variable with the greatest number range.

But this implementation trick only works in Little Endian systems, because in Little Endian system the used memory space starts with the lowest byte of the data value. Therefore the first byte of the allocated memory space represents a short int or the lowest byte of a double int (see Figure 4.5).

Unfortunately this trick doesn't work in Big Endian system, because the lowest byte of a variable is placed on the last position. Therefore even as a Big Endian short int and a Big Endian double int represent the same value, the representation in memory differs greatly (see Figures 4.6a and 4.6b). This

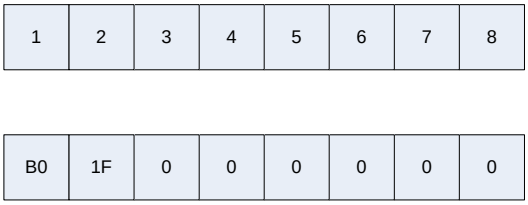
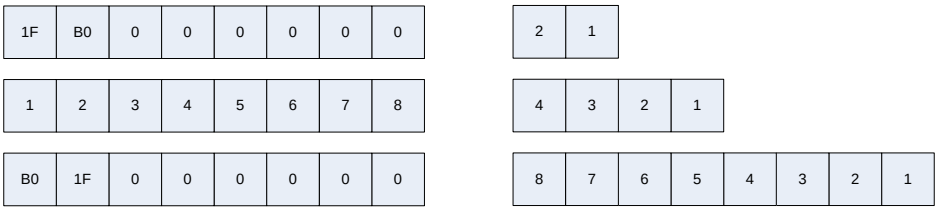


Figure 4.5: Little Endian Data Representation



(a) Big Endian Data Representation

(b) Big Endian – Little Endian Problem

Figure 4.6: Big Endian Problem

problem can be bypassed, by implementing two sets of functions that deal with the inner representation of the `union`, which can be switched by a compiler option, as a system uses either Little Endian or Big Endian data representation.

The functions `virtual bool fromString(const string &pa_rsValue)` and `virtual bool toString(string &pa_rsValue)` can be similarly re-designed to reduce the number of virtual functions in the subclasses.

For further code size reduction the class hierarchy is slightly changed. The class `CIEC_ANY_FACADE` is removed to save code space, as the gain of this class was to reduce the complexity for the handling of a data variable of type `ANY`. The elimination of the class forces some minor modifications of "Connections" package, but has no further impact on the data types.

Furthermore the class `CIEC_WSTRING` now inherits from the class `CIEC_STRING` as the implementation of most functions and operators are exactly alike, but with this design, the functions and operators don't have to be implemented twice, which not only saves code space, but also increases the maintainability of the code. The new class hierarchy design is shown in Figure 4.7.

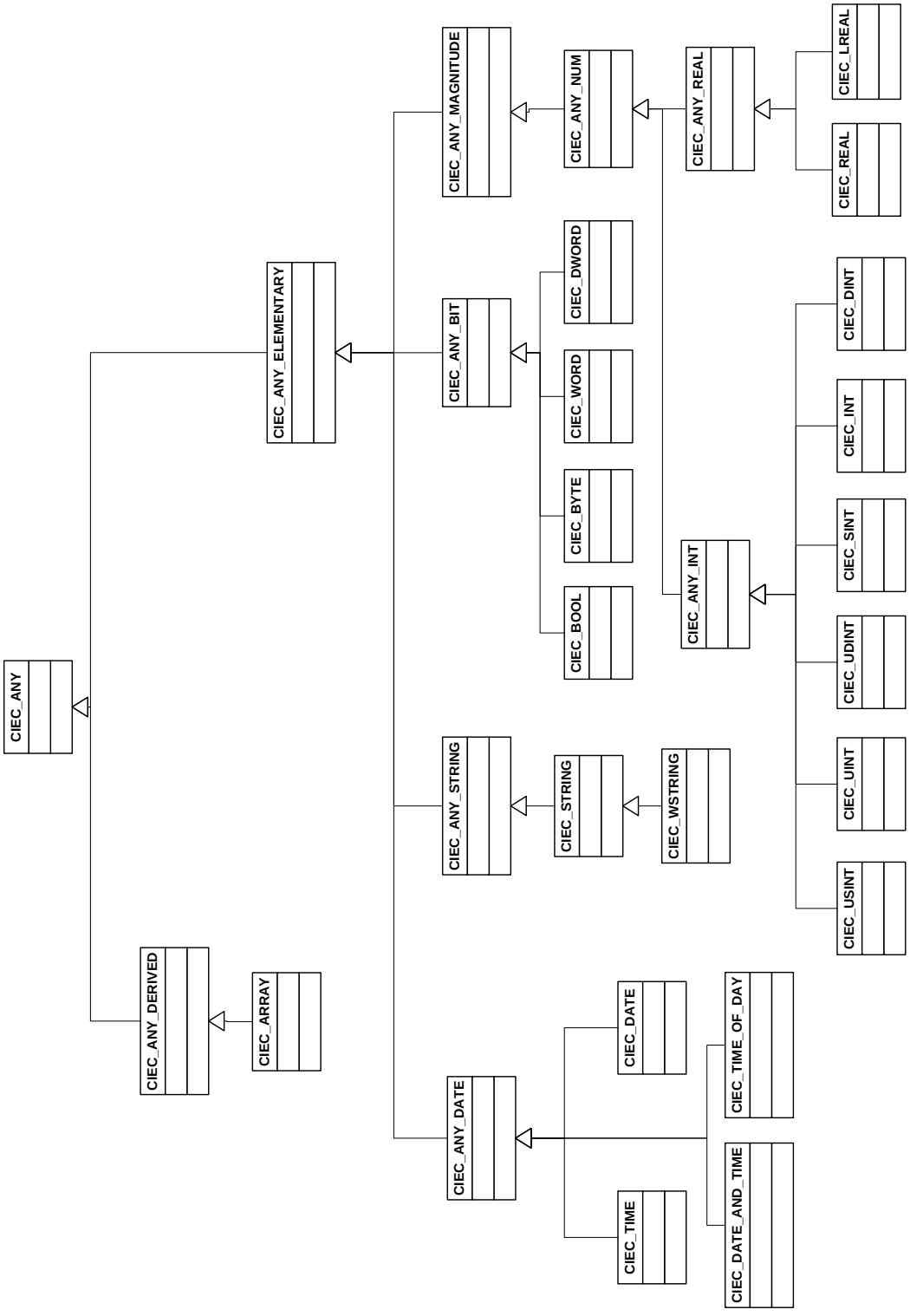


Figure 4.7: Revised FORTE Data Type Class Diagram

4.4 New Package: Serializer

As mentioned in Chapter 4.3.1 the data type base class `CIEC_ANY` contained a communication interface, which was not part of its role. As it is bad design to incorporate an interface which is outside the role of the class, because it decreases maintainability and reuseability, it has been removed from `CIEC_ANY`. But as the functionality is needed by FORTE the interface has to be re-implemented in a appropriate way. To determine how the communication interface shall be re-implemented, the provided functions and the typical usage of the interface have to be analyzed.

The purpose of the communication functions is to prepare data values to be sent over a network or serial interface, in the platform independent format ASN.1 as specified in the *IEC 61499 Compliance Profile for Feasibility Demonstrations*[HOL]. The communication functions are typically used by the communication FBs, which passes the data value to the communication functions to encode the value, or passes a ASN.1 coded data package to receive the decoded data value according to the compliance profile. As [HOL] only describes a sample encoding, it is thinkable that the encoding can be replaced by another one.

Based on this findings the class hierarchy, shown in Figure 4.8 for this "serializer" is conceived. The class hierarchy consists of the abstract interface class `ISerDeser`, which defines the functions `serializeData` and `deserializeData` as the interface for all concrete classes of the hierarchy. The interface is sufficient for the communications FB, as it is possible to encode and decode data packages. The subclass `CITASerDeser` implements the encoding and decoding rules of [HOL].

This design allows to easily change the encoding rules, as it is only necessary to create a new subclass which inherits from `ISerDeser` and to change the used concrete class for encoding in the corresponding communication FBs.

4.5 Function Block Management

The "Function Block Management" consists of two main parts, the `Object Handler` and the `Type Library`. The `Type Library` is a kind of list, which holds all FBs known to FORTE. It implements a so-called "Abstract Factory" design pattern, which allows to select an object by a name and then creates the selected object[GHJV96].

The `Object Handler` is the second half of the Function Block Management. Its task is to interpret and process the incoming management commands, in-

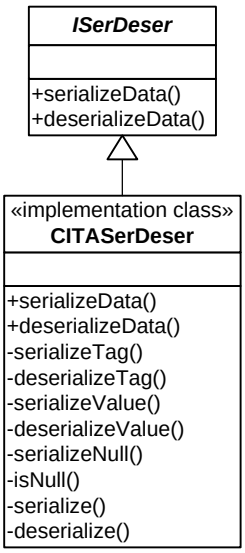


Figure 4.8: Serializer Class Diagram

voke the necessary functions to perform the requested command, and then send the corresponding response.

The goal of the redesign of the `Object Handler` is to remove the ability of FORTE to reconfigure a FBN at runtime, as a firmware device shall contain a fixed FBN with no need to be changed at runtime. To achieve this goal several steps have to be set. First the `Type Library` can be removed, as no FBs have to be created at runtime. Second the now unnecessary management commands, such as FB creation or deletion, have to be removed. At last the class hierarchy of the `Object Handler` has to be changed in a way that it is possible to use the `Object Handler` with the reduced management command set (called `Reduced Object Handler`), as well as the `Object Handler` with the original set of management commands.

4.5.1 Original Design

As the first step the class hierarchy of the `Object Handler` is analyzed, which is shown in Figure 4.9. As can be seen in the Figure 4.9 the class `C61499-ObjectHandler`, which implements the functionality of the `Object Handler` inherits from the class `CFunctionBlock`. As a inheritance relationship describes a specialization relationship[2] between classes, and the fact that an

[2]sometimes also described as an “is-a” relationship

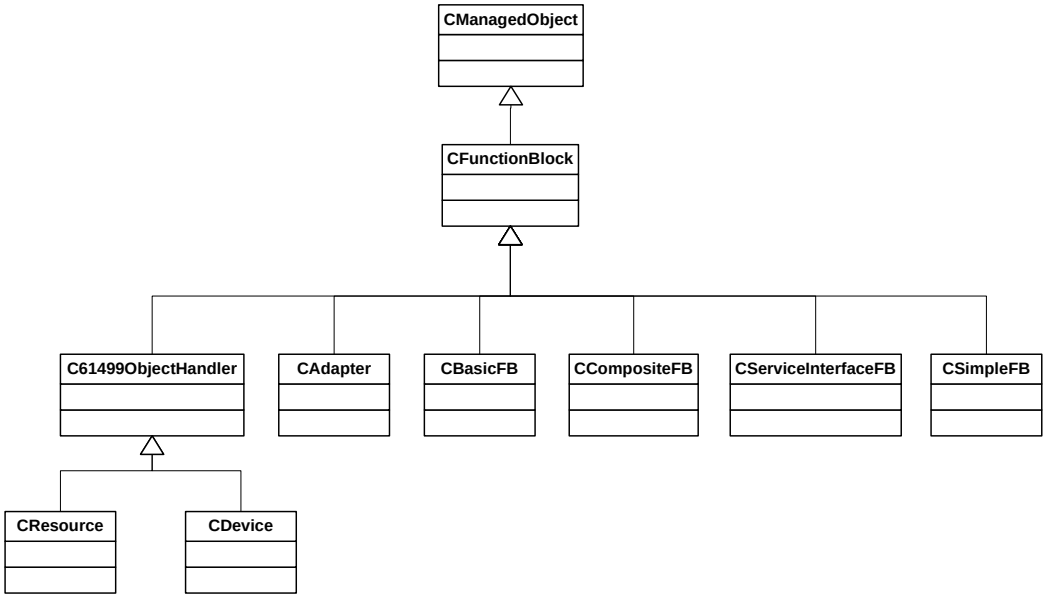


Figure 4.9: Function Block Management Class Diagram

`Object Handler` is not a type of function block, the inheritance relationship between those classes is plainly wrong.

Another oddity in the class hierarchy is the inheritance between the class `C61499ObjectHandler` and class `CResource`, and between the class `C61499-ObjectHandler` and class `CDevice`. The classes `CResource` and `CDevice` implement the resource and device behavior defined in IEC 61499, and as both classes don't represent a type of `Object Handler`, again the inheritance relationship between those classes is plainly wrong.

The next step is to analyze the interface provided by `C61499ObjectHandler`, which consists of following functions:

- `virtual EMGMResponse executeMGMCommand(SManagementCMD &pa_oCommand)`
- `EMGMResponse createFB(CStringDictionary::TStringId pa_nFBNameId, CStringDictionary::TStringId pa_nFBTypeId, const char *pa_acAppl)`
- `EMGMResponse createConnection(TUINT32 pa_nSourceId, TUINT32 pa_-nDestId)`
- `EMGMResponse deleteFB(CStringDictionary::TStringId pa_nFBNameId)`
- `EMGMResponse deleteConnection(TUINT32 pa_nSourceId, TUINT32 pa_-nDestId)`

- `EMGMResponse writeValue(TUINT32 pa_nDestId, const char *pa_acValue)`
- `EMGMResponse readValue(TUINT32 pa_nSourceId, string &pa_sValue)`
- `EMGMResponse queryFBTypes(string &pa_sValue)`
- `EMGMResponse queryRESTypes(string &pa_sValue)`
- `EMGMResponse queryDTTypes(string &pa_sValue)`
- `EMGMResponse queryTypeVersion(TUINT32 pa_nSource, string &pa_sValue)`
- `EMGMResponse executeQueryReq(TUINT32 pa_nSourceId, TUINT32 &pa_nDestId, string &pa_sResponse)`
- `EMGMResponse executeQueryCon(TUINT32 pa_nSourceId, TUINT32 pa_nDestId, string &pa_sResponse)`
- `EMGMResponse executeStartReq(CStringDictionary::TStringId pa_nFBNameId)`
- `EMGMResponse executeStopReq(CStringDictionary::TStringId pa_nFBNameId)`
- `EMGMResponse executeKillReq(CStringDictionary::TStringId pa_nFBNameId)`
- `EMGMResponse executeResetReq(CStringDictionary::TStringId pa_nFBNameId)`
- `virtual EMGMResponse startManagedObject(void)`
- `virtual EMGMResponse stopManagedObject(void)`
- `virtual EMGMResponse killManagedObject(void)`
- `virtual EMGMResponse resetManagedObject(void)`

The received management commands are processed by the function `EMGMResponse executeMGMCommand(SManagementCMD &pa_oCommand)`, which invokes the corresponding function to execute the requested command.

The task to the functions `createFB(CStringDictionary::TStringId pa_nFBNameId, CStringDictionary::TStringId pa_nFBTypeId, const char *pa_acAppl)` and `EMGMResponse createConnection(TUINT32 pa_nSourceId, TUINT32 pa_nDestId)` is to create FB instances and connections between the FBs and FBs and parameters. The `createFB(CStringDictionary::TStringId pa_nFBNameId, CStringDictionary::TStringId pa_nFBTypeId, const char *pa_acAppl)` function can be removed, as the reduced version of FORTE shall not be able to create FBs at runtime. The analysis of the function `EMGMResponse createConnection(TUINT32 pa_nSourceId, TUINT32 pa_nDestId)` has shown, that even if it is not desirable to create connections between FBs at runtime, the function must be kept, as the process of connecting to FBs at start of the application and at runtime is identical.

The functions `EMGMResponse deleteFB(CStringDictionary::TStringId pa_nFBNameId)` and `EMGMResponse deleteConnection(TUINT32 pa_nSourceId, TUINT32 pa_nDestId)` delete the chosen FBs or connections. This functions can be removed, as the analysis of Chapter 3 showed now necessity for this functionality.

The `EMGMResponse writeValue(TUINT32 pa_nDestId, const char *pa_acValue)` and `EMGMResponse readValue(TUINT32 pa_nSourceId, string &pa_sValue)` functions implement the `READ` and `WRITE` commands for accessing the so-called `access paths`. This functions will be kept.

The several `QUERY` commands and the appropriate responses are implemented by the functions

- `EMGMResponse queryFBTypes(string &pa_sValue)`
- `EMGMResponse queryRESTypes(string &pa_sValue)`
- `EMGMResponse queryDTTypes(string &pa_sValue)`
- `EMGMResponse queryTypeVersion(TUINT32 pa_nSource, string &pa_sValue)`
- `EMGMResponse executeQueryReq(TUINT32 pa_nSourceId, TUINT32 &pa_nDestId, string &pa_sResponse)`
- `EMGMResponse executeQueryCon(TUINT32 pa_nSourceId, TUINT32 pa_nDestId, string &pa_sResponse)`.

As the analysis of Chapter 3.4 has shown, FORTE doesn't need to support all of the already implemented queries, and therefore most of the function can be removed from the interface.

The functions

- `EMGMResponse executeStartReq(CStringDictionary::TStringId pa_nFBNameId)`
- `EMGMResponse executeStopReq(CStringDictionary::TStringId pa_nFBNameId)`
- `EMGMResponse executeKillReq(CStringDictionary::TStringId pa_nFBNameId)`
- `EMGMResponse executeResetReq(CStringDictionary::TStringId pa_nFBNameId)`

starts, stops, kills, or resets the chosen FB or application. All of those functions, except `EMGMResponse executeKillReq(CStringDictionary::TStringId pa_nFBNameId)`, will be needed.

The last four functions

- `virtual EMGMResponse startManagedObject(void)`
- `virtual EMGMResponse stopManagedObject(void)`
- `virtual EMGMResponse killManagedObject(void)`
- `virtual EMGMResponse resetManagedObject(void)`

are part of the `Object Handler` interface due to the odd class hierarchy design. As an `Object Handler` is a FB in this design it has to implement the common interface for all FBs, and as a FB has to be able to be started, stopped, killed, and restarted the `Object Handler` also has to provide these capabilities in this design. As initially mentioned the class hierarchy needs to be changed, as an `Object Handler` is not a FB, and therefore these function can be removed.

4.5.2 Revised Design

Considering the design problem found in Chapter 4.5.1, before any interface changes are considered the class hierarchy has to be changed in a way that the design represents the correct relationships between the classes. As the `Object Handler` is clearly not a FB it has to be removed from the FB class hierarchy and be put into its own. As the `Object Handler` for the firmware execution environment represents a reduced version of the already implemented one, this fact should be represented by the class hierarchy. Therefore the new reduced `Object Handler` (called `C61499Class0ObjectHandler` can be seen as the base

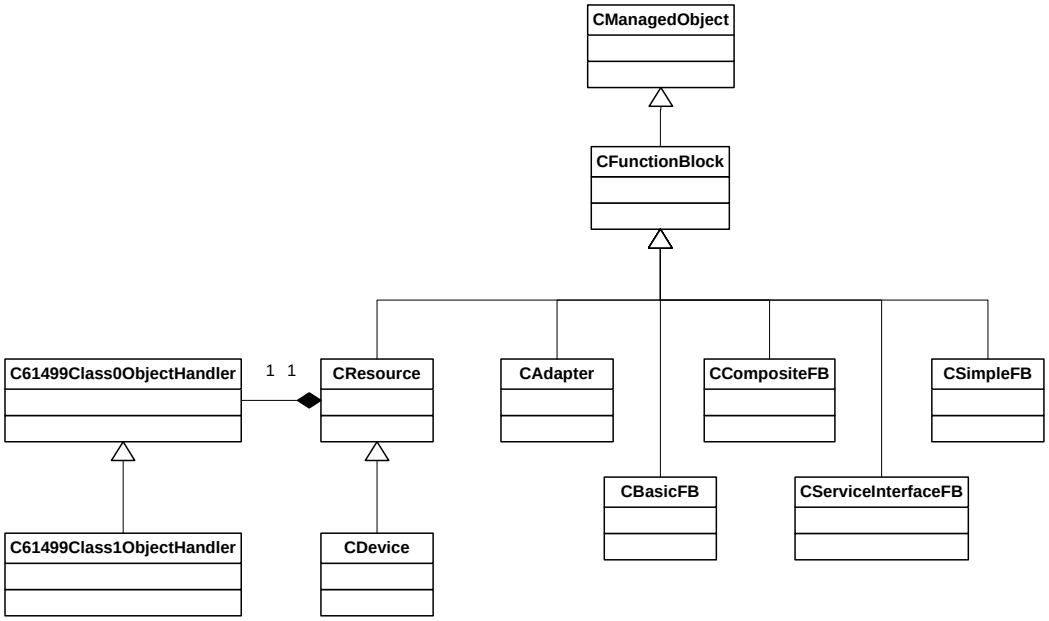


Figure 4.10: Revised Function Block Management Diagram

class for more advanced `Object Handlers` (the `C61499ObjectHandler`, now renamed to `C61499Class1ObjectHandler`), whereby reuse of the functions of the simpler `Object Handler` can be achieved.

The same train of thoughts is also valid for the relationship between `CResource` and `CDevice`, where `CDevice` represents a higher level structure with more functionality, but it also includes the functionality of `CResource`. This allows to design the relationship between those classes as a **"is-a"** relationship.

The last relationship that must be set is the one between `CResource` and `C61499Class0ObjectHandler`, and `CDevice` and `C61499Class0ObjectHandler`, which is a "`has-a`" relationship, as a resource or device has an `Object Handler` for managing their contained FBs. Each resource or device has one and only one `Object Handler`, which is represented by a one-to-one containment relationsship in UML. The new class hierarchy is shown in Figure 4.10.

The next design step is the definition of the interfaces of the `Object Handler` classes. The `C61499Class0ObjectHandler` implement the following interface:

- `virtual EMGMResponse executeMGMCommand(SManagementCMD &pa_oCommand)`
- `EMGMResponse addFB(CFunctionBlock* pa_poFuncBlock)`
- `EMGMResponse createConnection(TUINT32 pa_nSourceId, TUINT32 pa_-nDestId)`

- `EMGMResponse writeValue(TUINT32 pa_nDestId, const char *pa_acValue)`
- `EMGMResponse readValue(TUINT32 pa_nSourceId, string &pa_sValue)`
- `EMGMResponse queryAccessPath(string &pa_sValue)`
- `EMGMResponse executeQueryReq(TUINT32 pa_nSourceId, TUINT32 &pa_nDestId, string &pa_sResponse)`
- `EMGMResponse executeQueryCon(TUINT32 pa_nSourceId, TUINT32 pa_nDestId, string &pa_sResponse)`
- `EMGMResponse executeStartReq(CStringDictionary::TStringId pa_nFBNameId)`
- `EMGMResponse executeStopReq(CStringDictionary::TStringId pa_nFBNameId)`
- `EMGMResponse executeKillReq(CStringDictionary::TStringId pa_nFBNameId)`
- `EMGMResponse executeResetReq(CStringDictionary::TStringId pa_nFBNameId)`

The function `EMGMResponse addFB((CFunctionBlock* pa_poFuncBlock)` replaces the function `EMGMResponse createFB(CStringDictionary::TStringId pa_nFBNameId, CStringDictionary::TStringId pa_nFBTypeId, const char *pa_acAppl)`. The task of the function is to add a instantiated FB, created at startup of the device, to the list of a resource or a device, which is responsible for the FB (see Chapter 2.2.3).

To access the `Access Paths` it is necessary to know their names. The function `EMGMResponse queryAccessPath(string &pa_sValue)` returns the known `Access Paths` of the device, so that the user can read or write to them through the `READ` and `WRITE` management commands.

The remaining functions are the same as in the original design.

The `C61499Class1ObjectHandler` extends this interface to match the functionality of the old `Object Handler`, by the following interface:

- `virtual EMGMResponse executeMGMCommand(SManagementCMD &pa_oCommand)`
- `EMGMResponse createFB(CStringDictionary::TStringId pa_nFBNameId, CStringDictionary::TStringId pa_nFBTypeId, const char *pa_acAppl)`

- `EMGMResponse deleteFB(CStringDictionary::TStringId pa_nFBNameId)`
- `EMGMResponse deleteConnection(TUINT32 pa_nSourceId, TUINT32 pa_nDestId)`
- `EMGMResponse queryFBTypes(string &pa_sValue)`
- `EMGMResponse queryRESTypes(string &pa_sValue)`
- `EMGMResponse queryDTTypes(string &pa_sValue)`
- `EMGMResponse queryTypeVersion(TUINT32 pa_nSource, string &pa_sValue)`

This design allows to select a different `Object Handler` type for each resource and device. The class `C61499Class1ObjectHandler` can also be removed from FORTE, to generate a specialized FORTE version for firmware device. As the `C61499Class0ObjectHandler` doesn't need the `Type Library`, the code size of the specialized FORTE version is greatly reduced.

4.6 Standard Template Library Elements

The `Standard Template Library (STL)` provides a set of often needed classes for C++, such as container classes, associative arrays and strings. Those elements can be used with any built-in or user-defined data type that support a minimal set of operators (e.g., assignment, copy constructor). But for each type that shall be used with a `STL` element, a new class is created by the template-algorithm of C++. The main disadvantage of using `STL` elements in an embedded system is, that they are not optimized for such systems. The goal of the `STL` is, that the contained elements can be used in the majority of cases, and therefore the implementation is a trade-off between several optimization goal (see Chapter 4.2).[GHJV96]

The most frequently used `STL` elements in FORTE are the `string`, and the container classes `set`, `map` and `list`.

4.6.1 String

The `string` class offers many functions as concatenation, dynamic memory allocation, casts to and from built-in types, and much more. But most of the functions of the `string` class is not used. The strings of FORTE are mainly used to receive management commands, and to send management responses.

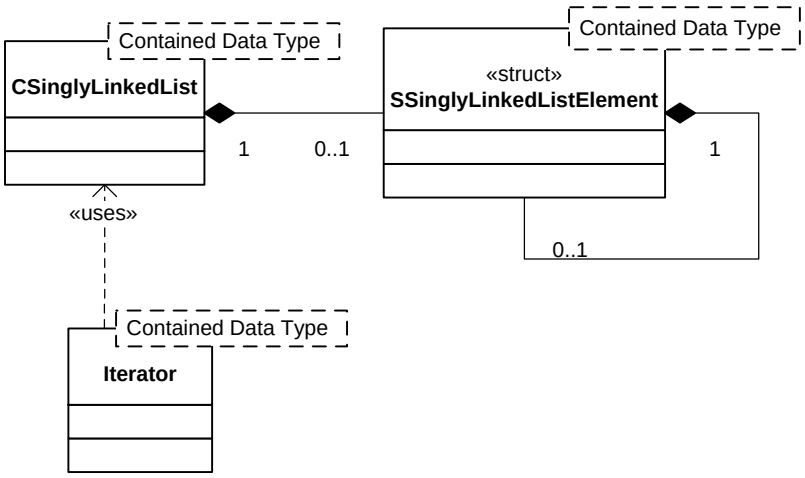


Figure 4.11: Singly Linked List — Class Diagram

`String` allocate the memory needed to contain the value dynamic, which is not desirable in embedded systems, because it can lead to memory fragmentation as embedded system seldom implement garbage collectors[3]. This can lead to a situation where only small chunks of continuous memory is available, and therefore, even if the sum of free memory would be sufficient, no large enough continuous memory could be allocated for a variable. In the worst case this can lead to a system crash.

As most of the `string` functions are not needed, and to save a great amount of code size, all strings are removed from the FORTE and replaced by character arrays. If functions like concatenation, or comparison of strings are needed the appropriate C functions are used.

4.6.2 Container

The usage of the container classes `set`, `map` and `list` in FORTE showed, that all container classes can be replaced by singly linked lists, as high access speed and random access are not necessary. The only objective for a container class in FORTE are small size, which is fulfilled by a singly linked list. As the string already has been removed, it is desirable to get rid of the other STL elements as well, because platform independency can be improved, as different compiler vendors deliver different implementations of STL elements. Therefore if FORTE implements its own singly linked list, the exact implementation is

[3]A program routine for reordering the allocated memory to reduce memory fragmentation

known and there is no further compiler vendor dependency.

The singly linked list consists of a container class (`CSinglyLinkedList`) which holds the pointer to the first list element (`SSinglyLinkedListElement`), if the element has been created already. Each list element contains a pointer to the next list element, as shown in Figure 4.11. The `Iterator` is used to traverse along the singly linked list and to access its elements. This is only possible in one direction, as each element holds only the pointer to the next element. Therefore if a previous element must be accessed, the `Iterator` has to restart at the beginning of the list.

The interface of the container class consists of:

- `void push_front(T const& pa_roElement)`
- `void push_back(T const& pa_roElement)`
- `void pop_front()`
- `void clearAll()`
- `bool isEmpty()`

The functions void `push_front(T const& pa_roElement)` and `void push_back(T const& pa_roElement)` adds the given element at the start or at the end of the singly linked list, where `void pop_front()` deletes the first element and `void clearAll()` deletes all elements. The function `bool isEmpty()` returns true if the list is empty, otherwise it returns false.

The `Iterator` implements a minimal set of operators for a one-way read/write access iterator, which are:

- `Iterator& operator++()`
- `T& operator*()`
- `T* operator->()`
- `bool operator==(Iterator const& rhs)`
- `bool operator!=(Iterator const& rhs)`

The `Iterator& operator++()` traverses the `Iterator` to the next element. The `operators bool operator==(Iterator const& rhs)` and `bool operator!=(Iterator const& rhs)` implement the access to the stored elements, and the last two operators `bool operator==(Iterator const& rhs)` and `bool operator!=(Iterator const& rhs)` implement the equality and inequality comparations.

4.7 Results

After the end of the code size optimization the success of the optimization measures has to be proved. Therefore the original version (FORTE V0.3), the redesigned version with full management command support (FORTE V0.36), and the redesigned version with reduced management command support support (μFORTE V0.36) have been build with different compiler optimization settings, which are `O0` for no optimization, `O1` for local size and speed optimization[4], `O2` for global size and speed optimization, `O3` for pure speed optimization, and `Os` for pure size optimization. The program size is split into code size (`text`), initialized data (`data`), and uninitialized data `bss`. The total program size is given by `dec`.

In Table 4.1 the code size of the different FORTE version are compared. FORTE V0.3 is the original version of FORTE before the optimization. FORTE V0.36 is the optimized version with full management command support and μFORTE V0.36 is the specialized version of FORTE for firmware devices. As can be seen the major part of the size reduction is due to code size reduction, and therefore the design changes are proven to be correct. The program size of the FORTE V0.36 has been reduced by 101064 bytes (33%) compared to the original version, without loosing any functionalities provided by the original runtime.

The reduced version μFORTE V0.36, suitable for embedded firmware execution, is even smaller, saving 172076 bytes (56%) compared to the original version, which is why this version is appropriate for single-chip solutions, where ROM and RAM size is limited.

4.8 Summary

Based on the requirements formulated in Chapter 3, the already existing runtime environment FORTE has been modified to meet those requirements. After a short overview of FORTE, optimization guidelines and goals have been formulated. The main goal of the optimization was to reduce code size, by the means of better implementations of some FORTE packages and by the reduction of functionality, such as real-time reconfigurability. Based on this criteria FORTE packages have been identified, which have to be optimized. Each package has been thoroughly analyzed, and based on this analysis the optimization has been performed, where special attention was given to the explanations of the design decisions in the new design.

[4] Although this option is seldom used

Table 4.1: Size Comparison FORTE — Values in Bytes

	section	O0	O1	O2	O3	Os
FORTE V0.3	text	336092	311804	332944	345280	282528
	data	4624	4628	4444	4444	4468
	bss	18376	18376	18280	18280	18280
	dec	359092	334808	355668	368004	305276
FORTE V0.36	text	242272	2386280	219360	223168	183512
	data	2544	2740	2548	2548	2572
	bss	18128	18224	18120	18120	18120
	dec	262944	259592	240036	243844	204212
μFORTE V0.36	text	151500	148252	131704	134872	113772
	data	2520	2716	2524	2524	2548
	bss	16880	16976	16880	16880	16880
	dec	170900	167944	151108	154276	133200

5 Outlook

New possibilities ans tasks are opening up through the new IEC 61499 runtime environment specialized for firmware devices. The next step after the code size optimization is to evaluate the execution performance of the revised design. The changes done in FORTE should not only decrease the used code space, but also increase the execution speed considerable, as the new design eliminated most of the used virtual functions. The latest investigations on virtual function performance indicate that a virtual function is approximately three times slower than a simple function call[O'R02]. Another reason for a thoroughly execution speed evaluation is to identify execution bottlenecks throughout the runtime, as small systems tend to have less execution speed than industrial PCs or PLCs. Therefore it is important to improve the execution performance of the runtime environment as well.

As the size of FORTE is now reasonable to run on small systems (e.g., embedded systems, firmware controller), the FB paradigm of IEC 61499 can now become a part of firmware and embedded systems design. Therefore it must be evaluated, if the new programming paradigm is suitable in the day-to-day business of embedded systems design and firmware development, and which further gains can be generated by the use of the FB paradigm.

The Component-based software engineering (CBSE) is a promising new software design paradigm, as it focuses reuse of software elements of the granularity of a FB. It should be researched if the principles of CBSE can be applied on IEC 61499 FBs and which prerequisites must be meet in the development of a FB, that it can be seen as a component according to CBSE. If this is possible IPMCS device vendors would have several benefits. They could program the device firmware with FBs and therefore automatically generate IEC 61499 compliant devices, and also generate a set of FB that can be reused in further products. The firmware interface, which could be represented by a SIFB, could be handed to the customer, who could use the SIFB in his application. As the customer only needs the interface and a behavior specification of the device, the FBN of the firmware device, and their algorithms would be not visible to the customer, whereby the intellectual property of the device vendor would be protected.

At last it would be interesting to investigate on the capability of IEC 61499

for the use in low-power system. The event-based execution model of IEC 61499 would be well suited for such systems, as algorithms are only executed, if the corresponding event arrives. Therefore a low-power system could go into a sleep mode as long as no events arrive. Such capabilities are interesting in the field of building automation systems, where more and more wireless, battery powered devices are used for e.g., temperature, moisture, and light sensors. The data of those sensors is then used to control the heating system, blinds, and the ventilation system. As those devices are battery powered it is desirable to reduce the power consumption to a minimum.

Battery powered wireless smart sensors are also used in so-called "Wireless Smart Sensor Networks", where each sensor performs its measurements, but additionally the wireless sensors form a mesh network. The measured values are passed to the next node as long as the value arrives at the destination node, which is commonly a so-called gateway node and is connected to another network, e.g., connected to a PLC or actuators.[Mah04]

6 Conclusion

Lot-size one and small batch productions are becoming more important, as the western high-labor cost countries can not compete with low-labor-cost countries in the field mass production. Therefore they are forced to be more flexible in their production and be on the edge of innovation with both production processes and products. The established standard IEC 61131 is not suited for the new task of high adaptability and therefore the standard *IEC 61499 – Function blocks* has been developed. Its core capabilities are run-time reconfigurability and distributed execution, wherefore it is suited for fast adaptation of the production facility to new products and processes. Unfortunately the IEC 61499, despite being introduced in January 2005, has not become widely accepted.

One reason for it, was that no functioning and free execution environment, and therefore no IEC 61499 compliant devices, have been available. The 4DIAC initiative released the free available open source runtime environment for IEC 61499 named FORTE. Although compliant to the standard and freely available FORTE had the disadvantage, that it has been to big for low-cost micro controllers. Therefore it was not possible to program devices, that have only little ROM available e.g., smart sensors or smart actuators, which is why no direct integration in IEC 61499 of such devices have been possible. That is why this thesis analyzed the requirements of firmware development and research if the found requirements could be fulfilled by IEC 61499.

The first step was to identify what requirements a typical automation device poses on a runtime environment, hence why the typical life-cycle of an automation device was analyzed. As hypothetical devices are not tangible and therefore important points could be missed, a sample application, a distributed heater control, has been introduced, whereof the programming phase and the operational phase has been thoroughly examined, by use case analysis. Based on the insights of the use case analysis, first the requirements have been deduced, then a conceptual design for an embedded execution environment for modular firmware structures has been defined.

As FORTE has been chosen to be the basis for the embedded execution environment, the whole runtime has been thoroughly inspected. With the conceptual design and the requirements in mind, the parts of FORTE which needed modification have been identified. The main goal of the redesign was to

decrease code size, by optimization of FORTE and the reduction of integrated functionality to the point, where only the basic requirements for firmware execution were met. This included the reduction of the number of supported management commands and optimization of some parts of the FORTE (e.g., data types). Along the way some disadvantageous design decisions have been found and replaced by more suited designs. The code size of FORTE has been reduced by 101064 bytes (33%), while retaining the full functionality of the original version, or by 172076 bytes (56%) in the reduced version.

The next steps could be to examine the execution performance of the revised FORTE version, or the possibilities for low-power operation for applications in building automation or wireless sensor networks. It would be also of interest if CBSE can be applied to the IEC 61499 programming model.

The now possible integration of various small devices in IEC 61499 systems opens up many possibilities for the use of embedded systems in IPMCS, building automation systems, wireless sensor networks, and other fields. As smart sensors, smart actuators and other embedded devices can now be directly included in IEC 61499 system, whereby the granularity of distribution can be further reduced. This improves the reconfigurability and reuseability of IEC 61499 systems, as instead of a PLC controlling the embedded devices, and therefore forming a decentralized architecture, the devices are now becoming part of the IEC 61499 system.

Bibliography

[Dou99] Bruce Powel Douglass. *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-49837-5.

[Dou02] Bruce Powel Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems.* Addison-Wesley, Boston, 2002. ISBN 0-201-69956-7.

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, Bonn, 1996. ISBN 3-8273-1862-9.

[HOL] HOLOBLOC, Inc. IEC 61499 Compliance Profile for Feasibility Demonstrations. http://www.holobloc.com/doc/ita/index.htm.

[IEC03] IEC. *IEC 61131-3 Programmable controllers Part 3: Programming Languages.* IEC, January 2003.

[IEC05a] IEC. *IEC 61499-1 Function blocks - Part 1: Architecture.* IEC, 1 2005.

[IEC05b] IEC. *IEC 61499-1 Function blocks - Part 4: Rules for compliance profiles.* IEC, 1 2005.

[ISO05] ISO. *Unified Modeling Language Specification , ISO/IEC 19501.* ISO, January 2005.

[Lew01] Robert Lewis. *Modelling control systems using IEC 61499, Applying function blocks to distributed systems.* IEE - The institution of Electrical Engineers, London, United Kingdom, 2001.

[Mah04] Stefan Mahlknecht. *Energy-Self-Sufficient Wireless Sensor Networks for the Home and Building Environment.* PhD thesis, Vienna University of Technology, 2004.

[OMG09a] OMG. *OMG Unified Modeling Language^{TM}(OMG UML), Infrastructure*. OMG, February 2009.

[OMG09b] OMG. *OMG Unified Modeling Language^{TM}(OMG UML), Superstructure*. OMG, February 2009.

[O'R02] Martin J. O'Riordan. Technical Report on C++ Performance (DRAFT). May 2002.

[Öst01] Bernd Österreicher. *Objekt-orientierte Softwareentwicklung Analyse und Design mit der Unified Modeling Language*. Oldenburg Verlag München Wien, 5 edition, 2001.

[Str99] Bjarne Stroustrup. *The C++ Programming Language Third Edition*. Addison-Wesley, Murray Hill, New Jersey, 1999. ISBN 0-201-88954-4.

[Szy02] Clemens Szyperski. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[Vya07] Valeriy Vyatkin. *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*. ISA, New Zealand, 2007. ISBN 978-0-9792343-0-9.

[Zoi07] Alois Zoitl. *Basic Real-Time Reconfiguration Services for Zero Down-Time Automation Systems*. PhD thesis, Vienna University of Technology, Vienna, 2007.

[Zoi09] Alois Zoitl. *Real-Time Execution for IEC 61499*. ISA, O3Neida, 2009.