TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

Integration of EIB within OSGi Environments

ausgeführt am

Institut für Rechnergestützte Automation
Arbeitsgruppe Automatisierungssysteme
der Technischen Universität Wien

unter der Anleitung von

ao. Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Kastner

durch

Georg Neugschwandtner
Aspettenstraße 30/4/12
2380 Perchtoldsdorf

Wien, im Mai 2004

Abstract

Field area networks extend computer networking to the level of sensors and actuators. EIB (European Installation Bus) is an open and well-established representative of this group used in building automation for connecting intelligent wall switches and light dimmers. Integration with other networks is a key point in leveraging its potential. The OSGi (Open Services Gateway Initiative) defines a dynamic, service-oriented framework for Java-based software components. It is especially intended to deliver flexible functionality to gateways interconnecting wide-area and local networks, while remaining independent of any particular network technology or application scenario.

The present thesis is concerned with bringing together these two technologies. For this purpose, the relevant background is explored first, covering aspects of home automation in general as well as EIB and the OSGi platform specifically. Then, the parts of the EIB network stack especially relevant for communication over gateways are identified, and suitable ways to represent them within the context of the OSGi environment are developed. A modular approach is taken towards exposing this functionality. At its foundation, a low-level interface allows the full use of all features of EIB while already relieving higher-level components from many aspects of data frame assembly as well as hiding the peculiarities of different communication controllers. A prototype implementation of this base component is presented as well, providing a convenient generic foundation for various kinds of EIB related development work.

On top of this base abstraction, further modules can add exactly the functionality needed for a particular application. Specifically, a component allowing to access application-related functionality in EIB devices is described. Maximising the usefulness of this interface, a technology-neutral, generic abstraction for the functionality available in building automation installations is proposed. This representation, aimed at systems of limited size, allows data points to be freely grouped according to topological and functional points of view.

Kurzfassung

EIB (Europäischer Installationsbus) ist eine offene und weit verbreitete Netzwerktechnologie für den Feldbereich. Sie kommt im Bereich der Gebäudeautomation zum Einsatz, wo sie Sensoren und Aktuatoren wie intelligente Schalter und Dimmer verbindet. Ihr Potential kommt insbesondere bei der Integration mit anderen Netzwerken zur Geltung. Das OSGi-Konsortium (Open Services Gateway Initiative) beschreibt ein dynamisches, serviceorientiertes Framework für Java-basierte Softwarekomponenten, welches insbesondere dazu gedacht ist, Gateways an der Schnittstelle zwischen lokalen und Weitverkehrsnetzen mit flexibler Funktionalität auszustatten. Nichtsdestoweniger ist es unabhängig von spezifischen Netzwerktechnologien oder Anwendungsfällen.

Die vorliegende Arbeit beschäftigt sich damit, diese zwei Technologien zusammenzuführen. Zu diesem Zweck erfolgt zunächst eine Aufarbeitung der relevanten Grundlagen, wobei Aspekte der Heimautomation im Allgemeinen ebenso behandelt werden wie Spezifika des EIB und der OSGi-Plattform. Daraufhin werden jene Teile des EIB-Protokollstapels identifiziert, die besondere Relevanz für die Kommunikation über Gateways aufweisen und geeignete Wege für deren Darstellung im Rahmen der OSGi-Umgebung entwickelt. Für die Abbildung dieser Funktionalität wurde ein modularer Ansatz gewählt, dessen Basis von einer Schnittstelle gebildet wird, die, obwohl sie den vollen Zugriff auf alle Merkmale des EIB erlaubt, für übergeordnete Komponenten bereits zahlreiche Aspekte der Konstruktion von Datenrahmen übernimmt und Eigenheiten unterschiedlicher Kommunikationshardware verbirgt. Eine Prototypimplementierung dieser Basiskomponente, die eine allgemein einsetzbare Grundlage für verschiedene Entwicklungen im Umfeld von EIB bildet, wird ebenfalls vorgestellt.

Basierend auf dieser grundlegenden Abstraktion können weitere Module exakt jene Funktionalität hinzufügen, die für eine bestimmte Anwendung relevant ist. Insbesondere wird eine Komponente beschrieben, die den Zugriff auf anwendungsbezogene Funktionalität in EIB-Geräten erlaubt. Um den Nutzen dieser Schnittstelle zu maximieren wird eine technologieindifferente, generische Abstraktion der von Gebäudeautomationssystemen bereitgestellten Funktionalität vorgeschlagen. Diese Darstellung hat Anlagen beschränkter Größe zum Ziel. Sie erlaubt die freie Gruppierung von Datenpunkten nach topologischen und funktionalen Gesichtspunkten.

I would like to express my heartfelt gratitude to those
who helped this thesis come into being:

To my supervisor, who was always available for discussion and feedback;
To my parents, who fully and continuously supported my studies;
To my friends, whose encouragement kept me going;
And to Irmgard, for standing by me.

Contents

1 Introduction and Motivation **1**

1.1 "Smart" Wiring 1
1.2 Gateways and OSGi 3
1.3 Benefits 6
1.4 Task and Structural Overview 7

2 Home and Building Automation **9**

2.1 The Networked Home 10
2.1.1 The Present: Automated Devices 11
2.1.2 Making the Connection: Control Interfaces 12
2.1.3 The Future: Automated Homes and Remote Services . . 13
2.1.4 The Role of Gateways 14
2.2 Building Automation 16

3 EIB **19**

3.1 Overview 20
3.1.1 Core Technical Properties 20
3.1.2 Organizational Background and Standardisation 22
3.1.3 Configuration 23
3.1.4 Competitors 24
3.2 Group Communication 26
3.2.1 Group Objects and Shared Variables 28
3.2.2 Interworking 30
3.3 The OSI Reference Model 34
3.4 Twisted-Pair EIB 37
3.4.1 Topology 38
3.4.2 Standard Message Cycle 39
3.4.3 Format of the Standard Data Frame 40
3.4.4 Other Frame Formats 42

3.5 Other Physical Media . . . 43
3.6 Medium Independent Layers . . . 45
3.6.1 Overview . . . 45
3.6.2 Horizontal Run-Time Communication . . . 46
3.6.3 Network Management . . . 47
3.6.4 Interface Objects . . . 48
3.7 Node Development . . . 50
3.8 Integration . . . 52
3.9 The Future of EIB: KNX . . . 56

4 The OSGi Platform 59
4.1 Related Concepts and Technologies . . . 61
4.1.1 Component-Based Software Engineering . . . 61
4.1.2 OSGi as a Component Model . . . 62
4.1.3 Service-Oriented Architectures . . . 63
4.1.4 Service Orientation in Practice . . . 65
4.2 Framework Architecture . . . 68
4.2.1 Bundles . . . 69
4.2.2 Services . . . 70
4.2.3 Core Design Features . . . 72
4.3 Standard Services . . . 73
4.3.1 Framework Extensions . . . 73
4.3.2 Device Access . . . 74
4.3.3 Communication . . . 76
4.3.4 Management . . . 77
4.3.5 Wiring . . . 78
4.3.6 Measurement . . . 78
4.3.7 Structured Data Storage . . . 79
4.4 Challenges in Service Design and Implementation . . . 79

5 OSGi/EIB Integration 82
5.1 High-Level Design Considerations . . . 82
5.1.1 Requirements and Possible Benefits . . . 82
5.1.2 EIB and the OSGi Device Access Model . . . 84
5.1.3 Consequences and Overview . . . 85
5.2 EIB Frame Service . . . 88
5.2.1 Service Interface . . . 90
5.2.2 Prototype Implementation . . . 92

5.3 EIB Group Communication Service 95
5.4 Advanced Driver Architecture 96

6 A High-Level Control Abstraction 99
6.1 Architecture Overview 100
6.2 A Pool of Data Points 102
6.3 Functional Aspects 103
6.4 Presentation 106

7 Conclusion and Outlook 109

List of Abbreviations 112

Bibliography 113

List of Figures

1.1 Centralised vs. Bus-Based Automation System 2
1.2 Residential Gateway Using Software Components 4

2.1 Network Levels in Building Automation 18

3.1 EIB Regular Operation: The Publisher/Subscriber Model 27
3.2 EIB Regular Operation: Group Addressing, Advanced Example 28
3.3 EIB Regular Operation: End-to-End Overview 31
3.4 EIB Example Device: Switching Actuator 32
3.5 The OSI Model: Data Units 36
3.6 The OSI Model: Service Primitives 37
3.7 EIB Network Topology 38
3.8 Medium Access Timing for Twisted-Pair EIB 40
3.9 EIB Standard Data Frame 40
3.10 EIB Network Stack and Application Environment 45
3.11 EIB Interface Objects 49
3.12 EIBnet/IP 53

4.1 Architectural Overview of the OSGi Platform 68
4.2 OSGi Component Interaction: Bundles and Services 71
4.3 The OSGi Service Registry 72
4.4 OSGi Device Access: Driver Refinement 75

5.1 Hardware-Software Collaboration 89
5.2 EIB Frame Service: Information Flow 90
5.3 EIB Frame Service: Class Diagram 92
5.4 EIB Frame Service: Prototype Implementation Class Diagram . 93
5.5 EIB Frame Service: Screenshot of Prototype Client 95
5.6 EIB Group Communication Service: Information Flow 96
5.7 “Smart” Driver Architecture 97

6.1 System Architecture 101
6.2 Data Point Properties 104
6.3 Data Points: Semantics Distribution and Binding 106
6.4 Data Point Presentation: Possible User Interface 108

1 Introduction and Motivation

The term "Smart Home" has grown increasingly popular over the last years. Although it is also associated with the aspect of using innovative building materials and techniques, mainly in order to reduce energy consumption, the key ingredient which makes a home a "smart" one seems to be extensive automation. A home where lights and heating turn up to the preferred level automatically when the tenant comes back from work or maybe even the refrigerator sends a notice to a mobile phone to warn that its door has been left open certainly means more comfort for the average consumer. But it means even more to high-risk groups like the elderly or disabled people. For them, it opens up the possibility to live on their own, staying in their familiar environment. This is of ever-increasing importance as the average population ages.

1.1 "Smart" Wiring

For this vision to become reality, it is obviously necessary to have the required electromechanical and electronic devices like motorized windows blinds and motion sensors installed in the first place. Wiring them the traditional way, however, can only fulfil very basic requirements. For example, it is not possible to have a single switch near the door which allows to turn off all lights in the house and close all blinds.

For this, it is necessary to break the direct connection between the switches and the devices they control. This can be achieved by installing a central unit to which both all switches and all devices to be controlled are connected. This control unit would contain relays to power the lights, which can be activated and deactivated in any suitable pattern derived from the switch inputs.

This way, it is no longer one certain wall switch that closes the contact to have a lamp supplied with power. This association is maintained within the control unit. Separating logical from physical connections offers a high degree of flexibility: The lamp may easily be controlled from multiple switches, each far from the other and possibly even on another electrical circuit. In case it is desired to have another switch control the lamp, it is no longer necessary to

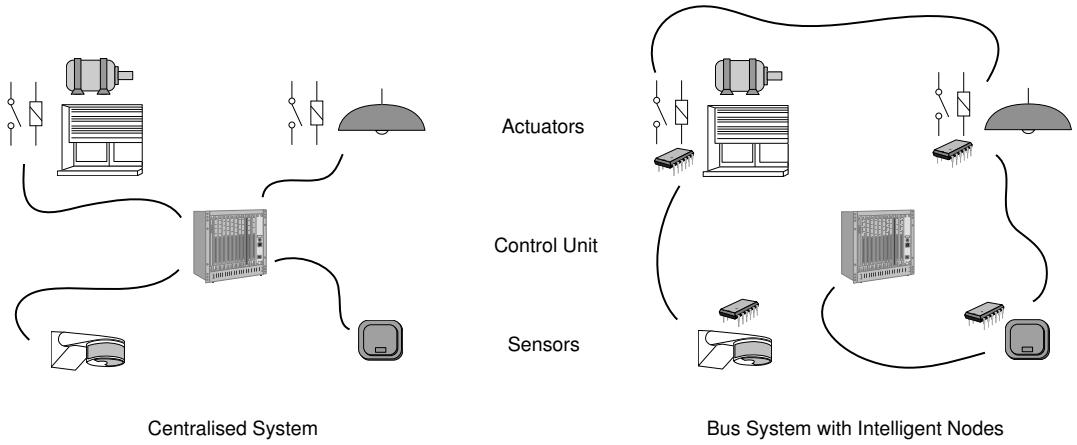


Figure 1.1: Centralised vs. Bus-Based Automation System

physically change the wiring. For such a modification, it is sufficient to change the set-up of the control unit.

Such an approach, however, requires considerable lengths of cable. Yet, many of them will only carry control information. Separating it from power distribution will significantly reduce the size of the necessary cable harnesses, since control information can be transmitted using much lower voltage and current and thus using finer cables. For switches, this is straightforward. For power consuming devices, it can be achieved by separating the control relays from the central unit.

Even then, many of these wires will only transfer one single piece of on/off information—by either being live or not—, which is still inefficient. This can be improved by replacing these connections with a computer network. For this, all switches and devices have to be made *intelligent* by adding the necessary communications hardware to make them self-contained network nodes. Thus, they can share a single cable to transport all of the control information. Such a solution is often referred to as "smart" (in contrast to "traditional") wiring, as it allows a maximum of additional functionality with a minimum of additional cabling. Figure 1.1 illustrates this transition. Intelligent devices even can maintain the bindings between cause and desired effect themselves, eliminating the need for a central unit.

Networks like these reach out to the lowest level of automation technology containing the *sensors* and *actuators* which directly interact with the environment, known as the *field level.* Consequently, they are referred to as *field area networks (FANs).*

FANs need to transfer control data robustly in a cost-effective way. While control messages contain only small amounts of data, challenges are to be found elsewhere. In building automation, data have to be collected from and distributed to dispersed stations, which only participate infrequently in communication. In the automotive area, as another example, range is limited but messages are far more frequent and often have to meet real-time requirements. Several specialized network technologies tailored to these specific requirements are available on the market.

Sensors collect information from the environment. In building automation, this includes wall switches, motion and temperature sensors. Actuators allow the automation system to act upon its environment. Which devices are to be regarded as actuators depends on what is considered the "environment" to be influenced. If it is the perceived light intensity or the state of the window blinds (which would match the—unanimously accepted—stance taken with sensorics), then individual luminaries and blinds drives would count as actuators. The popular position is, however, to consider these devices part of the environment and refer to their control relays as actuators.

In most field area network technologies, stations are connected to one single cable. Due to their bus-style network topology, these designs are frequently referred to as *field buses.* Many of them offer both a more flexible topology and more complex functionality than this rigid term would suggest, however. The European Installation Bus (EIB) is a well-established representative of this group expressly aimed at enhancing electrical installations in buildings.

1.2 Gateways and OSGi

The additional possibilities field area networks can bring to electrical building installations are definitely an improvement. Yet, integrating them with other networks offers further evident benefits.

If, for example, video and audio equipment in the home are connected via their own "home entertainment network", interconnecting it with the installation FAN would allow the sound system to play the tenant's favourite easy-listening music whenever the lighting scene for relaxation is activated. The other way round, the home theatre system may have the window blinds closed and the lights turned down when setting to watching a film. To achieve this, messages have to be converted between the two networks, which will most prob-

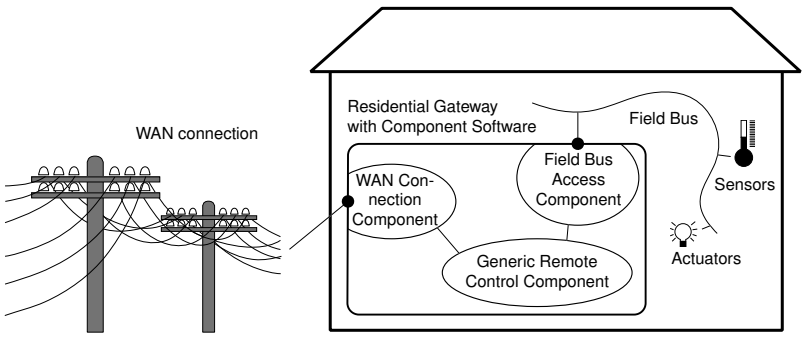


Figure 1.2: Residential Gateway Using Software Components

ably be entirely incompatible. A device which accomplishes this task is termed a *gateway.*

While this is already attractive, some of the most interesting capabilities a "smart home" can offer require a connection from the devices installed within to the outside world. Popular examples include checking whether all doors were really left locked and closed from one's way to work and pre-heating the sauna while still commuting home. Again, a gateway is needed to connect the FAN to whatever WAN (Wide Area Network) seems appropriate, be it the telephone system or the Internet. Such a *residential gateway* may also provide WAN connectivity to other local networks, forming a "bridgehead" where all information sources converge.

This key position makes the residential gateway highly attractive to companies which see a business opportunity in selling services to the tenant via the devices in his or her home. Unlike telephone and cable TV companies, which still had to take care of the physical network connection of the end devices for their services, they would leverage already existing local networks to provide services related to, for example, safety and security, electronic commerce (like automatic replenishment for a "smart fridge"), health care monitoring or communication.

While a number of gateways which allow the remote monitoring and control of an EIB installation are readily available, their capabilities are fixed. Thus, it is not possible to extend them in case the envisaged application is not covered by any of the available products. In this case, a completely new solution would have to be developed —even if a large part of the entire project would actually be the same as in an existing product.

This is inconvenient for service providers. Adding more flexibility to the gateway is not a problem technically, but will incur additional complexity.

Dispatching a technician at every service subscription or cancellation is not acceptable, let alone requiring end users to make the necessary adjustments. Outsourcing this complexity to data centres while keeping to standard gateways on-site is no panacea either, since it does not allow flexible preprocessing and filtering of data before they leave the subscriber's home. What is needed is a gateway with modular functionality (as in Figure 1.2) which can be administrated from a distance so that once the local interfaces are set-up by a qualified technician, software modules using these interfaces can be deployed remotely.

The Open Services Gateway Initiative (OSGi) has taken to the task of providing such an environment. It has defined an open service-oriented software component framework where software components can interact in a defined way and can be installed and uninstalled on demand at run-time. Although Java-based, the software platform is designed to be usable in the resource-limited environment of an embedded device. Since all management aspects of an OSGi platform can be controlled remotely, it effectively enables end users to deal with the subscription of services rather than configuration and management issues.

For an example of how such a service could look like, assume that a home is equipped with a video intercom system for the doorbell. This intercom system is connected to a residential gateway, which contains an OSGi service platform. A telecommunications company may offer to extend this intercom system to the tenant's mobile phone in case he is not at home. When a home owner subscribes to this service, a piece of software is downloaded to the OSGi platform (probably for a monthly fee).

Whenever the door bell is pushed but not answered, the software opens a connection to a back end server at the telecommunications company, where the video and audio from the door station are processed into a form suitable for mobile equipment. From there, they are relayed to the mobile phone, so that the tenant may talk to the person waiting at the door as if he or she was at home. In case it turns out to be a friend, he/she may decide to let him or her in—which is made possible by the door lock being connected to the residential gateway as well. The back end server may also offer to record a message in case the subscriber is not reachable on the phone.

In a similar way, a service provider could act as a trusted agent in a home care situation, deciding which measures to take and for whom to open the door in case of an emergency (indicated by the patient pressing an emergency button or telemedical monitoring).

To perform its task, the software component implementing these services needs to access the intercom system and the door opener through well-defined interfaces. These have to be provided by appropriate *driver services,* as they are known in OSGi terms. Likewise, a suitable interface for EIB devices is needed to access their functionality.

It is important to note that it is not necessary to place an OSGi-based gateway entirely under the (remote) control of a services provider. Although OSGi places its main focus on this business model, great care is taken to keep the specified environment open to alternative forms. These include placing the entire management of the gateway into the hands of the end user. In the case of home automation, such considerations are not only a matter of convenience, but trust, since giving access to the devices in a "smart home" may be as good as handing over the door key.

1.3 Benefits

Obviously, an OSGi driver service for EIB will allow OSGi-based service offerings to include the capabilities of EIB devices. For a control network like EIB, interesting applications are to be expected mainly in the areas of remote control (including load management), device diagnostics and maintenance and safety/security.

A service exposing the functionality offered by building control networks in a way which is independent of the underlying FAN design would be of special benefit. Such an interface would allow service providers to easily extend their offerings to incorporate various control network technologies.

Outside this specific setting, OSGi still remains an open and proven platform supporting the dynamic configuration of software components with powerful remote management capabilities. An OSGi driver service will provide a reusable component enabling software designers to interact with the EIB on a high level of abstraction while retaining a small footprint for the overall solution.

Besides all kinds of experimental work, this could especially jump-start EIB gateway development. The OSGi environment makes it easy to combine various drivers and other application components. Thus, the modular concept of OSGi allows gateways to provide flexible and extensible functionality and maximise their use by integrating multiple, diverse network technologies.

Presenting the control network data in a way which allows them to be accessed by multiple client components would provide further benefits. This

would make it easy to provide multiple control points from where to interact with the devices installed—no matter whether locally or from remote. While one service may provide a control panel on the TV set, another may choose to present the data via a secured WWW page.

Although remote services are not (yet) of significant interest in functional buildings, a software component for EIB access may nevertheless prove useful in this context as well. In larger installations, it could be combined with drivers services for higher-level backbone and automation networks to create custom tailored gateways with reduced implementation effort.

1.4 Task and Structural Overview

The task of this thesis is to provide a solid grounding in EIB and OSGi and investigate ways for their integration. A suitable representation of the functionality provided by EIB devices to software operating on the OSGi platform is of particular interest. Also, an implementation of fundamental functionality to test the basic concepts in practice shall be provided, which can serve as an extensible basis for further research and development.

Consequently, the prerequisites necessary to understanding the further exposition are presented first. Chapter 2 examines the setting where the resulting gateway software is to be placed. Here, clarification of the meaning of the term "Home and Building Automation" is sought. Also, the scope of automation networks in the home is discussed, with a special focus on the role of gateways.

Chapter 3 discusses the key concepts and properties of EIB. Its unique model of communication is presented as well as an overview of the complete network stack and aspects of configuration and integration with other networks (including an overview of existing gateway solutions).

Chapter 4 covers the OSGi Service Platform specification, with a special focus on related software engineering concepts and including specific challenges when developing software for this environment. As with Chapter 3, the goal is to give a mainly technical overview while going into enough detail to allow understanding the design decisions taken further on.

Chapter 5 then motivates and presents the design of the base components developed for accessing EIB devices from within OSGi environments. It also discusses the issues surfacing when trying to bring these worlds together. The prototype implementation of the fundamental module which unifies the handling of the various physical media and bus access hardware solutions available

is covered as well as an experimental concept for making the configuration of EIB devices accessible in an open way.

Finally, Chapter 6 proposes a technology-neutral, generic representation of narrow-band building control functionality for small to medium sized, chiefly residential, installations. Like the interface presented in Chapter 5 hides the details of the underlying EIB hardware, this abstraction even further minimises the amount of context-specific knowledge needed for the development of client services.

2 Home and Building Automation

Webster's Dictionary [21] provides (among others) the following, rather unwieldy definition for the word "automation":

> [... the] automatically controlled operation of an apparatus, process, or system by mechanical or electronic devices that take the place of human organs of observation, effort, and decision.

When taking a look at what people do day in, day out in all kinds of buildings, one can easily find a number of situations where machines could step in to help, like:

- turn off the light as you leave,
- turn down the heating as long as a window is open for airing or
- close the blinds when the summer sun blazes into the room.

To make this possible, one has to allow machines to make the necessary interventions. For example, one has to replace manually operated light switches with relays and install motor-driven window blinds. Besides these "organs of effort", one has to add "organs of observation", like presence detectors, window contacts, but also simple push buttons to receive explicit commands from the user.

All these sensors and actuators have to be connected to a central automation system (the "organ of decision"). Intelligent sensors and actuators may also handle these decisions on their own, in a distributed way.

The choice of examples above illustrates the functions widely associated with home and building automation—above all, lighting and window blinds; at second thought, probably, sophisticated heating, ventilation and air conditioning (HVAC) solutions. However, there are many more functions which can be automated: Doors can be locked and unlocked (maybe even opened and closed) remotely or according to timer programmes. Even if remote operation of door locks is not desired (e. g. for cost or security reasons), their state can at least

be made available for monitoring via cell phone or from the porter's desk to save the trouble of having to check them personally. In a residential setting, the dishwasher could have the TV set display a notification that the load has finished while you were watching, or an automatic irrigation system could look after watering the garden plants. In a larger building, the HVAC system may automatically contact the service engineer if some part has failed.

Although the terms *"home"* and *"building"* have so far been referred to more or less in the same breath, they have considerably different meanings attached to them in the field of home and building automation. Here, "home" refers to a small scale installation in residential context.[1] It is most often associated with an owner-occupied detached house or flat. "Building", on the other hand, is generally short for "functional building", implying the presence of a significantly more complex automation system. The following paragraphs will examine some characteristic features of these two settings and explain why gateways are essential components in both.

2.1 The Networked Home

In the home, electric and electronic devices which could potentially be integrated into an automation system can be divided into several groups according to their function:

- Lighting and window blinds
- HVAC systems, including water heating
- White goods (household appliances), like a washing machine or stove
- Brown goods (audio/video or home theatre equipment, game consoles)
- Communications equipment, both on and off premises (intercom system, telephone)
- Information processing and presentation equipment (PCs, Tablet PCs, PDAs, ...)

[1] While this term seems reasonably well-defined, the ubiquitous "smart home" is more of a moving target: What makes a home "smart" seems to be related to whatever aspect of technology is currently popular. Some years ago, this was home automation, while nowadays the focus has shifted more towards information (and infotainment) systems, like the "smart fridge".

- Security and access control
- Safety alarm systems[2]
- Special domains, for example stair lifts

Technically, PCs are ideally suited for purposes of visualization, monitoring and control. They can offer rich user interfaces for tasks like the setting up of complex timer programmes or reviewing history logs and also have ample processing power and data storage capability. In practice, however, the unstable nature of a general-purpose home PC definitely warrants the installation of dedicated control panels, if such functionality is desired.

When considering the necessary communications infrastructure, one should not let oneself be confused by the fact that some of the above devices handle high bandwidth data, like media streams. For automation purposes, only control information is relevant[3], which is comparatively narrow-band in nature.

2.1.1 The Present: Automated Devices

Within every of these groups, automated solutions are available. Many of them are commonplace in the average first-world household and, though some are of significant complexity, generally not perceived as something special. Time definitely was when washing machines were being advertised as "fully automatic".

Yet, control data remains confined within the boundaries of individual devices. Although some of them have remote sensors or detached control panels, these are merely "satellites" of their main units. The ambient temperature sensor for the central heating boiler cannot perform useful duties on its own, and neither can it usually be attached to any other device to do so. The same holds, for instance, for remote display panels available for certain "avant-garde" white goods.[4]

[2] *Security* alarm systems will alert to actions with malicious intent, like burglaries. *Safety* alarms provide warning in case of failing systems or accidents (strictly speaking, consequences of negligence), like a gas leak or fire.

[3] Recall the dictionary definition on page 9.

[4] Alarm systems are a special case: By the nature of such systems, remote sensors have an integral part in their overall functionality. Thus, standard interfaces have evolved, providing a certain amount of exchangeability concerning these sensors. Yet, these interfaces are as simple as requiring a single electrical connection to be made or broken, and standardization thus merely concerns permissible voltages and currents. Still, the remote sensors are quite useless without the main unit.

Automated event sequences encompassing two or more independent devices are rare, and existing approaches are most often manufacturer-specific. Generally, devices are not even able to supply information to others. If they do—like it is the case with brown goods, which routinely exchange audio and video signals—no control interaction is involved.

Besides the communication between devices within the home, many applications could also profit from a control connection to the outside world. Today, most households possess wide-area communications links—-at least the telephone line, and to an increasing amount broad-band Internet access as well. Yet, the lack of internal connectivity bars local devices from leveraging those unless they are equipped with a dedicated interface. Such dial-out boxes are currently popular with alarm systems, but not so with other appliances, since their installation requires significant effort.

2.1.2 Making the Connection: Control Interfaces

Obviously, any automation concept promising to integrate multiple devices has to find a defined way of accessing their functionality. Finding such an interface can be quite demanding for more complex devices, like a washing machine.[5] Fortunately, there are many devices whose function—and thus the interface—is trivial to describe: either they are powered, or not. These simple devices are mostly found in the field of lighting, like lamps or light switches, but also in others, like window contacts or irrigation pumps. They can be integrated into an automation system simply by providing a relay, which supplies or withdraws power, or by checking whether current is allowed to flow.

This interface is actually so straightforward that its simplest implementation, time switch adaptor plugs, are available at every supermarket. But although the functionality of a single lamp is trivial to describe, it is not as trivial to have many of them operate in a coordinated way. Thus, a seemingly simple function like lighting scene memory still possesses high novelty value. A multitude of products and systems are on the market, which significantly differ in features and performance. This does not only concern whether control data are transmitted using separate cables, via the mains network or by radio. It also touches issues like scalability, open interfaces to the outside or the capability of transferring more than simple on/off information, like temperature values for HVAC control.

[5] Standard Interfaces for white goods are only just beginning to emerge (cf. Section 6).

Even the simpler ones among these systems are advertised as "home automation solutions" in a way that makes it easy to overlook other aspects of automation in the home. In contrast, when appliances offer new automated functions, advanced as they may be—for instance, to give a state-of-the-art example, if a dishwasher examines the dishwater for remaining food particles and lengthens or shortens the program accordingly—, the novel feature will not be referred to as an advance in automation.

Obviously, the term "home automation" is actually reserved to marketing inter-device automation (in contrast to intra-device automation, as with a washing machine) solutions. The reason may very well be that it already conveys the notion of automating the entire residential environment, not just a single appliance. Still, it seldom appears in everyday usage, probably due to its technicality.

2.1.3 The Future: Automated Homes and Remote Services

In the future, control interfaces will be standard on all devices in the household. This will allow to take advantage of synergetic effects, making the whole installation actually more useful than the sum of its parts alone. For example, the machinery necessary for providing lighting scenarios—a comfort detail—can also be used to turn on all lights in case of an alarm triggered by the security system, providing an additional burglar deterrent. Applications like this one which bring together functionality from two or more of the fields mentioned above are especially rare at present. There are no established standards to bridge the gap, and since these fields are traditionally covered by different manufacturers, proprietary solutions cannot step in, either. Likewise, they can be expected to increasingly communicate with the outside world, both as transmitters and receivers of control data. This may, for instance, include an oven which receives tailor-made cooking instructions for an instant meal via the Internet after scanning a bar code (or RF tag) from its packaging. Certainly, communication will not be limited to control data, but include information and media as well (consider instructions for the preparation of the instant meal shown in large print on a nearby display as an example).

Technically, all of this is already possible, and devices offering a significant part of this functionality are already available on the market. Still, they are far from being commonplace. Both product range and market can reasonably be expected to broaden (and prices to be falling), as it was the case with cell

phones over the last decade. When exactly this future is to become present, is unclear, however.

2.1.4 The Role of Gateways

It would in theory be perfectly possible to have all in-home devices equipped with the same single network technology to ensure universal connectivity. From a practical point of view, though, it is very probable that several different ones will be in parallel use. Three main reasons for this will be outlined in the following.

First, different applications have different demands on their communication infrastructure. Transmitting live video, for instance, requires significantly higher bandwidth than switching a lamp on or off. But the capability to handle high bandwidth comes at the price of restrictions in other aspects, given a fixed level of technological maturity. For transmitting high-quality live video over a wire, one currently has the options of handling unwieldy cables (as it is the case with twisted-pair Ethernet cable) or limiting the range of the network (as with "FireWire", IEEE 1394). Differences in requirements are not limited to bandwidth, but also concern protocol issues. For example, mechanisms to handle isochronous data (or other Quality of Service criteria) are not necessary for control networks. Implementing features without practical value would mean wasting resources in network nodes. Thus, although eventually IP networks will probably reach the field level, there will still be a niche for field area networks in home and building automation for a considerable time.

Secondly, even if applications could be based on the same network technology from a technical point of view, one has to consider the matter-of-fact power of market development. The device groups listed at the beginning of this section are traditionally (and probably will be in future) covered by different manufacturers. Understandably, they are reluctant to give up their well-established solutions for the benefit of a common standard. Another, more subtle, example of such a non-technical obstacle is that insurance companies prefer security systems to be self-contained for easier certification. This complicates using possible synergy effects with narrow-band control applications, although these two groups would be a fine match regarding sensors, actuators and required bandwidth.

Third, one also has to consider the fact that the life of buildings is significantly longer than the innovation cycle in communications technology. There may be cases where old technologies do not offer certain desired new function-

ality, but still perform satisfactorily. Therefore, the new technology is often installed in parallel. For example, integrating loose goods into a home automation system would offer interesting possibilities, like vacuum cleaners which automatically stop or turn low when the door bell rings, floor and desk lamps which are included in lighting scenes or the ability to monitor how much power was consumed for ironing clothes. To achieve such integration, "plug and play" support is instrumental. Yet, current field bus designs (like EIB) only target fixed installations, and thus neither have nor need the respective support. A number of apt candidate technologies exist to solve this problem, which will all be able to coexist with present-day field bus installations. Yet, which one will become standard will not be decided any time soon.

Obviously, devices which can connect diverse networks play an essential role. Gateways can unite different network designs to form a single "home area network" without requiring all devices to use a single technology. The individual sub-networks can be tailored to the respective fields of application (and be provided by different manufacturers). A gateway also allows to integrate new technologies while protecting existing investments. As a residential gateway, it will provide wide area access to local devices. In this case, connecting it to all in-house networks will maximise its benefit. Such benefit is not limited to remote control. In enabling devices to communicate with the outside world, a suitable residential gateway allows companies to offer value-added remote services for products (like automatically adjusting the oven for preparation of an instant meal, as mentioned in the previous section) or to retrieve maintenance information.

Holding such a key position will, however, not only maximise the potential benefit, but also the negative consequences in case of functional deficiencies. The gateway has to be designed to handle the expected communication load without being a performance bottleneck. Also, it introduces a potential single point of failure. Therefore, it should not interfere with the operation of the networks connected more than necessary to allow basic operation to continue in case of such a failure. Especially with residential gateways, appropriate privacy and security measures have to be taken, since they provide a wealth of information and extensive control facilities in a particularly exposed location.

2.2 Building Automation

As mentioned earlier, the term "building" generally signifies a functional building (like an office, school or hospital) and implies a larger automation installation. Here, the focus is different from the residential setting discussed so far. While an increase in personal *comfort, safety and security* is seen as the main use of a home automation system, building automation systems are installed for the reason of *economic utility*.[6] The ability to control and optimise the use of a large number of energy consuming devices, allows immediate considerable financial savings to be made.[7]

Besides this aspect of increasing economic efficiency through saving energy, there is also the aspect of saving money by saving time. Examples here would be pre-set defaults for blinds and lighting in a conference room to reduce the set-up time for presentations or reducing the need for security personnel to make their round by showing from the porter's desk where in the building lights are still on, blinds down or doors open. Finally, there are also cases where electrical installations reach such a degree of complexity that building automation systems are needed to fulfil the demands made on their functionality at all, like in theatres or convention halls.

The list of devices which are eligible for participation in a building automation system is not different from the one drawn up for home automation on page 10. What does differ is the relative strength of the groups (consider white and brown goods on the one and lifts on the other hand) as well as envisioned application scenarios (for example, integration of the home theatre system with blinds and lighting versus escalation management of technical alarms). This results in device groups being considered with different priority for integration into an automation system.

As another difference, functions which can in the home be provided by single devices will be provided by entire systems as buildings get larger (and, thus, the complexity of these tasks increases). These systems are themselves automated to a high degree and often use some kind of private network to connect the participating devices.

[6] This also includes higher rental fees expected for "value-added" property, which is one reason for comfort options present in functional buildings as well.

[7] The large scope of control is key: Although a block of flats will have considerable overall power consumption as well, the individual flats form separate spheres of control. Yet, one would again apply the term "building automation" if the property manager of such a block of flats is using an automation system to centrally control the exterior lighting, elevators or local heat distribution.

On the whole, the considerations presented in the previous section considering the role of gateways hold as well for building automation. Yet, there is a difference: While it is a relatively new idea to collect information at a single point in homes, larger buildings traditionally already have such a point where information sources converge—in most cases, this is the porter's desk. Also, there is less need to hand this data over to external service providers, as maintenance or security personnel is permanently present and can be reached using internal communications infrastructure. There is, of course, no clear-cut dividing line for these transitions as functional buildings can be of very different size and management policies vary.

Yet, bringing monitoring and control of several building disciplines together in one place does not automatically mean integrating them into a single, unified management system. Doing so can increase service quality and efficiency, for example by automatically relaying the fault notification from the HVAC system to the pager of the resident technician. Also, it allows to create a common user interface which will provide better usability and may also save money spent otherwise on training necessary for the handling of different systems. Moreover, data can easily be relayed to other locations, like the facility manager's office to provide a basis for calculation and planning. Obviously, gateways are needed to provide this integration.

The use of gateways to allow the free choice of the appropriate communications infrastructure for different applications has already been discussed. An additional consideration has to be included when dealing with large installations: Field buses are ideally suited to connect reasonably large numbers of simple-structured devices. Yet, their performance is too limited to accommodate the overall amount of data generated by control events in very large buildings, especially if all these data are to be concentrated in a single point for monitoring. This problem is addressed by dividing the field network into smaller units (for example, one per floor) and connecting these zones through a higher-performance backbone network.

Thus, a hierarchy of networks emerges, which is illustrated in Figure 2.1: The field level, which connects sensors and actuators; the automation level, which serves as a backbone for the field level networks, but may also have more complex devices connected directly; and the management level, which

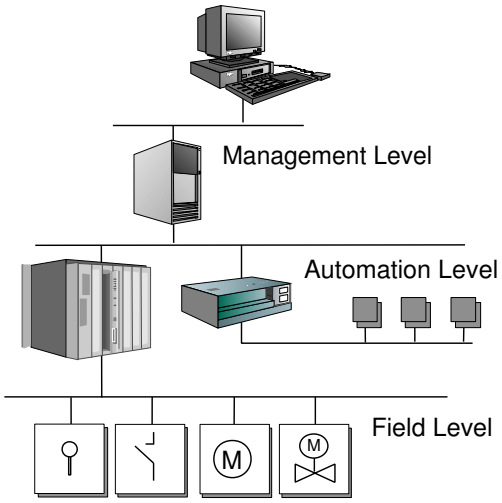


Figure 2.1: Network Levels in Building Automation

provides an unified top-level view.[8] Gateways are needed to mediate between the different network technologies used at these levels.

There are very small chances for this conversion to be perfect, since gateways are by definition sitting atop the communications stack. For example, certain information items may be compulsory on one side but optional on the other. They will have to be replaced by default values, but requirements on both sides of the gateway—and thus sensible default values—may change. Therefore, the advantage of flexible and possibly modular technology is present here as well.

[8]This hierarchy and the level names used are those chosen by CEN TC 247, a European standardization committee concerned with building automation. Depending on circumstances, different structures may be appropriate.

3 EIB

EIB (European Installation Bus) is a field area network designed to enhance electrical installations in homes and buildings. Thus, its technical concept is focused on incorporating devices which are installed in a fixed place and permanently connected. Main application areas of EIB solutions are lighting, the control of window blinds and HVAC systems.

The following chapter explores key characteristics of EIB with regard to the task of creating a gateway software component. First, an overview of core technical and organizational properties is given. Next, the group communication model unique to EIB is introduced, which is essential to understand for being able to combine the functionality of individual nodes into a working system. Then, the network stack is discussed. Of the available communication media, special emphasis is given to twisted-pair wiring. Being the physical medium EIB was originally designed around, it is the key to understanding the protocol. Concerning medium independent layers, focus is placed on the semantics of the services defined by the Application Layer. To finish off the developer's perspective, the available infrastructure for node development is discussed as well as readily available solutions for the integration of EIB into heterogeneous environments—the backdrop of an EIB OSGi driver service. The chapter finishes by looking at how EIB now forms a part of the KNX standard.

For further reference, the *EIB Handbook* [16] (respectively its successor, the *KNX Handbook* [27]) is the authoritative source.[1] Secondary literature on EIB is almost entirely limited to German-language books directed at integrators and installers, like [30] and [41]. While these texts go into great detail in describing the planning, setting up and troubleshooting of an EIB system, [14] addresses audiences more interested in technical background information.

[1] Key parts of the EIB Handbook covering the protocol stack are currently also available for free download from the EIBA web site.

3.1 Overview

The design of EIB is powerful enough to create tailor-made complex installations for large buildings. Nevertheless, it is also being marketed for upscale home automation. Rather high component prices, however, limit its spread in the latter field. The technology is well established in Europe's German-speaking countries, with a great variety of components being readily available from multiple companies. Although their products are compatible, companies use different brand names to refer to EIB technology in their product ranges.

3.1.1 Core Technical Properties

The clear alignment of EIB towards electric installations in buildings is visible in a number of properties. Most obviously, devices come in housings appropriate for this field. *Rail mount* housings, for example, allow them to be mounted in a distribution box easily. Here, the standard DIN rail can be extended to provide bus connectivity in addition to being a mechanical fixture. This is accomplished by a piece of printed circuit board inserted into the recess of the rail. The copper tracks this *data rail* provides remove the need for stringing bus wires between devices sharing a rail.

Also, many *flush mount* devices are available. While rail mount housings are frequently used for switching and dimming actuators, flush mounting is especially suited to sensors, like push buttons or a temperature adjustment knob. With EIB, such devices are frequently split into two parts: Standard modules providing bus connectivity (so-called *Bus Coupling Units, BCUs)* which fit into a standard installation socket are combined with different *application modules*, which contain the user-visible part. The necessary connector *(Physical External Interface, PEI)* is standardised as well. This principle is very similar to the one used with conventional flush mount switches, where the same switch mechanics can be used with various cover plates. It significantly facilitates the creation of design variants.

These standardised housing form factors ensure that integrators have a choice of interchangeable components by different manufacturers for most applications. This does not only offer increased flexibility when creating an EIB system: The dramatically reduced risk of vendor lock-in also proves beneficial in maintenance.

The bus wiring was designed to comply with the demands of professional electricians as well. EIB uses cables which can be threaded into pre-laid con-

duiting and allows arbitrary branching—just like the standard mains distribution. Nonetheless, a fully fledged EIB installation can cover up to several thousand metres[2] and contain tens of thousands of devices, with data traffic still remaining reasonably robust against interference. More recently, the ability to use the mains for communication instead of dedicated wiring *(powerline communication)* and radio transmission[3] were added to the list of available physical media, offering alternative paths for renovation tasks and situations where no cables can be tolerated at all. All of them can be mixed and matched within an installation as needed.

With all currently available physical media, EIB operates at data rates around 10 kBit/s. While this is sufficient for the applications envisaged, the limited margin on allowable bus load demands careful planning of more complex installations. Such considerations have to include the traffic generated by individual devices as well as the total number of nodes. Certainly, more advanced transmission techniques could have allowed higher data rates. Yet, the standard dates back to before 1990, when implementing these (provided they were already researched) would most probably have resulted in unacceptable hardware cost. Today, one should expect the comparatively simple design to allow communication controllers to be sold cheaply. However, prices have been maintaining a high level for years, and there is currently no sign of them going to drop in the foreseeable future.

Designed to provide reliable performance, EIB is a *peer-to-peer* network system. Access to the communication medium is handled using a distributed algorithm. Likewise, the mapping between sensor inputs and desired actuator actions is maintained in a decentralised way. By avoiding the need for a central station controlling these aspects (as still present in Figure 1.1), such a design eliminates a potential bottleneck and single point of failure. It allows functionality to degrade gracefully in case of node failures. This is a desirable quality in the functional areas covered by EIB. For instance, the failure of a single light can in most cases be easily tolerated, but complete loss of lighting due to anything but a mains power failure will not be acceptable.

[2]EIB is by no means intended to be a wide area network. However, a network reaching multiple locations in every room of a multi-storey building will exhibit a considerable accumulated lead length as well.

[3]Radio transmission is already defined under the umbrella of KNX, the follow-up standard to EIB (cf. Section 3.9). Nevertheless, it is backward compatible.

3.1.2 Organizational Background and Standardisation

Until recently, the EIB specifications were managed by EIBA (EIB Association). EIBA was founded in 1990 by a number of European installation technology companies involved in the development of EIB to oversee the then-new standard.

In 2002, EIB merged with *BatiBus* and *EHS* (European Home System) to form the new KNX standard. KNX accommodates these legacy standards in an inclusive manner. Thus, every device designed in correspondence with the EIB Handbook [16] is automatically KNX compliant. Although EIB is now correctly known by the name of *KNX TP1/PL110 S-Mode*, "EIB" will definitely stay as a label for a specific, subset of KNX functionality for quite some time and will be used in this manner in the remainder of this thesis.

The definitive and binding specification for KNX [27] is centrally maintained as a consortium standard by *Konnex Association.* It encompasses the complete network protocol as well as application-level interworking profiles. It also regulates basic system components like transceiver ICs and BCUs. The specification is openly available to interested parties at non-discriminatory conditions.

KNX is also documented through formal standards. The relevant family of European Standards is EN 50090. It is maintained by CENELEC TC 205 (Home and Building Electronic Systems, HBES) in cooperation with Konnex Association. The normative process is also coordinated with CEN TC 247, which is concerned with building controls and has in the past also defined standards covering EIB technology (in particular [18], which also includes the BatiBus and EHS protocols). However, the information available by way of formal standards will inevitably always be less comprehensive and up-to-date than the material available directly from Konnex Association.

Konnex Association does not only coordinate further development of the specification, but is also involved in intellectual property rights (IPR) management. This activity is necessary since implementations of the standard will necessarily touch IPR property (i. e. patents) of companies which have contributed material to the common specification. Konnex Members automatically gain the necessary IPR licenses with no additional expenses besides the fixed membership fee.

The association also maintains certification schemes and associated logo programmes for standard compliance of devices. For this purpose, the specification also contains elaborate conformity test procedures on both the software

and hardware level. Detailed minimum requirements on other basic system components like cables and connectors are given as well.

The certification activities are not limited to devices, however. The flexibility of EIB necessarily entails a certain complexity (and intricacy, in places). Being able to influence the operation of the system in many aspects opens up equally many possibilities for error. Lest large numbers of half-working EIB installations should ruin the reputation of the technology as a whole, manufacturers (and Konnex Association as their representative) have a strong interest to see it in the hands of well-trained professionals only. To this end, standardised training programmes are offered. Only after passing the associated examination participants are entitled to bear the "EIB Partner" logo. These courses are held by independent training centres, which in turn have to apply to Konnex Association for certification to ensure an appropriate quality level. Obviously, particularly easy access to all aspects of system configuration (maybe even to end users) is neither necessary nor desired in this concept.

3.1.3 Configuration

For planning an EIB network and properly setting up the participating devices, EIBA[4] maintains and distributes a single, official standard PC software package. This Microsoft Windows-based software is called the *ETS* (EIB Tool Software, Engineering Tool Software). The ETS assists with defining the project in a structured way and is able to configure certified EIB devices of any manufacturer regarding appropriate behaviour and communication relationships. All of this is possible via the network, without the need for a special physical connection to the target device. It also provides bus monitoring functions for troubleshooting.

Compliance with this tool is a certification requirement for EIB (and KNX) devices. Manufacturers are required to supply the necessary device descriptions along with their hardware. Tools for their creation are provided by EIBA as well. The ETS can also be extended by way of plug-ins, should the configuration of a device require it. Only a minor number of devices need further, additional set-up tools. For interacting with such external tools, the ETS supports the export of project files.

[4]Despite the transition to KNX, with Konnex Association taking over all other tasks, EIBA retains responsibility for specification, marketing and sales of the software tool set for organizational reasons. Konnex Association is required to lay the necessary foundation regarding standardization and certification requirements to actively promote the single tool.

Thus, the ETS ensures uniform handling of planning and configuration for every certified EIB component. This vendor-neutral approach is a distinctive feature of EIB. It makes multi-vendor systems usable in practice as it allows an integrator to mix and match products from the entire selection available without incurring excessive effort in setting up the system. Besides this obvious advantage, a vendor-neutral configuration tool also facilitates the standardization of trainings as well as actually holding them. It also introduces a stability factor, offering a predictable long-term perspective.

For all its benefits, the common tool approach definitely involves trade-offs as well. Obviously, it closes the door on competition as a possible source for innovation and improvement. Such improvements may include alternate modes of interaction, like a simplified mode for beginners offering fewer options—and thus fewer possibilities for erroneous configuration. In trying to broaden the market, various manufacturers attempt to fill this gap by offering different proprietary configuration approaches. These allow to circumvent the ETS when restricting the system to the specific manufacturer's subset of EIB devices. Further configuration abilities can still be accessed by using the ETS, if needed.

While one software tool is closely modelled on the ETS, another adopts a more graphical approach with dramatically reduced configuration possibilities. Other solutions entirely do without a PC, delegating configuration to a controller component, which is integrated in a device providing certain runtime functionality. In one case, this is a rail-mount actuator—which as a controller will even automatically detect connected sensors and actuators—, while the other is a simple control desk.

As an aside, it should be noted that due to the very nature of EIB (disregarding alternative configuration approaches) the ETS project file is needed even for minor changes to the configuration, like the switch-on time for a stairway light, adding an additional switch or replacing it in case of failure. Without this data, modifications will incur excessive work or even be impossible. Thus, this project file should be considered an integral part of the system.

3.1.4 Competitors

Of the already limited number of field area network designs addressing the domain of home and building automation, some further confine themselves to specific aspects of the field, like DALI (Digital Addressable Lighting Interface) [24], or only allow systems of very limited size, like X10. When narrowing the

choice to universally applicable peer-to-peer field bus systems comparable to EIB in flexibility and performance, LON and LCN emerge as main contenders.

LON (Local Operating Network) was designed with utmost flexibility in mind. It should be applicable to all kinds of control systems where field bus technology could potentially be of benefit. For example, it offers a comprehensive range of network media, data rates and transmission ranges. On the protocol level, aspects like the selection of services to be supported, the presentation of data or even the address length are free to choose for the system designer. Its communication protocol, called *LonTalk,* is disclosed via United States and European formal standards [3, 18]. To denote the overall solution built from LonTalk, various physical media definitions and the API of standard hardware components, the "LonWorks" trademark is used.

The downside of this flexibility is that two arbitrary LON-based devices are usually far from even being able to exchange data, let alone interpret it in a meaningful way. To ensure interoperability in spite of these numerous degrees of freedom, LonMark profiles prescribe a specific subset of LON technology to be used as well as behavioural aspects. None of these requirements are laid down as a formal standard, however. Under this umbrella, a number of interoperable solutions for building automation have emerged. Still, LON-based solutions generally require higher competence from system integrators. They are therefore considered to be better suited to complex applications where far-reaching control of technical aspects is required. As a result, LON technology is very rarely applied in home environments. Nonetheless, it has definitely gained a foothold in building automation.

In contrast to LON, *LCN* (Local Control Network) was developed with a very specific focus on home and building automation. Consequently, it shares some properties with EIB, like devices being offered in rail-mount or installation socket compatible flush-mount housings. LCN, however, takes a quite different technical approach in places, with the manufacturer frequently highlighting them against (actual or perceived) weaknesses of EIB. Although some are not necessarily the clear advantage that they may seem at first sight, LCN actually possesses some rather interesting features.

One difference, for example, is that it does not split functions into separate modules as it is popular with EIB. Rather, LCN uses monolithic nodes which contain both sensors and actuators as well as logic and timer functions. Such a module can immediately replace a conventional wall switch. However, comparable sensor/actuator combinations are available for EIB as well, even if they are not that popular. On the other hand, LCN uses a free strand in standard

mains distribution cables for communication instead of requiring to lay separate cabling. This is actually especially convenient since four-wire cable is popularly used for installation at any rate. LCN also embeds more information in network telegrams, making ad-hoc modification of actuator behaviour easier.

Nevertheless, assessing the potential of LCN technology is difficult since the specifications are proprietary and not disclosed. LCN also lacks the backing of large component manufacturers. It is practically a single-vendor system.

3.2 Group Communication

EIB group communication allows to pass a piece of information to an arbitrary number of receivers by way of a single message. This is achieved by making use of a *publisher-subscriber* model. The sender uses a logical group address as destination address. Receiving stations know which group (or groups) they belong to, and accordingly either ignore or process incoming messages.

Thus, a sender does not require information which nodes will actually be receivers of its message. Any node can elect to *subscribe* to a group without the *publisher* knowing. Actually, it is neither necessary nor possible for a node to determine which other nodes will act as publishers or subscribers to a specific group address. The knowledge concerning which nodes participate in a certain group communication relationship is distributed over all nodes in the system.

Figure 3.1 illustrates the resulting communication model. It shows how an additional publisher and subscriber can enter a group communication relationship without any modification to the original communication partners.[5] Obviously, the set-up presented would not be possible when using maintained contact switches. Their rocker would remain in position even if the state of the lights was changed using the other switch, creating an inconsistency. Therefore, intelligent building installations generally use momentary contacts in switch sensors to avoid this kind of restriction when influencing the state of an actuator from multiple places. This does not only include obvious cases like corridor lighting, but also every kind of central control functionality.

Now, one might argue that switches need not consist of two separate halves for issuing “on” and “off” commands as they do in the example. A single button toggling the state of the light at every press by alternately transmitting “1” and “0” messages would suffice. Although this is correct, such an

[5] In practice, one usually would not install a second actuator for the sole purpose of controlling two lamps in parallel, but of course connect them both to the power output of the first.

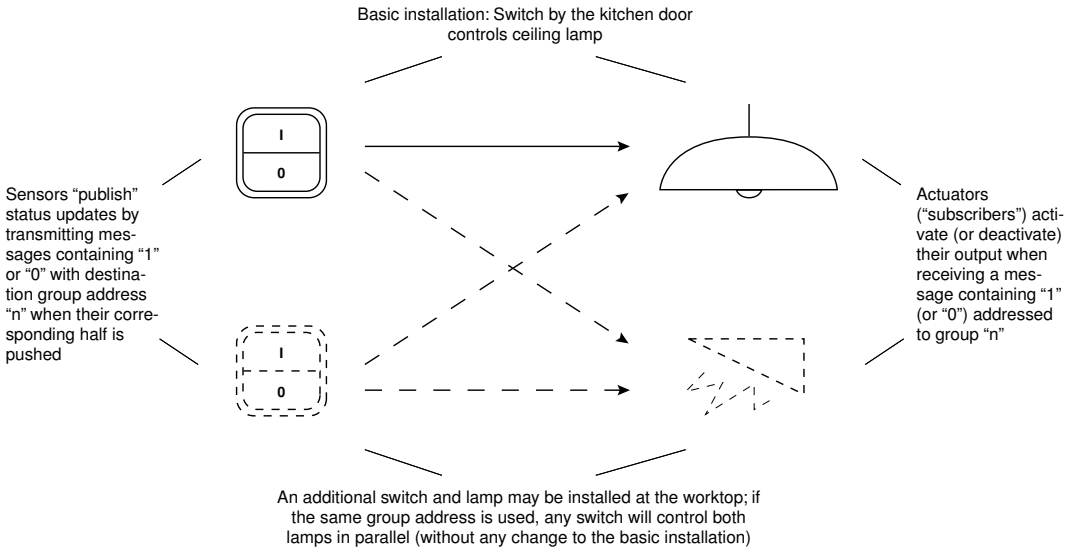


Figure 3.1: EIB Regular Operation: The Publisher/Subscriber Model

approach introduces additional complexity. It requires the switch to keep track of the controlled actuator's state, even if the latter is changed by another node. Otherwise, it would again exhibit inconsistent behaviour, requiring two presses whenever the state it assumes the actuator to be in does not correspond with reality.

In practice, one usually would not install a second actuator for the sole purpose of controlling two lamps in parallel, but connect them both to the power output of the first. Extending the simple example shown in Figure 3.1, an additional requirement shall be added which will put the second actuator to good use: While the switch near the door should only turn on the ceiling lamp, the switch at the worktop should turn on both lights to create ergonomic working conditions. The switch at the worktop should however only turn off the worktop lighting, while its counterpart near the door shall turn off both lights, as it will mostly be used when leaving the kitchen.

Figure 3.2 illustrates how to achieve this. Still, one group containing both actuators is needed, which can be used by the "off" half of the switch by the door and the "on" half of the one at the worktop to switch both lights. In addition, two new groups have to be formed with the respective actuators as sole members. These allow the lights to be turned on and off separately. Each actuator is now member of two logical groups.

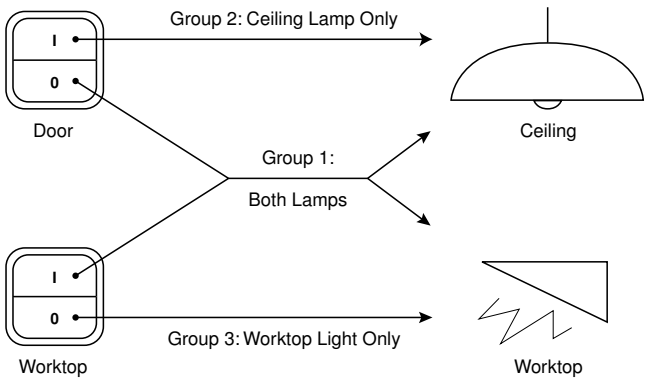


Figure 3.2: EIB Regular Operation: Group Addressing, Advanced Example

Again, the desired effect could also be brought about using single push buttons. This however can definitely be considered an advanced exercise requiring in-depth knowledge of EIB. Problems like this are usually approached using PLCs (Programmable Logic Controllers) with EIB interface.[6]

3.2.1 Group Objects and Shared Variables

EIB uses a *shared variable* model to express the functionality of individual nodes and combine them into a working system.[7] Every device publishes several application related variables which expose specific aspects of its functionality. They will usually either be *data sources* providing information to other devices, or *data sinks* which carry out certain actions according to the information received. Examples for the former would be the position of a switch; for the latter, the control input of a relay. These application related variables are

[6]Instead of using a fully fledged PLC, one can also manage with a simpler variant providing only a predefined set of certain logic functions, known as "logic modules" in the EIB world. The key to the solution lies in the following observations (S_D and S_W being the switches at the door and worktop respectively, L_C the ceiling and L_W the worktop light):

L_W cannot be turned on using S_D
Every time a press of S_D results in turning off L_C, L_W has to be turned off, too

Thus, one can use S_D to toggle the state of L_W and use a logic module to echo all (and only) "off" messages to L_C. The same holds for S_W, vice versa. Another variant would be to make use of the fact that after toggling one light, the other one will always be in a constant state (i. e. L_W always off after pressing S_D, L_C always on after pressing S_W). EIB switches usually can output different commands to different group addresses at different actuation edges. Thus, they could toggle one lamp when being pushed and force the other to its constant state when being released. While this will save the logic module, the lamps will not switch at exactly the same time.

[7]Despite the state-based semantics of this model, the underlying communication is event-based.

referred to as *group objects*.[8] It should be noted that these usually represent a single value only and thus are loosely related at best to the notion of an "object" in object-oriented programming.

Group objects of various devices are grouped at set-up time to form network-wide shared variables. The values of all group objects assigned to the same group will be held consistent by the nodes' system software. Continuing the above example, this would allow the switch to control the relay by linking the state of their respective group objects. Group membership is defined individually for each group object of a node.

Group objects can also belong to multiple groups.[9] This is a key feature of EIB, which allows elegant realization of central functions. By including all light switching actuators in the house in a common group besides their standard control group, a master switch by the door can easily turn them all off when the owner is leaving the house. In this case, however, the illusion of a purely state-based model can no longer be maintained. Since a group object cannot assume two—possibly different—shared values at once, it will keep the state of the last update.

Usually, data sources will actively publish new values, although a query mechanism is provided as well. For this, a group member which is to respond to the query with the value of its associated group object has to be chosen manually. There are also situations where this mechanism cannot be usefully applied. This includes the case of overlapping groups.

No limitations exist concerning the semantics associated with the individual group objects. This binding is entirely within the local responsibility of a node application. Thus, the same message from a master switch may turn off a light as well as cause a sun blind to go down.

Every group of communication objects is assigned a unique group address. This address is used to handle all network traffic pertaining to the shared value. Thanks to the publisher-subscriber principle, the group address is all a node needs to know about its communication partners.

Installations will often run into large numbers of group addresses. Therefore, it is useful to assign these using a scheme indicating the functionality associated. Usual practice is to have the four most significant bits form the *main group*, while the eleven (or alternatively eight) least significant bits describe the

[8] Historically, group objects are also known as "communication objects".

[9] Currently available BCUs allow update notifications to be sent to a single group only. This however does not restrict the expressivity of the model, since one can always form a new group which includes all intended receivers.

sub group. In the case of an eight-bit sub group, the remaining bits form the *middle group*. The group address 32767 therefore reads 15/2047 in two-field or 15/7/255 in three-field notation. This division however is relevant exclusively for visual presentation, not operation. Any field values (or entire group addresses, for that matter) can be chosen freely. Returning to the example shown in Figure 3.2, the three group addresses could share a common main and middle group representing "Lighting" and "Kitchen".

Figure 3.3 illustrates what was discussed so far by showing which steps are involved in turning on a light using EIB. Simple user applications, like those for polling a switch and operating a relay, are usually hosted by the BCU in addition to the standard implementation of the EIB network stack. They can use the BCU's Physical External Interface for exchanging arbitrary signals with the application module.

The look-up process the network stack performs concerning the association of group objects and group addresses is simplified for the purpose of this illustration: Actually, no such integrated lookup table exists, but multiple ones are used, as will be detailed in Section 3.6.

3.2.2 Interworking

For the EIB network stack, the contents of shared variables are opaque octet strings only. Yet obviously, all group objects of a group need to use a common encoding to assure that the shared value will be interpreted in a consistent way.

For this purpose, the *EIB Interworking Standard (EIS)* defines a standardised bit-level syntax for various variable types, including Boolean values, signed and unsigned integers of multiple widths, time/date and floating-point values. At set-up time, the ETS ensures that only group objects of compatible type are combined. For this purpose, EIS also defines code numbers for various physical units.[10] Yet, no information concerning the type or unit of a shared variable is ever exchanged over the network.

EIB devices interact by exchanging trigger information only, mostly Boolean values. The same Boolean value may directly control the state of a relay, but it may be used to trigger a time delay switch as well—without the publisher knowing. Instead of being encoded in control messages, most of the behaviour of a device is defined at set-up time (using the ETS) and stored in non-volatile memory.

[10] Partially, these numbers also cover more than the mere unit, like "wind speed" (or direction) instead of simply km/h or degrees, but this is not done on a systematic basis.

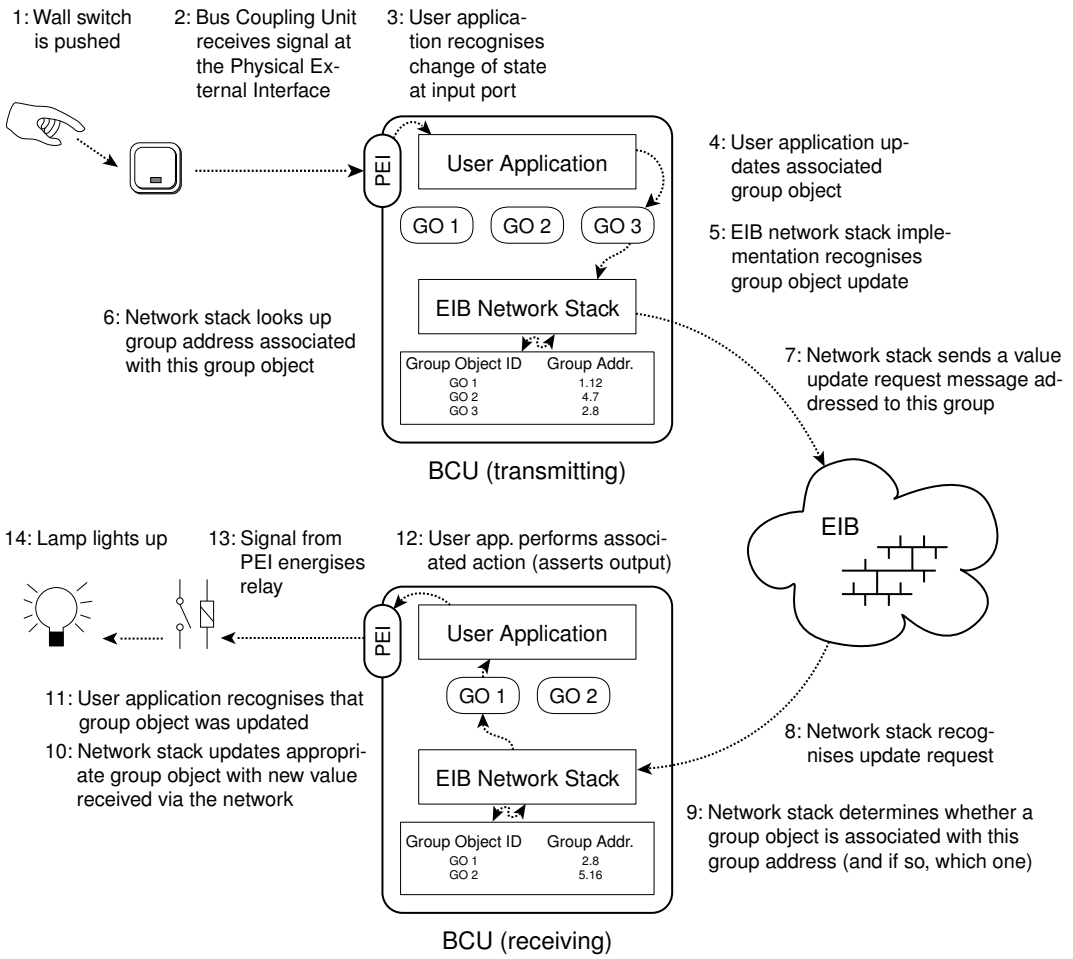


Figure 3.3: EIB Regular Operation: End-to-End Overview

For illustration, Figure 3.4 shows how this is done for a switching actuator. The top level unit governing device functionality is the *application.* Available parameters and group objects depend on this choice. *Application parameters* are used to fine-tune the desired behaviour. Only values which are expected to change frequently are made available for run-time exchange via group objects, which in this example are all of Boolean type.

By concentrating on the average case, this approach helps to keep network traffic low at the expense of ruling out dynamic change of less-often used parameters. It also keeps the interface between applications simple, increasing the likeliness that any two devices will be able to exchange data. Together with the fact that individual group objects can be bound freely, it offers high flexibility. Yet, it makes EIB rely on qualified integrators to build working solutions by manually selecting and matching appropriate behaviour.

Application Program	Parameters	Group Objects
Switch	Logic Operation (None/AND/OR) Contacting Behaviour (Make/Break) Power-on Behaviour (Off/On/Last State)	Switch (in) Logic Input (in) Feedback (out)
Stairway Lighting Switch	Logic Operation (None/AND/OR) Contacting Behaviour (Make/Break) Stairway Lighting Duration (minutes)	Switch (in) Logic Input (in) Feedback (out)
Delayed Switching	Logic Operation (None/AND/OR) Contacting Behaviour (Make/Break) Switch-On Delay (minutes) Release Delay (minutes)	Switch (in) Logic Input (in) Feedback (out)

Figure 3.4: EIB Example Device: Switching Actuator

For the most part, EIS does not standardise any behavioural aspects. It does, however, define profiles (referred to as "types") for the control of actuators for light dimmers and sun blind drives. Their definitions each describe a set of interacting shared variables, called *sub functions.* For "Dimming," these are

Position: A Boolean value; turns off the dimming actuator or sets it to full brightness

Value: A 8-bit unsigned integer value; sets the actuator to the given output level (0 . . . off, 255 . . . full brightness)

Control: A signed integer value ($-7 \ldots +7$); every message increments or decrements the output set point (also an 8-bit unsigned integer) by $2^{|n|+1}$. The actual output value is to approach the set point at a rate of 64 units/s. A zero control message will stop such a transition at the currently present value.

Devices implementing this profile will usually also allow the integrator to define differing behaviour, for example to assume the last brightness value when being switched off and on again instead of going to full brightness.

The EIS type "Drive Control" has the following two sub functions, both of which are based on Boolean values:

Move: Sets the blind into upward or downward motion (respectively causes it to open or close, if it operates in horizontal direction).

Step: While the blind is moving, any message shall cause it to stop; after being stopped, messages shall cause short-timed gradual upward or downward movement. This type of movement is usually associated with adjusting the slat angle.

Rather than actually modelling the behaviour of a drive actuator, this profile seems to have been created with the intent of supporting the popular behaviour of wall switches for sun blind control. These often consist of a double throw rocker and will set the sun blind into opening or closing motion after a long press. Any short press will stop the motion. A variant is to have the blind move only while the rocker is being held down and stop as soon as it is released. In either case, further short presses will adjust the slat angle. The two sub functions correspond immediately to these long and short actuations.

Contrary to the general high-level communication model of EIB, the Control and Step sub functions do not follow state-based semantics. With these sub functions, actuator behaviour is not associated with the current value of a shared variable, but with the individual messages otherwise used for delivering value updates. In this *event-based* model of communication, two (or more) subsequent updates to a group object containing the same value are no longer automatically idempotent.[11]

There is no obvious, compelling reason for this decision. For example, the command for relative set point change provided by the Control sub function could be replaced by an "in-motion" state (possibly even including the desired dimming speed) comparable to the Move sub function. It can be assumed that an event-based approach simply seemed more convenient to the designers of these profiles.

With one exception (which will be detailed below), these profiles do not provide for retrieving status information. Unlike with a simple switching actuator—without time delay functionality—, querying the inputs will not help either. With the command-oriented sub functions, there simply is no state which could be queried. The drive Move sub function does not provide very much interesting information. It will only reveal the last direction the blinds were moving in; they may have reached the end position or may have been stopped in midway. The group object associated with the Value sub function is not required to be updated when the output level changes.

The exception mentioned is the Position sub function. The group object associated with this sub function is required to change its state to "off" when

[11] They still can be, as in the case of multiple Control stop messages; but this depends on the semantics of the individual control event.

the actuator output level goes to zero in response to control activity on the other sub functions. Likewise, it shall change to "on" on every transition from zero to a non-zero brightness. This status is not only intended for query purposes, but also for active transmission on every such change.

While this seems appealing at first sight, such a bidirectional communication object requires especially careful attention when designing the communication relationships of an EIB system. As an example, consider a lecture hall equipped with separate lights for the lecturer's desk and the auditorium. These are to be dimmed independently. In addition, a master switch shall be provided to quickly turn on or off both lights. This can be achieved in a straightforward manner by forming three groups. While two of them connect the Control sub function of the individual dimming actuators to the switches controlling them, the third includes both Position sub functions and the master switch.

Now assume a pilot light showing the state of the desk light is to be added near the master switch since it is hard to see whether it was left on from this position on a bright day. Simply allowing the desk light dimming actuator to transmit its Position status and include the pilot light with the master switch group may seem the obvious solution at first sight. Yet, this way the status information will not only control the pilot light, but the auditorium light as well. The latter will, for example, be forced to full brightness whenever the desk light leaves the "off" state. Thus, a separate group has to be formed.

Very probably to reduce the potential for such errors, the EIS suggests to keep group objects unidirectional (i. e. either data sources or sinks) and provide separate group objects for any status output. It is likely that the design of the EIS dimming profile preceded the inclusion of this recommendation.

3.3 The OSI Reference Model

Since sending messages over a network is a complex issue, it is useful to break it up into a hierarchy of multiple layers providing subsequent abstractions. The OSI (Open Systems Interconnection) Reference Model [25] was developed by ISO in the late 1970s to provide a guiding principle for developers, the intention being that designing communication systems according to a common model should facilitate data exchange between them. Although it is heavily influenced by the (then dominating) structure of telephone networks and does not map well to today's computer networks and programming practices in many aspects, it is still a valuable and accepted reference for discussing the structure

of a communications system. Nevertheless, it is common consent that the OSI model should not be taken as a straight-away model for implementation. Since EIB is aligned with the OSI model, this section will introduce some of its key concepts and terminology for use in the following sections covering the EIB protocol. For a more detailed discussion, see (for example) [44].

As is widely known, the OSI reference model uses seven layers, summarized here in bottom-up order:

- The purpose of the *Physical layer* (1) is to transfer single bits over the communication medium. Here, the physical properties of the communication medium (including connectors) and signalling method (for example, voltage levels) are of interest.
- The *Data Link layer* (2) provides point-to-point connections between two stations and is concerned with issues like medium access control.
- The *Network layer* (3) handles routing. Its duty is to determine which of the potentially many ways possible telegrams should take through the network to reach their destination.
- The *Transport layer* (4) is responsible for providing reliable end-to-end connections between sender and receiver.
- The *Session layer* (5) is associated with issues like user identification and authentication and the management of the possibly multiple parallel or subsequent connections associated with a communications session.
- The *Presentation layer* (6) handles issues of data representation and encoding.
- Finally, the *Application layer* (7) offers services of immediate use to the application. Depending on the envisioned use of the communication system, this could be file transfer, electronic mail, turning on and off a light, or modifying a piece of virtual shared memory.

Layers 5 and 6 did not become very popular with network designers for various reasons. For EIB (as with many others), they are left empty. *Repeaters, bridges, routers* and *gateways* connect network segments or different networks at the Physical, Data Link, Network and Application layer, respectively.

Every layer provides a precisely defined set of *services* to the one above it. To implement them, it will in turn make use of the services of the layer below. This way of communication defines the "physical" path data units take

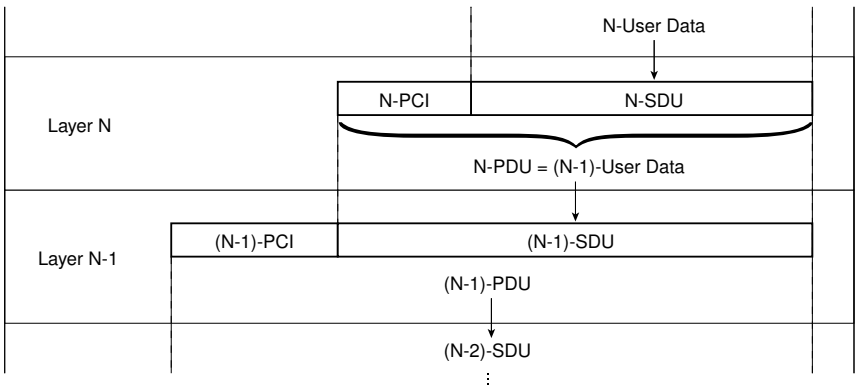


Figure 3.5: The OSI Model: Data Units

and only takes place between adjacent layers. Conceptually, however, any layer communicates with its *peer layer* in the remote communication partner only. The particular *protocol* they use is not of concern to layers above and below.[12] Thus, communication systems built according to this model will contain a "stack" of independent protocols. The popular terms *protocol stack* and *network stack* have their origin here.

The data units exchanged at layer N are referred to as *(N)-protocol data units* or *(N)-PDUs.* They contain the *protocol control information (PCI)* needed by the peer layers to contain their joint operation. In most cases, they will also contain *user data* transmitted on behalf of the higher layer (or application). The PCI is usually attached to these *service data units (SDUs)* as a header, or sometimes also trailer. PDUs are exchanged with the peer layer using the services of the next lower layer (N-1), which for its purposes will regard them as SDUs. Figure 3.5 illustrates this process. Layer 2, 3, 4 and 7 PDUs are usually referred to as Frames, Packets, Telegrams and Messages, respectively.

Service users and service providers (neighbouring higher and lower layers, respectively) interact by exchanging service *primitives.* These are:

Request (.req), made by the service user to invoke a function,

Indication (.ind), issued by the service provider to indicate an event, possibly a remote request,

Response (.res), sent by the service user to answer a remote request, and

Confirmation (.con), returned to a service user upon receipt of a response.

[12] The OSI reference model does not define any protocols. While such were published as separate standards, they have never taken hold.

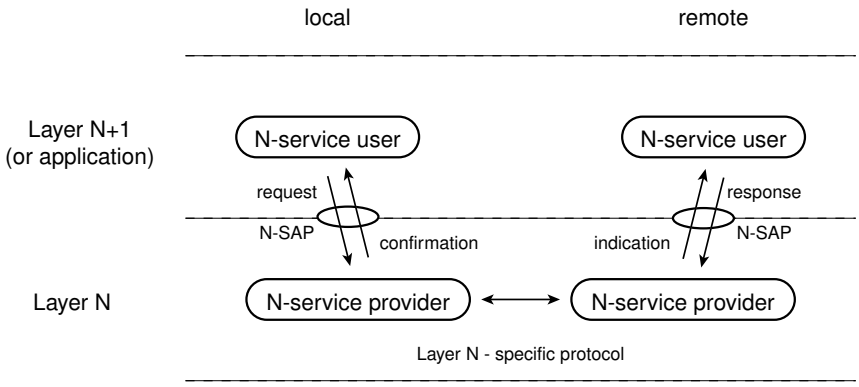


Figure 3.6: The OSI Model: Service Primitives

When a service involves the exchange of all four primitives, as shown in Figure 3.6, it is referred to as a *confirmed service* in OSI terms. A service which does not include a response from the remote service user (and consequently, no confirmation), is called an *unconfirmed service.*

The EIB specification extends this scheme by introducing additional types of confirmations. In addition to a response from the remote service user, these can be based on a response of the remote peer layer or simply the local lower layer confirming the successful execution of the request (called a *local* confirmation). The service user will receive a confirmation after *every* request. Moreover, a local confirmation for responses is added. Unfortunately, the specification documents do not refer to the resulting possible modes of communication in a consistent manner.

Primitives are invoked through uniquely identifiable *service access points (SAPs).* A service can provide any number of SAPs. The SAP it is issued at is implicitly associated with a primitive. This concept becomes clearer when thinking of SAPs as phone sockets of a private branch exchange, which are associated with specific extension numbers.

3.4 Twisted-Pair EIB

The mainstay physical medium of EIB is twisted-pair (TP) cabling. A single wire pair (2 x 0.8 mm diameter) is used for both data and power transmission. Although recommended, shielding is not mandatory. Proper sheathing and resistance against mechanical and thermal stress is required for EIB certified cables to allow them to be threaded into pre-laid conduiting, in immediate

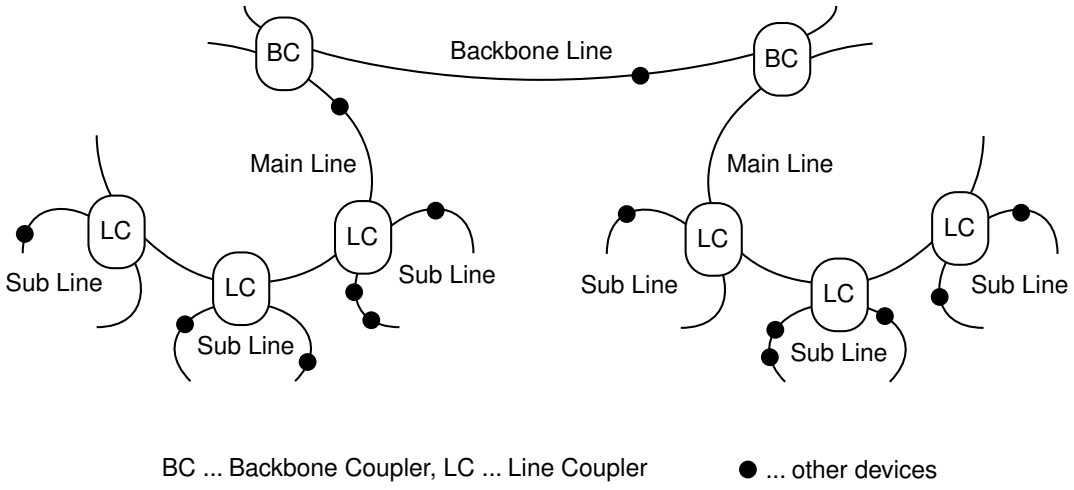


Figure 3.7: EIB Network Topology

vicinity of the mains wiring. Still, SELV (Safety Extra-Low Voltage) conditions are maintained on the EIB network itself.

3.4.1 Topology

The physical segments of an EIB network are called *lines.* A line can accommodate up to 256 devices in free topology. Loops are permissible, but should be avoided. Terminating resistors are not required. The maximum allowable accumulated cable length of a line is 1,000 m. Since EIB networks provide link power, one *bus power supply unit (BPSU)* per line is required. BPSUs contain a 30 V DC supply and a choke for signal shaping.

Since the original EIB specification (referred to as *TP64*) allowed only 64 devices per electrical segment, devices with two segment connections forwarding all messages in both directions were introduced. Thus, a line would be made up of up to four electrical segments. Although more of a bridge in OSI terminology since they buffer telegrams, these devices were termed *line repeaters.* With the current *TP256* specification, lines coincide with electrical segments. Line repeaters can still be useful in special cases to extend the overall length of a line up to 4,000 m or even further if non-standard—yet compatible—solutions like fibre optic transmission are used, which are offered by certain manufacturers.

Lines can be connected by routers (referred to as *couplers*) to form a tree structure, as illustrated in Figure 3.7. Up to 15 lines (in this case frequently referred to as sub lines) can be connected by a *main line* to form a *zone.* A maximum of 15 zones can in turn be coupled by a *backbone line.* No loops are

allowed.[13] Usually, levels of this hierarchy will be matched to the division of a building (or complex of buildings) into its structural elements like individual buildings, staircases, floors, offices, apartments or rooms. Overall, an EIB installation can accommodate more than 60,000 end devices.

Couplers will only forward messages which would not reach their destination otherwise. Thus, locality of information is exploited to reduce the overall load on the network by segmentation. This is not sufficient, however, if central monitoring and control of the entire installation is desired. In this case, the amount of traffic flowing to (and possibly from) the monitor station will quickly exceed the load limit of the EIB network on the backbone line. The obvious solution is to switch to a higher-speed backbone connection. For this purpose, methods for tunnelling of EIB data frames over Ethernet *(EIBnet)* [19] and arbitrary networks using the Internet Protocol *(EIBnet/IP)* are included with the EIB specifications.

3.4.2 Standard Message Cycle

Stations share the communication medium by way of asynchronous time division multiplexing. The contention protocol used is CSMA (Carrier Sense/ Multiple Access) with bit-wise arbitration.

EIB employs balanced baseband signal coding. A logical "0" is encoded as a negative voltage imprinted on the DC supply voltage, created by a current pulse within the sender. It is always followed by a positive compensation pulse supplied by the BPSU choke, yielding a constant component free signal cycle overall. A logical "1" is identical with the idle state of the transmission medium. Thus, the transmission of a logical "0" will always dominate the medium.

A sending station will always examine the state of the bus line in parallel (listen-while-talk). In case it encounters a "0" value it did not transmit, it assumes a collision and cancels the transmission. The other station can continue its transmission undisturbed.

Data are transmitted with a rate of 9.6 kBit/s, yielding a bit time of 104 μs. Figure 3.8 illustrates the timing of the Layer 2 message cycle. Data frames are of variable length. After a pause of 15 bit times, the receiving node responds with an acknowledgement frame. It consists of a single character containing information whether reception was successful, erroneous or unsuccessful since

[13] Strictly speaking, though, the hierarchy is only tree-like since sub lines may be attached to the backbone directly. This rather odd feature should not be used in practice and has been omitted from the illustration.

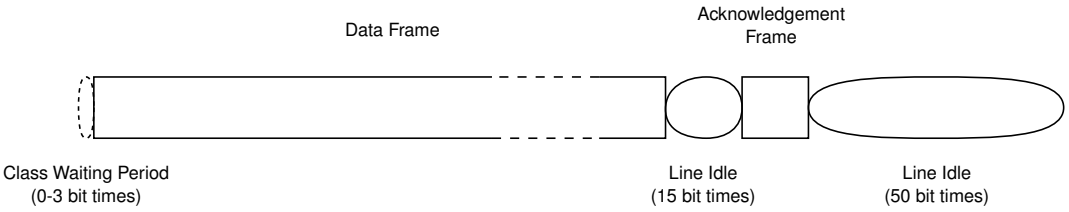


Figure 3.8: Medium Access Timing for Twisted-Pair EIB

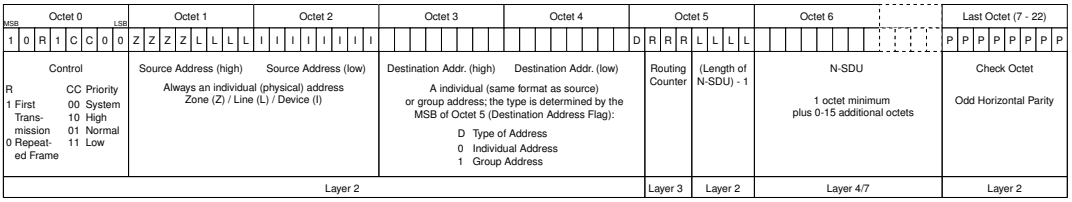


Figure 3.9: EIB Standard Data Frame

the receiver was too busy to process it. Before any station may attempt to transmit another frame, the line has to be kept idle for at least 50 bit times.

Stations intending to send a message with standard or low priority have to wait for three additional bit times, giving precedence to messages with system or high priority and frames repeated due to negative or missing acknowledgement. In the latter case, it is usual practice to re-send the frame up to three times. Acknowledgement frames (also referred to as *immediate acknowledgement*) are associated with the Data Link layer and significant for communication on the same electrical segment only. Couplers will therefore positively acknowledge every frame they pass on, buffer it and autonomously repeat its transmission on the other interface, if necessary.

3.4.3 Format of the Standard Data Frame

Twisted-Pair EIB makes use of character-oriented asynchronous start-stop signalling. Characters contain eight data bits, even parity and one stop bit (8e1) and are always followed by two idle bit times.

The format of the standard data frame is shown in Figure 3.9. The character corresponding to octet 0 is sent first (corresponding to the direction of reading). Within every octet, however, individual bits are sent in ascending order of significance, i. e. from right to left when looking at the illustration. The illustration also specifies the OSI layers associated with the functionality supported by its various fields.

Octet 0 contains the frame *priority,* which was already mentioned. Priority bit patterns are chosen so that frames with higher priority will win the arbitration cycle over ones with lower priority. The *repeat flag* marks frames which are retransmitted due to the sender not receiving a positive acknowledgement. Without this information, a receiver could not tell a repeated message from a mere duplicate in case it received both messages intact. This can, for example, happen when the acknowledgement frame is destroyed.

EIB uses two distinct types of station addresses, individual (also referred to as physical) and group. The frame source address will always be an individual address, while its destination may be specified as an individual or group address.

The *individual address* of an EIB node is closely related to its position within the topological structure of the network. It specifies the number of the zone and (sub-)line the device resides in as well as its device number within the line. Zero values in any of these fields designate special entities: Line couplers (device number is zero), main lines (line number is zero), backbone couplers (line and device number are zero) and the backbone line (zone and line numbers are zero).

Since individual addresses are unique within an installation, any collisions will be resolved after the transmission of the frame source address is completed. The individual addressing scheme is maintained on other physical media supported by EIB as well to ensure a consistent topological view. Especially on open media like Power Line, however, this topology will only be logical instead of both logical and physical as with twisted-pair cabling.

Group addresses are purely logical identifiers. Stations privately maintain a list of groups they are members of, comparing destination addresses of incoming frames against it to determine whether they are addressed. Every device is member of group 0, which is used for broadcast messages. In case of a group as destination address, multiple acknowledgement frames may be returned. Since they will be superimposed on one another, acknowledgement character bit patterns are chosen specifically to ensure negative acknowledgements will overwrite positive ones. Identification of duplicate frames by way of the repeat flag is especially important in this case, since a single addressee (which may have been busy) returning a non-positive acknowledgement will cause the sender to retransmit. Yet, all others may have received the frame in perfect condition.

For frames with an individual address as destination, couplers can immediately determine whether an incoming frame is to be forwarded to the other line by comparing their own individual address with the frame destination. For

frames addressed to a group, they maintain a routing table which is automatically created by the ETS.[14]

The *routing counter* ensures that a frame will appear in seven physical segments at maximum to avoid frames circulating endlessly due to errors in the network set-up. Its initial value (usually six) will be decremented by every coupler (and line repeater) passing on the frame. Frames whose routing counter is at zero will not leave the segment. The special value of seven is never decremented, allowing to disable this mechanism for diagnostic purposes. Apart from properly initializing the routing counter, the OSI network layer is empty for EIB end devices (i. e. devices other than couplers and line repeaters).

Finally, the length field specifies the size of the Network Layer Service Data Unit (N-SDU), which can be one to 16 octets. The frame is concluded by the check octet, which contains an odd horizontal parity value calculated over all preceding frame octets. Together with the even vertical parity information contained within the protocol characters it enables the receiver to perform cross parity checking. This way, arbitrary double-bit errors can be detected. In principle, correction of single-bit errors is also possible, but not performed by current communication controllers.

3.4.4 Other Frame Formats

The standard message frame presented is, although by far the most frequent, not the only format possible. All formats use the same control octet, but with different values for the two most significant bits. "00" at this position identifies an extended data frame, which extends the maximum permissible frame length beyond the 23 character limit and contains additional type bits for future protocol extensions.

Poll requests ("11") allow a station to request status data from up to 15 other nodes with minimal protocol overhead in master/slave fashion. All participating devices have to share the same physical segment, and the data returned are limited to one character each. Responses have to be sent in a narrow time slice following the request frame (the message cycle can be roughly likened to a standard data frame with multiple subsequent acknowledgement characters). Every slave station is assigned a constant, specific position in the answer se-

[14] This table cannot be built automatically, since no service exists which would allow to determine the individual addresses of group members. No services using group addressing are acknowledged on higher layers than Layer 2, which would be the prerequisite for an "auto-learning" router. Given the static structure of EIB communication relationships, there also is no need for such a function.

quence ("slot number"). The master station inserts fill characters for slaves which fail to respond to ensure synchronisation. Polling groups are especially useful for high-frequency liveness checking of sensors in security applications, and are only supported on twisted-pair EIB.

3.5 Other Physical Media

Powerline (PL) EIB uses the existing 230 V mains wiring as its transmission medium. Data are transmitted on the phase and neutral conductors using spread frequency shift keying (SFSK) with a middle frequency of 110 kHz. A logical "0" is encoded by injecting a 105.6 kHz sine wave, while a frequency of 115.2 kHz corresponds to a logical "1". Data Link layer octets are transmitted as 12-bit characters, with the additional four bits being utilised for error correction. The data rate used is 1200 bits/s.

Since senders cannot dominate the medium, access conflicts cannot be resolved during transmission. Thus, EIB-PL uses a CSMA/CA (Carrier Sense/ Multiple Access with Collision Avoidance) mechanism which strives to minimise the chance for collisions from the outset. This is achieved by requiring stations to obey additional graduated, randomly chosen waiting periods before starting transmission.

The frame format adds some medium specific extensions to the TP frame. The latter is prepended with a 4-bit training sequence for input gain control and a 16-bit preamble necessary for medium arbitration. The TP frame is then followed by a 16-bit domain address, the upper 8 bits of which are reserved.

The domain address (also referred to as system ID or installation ID) allows to subdivide the physical network—i. e. the mains cabling—into logically independent medium instances. Ideally, however, it should only be used in addition to physical separation by band-stop filters for privacy and performance reasons. Other than that, the addressing scheme is identical with twisted-pair EIB. Zone and line number of individual addresses are usually referred to en bloc as the *subnetwork address.*

With EIB-TP, multiple acknowledgement frames returned in response to a data frame using group addressing are superimposed onto one another. This is not possible on the powerline medium. Therefore, a single *group responder* has to be selected during set-up for providing the immediate acknowledgement.[15]

[15] Note that this setting only concerns the Data Link layer. It is not related to the selection of appropriate responders to queries for the state of shared variables.

For installation of EIB-PL on a triple phase mains network it has to be ensured that data frames can also be received on devices connected to another phase. This can be achieved either via a passive element (phase coupler) or an active repeater. The EIB-PL repeater will echo frames which are not followed by an immediate acknowledgement to all three phases. Since this will include the one where the transmission originally originated, it has to be considered in the message cycle. In an installation containing a repeater, transmitting stations do not automatically resend frames for which they receive no Layer 2 acknowledgement, but first wait for the repeater to do so.

When powerline is used as a supporting medium in a twisted-pair EIB installation, PL subnetworks are usually included at the level of sub lines, with TP being used for the upper hierarchy levels. In this case, it is useful to separate PL subnetworks with band stop filters for performance reasons even if they share the same domain address.

Besides powerline, *radio frequency (RF)* signalling can also be used as an alternative transmission medium. The first EIB-RF design was completed and ready for production at the turn of the millennium, but was never standardised since the responsible company decided to withdraw from the field of home and building automation. Today, it is of historical interest only. Since 2003, the KNX standard includes a completely new design for the RF medium, which can also be used to augment existing EIB systems. It uses an extended addressing scheme which differs significantly from twisted-pair EIB, but an automatic translation scheme is provided for integration. This standard also provides for unidirectional devices to enable the construction of low-cost, battery powered sensors.

Regarding standardisation of the use of *infrared (IR)* light as a physical medium, only a draft document exists. Besides providing a bidirectional mode (which relays standard TP frames over an IR link with minimal modifications), it specifically addresses the needs of remote control handsets.

For both RF and IR transmission, various proprietary solutions offering tight integration with EIB are readily available. It should however be noted that all these additional physical media, even if they allow devices to connect without explicitly setting up a physical connection, will not make EIB a “plug and play” system since it still lacks the necessary protocols.

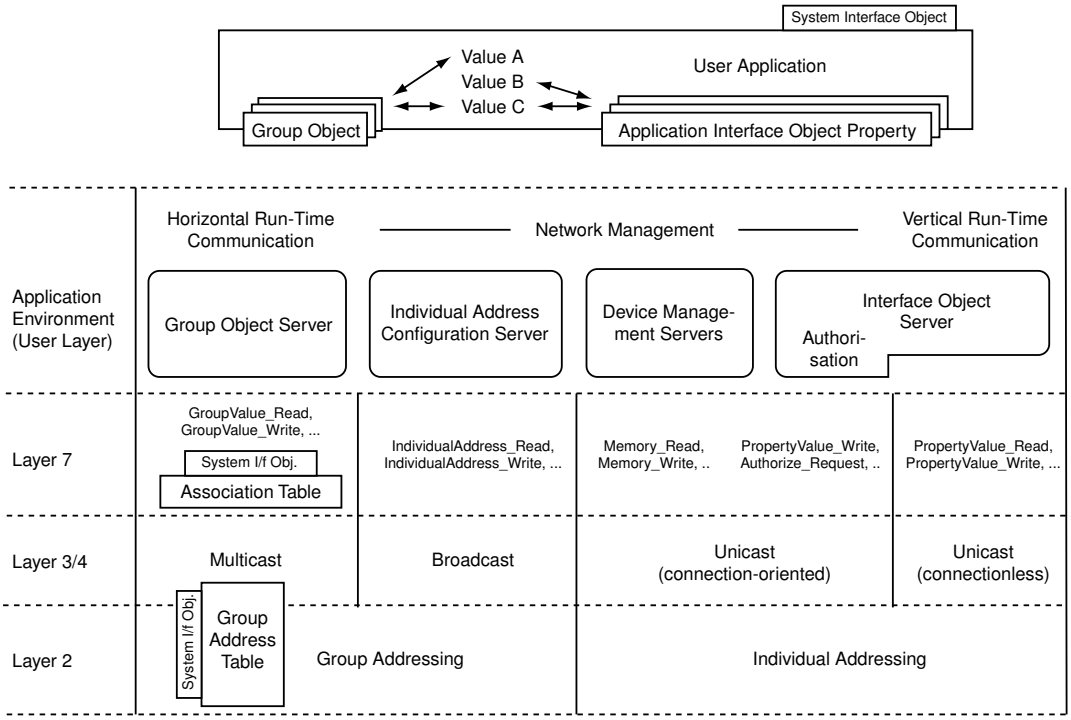


Figure 3.10: EIB Network Stack and Application Environment

3.6 Medium Independent Layers

The EIB protocol stack is aligned with the OSI model. Its structure is illustrated in Figure 3.10. The Session and Presentation layers are left empty, as it is frequently the case with field area networks. Beyond the OSI Application layer, EIB offers a standardised application environment, referred to as "User Layer".

3.6.1 Overview

The design of EIB is based on the assumption that regular operation will frequently include addressing groups of communications partners simultaneously. Consequently, logical group addressing based on the publisher-subscriber principle is provided on the lowest layer possible, making it a native protocol feature. Medium access control and frame format on twisted-pair cabling as well as the publisher-subscriber principle have already been introduced in Sections 3.2 and 3.4.

The necessary table of group addresses a node is subscribed to is maintained by the *Data Link layer* as well. Incoming frames will be passed to upper layers

when their destination address appears in this table or matches the individual address of the node.

The *Transport layer* provides broadcast, multicast and unicast unreliable datagram services as well as reliable point-to-point connections. Multicast and broadcast services are implemented on top of the group addressing facility of Layer 2, while the remaining services use individual addressing. All datagram services rely on the best-effort semantics offered by Layer 2 and are restricted by its maximum frame length.

While Layers 1 to 4 implement protocol functionality, the purpose of the Application layer is to define end-to-end semantics of services for maintenance and regular operation, which will be detailed in the following passages. Service requests by and responses from the Layer 7 user are mapped to Protocol Data Units and passed to the peer Application layer of the receiver by way of appropriate Transport layer services. Likewise, incoming PDUs are mapped to appropriate indications and confirmations.

The *User layer* places an additional level of abstraction between Layer 7 and the user application. On the one hand, it eases the handling of shared variables, allowing the user application to use them in a way similar a local variables, while being notified on external updates. On the other hand, it autonomously handles incoming network management calls, allowing application programmers to concentrate on the actual application. Conceptually, this functionality is broken up into a number of *Server* entities.

3.6.2 Horizontal Run-Time Communication

This part of the Layer 7 provides two services for the communication between sensors and actuators according to the shared variable model (as described in Section 3.2). They use the Transport layer multicast facility (which in turn is based on Layer 2 group addressing).

`GroupValue_Read` is a confirmed service to retrieve the current value of a shared variable from the network. `GroupValue_Write` is an unconfirmed service to indicate to group members that the value of a shared variable has changed and provide them with the new value.

Application layer primitives refer to group objects only, which therefore act as Application Layer Service Access Points (A-SAP). The association between

group objects and group addresses is maintained by Layer 7.[16] This clearly separates network management issues (i. e. communication relationships) from application functionality. In the following, the service primitives involved are introduced.

`GroupValue_Write.req` causes the network stack to transmit the value associated with the A-SAP for perusal by interested subscribers. It is usually invoked whenever this value has changed.

`GroupValue_Write.ind` informs the Layer 7 user that a value update for a certain A-SAP was received.

`GroupValue_Read.req` causes the network stack to transmit a poll request for the value associated with an A-SAP. Any number of responses (including none) can be expected. These may also contain differing values.[17] The system integrator is responsible for ensuring useful responses by selecting a single station—which can be assumed to be in possession of the correct value—to answer. For this purpose, Layer 7 users may ignore `GroupValue_Read` indications.

`GroupValue_Read.con` informs the Layer 7 user about the value associated with an A-SAP as seen by another group member. It corresponds to `GroupValue_Write.ind`, but does not imply that new information is available. It The `GroupValue_Read.req` which caused the activation of this primitive need not have been issued locally.

`GroupValue_Read.ind` informs the Layer 7 user that a request for the current value associated with an A-SAP was made by another node. It is free to answer (or ignore) the request.

`GroupValue_Read.res` corresponds to `GroupValue_Write.req`, but is only sent in response to `GroupValue_Read.ind`.

3.6.3 Network Management

Commonly, the term *Network Management* is used to describe the making of all necessary provisions for transmitting data actually useful from a user's

[16] Actually, the Application layer does not directly operate on group addresses, but on Transport layer SAPs. These identifiers are mapped one-to-one to group addresses by Layer 4. This additional level of indirection is necessary for implementation reasons only.

[17] This is due to the fact that group objects may be associated with multiple groups, as discussed in Section 3.2.

perspective. In the EIB management model[18], a distinction is made between management procedures whose semantics are sufficiently defined by Application layer services and others which demand specific knowledge about the internal structure of the device being managed. The latter are referred to as *Device Management* functions. Implementation independent network management is effectively restricted to the assignment of individual addresses.

`IndividualAddress_Write` allows to set the individual address of a node. The desired value is transmitted via broadcast. Usually, the node to be affected is designated manually by pressing a button on this device, which will cause it to enter so-called *programming mode*. Devices not in programming mode will ignore the telegram. A variant of this service identifies the target node by including its unique serial number in the telegram. Corresponding services exist to read out the individual address of devices placed in programming mode or possessing a certain serial number.

The purpose of `Memory_Read` and `Memory_Write` (on unicast connection-oriented communication relationships) is to read respectively to modify the content of memory locations within the communication controller. `DeviceDescriptor_Read` provides a unique identifier for the implementation structure of a target device, which specifically includes memory locations of management resources like the group address table.

Optional corresponding services (`UserMemory_Read` and `_Write`) exist to access memory in an external application processor in cases where the user application is not hosted by the BCU. By way of these services, a logical address space of 64 KB can be addressed. Mapping of these logical to actual memory locations is within the responsibility of the application processor. `UserManufacturerInfo_Read` serves to identify the associated memory map in this case.

Traditionally, management resources were solely accessed by directly writing to the memory of the communication controller using these services. On current communication controllers, these resources can be accessed in a more structured way through the use of interface objects.

3.6.4 Interface Objects

EIB interface objects serve to provide access to management resources in an implementation-independent way. Instead of requiring data points (called *prop-*

[18] Although being aligned with the OSI model, EIB—like all networks in widespread use today—does not attempt to implement the OSI network management model.

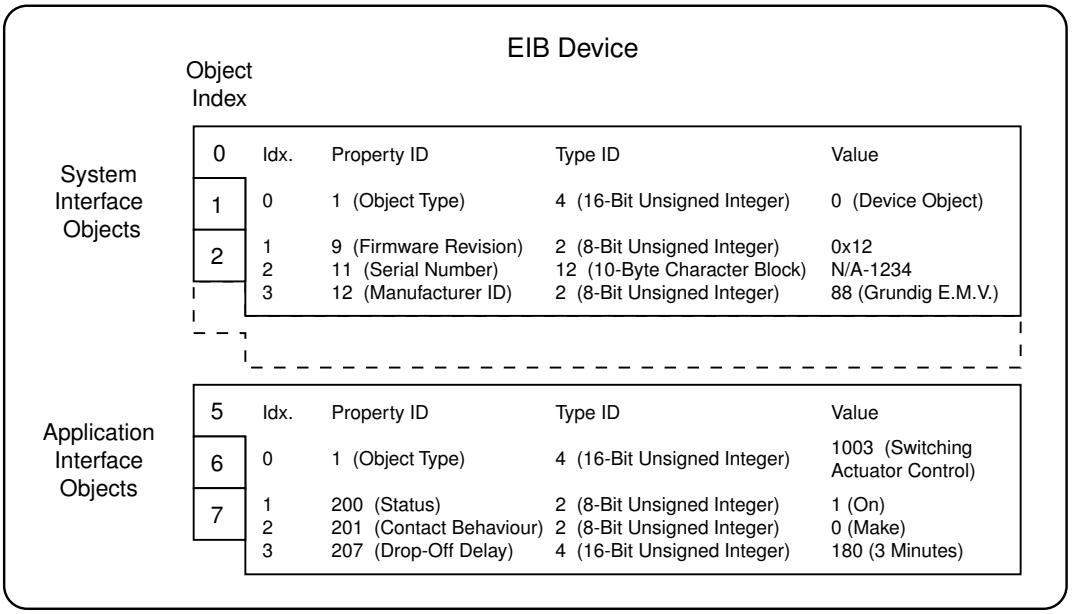


Figure 3.11: EIB Interface Objects

erties) to reside at specific memory locations, it allows a *property client* to refer to them via standardised IDs. This will typically be a PC based tool or controller, but could as well be the local user application. Properties of related functionality are grouped into objects. The resulting structure is shown in Figure 3.11.[19] Every property ID implies certain semantics associated with the property value. For example, property ID "12" holds the manufacturer of the device. This information is encoded as an 8-bit unsigned integer, which is uniquely associated with the name of the manufacturer.

The value of a property is retrieved by passing its object index and property ID to the `PropertyValue_Read` service. To determine which interface objects are available, the property client iterates through object indices, retrieving the object type by requesting the value of the property with the well-known ID 1. While the meaning of property ID 1 is (obviously) independent of the object type, other property IDs may be unique to a specific object type.

For every object type, a basic set of mandatory properties is specified, which the property client can immediately access by stating their ID. Modifications are possible using the `PropertyValue_Write` service, which works in analogy to `PropertyValue_Read`.

For information about additional properties available, the property client can step through property indices, passing them to the `PropertyDescrip-`

[19]Note that property and type IDs shown are partially fictitious and for illustration only.

`tion_Read` service. Meta data returned include both the property ID and a type identifier which allows to correctly display the value data even if the property ID is unknown. This self-description mechanism reduces the a priori knowledge necessary to interact with a yet unknown device and thus significantly improves ad-hoc management.

Access to properties can be secured by assigning them access levels. Every access level is associated with a single, static 32-bit secret key, which has to be presented by a property client before access is granted. In this case, connection oriented access is necessary while otherwise properties can be accessed on connectionless communication relationships as well.

System interface objects are related to system management and include the Device Object—holding general information like the device serial number—, the Address Table Object, the Association Table Object and the Application Program Object.

In addition, every device can provide any number of *application interface objects* related to the behaviour of the user application. On the one hand, their properties can hold application parameters that are normally modified during set-up time only. On the other hand, they can contain run-time values normally accessed through group objects.

Although this facility could in principle also be used for horizontal communication (i. e. between sensors and actuators), no situation is conceivable where this would be of benefit. Instead, this mode of communication is intended for *vertical* access. For example, it allows a central monitoring station to retrieve parts of the process image spontaneously. When using group objects, this demand would have to be prepared at set-up time by establishing a proper group communication relationship.

3.7 Node Development

Standardised communication controllers—BCUs—have a key role in enabling uniform handling of devices by ETS. Therefore, they are highly relevant to system developers as well.

When targeting the development of a new field device (sensor or actuator), BCUs and *BIMs* (Bus Interface Modules) provide a convenient platform. They implement the network stack including the User Layer and can also co-host simple user applications. This allows to create new applications without getting involved with the details of network communication more than absolutely

necessary. Both commercial and open-source IDEs (Integrated Development Environments) are available.

BCUs and BIMs are based on MC68HC05 and MC68HC11 family microcontrollers, which are pre-loaded with EIB system software. Their digital I/Os and analogue inputs are connected to the standardised 10-pin PEI connector. While BCUs come with housing and EMC[20] shielding, BIMs do not, allowing for tighter integration into manufacturer-specific solutions. Available versions vary in EEPROM and RAM size (which, for example, affects the maximum size of the group address table) and execution speed. Also, two system software variants exist, which mainly differ in that System 1 does not support interface objects. Also, System 2 employs more complex load control procedures for management resources, like the application program. For high-volume designs, the transceiver ICs and microcontrollers used in BCUs and BIMs are also available separately.

Since the processing power of a BCU is limited, more complex user applications have to be run on a separate microprocessor. In this case, the application processor can access all EIB-related functionality provided by the BCU (including support for group objects) using the PEI as a serial interface. Alternatively, the processor can implement the entire communication stack down to the MAC (Medium Access Control) sub-layer itself, using a standard transceiver IC for connection to the EIB medium. Especially the Layer 2, however, is complex to implement, not at least due to its tight timing requirements. This also entails high certification costs.

Therefore, the TP-UART IC offers a more convenient solution. It interacts with the application processor on the level of Layer 2 frames via an asynchronous interface (UART). Of the Layer 2 services, only the determination whether its node is being addressed or not is left to the application controller.

While use of the TP-UART will significantly soften the timing constraints to be met, they are not removed entirely. For example, the application processor has to respond whether an acknowledgement frame is to be generated within less than 1.7 ms of receiving the destination address. This is due to the fact that the TP-UART just relays frames without further processing or buffering (except to compensate for speed differences between the EIB and the application processor interface). As a positive side effect, this offers more flexibility regarding the frame format. For example, whether a frame is in standard or extended format is entirely irrelevant to the TP-UART.

[20] Electromagnetic Compatibility

Certified software implementations of the remaining EIB protocol stack are available for MSP430 and AT-Mega microcontrollers. These will also provide standardised "device models" which allow the ETS to handle the configuration of nodes built around this solution. Essentially, this amounts to an emulation of BCU behaviour.

For applications with even higher demands on processing power or on the human-machine interface, PC-based solutions come into play. Connection to the EIB is usually accomplished using serial communication with a BCU. Additionally, USB interfaces are available for "legacy-free" PCs. As an alternative, approaches accessing the EIB by way of an intermediate network gain importance, as will be detailed below.

For Microsoft Windows based systems, EIBA offers a certified software component called *Falcon* which provides a high-level API for accessing functionality throughout the network stack. For the Linux operating system, which provides an interesting perspective toward cost-effective embedded platforms, both commercial and open-source kernel level drivers for BCU access as well as TP-UART based serial interfaces are available. Finally, the Eiblet API [35], with an implementation available free of charge, allows Java applications to interact with an EIB system at multiple levels of abstraction.

3.8 Integration

Even if many appliances integrate EIB natively and considerable improvements can be achieved through its use alone, integrating it with other networks offers further evident benefits. This includes tunnelling to support various remote access scenarios, routing to extend both range and bandwidth, and gateways.

As a field area network, EIB is well-suited for transporting narrow-bandwidth control data. Yet, large building automation systems will generate lots of data, which are to be collected in a single place for data acquisition and central control. Since high-volume data transfer is out of scope of FANs, other network technologies have to step in and act as a backbone to smaller FAN segments.

A straightforward approach is to use them as an alternative transport medium, encapsulating FAN protocol frames in the host protocol. This method, called *tunnelling*, can be used for connecting FAN segments with a high-performance backbone to overcome limitations of the FAN protocol regarding bandwidth, range or both. It also offers an alternative to physically de-couple applications like remote configuration or visualization from the bus line.

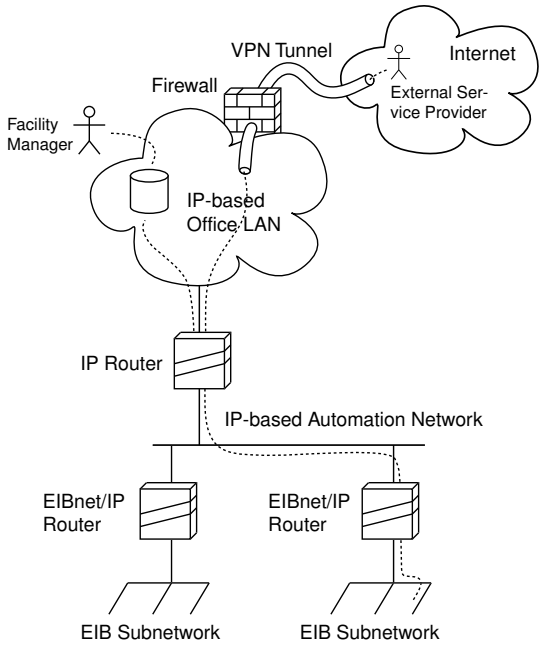


Figure 3.12: EIBnet/IP

Although tunnelling can perfectly cover certain fields of application, it has two main drawbacks. First, the fact that the host network remains transparent to FAN end devices will inevitably cause timing problems when the latency of the former exceeds the range acceptable for the FAN protocol. Second, it is only applicable to scenarios where all participants can handle the FAN protocol. In cases where this requirement is not convenient or acceptable, information has to be converted at the application level using *gateways.*

Regarding Internet connectivity, official IP tunnelling support for EIB has been existing for years. Since it was marketed as extending ETS for remote access *(iETS)*, the server component on the EIB side is called iETS server. Falcon can use this transparently as an alternative bus access method instead of a locally connected BCU.

Currently, a working draft for IP tunnelling and routing *(EIBnet/IP)* is undergoing voting for inclusion into the EIB specification. With EIBnet/IP *routing,* EIBnet/IP routers take the place of backbone or line couplers.

IP multicast is used by an EIBnet/IP client to discover EIBnet/IP servers and for routing. Retrieval of further descriptive information on an EIBnet/IP server and tunnelling is handled via IP unicast. Although EIBnet/IP leverages the practical omnipresence of IP technology, it will in most cases be advisable to keep the automation network logically separated.

EIBnet/IP explicitly does not address security issues to keep the protocol lean and reduce the processing power required in EIBnet/IP routers. This is done on the assumption that the backbone IP network will be tightly supervised and highly restricted or closed to network traffic from the outside. The use of EIBnet/IP in an Intranet or over a VPN (Virtual Private Network) connection only is suggested as a key countermeasure against all threats.

Besides allowing more fine-grained handling of security issues, such separation also simplifies performance considerations. It may be achieved by routing and/or firewalling, with authorised users able to open VPN tunnels as shown in Figure 3.12. It is the task of the firewall to handle authentication, authorization and encryption to avoid burdening EIBnet/IP servers with these tasks.

For future releases, the integration of gateway functionality into EIBnet/IP with the capability to handle both regular and maintenance operation is planned as well. Until then, a variety of proprietary implementations for interfacing not only with IP but all kinds of other networks (including ISDN,[21] POTS,[22] PROFIBUS-DP[23] and Bluetooth) are available on the market. Albeit, these are, as a general rule, limited to group communication.

For BACnet [1], a network protocol dedicated to building automation and control, the situation is different. The mapping between EIB and BACnet objects is laid down in both standards [2, 19]. Together with the specification of EIB tunnelling on Ethernet (EIBnet), a defined integration of these systems is ensured. Still, semantic differences remain to be bridged by the system integrator through proper gateway set-up.

For high-demand visualization and facility management, various PC software packages exist which will connect to the EIB locally or via tunnelling, if necessary through multiple ports. Besides standard soft PLC, logging and alert escalation functionality, some products will also automatically dispatch work orders and handle room management.

Additionally, any OPC-enabled software can be used by way of one of the OPC servers for EIB available from various vendors (including EIBA). OPC (OLE for Process Control, Open Process Control) is a de-facto standard for providing an open interface for process control, automation and visualisation on Windows platforms. Originally based on the Microsoft OLE (Object Linking and Embedding) component object model, OPC server components provide

[21] Integrated Services Digital Network (digital telephony)
[22] Plain Old Telephone System (analogue telephony)
[23] A field bus popular in industrial automation

standardised methods for accessing data residing in PLCs, field bus devices and other process control equipment.

Still another way of connecting to the EIB is through one or more gateways via an intermediate automation level network. A variety of compact, rail mounted embedded IP gateways is available, which will additionally provide other features like:

- ISDN TA[24] or POTS modem in addition to Ethernet interface (for dial-up Internet access or WAP server)
- HTTP/WAP server
- Various messaging possibilities, including escalation management (e-mail, fax, SMS, voice call)
- Logic functions (including timer functions and lighting scenarios), partially with visual set-up
- Logging (data acquisition)
- Time servers
- Visualization
- Media integration (brown goods)

Which of these functions are actually implemented in a particular device is related to the intended sales market. Some of these functions are not relevant for large-scale building automation, since they will be handled by the central server (like visualization and data acquisition), yet nonetheless of interest for smaller buildings. In the home environment, entirely different functionality comes to the front, like for instance being able to make a voice call to the gateway to receive a one-time PIN which can be used to access the gateway from an Internet café without fear of eavesdropping. For home use, outsourcing of visualization functionality to data processing centres is offered as well. Besides personalised portal sites, they will provide remote services like dispatching voice calls in response to an alert condition.

[24] Terminal Adapter

3.9 The Future of EIB: KNX

In 2002, EIB merged with *BatiBus* [18] and *EHS* (European Home System) [18] to form the new KNX standard. Although EIB is now correctly known by the name of *KNX TP1/PL110 S-Mode*, "EIB" will definitely stay as a label for a specific subset of KNX functionality for quite some time.

Besides the "KNX Handbook" [27] provided by Konnex Association, KNX is documented through formal standards as well. The relevant family of European Standards is EN 50090. It is maintained by CENELEC TC 205 (Home and Building Electronic Systems) in cooperation with Konnex Association. The normative process is also coordinated with CEN TC 247, which is concerned with building controls and has in the past also defined standards covering EIB technology.

From the perspective of EIB, the transition to KNX brings about certain extensions to the well-known protocol stack, none of which are mandatory to implement. Regarding physical media, an additional twisted-pair *(TP0)*, an additional powerline *(PL132)* and the RF medium already mentioned in Section 3.5 were added. For both TP media, a specification which allows bus power to be provided in a distributed way by any device instead of requiring dedicated BPSUs was introduced.

KNX also offers new configuration modes. While compliance with the single tool ETS (referred to as *S-Mode* (System) configuration) will be a requirement for all KNX devices, additional mechanisms are available in parallel.

E-Modes (Easy) aim to provide installers with easier and less error-prone (albeit less powerful and flexible) ways of linking devices without need for ETS. This is accomplished by

- Push buttons: The installer pushes special buttons on the devices which are to work together.
- Logical tags: Devices which have the same tag number set (for example, via a code wheel) will cooperate.
- Central controllers: This mode, making use of a dedicated controller device, is similar to the proprietary approaches outlined in Section 3.1.3.

While the E-Modes like EIB/S-Mode are targeted at the installation domain, *A-Mode* (Automatic) is intended for the integration of loose goods, in particular white goods. A-Mode devices will autonomously integrate themselves into

the network in a "plug-and-play" fashion, establishing the necessary communications links without user intervention.

Since E- and A-Modes cannot rely on an integrator's competence to combine the data points of node applications into a properly working system, interworking issues are given ample room in the KNX specification. *Functional blocks* for various application domains are being specified, which describe sets of group objects and interface object properties with clearly defined semantics. Instead of the free binding possible in S-Mode, E- and A-Modes always link data points at the granularity of functional blocks. Functional blocks also prepare the ground for a more visual way of configuration using PC-based tools.

A functional block encapsulates a specific solution for one given task of an application. For example, the application model "Heating" includes a functional block "Hot Water Boiler". Unlike the legacy EIS types for dimming and drive control, the data points of functional blocks specifically include (typically persistent) behavioural parameters. For example, the functional block "Dimming Actuator Basic" also includes data points for setting various delays. It also goes far beyond the EIS profile in other aspects, although almost all added functionality is defined as optional. Although, while the profile was reworked significantly, certain not immediately convincing design decisions were carried on. While feedback is finally mandatory, it can still optionally be implemented by making the set-point control data points bidirectional. Also, step-wise dimming is still controlled in an event-based manner by transmitting the desired relative set point change.

Devices are not expected to implement the KNX specification in its entirety. Rather, they are expected to select a subset *(profile)* appropriate for the sales market intended. Consequently, certification is also done against specific profiles. With simpler configuration modes and other aspects aimed at accommodating resource-limited devices within the KNX standard, KNX opens up paths into the mass market.

Also, endeavours to clarify and restructure the specification can be identified. This specifically includes issues of terminology. For example, where the EIB specification would refer to "communication objects" and "EIB distributed objects", KNX uses "group objects" and "interface objects", respectively. Also, EIB "physical addresses" are now known as KNX "individual addresses", acknowledging the introduction of open media where these addresses no longer correspond with the physical topology of the network. Although the present thesis is concerned with EIB, it fully endorses the new, clearer terminology wherever applicable.

As an aside, the EN 50090 family of standards also includes a draft describing issues related to residential gateways. It however does not seem to have reached any further significance.

4 The OSGi Platform

The Open Services Gateway Initiative (OSGi) is a non-profit technology consortium formed in February 1999. Its core task is to develop and promote an open specification for a gateway platform supporting the delivery of remote services (a *service gateway*). This specification is available free of charge from the OSGi website. Implementations are not subject to payment of royalties to the OSGi, although certain elements may be subject to IPR claims by third parties. The OSGi is also involved in the certification of compliant implementations.

The OSGi platform is centred on a lightweight service-oriented component model. It provides a Java-based environment where multiple software modules execute concurrently, collaborating to provide their functionality both to one another and to the outside. These can be added, modified and withdrawn at runtime, allowing the overall behaviour of an OSGi-based service gateway to be dynamically configured.

The OSGi architecture follows a three-tier computing model. In this model, gateways bring together local devices and remote back end servers. Its primary focus is on the delivery of electronic services over public networks to the consumer environment. It is designed to handle the complexity of an operator based network with multiple independent, external service providers. OSGi platforms thus allow to mix and match service components from different vendors to exactly suit a particular business plan or specific end-user demands.

Supporting this architecture where gateway operators control large numbers of service platforms situated on the customers' premises, the design includes a powerful concept for remote management. Its policy-free approach allows to control all management aspects through a communication channel free of choice. This specifically includes aspects of provisioning.

Also, considerable attention is paid to security issues. The OSGi platform security model specifically addresses the fact that software provided by possibly competing vendors will be installed in parallel. It places special emphasis on preventing components from interfering with one another, whether unintentionally or on purpose.

It is the aim of the OSGi to leverage existing computing and network infrastructures through a platform-independent, network agnostic design. The same inclusive attitude is maintained towards existing standards.

Addressing the resource limitations present in embedded gateway configurations, the minimum Java package footprint required is a subset of the Java 2 Micro Edition Connected Device Configuration (CDC) 1.0 / Foundation Profile 1.0.

Although the open service provider model described forms the primary reference architecture, setting the course for the development of the platform specification, OSGi-based solutions are expressly not limited to it. For example, the service platform need not be placed at the end user. It can remain within immediate control of the operator, possibly as a logical instance on a back end server. Such an approach is suited more to services related to network operation, like bandwidth monitoring. An example for such a virtual gateway model is presented in [23].

In an industrial environment, OSGi platforms provide a way for integrating proprietary islands of control into a coherently manageable system. Equally, they can be of benefit in a scenario where management is done entirely by the end user. For example, the renowned Eclipse IDE uses an OSGi framework to support its dynamic plug-in concept. This shows that although the platform is (or once was) designed with gateway applications in mind, the core framework design in particular is generic enough not to preclude other use.

The Service Platform specification maintained by OSGi is now available in its third edition [34]. The framework proper has remained effectively unchanged since its initial release in May 2000. In addition, the specification defines a set of standard service APIs. Besides supplementing the core framework with respect to management issues, they provide support for typical issues in gateway applications like logging or communication. This modular approach also helps ensure backward compatibility, which is a declared goal of OSGi.

While the initial focus of OSGi specifically concerned residential gateways and home automation, nowadays the focus has shifted towards a more horizontal approach applicable to other markets as well. This specifically includes automotive systems, but also areas like consumer electronics, mobile phones and security products. This change is reflected in the different standard services which were added to subsequent specification releases.

Several commercial and open-source implementations of the framework exist, most of them accompanied by implementations of the standard services. They are being employed in a significant number of projects. The OSGi web

site provides a comprehensive overview of related activities of OSGi member companies.

The remainder of this section introduces key aspects of the framework and describes a selection of standard services. Besides, it presents related concepts and discusses specific challenges arising when designing OSGi-based software.

For further information (besides the authoritative source [34]), [11] provides practical hints for development. Although it discusses the first specification release, it gives valuable insights on framework concepts. A brief overview of possible use cases in residential environments, reference architecture entities and framework mechanisms describing the second release can be found in [32]. The services included with the third specification release are covered in [29], which also describes a number of commercial products incorporating OSGi technology and outlines a specific example for OSGi-based development in academia, highlighting the resonance it has found within the scientific community.

4.1 Related Concepts and Technologies

This section will discuss two major software engineering concepts forming the backdrop to the design of the OSGi platform specification. Software "components" and "services" are popular terms—component technologies are even in widespread use—, yet both seem to lack a canonical definition to this day. The following paragraphs aim to illuminate this background of both OSGi and selected other Java-based technologies often referred to in one breath with it.

4.1.1 Component-Based Software Engineering

Component-based software engineering (CBSE) is a core principle of software reuse. Although it is today a widely accepted concept and supported by numerous technologies, no universally agreed characterisation for software components exists. A frequently quoted definition originates from [43]:

> "A Software Component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Another influential source is the notion of a component within the UML (Unified Modeling Language) [6], which considers it

> "[...] a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces."

This definition emphasises the far-reaching (although not entirely unanimous) consent that components should be *binary* units. Like OOP[1] advocates the separation of concerns and data hiding on the source level, component technologies are expected to heed these principles on the binary level. Components represent logically cohesive parts of a system. The functionality they encapsulate is only reachable through well-defined interfaces, ideally allowing components to be replaced transparently to modules using them.

Ideally, complete applications would be assembled by customising and connecting components without actually writing code in the traditional sense. If supported by the execution platform, this binding may even take place at runtime.

Different *component models* concentrate on various further aspects, like certain mandatory properties, interfaces, or allowable modes of interaction. For example, JavaBeans or ActiveX Controls require support for composition by visual builder tools. These aspects are closely related to the capabilities offered by the respective *component frameworks*. For instance, frameworks for Enterprise JavaBeans[2] provide components with support for security, distribution and persistence.

4.1.2 OSGi as a Component Model

As will be detailed in Section 4.2, software is deployed into an OSGi environment in the form of so-called bundles containing sets of Java class files. Gateway operators define the overall functionality of the gateway platform by choosing appropriate bundles, which will each provide a coherent subset. Bundles interact through well-defined interfaces (referred to as services) only. Which bundle offers a certain service is not of interest to clients.

Going by the definitions just cited, bundles clearly emerge as components within the OSGi framework. Yet, unlike in other component models, bundles cannot be instantiated multiple times. Instead, they can implement the same service interface multiple times, with variations in behaviour. Upon closer

[1] Object-Oriented Programming

[2] It should be noted that the "plain" JavaBeans and Enterprise JavaBeans (EJB) component models are drastically different despite their nearly identical names. While the former is geared towards rapid development of GUI (Graphical User Interface) applications, the latter is tuned towards server-side, database centric software.

scrutiny, other details appear besides this one which suggest that attributing the role of components within the OSGi framework to bundles alone would not give a complete picture of its properties and capabilities [8]. Still, it seems justified to consider bundles as components and services as their interfaces for the purpose of general discussion, as is done in the present thesis.

Another unconventional property of the OSGi model is that components are not manually bound to another to achieve the desired application. Therefore, the concept of a static assembly found in other component models does not exist.[3] Bundles in need of a service dynamically and autonomously discover and bind to it when it is made available by another bundle. The precise policy they follow in doing so is defined by the bundle developer. Accordingly, service dependencies are not tracked and managed by the framework.[4]

To support this dynamic process, bundles are given considerable autonomy. Each of them is associated with an independent thread of execution. Together with the lack of static composition this makes bundles look more like independent applications rather than components at first sight. Because of this, bundles are sometimes referred to as "service applications" (which however should be avoided if possible due to its potential for confusion).

Not attempting to manage dynamic dependencies is a key example of how the OSGi framework is designed to be lightweight by concentrating on essential functionality. While a framework like EJB—to pick a drastic example—undoubtedly provides more features, the resulting overhead would be detrimental instead of beneficial in the resource-limited environments targeted by OSGi. With the latter, lightweight design is applied in terms of concepts, code size and package footprint. If needed, additional functionality can still be added by way of appropriate bundles.

4.1.3 Service-Oriented Architectures

Service Orientation is a rather novel software engineering concept. As might be expected given the fact that the term "service" is at least as overloaded with different interpretations as is "component", a precise definition seems even farther from reach. At any rate, the essence of any such approach is to concentrate on the *service* a piece of software (or combination of hardware and software) can offer to a user—which may be a human or yet another piece of software. Concepts in this field are not limited to the better internal structuring

[3] Neither does the need for components to provide design-time support like JavaBeans do.
[4] An extensive effort to add service dependency management to OSGi is presented in [9].

of complex software. Especially recent ones strongly emphasise the commercial aspect: "Sell Services, not Software." This goes far beyond application servers or outsourcing payroll processing. In the future, business cases are expected to be created around the exchange of process input and output data (over networks) on much a finer scale. It was not until the advent of the Internet that such ideas could gain momentum.

The vision of *service-oriented computing* is to tailor the the technical infrastructure towards immediately providing business services. In such an all-encompassing *service-oriented architecture* (SOA), like described in [36], open marketplaces exist where *service providers publish* the services they offer. Potential *service users* can *discover* these and select an appropriate one for *binding* into different applications or business processes. *Service aggregators* compose different vendor's services into combined offerings, in turn becoming service vendors again. This services market is supported by an infrastructure of "meta services" addressing issues of composition and management.

This process is to be of highly *dynamic* nature, with on-demand procurement and purchase of software services being possible. Composite services would eventually be created in an ad hoc manner for one-time use, serving a particular user demand at a particular point in time. In [4], this is likened to buying a new car. Here, the supporting process is already flexible and fast enough to allow customers to create their desired car by combining option elements instead of having to select from a predefined stock.

Composition is a key concept in SOAs. To understand this, one has to consider that the underlying technical infrastructure is (by common consent) best exposed in an as abstract and generic way as possible. The value of these services is considered to grow the more implementation details they hide.

Considering the example of a print service, issues like the control protocol of the printer or when its toner cartridge has to be exchanged[5] are not of interest. Only key service parameters and qualities, like the paper format the output is to appear on or whether black and white or full colour output is desired, are to be visible to the outside.

Whether such a base service will actually be sold separately to a customer is a matter of circumstances. For example, proof printouts will be complimentary when a large ad campaign is being commissioned, but the same colour laser prints may as well be sold separately by a copy shop.

[5]This example assumes copy-shop like circumstances. Things would be different when the printer is offered free of charge for customers to use on their premises on the basis of a fee per printed page.

This means that the technical infrastructure will be represented as a pool of long-lived base services, which are carefully designed to expose as little (implementation) context as possible to make them usable in a variety of situations. The specific, possibly fleeting context of a particular business opportunity can then be reapplied by composing these into a derived offer [37].

Generally, services should represent highly cohesive functional units, both to present a meaningful unit of purchase as well as to reduce the complexity of the binding process. The semantic aspects of a service need to be precisely defined in an open contract, which also has to cover non-functional aspects, like parameters concerning quality of service. To enable the service user (or aggregator) to quickly choose among offerings, standardised representations for such contracts have to be sought.

To ensure an extensive services market, services need to lend themselves to composition in other ways as well. Specifically, they need to break the limits of platform or language specific frameworks. Also, it should be possible to mix and match services provided by different vendors.

4.1.4 Service Orientation in Practice

A number of software infrastructure solutions is designed to support the implementation of service-oriented architectures. Others just may be inspired by their underlying concepts in finding new ways for structuring software. This is often referred to as *service-oriented programming* (SOP) [5, 7]. At any rate, specifications will only address selected parts of the large spectrum.

At the core of all such approaches is the idea of concentrating on *what* a piece of software does, rather than on *how* it does it. So far, the picture is not very different from component-based approaches as described above, which also frequently use the term "service" to describe the functionality exposed by a component via an interface. As a specific example, [6] states that

> "[...] components provide [...] services through [...] interfaces, which, in turn, other components can discover and use."

Yet, while components and services are not orthogonal concepts and have a considerable degree of overlap, they are complementary in that service orientation is primarily concerned with *usage* patterns of software, while CBSE addresses *implementation* concerns. Although component software can be considered the foundation of service-oriented architectures, it is important to understand that application of one of these concepts does not imply the presence

of the other. Within a service-oriented architecture, the term "service" will always have overtones of additional concerns going beyond those traditionally associated with component interfaces.

Also, distribution was only added to the concept of CBSE at a later stage, while SOP implies the context of a (in most cases highly) distributed system from the outset. Consequently, the demand that distributed services should be as independent from a certain hardware platform, operating system and programming language as possible forms a focal point of interest. Supplementing the creed of service orientation presented above, this could be expressed as service providers publishing what they *do,* but hiding what they actually *are.* A suitable open interface abstraction allows the seamless integration of legacy systems, allowing software (or rather *system*) reuse at a high level of functionality [42].

This goal is typically accomplished by requiring service providers and users to communicate via well-defined network protocols. For example, SOAP (Simple Object Access Protocol) enables platform-independent interaction between *Web Services,* which currently eclipse all other service-oriented architectures in popularity.

Alternatively, a proxy software component could be deployed to client machines which exhibits a precisely defined interface, but hides the protocol used to communicate with the actual back-end application server. Although such an approach will forfeit the benefit of total platform independence, it may allow more efficient data handling, resulting in cleaner (and less error-prone) implementation and better run-time performance [20].

The use of run-time environments masking the underlying platform (like those of Java or .NET in particular) will help to increase the range of clients such a proxy will be able to run on. Also, the server should retain control over the proxy component to be able to update it in case, for example, the communication protocol is to be updated. Java *Applets* were the pioneering concept to answer this demand. *JNLP* (Java Network Launching Protocol, also known as Java Web Start) follows in their footsteps. With JNLP, the Applet approach is extended towards larger applications while maintaining key features like the sandbox, which is essential for running foreign, potentially malicious code. Downloaded applications will be cached, and incremental updates are automatically run when parts of them are changed on the remote server. Still, JNLP is aimed at traditional (if Web-centric) client-server applications which essentially confine themselves to presenting a front end to a data source located on a remote server, up to and including the user interface. The user is bound

to a single service provider at a time. Also, a remote service accessed this way is not available for further composition.

For the issue of how to keep service users up to date in case of modifications to a service to arise, the service has to be changed in the first place. Managing the functionality of a heterogeneous collection of highly distributed, always-on servers is a challenge in itself. The JMX (Java Management Extensions) architecture describes a popular API for remote management which is often used in conjunction with EJB solutions. It provides a flexible infrastructure for accessing management resources exposed by components via freely selectable communication protocols.

Returning from these technicalities of handling distributed services to the issue of dynamically bringing together service providers and users, *Jini* emerges as the probably most widely known Java-based technology which focuses on this key feature of advanced SOAs. Jini is designed to enable spontaneous networking, with as little human intervention as possible. Users can discover the presence of services they would like to bind to through dedicated lookup services, which also allow them to keep track of events like the arrival and departure of providers. The problem of discovering these lookup services in the first place is addressed as well. Communication between service providers and users is handled via a proxy mechanism as described above. Services are leased for a defined period of time. When a user does not renew its lease, a provider can assume it to be gone and reallocate the associated resources. Jini is (at in its original design) not concerned with services in their dimension as business cases.

The *Apache Avalon* Project is an example for service-oriented concepts being applied at the node level. Its goal is to provide a framework for server-side Java code with flexible functionality, building upon (although extending far beyond) the Servlet concept. With Avalon, service providers and users are software components residing on the same physical platform. Its framework includes facilities for service discovery, composition and life cycle management.

Finally, the specification of the *OSGi* service platform can be considered service-oriented in multiple ways (although the term actually never appears in [34]). First of all, it is specifically designed to support a distributed infrastructure of service providers, aggregators and consumers maintaining business relationships (although it is not concerned with details of any of these).

Beyond this, the programming model it promotes wholeheartedly espouses SOP as well. Not only are interface implementations referred to as services, but bundles may actually publish and withdraw them at any time. A service

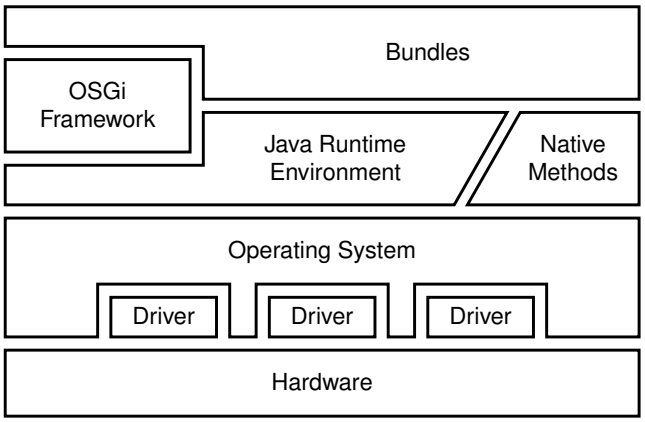


Figure 4.1: Architectural Overview of the OSGi Platform

registry keeps track of currently available services to enable discovery, composition and refinement by prospective users. The possibilities for registering service meta data are limited, however.

Bundles will often act as proxies for services hosted remotely by providers (yet another notion of "service" in the context of OSGi). Billing these on a per-use basis is allowed for. The coexistence of components provided by possibly competing providers is addressed through an appropriate security infrastructure. Remote management support includes the provisioning of components and switch-over to other gateway operators.

OSGi devotes its specification efforts entirely to defining an execution environment. Any protocols the platform may use for outside communication—with remote services, local networks or management clients—are not considered on purpose and left to be defined by appropriate components.

Although specialised technologies (like those touched upon above) may cover particular concerns of service orientation in greater depth, OSGi is unique in addressing them in a comprehensive and well rounded-out manner while keeping a small footprint. Where applicable, it still lends itself to combination with such solutions through its open design.

4.2 Framework Architecture

The OSGi platform is built upon a Java runtime environment, which contributes essential features for a dynamic component architecture like dynamic class loading with late binding. It also significantly assists in attaining key design goals like platform independence and security against malevolent code.

Also, the orientation of Java towards network programming in general (and the Internet in particular) corresponds well with the aims of the OSGi architecture. Figure 4.1 illustrates the relationship of the elements present within an OSGi service platform. Service application components, which are referred to as *bundles,* can access functionality provided by the framework, the underlying Java VM (Virtual Machine), and the operating system (by using native code libraries) as needed.

4.2.1 Bundles

Bundles represent the packaging and delivery unit within the OSGi framework. Format-wise, these are standard Java Archive (JAR) files, which may not only contain any number of Java classes and native code libraries (also for different hardware and operating system platforms—the framework will assist in selecting the appropriate ones), but also resources (like static HTML pages and images) and documentation.

Only a single instance of a bundle exists at any given time within the platform, which is uniquely qualified to the framework by the location its JAR file was retrieved from (typically an URL, which may point to a local file system as well as an HTTP server). Every bundle is associated with its own thread of control. The life cycle operations a bundle will be subjected to therefore encompass being installed (the JAR file is retrieved from the bundle source location to the OSGi platform), started, stopped, updated (the JAR file is replaced by a newer, backward compatible version) and uninstalled (all traces of the bundle are removed).

The JAR Manifest headers are extended to include OSGi-related information, like the so-called *Bundle Activator.* This class provides hooks into the bundle code which will be called by the framework when the bundle is started or stopped. At start-up, the activator class will be passed the *bundle context* object, which is the entry point for accessing all framework-related functionality. Since every bundle has its own context object, the framework can keep track of which one performed a certain operation (for example registering a certain service). The headers also provide information about which additional packages (including requirements for specific minimum versions) are necessary for the bundle to run. These will will in turn be supplied by other bundles. This mechanism replaces the static class path concept and allows to dynamically resize and update the runtime environment.

Normally, the framework ensures that a bundle can only be started if these dependencies are satisfied. To support the 'Class.forName' idiom used by (non-OSGi) libraries in case the names of required packages (e. g. drivers) are not known until run-time, it is also possible to specify packages for dynamic import. This will defer dependency resolution to class instantiation time.

Exported packages are made available to importers as soon as the exporting bundle is installed. They will remain exported as long as all importing bundles are uninstalled. This process is unrelated to the exporting bundle being started or stopped.

The framework ensures that every package is only exported by a single bundle, which may lead to the situation that a bundle will not use the ones it declared for export, but those of another bundle instead. This is necessary since every bundle has its own class loader for reasons of security and life cycle management. Thus, every bundle lives in a separate name space. While this prevents bundles from interfering with one another in case both accidentally (or maybe intentionally) pick identical package and class names, it also prevents the intentional exchange of objects. Thus, the loading of shared class definitions has to be delegated to the class loader of the exporting bundle by all others.

4.2.2 Services

Bundles collaborate[6] by mutually providing and using *services.* A bundle can provide any number of services (including none), and services can in turn be used by any number of bundles. In Figure 4.2, the Surveillance bundle uses the Sensor Event Notification service of the Field Bus Access bundle to receive alerts from a motion sensor. In case the surveillance system is armed, it will use the Lighting Scene Recall service to turn on all the lights in case an intruder is detected and notify the owner using the SMS service provided by the Messaging bundle. Entirely independent of these actions, the Home Theatre Control bundle can recall the proper lighting scene for watching a film if needed.

Within the context of OSGi, the term "service" in the narrower sense refers to a Java interface definition with (externally) agreed-upon semantics. A bundle using a service will only import the interface class. An appropriate implementation (called a *service object*) can be selected at run-time, potentially from multiple available choices. This approach de-couples specification and imple-

[6] While package sharing is necessary for bundles to have a common definition of classes whose instances they wish to exchange, it is not the intended model of interaction.

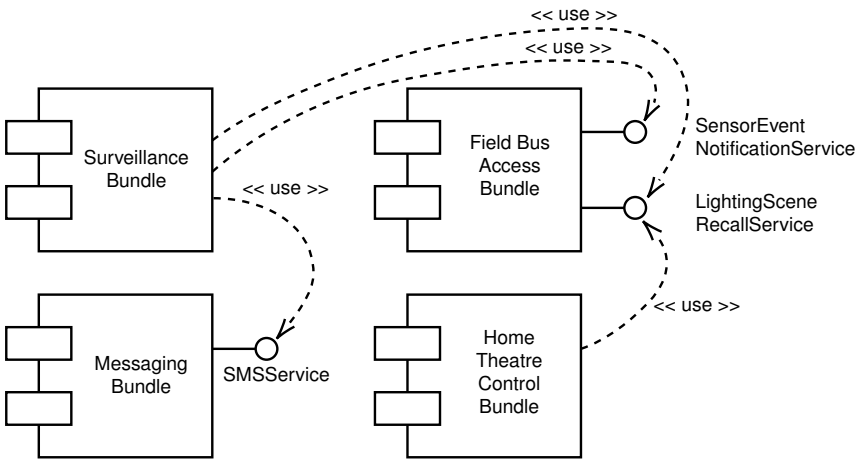


Figure 4.2: OSGi Component Interaction: Bundles and Services

mentation of a service and allows programmers to bind to its specification only. It is important to note that the bindings between bundles providing and using services are established (and removed) entirely at run-time.

Service discovery is provided through a registry maintained by the framework. Here, bundles can register any number of services. Since in many cases, possibly different bundles will register multiple service objects implementing the same service, a set of descriptive properties (key/value pairs) may also be recorded along with the registration. Other bundles interested in a specific service can then look it up using the interface name and, if needed, specifying filter criteria (in RFC-1960 LDAP filter syntax) over these properties. The framework returns a set of objects encapsulating references to corresponding services. These contain the entire set of properties as specified with their registration for the client to examine. Once a suitable service is found, the client bundle uses the `ServiceReference` to obtain a Java reference to the service object. This process is illustrated in Figure 4.3. It also shows how bundles only export the interface class to allow other bundles to register services with the same interface while retaining control over the service object.

The OSGi framework also provides support for *Service Factories.* Usually, all clients of a service receive a reference to the same service object that was registered. This process is invisible to the latter. Service Factories however will be notified by the framework every time a new bundle obtains the service or releases its last reference and can provide customised versions of the service object to every bundle. This allows to associate service related resources with

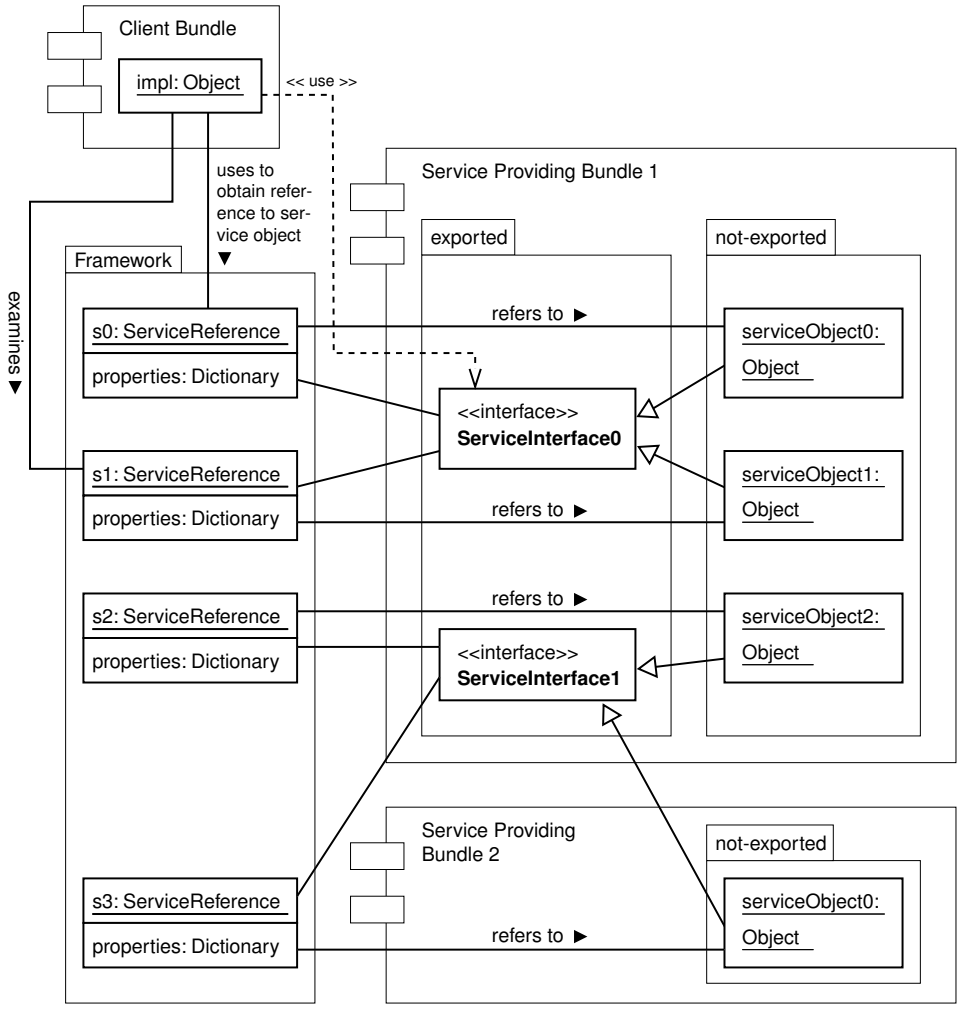


Figure 4.3: The OSGi Service Registry

client bundles, for example to keep them from interfering or to clean up after they have left.

4.2.3 Core Design Features

Bundles may register and get as well as withdraw and release services at any time. Since it is vital for bundles to react to these changes in their environment, the framework provides an appropriate *event mechanism.* Events are broadcast in the case of registration, withdrawal, or the change of properties of a service as well as in response to changes in a bundle's life cycle and when the framework was restarted or has encountered errors.

For remote administration, all administrative framework functions (in particular, bundle life cycle management, but also reporting features) are available through services. This "policy-free" approach leaves gateway operators free to choose a suitable communication channel for management by selecting an appropriate interface bundle.

Security in the framework rests upon the Java 2 security architecture. Though it is optional for a framework vendor to implement, significant consideration is devoted to it by the OSGi specifications. All parts of the API which are considered security relevant require appropriate permissions, which are granted on a per-bundle basis. The OSGi specification defines three additional standard permissions for access to the administrative functions of the framework, service registration and access and import/export of packages. Bundles are free to define additional custom permissions.

4.3 Standard Services

In addition to the framework proper the OSGi specifications define a number of standard services, of which an OSGi platform can contain any subset (including none at all). Some serve to map existing Java standards to the dynamic nature of the platform, while others are uniquely OSGi-related. In the following, a brief high-level overview over these services is given, roughly grouped by their purpose.

4.3.1 Framework Extensions

Since the OSGi framework API is to be kept backward compatible as much as possible, even very closely related functionality is added in the form of additional services.

The *Package Admin* service addresses the issue that uninstalled bundles cannot be removed from the environment while packages they have exported are still in use by active bundles. This both wastes resources and bars the importing bundle from using the updated version until package dependencies are resolved again when the framework is restarted. The Package Admin service allows to selectively refresh updated packages by automatically stopping, newly resolving and restarting all bundles which depend on them.

The start-up and shut-down ordering of bundles can be controlled using the *Framework Start Level* service. This can for example be used to implement a "safe mode" where only bundles fully trusted not to cause erratic behaviour

are started. The Framework will distribute appropriate events when packages are refreshed as well as when the current start level changes.

Extending the framework security architecture, which is only concerned with protection against malicious code only, the *User Admin* service adds capabilities for role-based authorisation of human users. It is designed to support various authentication methods. The *Permission Admin* service offers a standard interface for changing the platform policy configuration or even grant permissions just-in-time during bundle installation.

Last in the series of framework extensions is the *Service Tracker* utility, which helps to track services a bundle depends on. It allows to specify a set of services by service interface name (and properties filter expression, if needed) for which events are generated if a matching service is added to or removed from the registry (or its properties modified).

4.3.2 Device Access

The OSGi *Device Access* concept assumes devices connected to the gateway are to be represented as device services, provided by appropriate driver bundles. Driver bundles are expected to build a hierarchy of device abstractions (for example, a Camera Service on top of a Generic USB Device service). The process of creating such a hierarchy is termed *refinement.* While these services and bundles are not fundamentally different from others, the Device Access service aims to automate the refinement process. This includes the automatic selection, possibly download and installation of matching driver bundles based on the evaluation of specific service properties, without any operator or user interaction.

A service signifies that it represents a device by providing the `DEVICE_CATEGORY` property with its registration. The concept of a device category as defined by OSGi Device Access comprises a service interface and service registration properties with defined semantics together with match values to allow ranking potential drivers by how well they could refine (i. e. make use of) the capabilities of this device. A generic driver which only matches the device category, for example, could probably only provide basic functionality and will thus be ranked lower than another, which matches the exact make and model (given by the registration properties) as well and can be expected to provide access to enhanced features. Owing to the goal of application independence heralded by OSGi, the Device Access specification does not define any such device category, however.

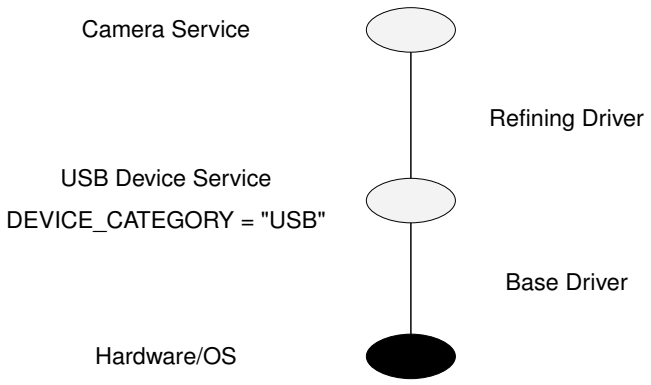


Figure 4.4: OSGi Device Access: Driver Refinement

Whenever a device service is registered with the framework, the Device Manager[7] presents its `DEVICE_CATEGORY` (together with all other properties) to all available Driver Locator services. These return locations of appropriate driver bundles. The Device Manager installs and starts them all and subsequently requests a match value from every driver, which may also communicate with the device in question in order to determine it. The driver with the highest match value is then asked to attach to the service and will register one or more refined devices. If no driver is found, the device service is informed and may withdraw its service registration and register anew with other properties (like a more general device category) to increase the chance for a refining driver to be available.

As an example, consider the situation illustrated in Figure 4.4. A gateway platform is equipped with an USB interface. A pre-installed bundle (called a base driver) monitoring the USB detects that a new device was connected to the bus, whereupon it registers a "generic USB device" service. The methods of this base device service allow as much communication and expose as many properties of the new device as possible without specific knowledge of device internals. For a USB device, this will include the device class, manufacturer and product IDs. This information is used to obtain the best matching driver. The winning driver bundle then binds to the base service object and registers a more abstract, refined device service (in the example, "camera service"), which makes use of the capabilities of the base device service.

The OSGi specification also contains recommendations to facilitate the integration of an OSGi platform into *Jini* communities and networks using *UPnP* (Universal Plug 'n Play). The Jini Driver service will discover available Jini

[7] A private, conceptual entity of the Device Access service.

services, retrieve their proxy objects and register them with the OSGi framework. Thus, OSGi bundles can use Jini services without even being aware of the the fact that the Service Platform is Jini enabled, as long as they know which service interface name to look for. For every UPnP device, an appropriate device service is registered, which can be used by every bundle that is familiar with the UPnP way of modelling devices. Specifically enabled bundles may also export services to Jini communities and publish UPnP devices. This part of the specifications is presented in [15].

4.3.3 Communication

The *HTTP* service provides a lightweight WWW server which allows bundles to provide both static resources and dynamic content. For the latter, it makes use of the Java Servlet API. It also allows easy integration of basic authentication and especially lends itself to provide simple browser-based access to status and commands of a service. The HTTP service offers the possibility to provide browser-based access to status and commands of a service. Requests are mapped onto registered providers of static resources or Servlets according to the part of the alias name space they registered for. Static resources will be translated into a local URL, which can in turn be mapped to a bundle resource or the local file system. The HTTP service also allows the association of appropriate MIME types to content.

Java provides a powerful communication mechanism by allowing applications to extend URL Stream and Content Handlers in run-time. The registration of such an extension, however, is a one-time-only action and cannot be revoked, which is inappropriate for use within the OSGi environment. Therefore, the *URL Handlers* service hides the underlying mechanism and provides a service that allows scheme and content type handlers to be removed along with the bundle providing them.

In addition, the OSGi environment specification has also adopted the Java 2 Micro Edition Connector framework, which enables applications to use communication protocols without having to deal with implementation details. For this purpose, a set of interfaces defining the capabilities necessary for a certain type of connection (for example, constructing, sending and receiving a datagram) are defined. Depending on the requested connection scheme (a protocol identifier, like `sms:` or `socket:`), an application is required to implement the appropriate interface. The OSGi *IO Connector* service adds the possibility to have bundles extend the supported schemes at run-time.

The specification also proposes a *name space* for the use with these and other transport mechanisms, which allows nested domains using different name server technologies and is capable of designating both entities within an OSG and arbitrary external entities.

4.3.4 Management

Since the concept of a local console is incompatible with the majority of use cases for OSGi gateways, the *Log* service provides a central place to store error and debug information. It resembles the logging facility introduced in Java 2 Standard Edition 1.4, but is more lightweight in nature. It logs entries with time, severity level, the ID of the calling bundle, a clear-text message and an optional Exception object in case of an error. A limited history of the log is kept by the log service, but it also provides an event mechanism to distribute log events as they occur.

To allow the customisation of bundle behaviour, the *Configuration Admin* service offers a uniform way to provide them with a persistent set of parameters. Bundles participating in this scheme register an instance of `ManagedService` with the framework. Its interface contains a callback method which the Configuration Admin service will use to provide the configuration target with its current parameter set. This is done both at the initial registration and whenever changes are made to the parameter set. Multiple configuration targets are supported for service factories. Adjustments made by way of the Configuration Admin service usually immediately affect the global behaviour of a bundle, that is, all services it provides to other bundles. Bundles can inform management tools about the configuration parameters they accept by implementing the *Metatype* specification. This information can for example be used to automatically generate appropriate management user interfaces or be mapped to other metatype languages for use with an existing management system.

The *Initial Provisioning* specification addresses the question of how a management agent (i. e. one or more management bundles which connect to the operator's back-end management system) is initially loaded onto a service platform. Besides bootstrapping, a method for re-provisioning is defined as well to allow hand-over of the service platform management to another operator, who is possibly using a different management system.

4.3.5 Wiring

The *Wire Admin* service supports to "wire" together services which register themselves as producers or consumers of data. As such, it effectively introduces the concept of static assemblies to the OSGi platform, albeit in a simple form. Example producers include a switch, a temperature sensor, but also a GPS receiver providing the current geographical position, while consumers may be a lamp, a heater or a navigation system.

Producers and consumers will usually not be aware of each other and do not need to hold each other's service object. They are, however, notified if a wire is attached to or removed from them. Both can specify multiple "flavours" (class names) for the data they can use to exchange information to increase the chance of compatibility. Both pull and push semantics are supported. In the first case, the consumer polls the producer (via the wire) for the current value. In the second, the producer calls the consumer (again via the intermediate wire object) every time the value it provides has changed. To avoid flooding a consumer with unnecessarily frequent updates, a wire may apply filter criteria based on time intervals or value thresholds.

For cases where such a huge number of information items is to be delivered that registering them as separate producers or consumers would overwhelm the service registry, the concept of composite producers and consumers is supported. Also, the Wire Admin service may be used to connect external entities (like a TV set and a DVD player) by exchanging addressing information only, without unnecessarily relaying all data through the gateway.

4.3.6 Measurement

The *Measurement* class is a utility to simplify the common problem of correctly handling measurements of physical values. It encapsulates a numerical value together with its (numerical) error, SI unit and time stamp. It provides support for calculations with such values by tracking the resulting numerical error as well as the derived unit in case of multiplication and division operations. Addition or subtraction of measurements with different units are prevented.

The *Position* class is defined with location-based services for vehicles in mind, which profit from a consistent way of handling geographic positions. It uses Measurement objects to contain latitude, longitude, altitude, track and speed of an object at some point in time and will usually be used in conjunction with the Wire Admin service.

4.3.7 Structured Data Storage

The *Preferences Service* provides a simplified subset of the Java 2 Standard Edition 1.4 Preferences API. It allows bundles to store information in a structured tree-like manner that will persist across the bundle life cycle. Unlike the standard Java properties utility class, it also supports different data types and default values.

The Extensible Markup Language (XML) is a popular way to describe complex structured data in an open and portable manner. Not all XML Parsers are equivalent in function, however, and not all bundles have the same requirements on an XML parser. Java 2 specifies a common way of registering XML parsers as extensions (JAXP, Java API for XML Processing). The *XML Parser Service* specification defines a utility to support this mechanism in an OSGi Service Platform. It allows sharing of XML parsers in the form of bundles and provides interested bundles with the possibility of finding a suitable parser.

4.4 Challenges in Service Design and Implementation

Certainly, the key challenge in service design is the design of a good service interface with respect to the qualities generally desirable in service-oriented architectures. Besides hiding (and thus leaving open) as many implementation details as possible, it should lend itself to a variety of contexts. The interface should present a generalised view of the task it encapsulates, leaving efficiency and performance considerations to be addressed by different implementations. It should represent a highly cohesive unit of functionality with minimum outside coupling while staying as orthogonal to other services as possible. A service should focus on a specific task; not more, but not less either.

Ideally, the service interfaces they expose should be the only points of contact between OSGi bundles. If not possible otherwise, objects shared with other bundles through the same reference should be kept immutable whenever possible to avoid inadvertent coupling. Especially on the OSGi platform, there is also the issue of protecting bundle secrets to avoid mischief being done by code assuming a wrong identity. Access to the context object of a certain bundle will, for example, allow other bundles to request services in its name—and possibly let it pay the associated service charge for them.

Also, the required Java package footprint should match the target platform. As a general rule—which applies to OSGi standard services as well as any other standard API—one should always consider whether an existing API could be suited to a routine task (like, for example, logging) before inventing one's own. Limited available resources on the target platform can complicate this decision, however, especially when only a relatively small subset of a feature-rich API could be utilised.

The highly dynamic and long-lasting nature of an OSGi environment presents even further challenges to the service designer. Above all, one has to account for services the own code depends on not being available. This situation does not only have to be addressed at bundle start-up. Since services are able to leave—and return—at any time, it is necessary to use the framework event tracking mechanisms to avoid calling "dead" code, be able to notify own clients and take up standard operation again automatically when the service depended comes online again.

The same diligence is necessary as well whenever resources are held on behalf of a client. A bundle cannot depend on client bundles to clean up on services they use before leaving the active state. Even if they are properly designed to do so, a software failure may prevent them. Although the framework will automatically release held services when a bundle stops, it has no way of accounting for resources allocated by way of these services.

In the simplest case, this affects object references within the JVM. Therefore, references held on other bundles' objects should be nullified when no longer needed to allow them being garbage collected. Otherwise, referenced but unused objects may cause the JVM to run out of memory over time [22]. While this could be addressed through the use of weak references, resources outside the JVM cannot be handled this way, since finalizers are not guaranteed to be called at any specific point in time [33].

It is therefore necessary for service providers to associate such resources with the bundle that requested them to be able to release them manually. While the standard way is to use a service factory, the *Whiteboard Approach* [11, 28] offers an interesting alternative. By having event listeners register with the framework rather than the event source itself, the latter can take advantage of the automatic clean-up done by the framework to assist in releasing the allocated resources, since it will be notified when the listener service disappears from the registry and can act accordingly. The Whiteboard approach especially lends itself to situations involving a publisher-subscriber relationship.

Finally, the multi-threaded environment entails the need for synchronisation—and consequently, the possibility of deadlocks. This can involve relatively simple cases like two bundles which mutually depend on services provided by each other and are programmed to delay registration until this dependency is satisfied. Other issues may be less obvious. For example, events broadcast by the framework in response to changes in the service registry are delivered synchronously. A bundle declaring the corresponding event handler synchronized and calling the framework in response to withdraw its own service—in the same handler method—will run a high risk of halting the framework.

Deadlock situations may, among others, be the cause for a thread calling foreign code (like an event listener) not to return. A bundle should to be prepared for this possibility. Generally, concurrency-related issues require careful attention as design flaws involving race conditions are almost impossible to discover by testing.

All these issues are aggravated by the fact that—by the very nature of a component framework—much of the code a bundle interacts with will be of foreign origin (and thus a "black box" with eventually uncertain behaviour).

5 OSGi/EIB Integration

The OSGi platform strongly encourages a component-oriented approach to software design. Following up on this approach, a modular concept for making the functionality of EIB networks accessible in an OSGi environment will be presented. First, this chapter will discuss the issues arising when these two technologies are to be brought together and motivate the proposed solution. The following sections describe the service interfaces of the two core components. The prototype implementation of the low-level bus access service is covered as well. Finally, possible extensions of these core services are outlined, including a concept with the goal of supporting the automated configuration of EIB networks.

5.1 High-Level Design Considerations

The worlds of EIB and OSGi are quite different conceptually. While OSGi heralds "zero administration", the ability to adapt to changes dynamically and with as little management intervention as possible, EIB strongly relies on a qualified administrator even for small modifications to the system set-up. Besides this general attitude towards change, their fundamental mechanisms of communication differ as well.

Before working out this aspect, however, this section reviews the requirements for an EIB representation within a gateway platform. Finally, it is examined which parts of the EIB protocol stack are especially relevant for fulfilling the demands made and the design approach chosen on the basis of these considerations is delineated.

5.1.1 Requirements and Possible Benefits

Owing to the purpose of a gateway discussed in depth in Chapter 2, the primary task of an EIB access service will be to allow the exchange of data with EIB devices in regular operation. In the spirit of the network-independent nature of OSGi, the data should not be locked into a representation specific to a

particular protocol or language, but be made available to other bundles in an open form for further processing. This may not only include conversion to another network protocol, but also other uses like for example local logging.

It would be desirable to minimise the context specific knowledge client bundles have to be provided with by offering suitable abstractions for services offered by EIB devices. This will include recording descriptive information with SAPs and mapping value representations to their Java counterparts as far as possible. For example, service users would probably like to turn on the light in the living room by setting an appropriately named value to "true" rather than writing a more or less cryptic hexadecimal value to a similarly cryptic group address.

The service should work with the broad base of existing devices. This does not only mean that different physical media should be accommodated, but also that it should not depend on protocol features which are not widely supported. Furthermore, the integration of the gateway into an EIB system should be made as easy as possible.

Using the gateway as a bridgehead for maintenance would be of interest as well. While attempts to dethrone the ETS as the tool of choice for setting up entire EIB systems are not likely to be crowned by success (and are out of scope for a gateway device anyhow), the modification of selected application parameters has a more realistic perspective.

Values which are safe to be changed without in-depth system knowledge, for example dimming speeds or the power-on duration of a hallway light, could be exposed for modification by end users. Yet, it is practically infeasible to change the configuration of most existing EIB devices without support from their manufacturers or EIBA, as parameter data are stored at undocumented memory locations. Interface objects would offer an open alternative, but are not widely implemented yet.

At any rate, the ability to use the gateway as an iETS (EIBnet/IP tunnelling) server would certainly provide added value. EIB routing (again, specifically regarding EIBnet/IP) could be attractive likewise. Given the fact that an OSGi service will present a convenient high-level abstraction for EIB access, it should also lend itself to experimental prototype development. Yet, while its design should not preclude the realisation of additional and future demands like those outlined, it still has to comply with the main requirement of providing a lean interface for data exchange in regular operation.

5.1.2 EIB and the OSGi Device Access Model

One might expect that a suitable mapping of EIB networks into the OSGi environment would consist of services representing individual EIB devices, possibly leveraging the OSGi Device Access mechanism. This is inappropriate for a number of reasons, however.

The OSGi Device Access mechanism is designed to allow the platform to automatically adapt to changes in its environment without operator intervention. To this end, it is necessary that the networks it is connected to support some kind of automatic device discovery mechanism and provide a standard way for devices to provide enough information to allow the selection of a matching driver. Traditional EIB, however, being designed for a field of application where change is expected to be infrequent, provides neither. Without this basis, the automatic device refinement process is pointless.

While these issues could be worked around (by manually registering the needed information, if necessary), one encounters a more fundamental problem when attempting to expose functionality of a node as an OSGi service, as suggested in most OSGi-related literature [11, 15, 34]. This approach contains the tacit assumption that a device driver can "talk" to its target device without further ado to trigger some action. But due to the publisher-subscriber concept, access to application functionality of EIB devices in regular operation is associated with shared variables rather than devices. This means one cannot specifically change the state of a certain actuator, one can only do so through changing the state of the group (or one of the groups) it belongs to.

Even if accessing regular functionality using ad-hoc point-to-point communication was possible—as it could be once the EIB Interface Object communication mechanism will be implemented by a significant number of devices—, the issue remains that some device may silently maintain an assumption regarding the state of an actuator. Should the gateway improperly interfere with control of this actuator, the assumption will no longer match reality and probably result in erratic behaviour. This problem is not specific to EIB, but especially frequent here, since status feedback is not required by the protocol. In fact, this is a deliberate omission since it would thwart the performance benefit of multicast addressing in large installations.[1]

These observations entail two main consequences. First, exposing EIB nodes as OSGi device services is the wrong approach as far as group communication

[1] For a large number of lights to be turned on or off at once, a single group telegram—which is processed by all switching actuators—suffices. Yet, requiring all these actuators to return separate confirmations would result in substantial traffic again.

is concerned. Since the service access points of an EIB network are actually values shared via group addressing, these should be the entities to be exposed by OSGi services. Secondly, group communication relationships within the EIB network have to be designed taking into consideration the functionality to be provided by the gateway. It has to be ensured that group addresses exist which allow it to address devices with precisely the required granularity and that no other nodes are left with a stale state assumption due to such an intervention. Should the system contain more than one line, routers also have to be configured properly to ensure that messages related to relevant shared variables are propagated from and to the gateway. All this means that expert knowledge is required to insert a device into an EIB network. Given the state of art, this knowledge will have to be supplied by a skilled individual.

5.1.3 Consequences and Overview

When looking at the EIB protocol stack (as shown in Figure 3.10) with the above discussion in mind, its clear vertical division should catch the eye. It enables to conveniently pick the parts which need to be supported.

Obviously, full support for horizontal run-time (i. e. group) communication is required. A client bundle has to be provided with support comparable to the one the EIB application environment provides to the user application. This includes maintaining the state of a shared variable, being able to transmit updates and receive change events. Vertical run-time communication—being explicitly intended for the purpose of central monitoring and control—would be of interest as well, but is only scarcely supported in devices. Moreover, this way of communication will not provide change notification.

On the other hand, the benefit of supporting device management functionality is limited. This does not only include the Application layer services necessary for configuring other devices, but also the respective User layer servers for making the configuration of the gateway accessible to EIB management clients, specifically the ETS.

Actually, supporting incoming device management requests seems appealing, since the gateway will need to be incorporated in the overall EIB network configuration. This, as was discussed above, needs specific management intervention. Theoretically, the EIB protocol is flexible enough to transmit arbitrary configuration data, like extended textual descriptions for shared variables. Integrators could simply record the necessary data in ETS, which would then be transmitted together with the group addresses assigned. No further steps

would be required to set up the gateway, minimising the additional knowledge required from integrators.

In practice, however, it is unlikely that the necessary ETS support (probably in the form of a plug-in) can be added any time soon due to the significant efforts—and financial implications—entailed by the compulsory certification. As an interim solution, a "dummy device" representing the gateway can be included when defining the network set-up in ETS. The exported project data can then be completed and transferred to the gateway using a separate tool. Since device management[2] is the only part of the EIB network stack which depends on connection-oriented relationships, this functionality can be left aside at first. Actually, this removes the need for the entire Transport layer, since multicast addressing is effectively provided at Layer 2.

When transparent accommodation of physical media and underlying hardware while retaining the option to add arbitrary elements of the protocol stack at a later time is desired, available options quickly boil down to implementing the Data Link layer interface. The services of the EIB Data Link Layer do not place tight real-time requirements on clients and allow uniform access to the physical media available. Exposing this interface as an OSGi service provides the desired hardware abstraction while retaining maximum flexibility in handling upper-layer protocols.

This interface can immediately be used for tunnelling and routing applications (including EIBnet/IP), provided the source address of outgoing frames can be set freely. It can also support a component supporting group communication, which would take care of the assembly of the proper Layer 7 PDUs and possibly caching of shared variable values. Such a group communication service would also be a good place to provide support for the conversion of variables from EIS into Java syntax.[3] As the EIB Network layer is practically empty for end devices, such a module would be able to sit directly on top of the Data Link layer abstraction. Likewise, a component for accessing interface object properties could be added equally easily.

Actually, all these components can even operate in parallel as long as the base abstraction allows multiple clients. This allows the separation of concerns regarding network operations in a vertical way in addition to the exclusively horizontal divisions of the standard protocol stack. Obviously, the resulting

[2] Access to protected interface object properties is limited to connection-oriented communication as well. "Safe-to-change" properties can be assumed to be freely accessible, however.

[3] Such functionality is already outside the EIB network stack, strictly speaking.

flexibility has to be handled carefully. The gateway operator has to ensure that competences are clearly divided between such components, either based upon of the addresses they are prepared to process (group or individual) or by means of handling non-intersecting protocol aspects only which can be unambiguously discerned by their respective PDUs. Also, complications are bound to occur since such a division was not intended by the designers of the EIB network stack. For example, while extending the standard Transport layer to allow concurrent outgoing connections is straightforward, this issue is not as trivial for incoming connections, which do not provide a means for determining the appropriate handler (as do for example IP port numbers).[4] Especially as long as one keeps to the "natural" vertical divisions of the protocol stack, however, the fine-grained decomposition possible provides clear benefits in enabling the resulting overall solution to be as lightweight as possible.

Adopting such an approach also suggests multiple horizontal divisions, which raises concerns regarding performance. Although the latter would certainly be optimised by implementing an EIB access service largely in native code with no more than a thin Java interface layer, it seems a viable approach to sacrifice some of it for added flexibility. Since interactions between OSGi services are effectively implemented as direct method calls, the incurred overhead can be expected not to be inadequate. Actually, [38] describes a modular gateway architecture whose individual modules communicate via IP, even carrying the overhead of the network stack at each call—obviously with no further problem. Moreover, the low data rate of EIB will not allow more than a few dozen messages per second[5] (of practically negligible size each) to enter the system in the first place. So, it seems viable to opt for utmost flexibility by putting an abstraction hierarchy in place, with the low end only encapsulating which cannot be addressed adequately from within the Java environment, specifically tight real-time requirements and hardware-dependent issues. Further components can then fill in exactly the parts of the network stack (and functionality beyond it) as needed.

The following sections will describe a low-level interface and a group communication support component aligned with this concept. While this covers the parts of the EIB network stack which have been identified as most important

[4] A possible solution to determine the appropriate handler service would be to always accept an incoming connection and offer the first frame—which contains the Application Layer PDU—to all clients in turn, then break the connection if no handler accepts.

[5] The theoretical peak value for Twisted-Pair EIB is 48 `GroupValue_Write` messages per second.

(and provides a clear path for adding the remaining parts), the question how individual devices are to be represented suitably remains open still.

As was discussed, due to the design of the EIB network stack representing individual devices as OSGi services will only prove beneficial when maintenance aspects are of interest. Even then, such an approach is not straightforward since EIB provides neither discovery nor sufficient self description mechanisms.[6] Yet, Section 5.1.2 suggested that these issues could be worked around. The finishing section of this chapter attempts to live up to this promise by outlining a vision of an advanced driver architecture which would expose maintenance aspects of EIB devices in the form of a hierarchy of device services based upon successive refinement. These service, would also provide open, high-level access to the configuration of the EIB devices they represent. Still, the restriction holds that such an architecture could only come to life with the necessary configuration data being supplied by manufacturers or EIBA.

Regarding possible ways for presenting application-related functionality, it seems clear that it will have to be exposed separately from any services representing individual devices since group addressing is so deeply rooted in the EIB protocol. Also, semantic information has to be provided to client services to enable them to actually make use of the shared variables made accessible to them. Also, these variables need to be grouped and sorted according to their functionality. These issues merit further investigation and will be covered thoroughly in Chapter 6.

5.2 EIB Frame Service

The *EIB Frame service* (EFS) provides the capability to exchange EIB Data Link layer protocol frames with other EIB devices. Its purpose is to encapsulate hardware dependencies and particularly timing-sensitive aspects. This is achieved by modelling its interface on the services provided by the EIB Data Link layer, which also provides uniform access to the different physical media available. Although access to and custom handling of almost all elements of the protocol frame is provided, the client is spared most of the tedium of frame assembly and disassembly.

Besides the Data Link protocol control information, the EFS also takes care of encoding and decoding the Network Layer control field (routing counter). Since end devices do not need to further process this information, the EIB

[6] Interface Objects will alleviate the second problem, but are not widely implemented yet.

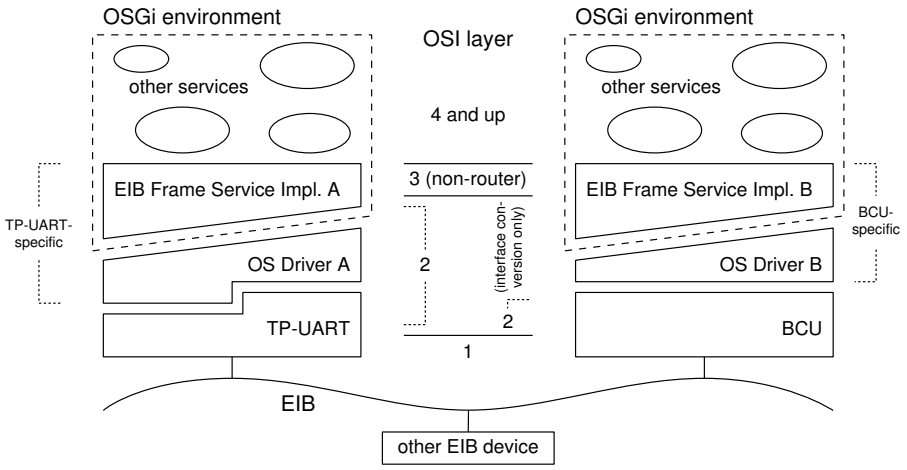


Figure 5.1: Hardware-Software Collaboration

Frame Service also covers Network Layer functionality for them. Routers, on the other hand, have the freedom to implement arbitrary routing functionality on top of the EFS.

As illustrated in Figure 5.1, an implementation will have to be specific to the EIB communication controller used and the corresponding low-level device driver of the host operating system. Depending on them (and the amount of specification conformance desired), it will provide a particular subset of the features accessible through the common service interface. For example, a TP-UART based solution can easily allow the dynamic configuration of a node's physical and group addresses, while a BCU is not designed for this information to be changed frequently.

To allow a client service to check on the actual capabilities of the service object implementing this interface, the latter will advertise its actual capabilities (for example, the number of group addresses supported) as service properties together with its service registration. Should the client service choose to request unimplemented functionality nevertheless, an exception will be raised. Global settings are managed via the Configuration Admin service. For example, the default source address for outgoing frames is made available as a configuration property. When it is changed, a TP-UART based node will simply change the corresponding internal variable, while a BCU-based one will need to write the change to the communication controller's EEPROM using local maintenance mode. Since this is an expensive operation, the BCU-based implementation will not accept requests for frame transmission with another than the default source address, effectively restricting changes to bundles which possess the necessary

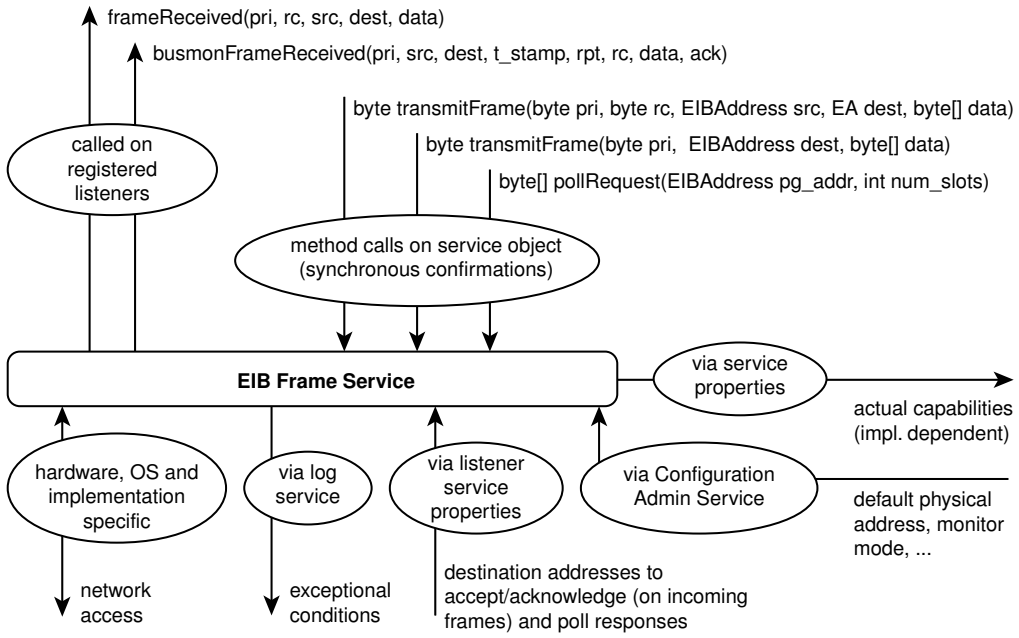


Figure 5.2: EIB Frame Service: Information Flow

administrative permission to access the Configuration Admin service. Other examples for configurable settings include whether the EIB hardware interface is operating in standard or bus monitor mode or whether a repeater is installed in an Powerline EIB system.

5.2.1 Service Interface

Figure 5.2 shows how the EFS interacts with clients and the environment. It is withdrawn when communication with the EIB is not possible for any reason and re-registered as soon as connectivity returns. Such exceptional conditions, which may be of interest to an operator, are passed to the log service.

To transmit a frame, a client service simply calls the appropriate method on the EFS service object. This method will not return until an acknowledgement from the remote Data Link Layer was received or the local Data Link Layer gave up retrying. The result is then returned as the method return value.

To satisfy special requirements, one method allows to specify a maximum of control information: priority class, routing counter value, source address, destination address and user data. For normal use, an overloaded method is provided where the EIB Frame Service will supply default values as configured.

In order to allow different parts of the network stack to be handled by separate components, the EIB Frame service provides support for multiple client services using the Whiteboard approach. To receive incoming frames, client services register themselves with the OSGi framework as `EIBFrameIndicationListener`, passing as service properties the source or destination addresses of frames they are prepared to process. The EIB Frame Service tracks these registrations and causes acknowledgement frames to be sent in response to matching incoming data frames. The latter are then distributed to all clients which declared interest.

When a client is stopped, its listener service registration is automatically removed by the framework. Since it also notifies users of a removed service, the EIB Frame service can accordingly modify its list of addresses. Thus, the list of incoming frames to acknowledge (which can be considered resources external to the framework as discussed in Section 4.4) can be kept accurate over time. Using the Whiteboard pattern to leverage the service registry for storing information related to service interactions also has further benefits. Client services do not have to re-register as listeners after periods of unavailability of the partner service, and the communication relationship is immediately visible for management purposes.

Since bus monitor mode indications differ in significant details from indications generated in regular operation mode (in particular the inclusion of repeated frames), a separate handler method is included on the listener interface. Listeners need not implement it if not needed. With usual bus access hardware, entering bus monitor mode entails the deactivation of normal communication, although set-ups are imaginable which may provide both at the same time. The interface is designed to accommodate this possibility. The listener interface also contains a handler for error indications (which will for example be delivered in case the properties describing the relevant addresses could not be parsed).

The EIB Frame Service interface also accommodates polling mode. In the unlikely case that the local node is to act as slave, poll octets to be returned are provided as service properties of a listener service, which is otherwise empty except for an error handler.

Source and destination addresses are encapsulated by the immutable utility class `EIBAddress`, which also provides high-level methods for constructing, parsing and comparing them. It also covers the needs of Powerline EIB and allows to describe address ranges. `EIBAddress` also defines and supports an unambiguous string notation for the representation of EIB addresses.

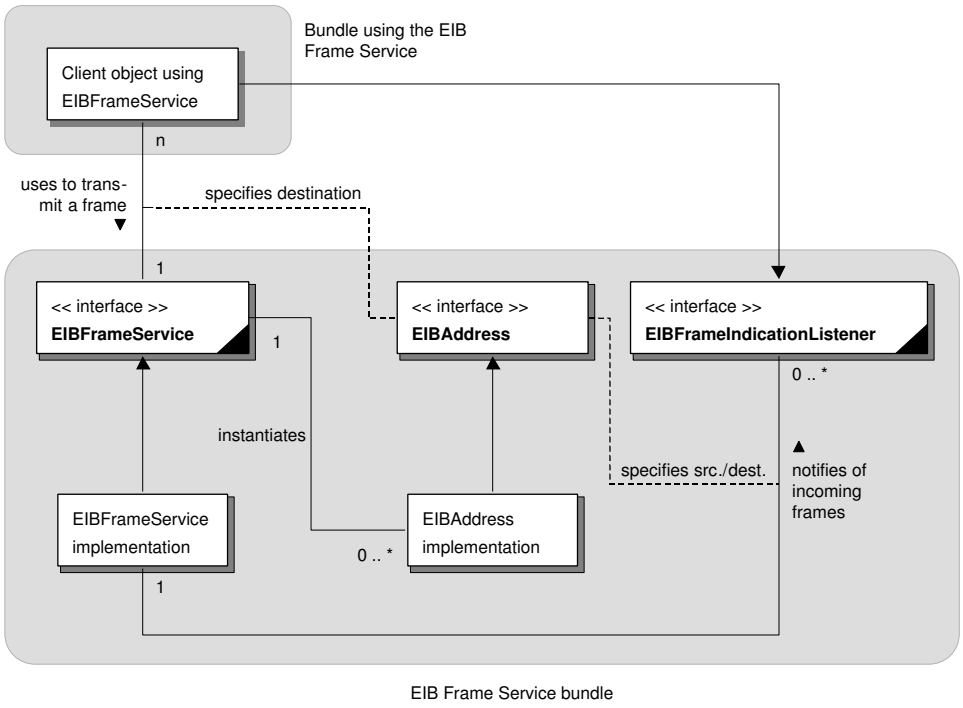


Figure 5.3: EIB Frame Service: Class Diagram

Figure 5.3 shows the relationship of the interfaces and classes making up the EIB Frame Service in the style used in [34]. The interfaces which are exported are shown in bold. They reside in a package separate from the one holding the implementation classes (although they belong to the same bundle). The service interfaces which are registered with the framework are marked with a black triangle in the lower right corner.

To minimize unwanted dependencies between the EIB Frame Service and a client service, the factory pattern is employed for `EIBAddress` objects. Instead of invoking a constructor on `EIBAddress`, client services request instances via a method on the `EIBFrameService` interface. `EIBAddress` objects are used to qualify received and transmitted frames. This is indicated by the dashed lines. For simplicity, the polling group responder interface is omitted from the illustration.

5.2.2 Prototype Implementation

A prototype implementation of the EIB Frame Service was realised on a Linux system using the TP-UART driver available from [13], which provides access to

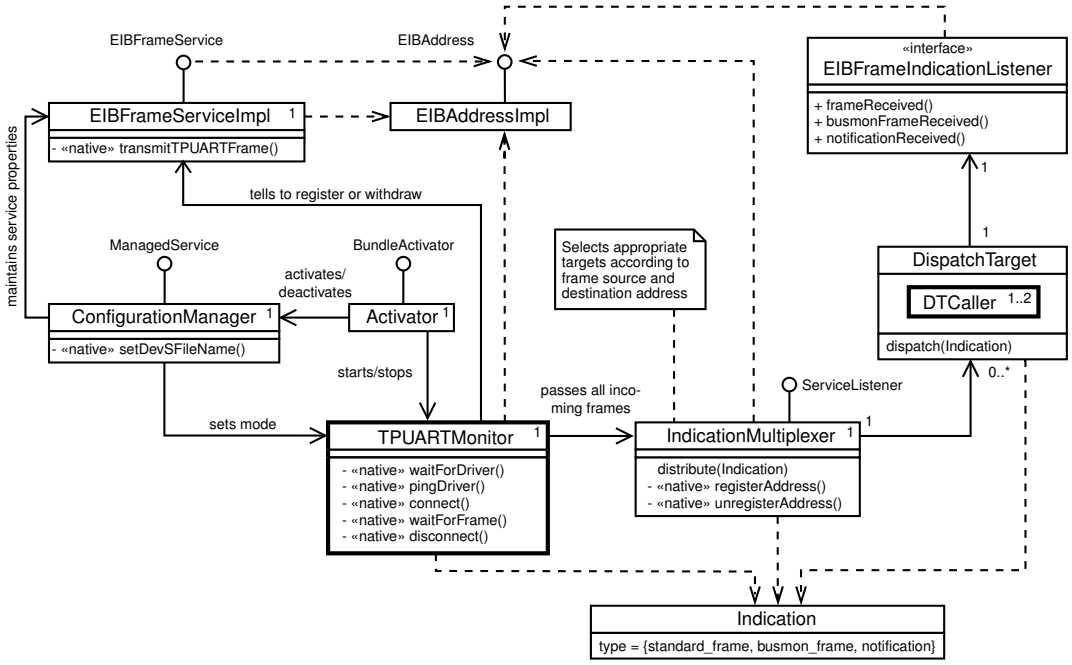


Figure 5.4: EIB Frame Service: Prototype Implementation Class Diagram

the TP-UART as a character-oriented device. The Gatespace SGADK, which is no longer distributed, provided the necessary OSGi platform implementation.

The class structure is shown in Figure 5.4. Access to the low-level driver had to be handled entirely by native methods, since it relies on `ioctl()` calls to define the destination addresses of received frames it will pass on and acknowledge. Also, blocking reads were used to avoid busy waiting for incoming data. Yet, the bundle had to remain able to respond to stop requests. Defining the read operation in a native method allowed to let it time out using the `select()` system call. After every time-out, a check is made whether a stop request was received. If this is not the case, the cycle starts anew.

This design proved especially useful as it turned out that the driver would not check for error conditions (for example due to the failure of bus power) after the initial `open()` call. Therefore, the device file is closed and reopened ("pinged") regularly in this loop to allow withdrawing `EIBFrameService` when EIB access is not available. In this case, the driver is continuously polled as well to detect connectivity to reappear again.

This main loop is contained in `TPUARTMonitorThread`, a singleton[7] which is started by the bundle activator (contained in `Activator`). At bundle start-up,

[7]Singletons are denoted by a "1" in the upper right corner. `TPUARTMonitorThread` is also an *active class*, as can be seen from its strong border.

the `ConfigurationManager` is activated as well, which causes it to register its callback with the Configuration Admin service. The name of the device special file is passed to the bundle as a configuration property and is maintained as a static variable within the native library. Since the OSGi specifications call for configuration properties to be echoed to the service registration, the `ConfigurationManager` also maintains the service properties of the `EIBFrameService`. Only a single implementation of the latter exists. Its method for transmitting frames is synchronized, which ensures that write accesses to the low-level driver will not overlap.

Whenever a frame is received, `TPUARTMonitorThread` constructs an appropriate `Indication` according to the current mode of operation (standard or bus monitor) and passes it to `IndicationMultiplexer` for distribution. This class is responsible for maintaining the list of `EIBFrameIndicationListener`s present (by listening to service events issued by the framework) and configuring the low-level driver accordingly. The state of every listener service is encapsulated as an instance of `DispatchTarget`. Every `DispatchTarget` is associated with its own `DTCaller` thread, which is used to invoke the callbacks of the listener. This allows events to be dispatched in a timely manner despite slow or malfunctioning listeners. Should a `DTCaller` not be ready again when the next indication is to be delivered, a second one is created for the sole purpose of delivering a failure notification to the respective listener, which is suspended from receiving further indications until both caller threads have returned.

For testing purposes, a prototype EIB Frame Service client bundle was implemented as well. It presents a graphical user interface for interacting with the service, which is shown in Figure 5.5. Its main window indicates whether an EFS instance is currently registered (multiple concurrent instances are not supported). If so, its service properties are shown. A minimal client for the Configuration Admin service is implemented as well to allow changing the operation mode of the EIB Frame Service. Any number of Frame Service client windows can be created on request. Each of them implements an independent `EIBFrameIndicationListener`, processing and displaying either bus monitor or standard mode indications. Standard mode clients also allow the transmission of EIB data frames. For the graphical user interface, AWT[8] was chosen over Swing for its thread-safe nature. This simplified the implementation, as changes to the display can be made directly from the indication handler method.

[8] Abstract Windowing Toolkit

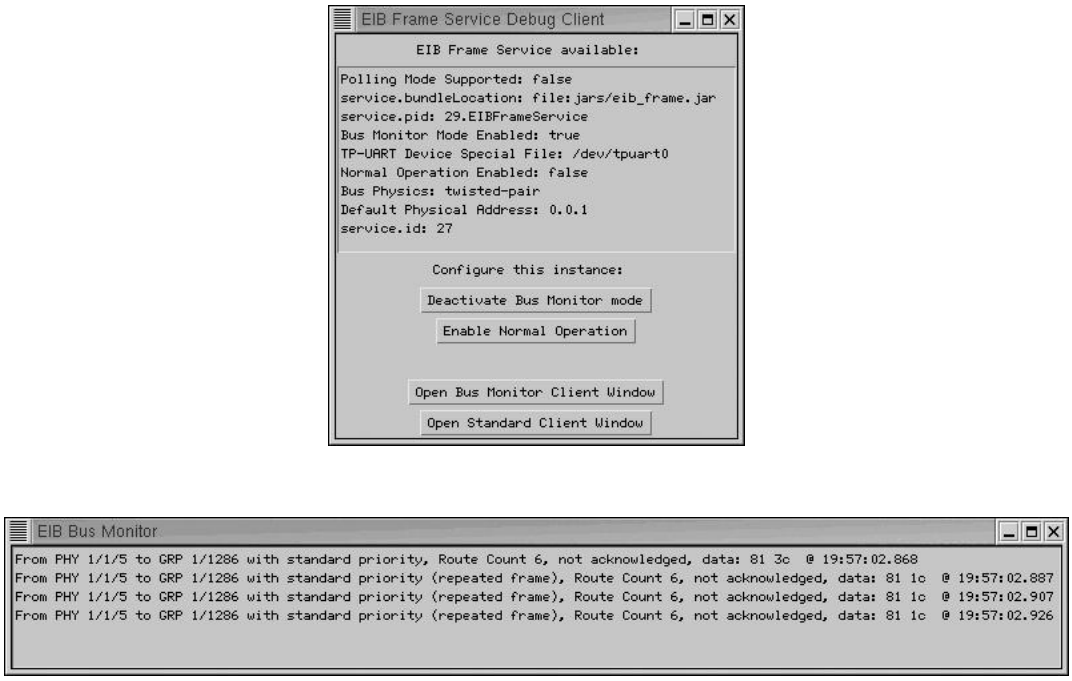


Figure 5.5: EIB Frame Service: Screenshot of Prototype Client

5.3 EIB Group Communication Service

A service supporting EIB horizontal runtime communication obviously has to provide an implementation of the appropriate services of the EIB network stack as described in Section 3.6.2. Since the necessary Application layer services are specifically designed for synchronising group objects with a maximum length of 14 octets only, every message related to publishing or requesting a shared value fits within one Data Link layer frame. Consequently, this task merely involves the correct interpretation and generation of six bits of Layer 7 protocol control information.

Translating transmitted values between the EIS bit-level syntax and appropriate Java data types is another important task of the *EIB Group Communication* service illustrated in Figure 5.6. Besides transmitting updates of a shared variable to other group members, it will also monitor any given group address, maintaining a copy of the last value distributed to this address via the network for the use of client bundles.

Clients wishing to monitor the state of a particular shared variable register an appropriate listener service with the framework, including the group addresses to monitor as service properties and are notified in case of an update. This

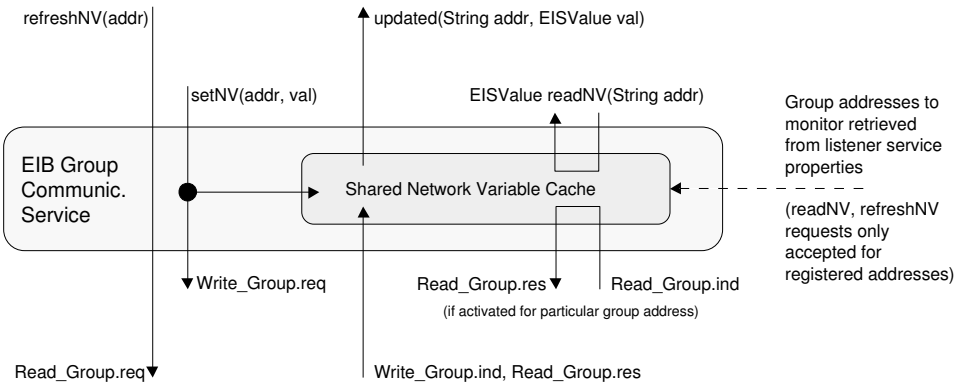


Figure 5.6: EIB Group Communication Service: Information Flow

pattern of event distribution is very similar to the one employed by the EIB Frame Service.

Upon a request for the state of a particular shared variable, its cached value is returned. Read requests to be issued to the network have to be called for explicitly. They will automatically refresh the cache. In both cases, it is required that the particular group address was previously registered for monitoring. The local cache is refreshed as well when a new shared variable value is written to the network. In this case, clients are notified of the change in the same way as they would be when the update is received via the network to ensure a consistent view.

In much the same way as `EIBAddress`, a utility class provides the conversion of EIS to Java types. The manager service passes byte arrays encapsulated by this utility class to listeners, which have to apply the proper conversion by calling the appropriate `getAs...()` method.

The EIB Group Communication service will also act as group responder, if requested. It is necessary to keep in mind, however, that it can act only within the limits of the given topology and routing set-up. It is therefore necessary to ensure that group addresses to be written or listened to can actually pass from their source to their destination.

5.4 Advanced Driver Architecture

This section outlines a vision of representing EIB devices which would permit to make their configuration accessible to other services in an open, high-level way. Also, it would enable the remote management of EIB systems where tunnelling is not applicable due to excessive network delay.

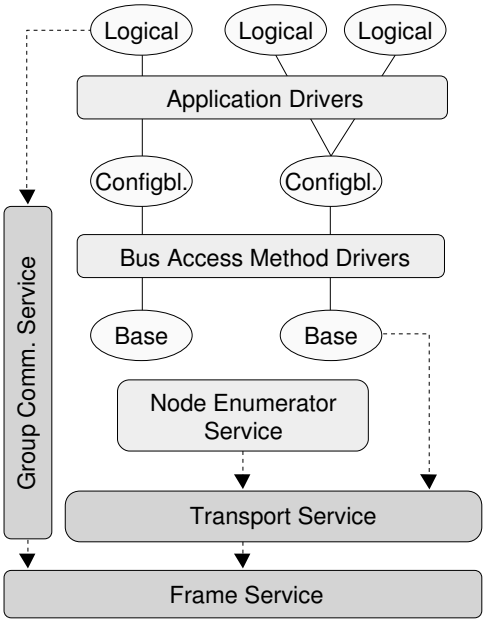


Figure 5.7: "Smart" Driver Architecture

The architecture presented in Figure 5.7 makes use of OSGi device refinement to identify devices and expose maintenance related functionality in subsequent steps. It assumes the existence of a Transport layer service. Appropriate Application layer PDUs are assembled by the driver services themselves.

The refinement process starts with the Node Enumerator Service being provided with the address of a previously unknown device (it could also conduct scans of given address ranges to find one—or more—of them). It will then register a base device containing environment related information, like its physical location (not the location of effect), wiring (e. g. whether a combined device is wired as a blinds or a series actuator) and its individual node address (which it will make available for change). Base devices can also provide their device descriptor (also referred to as mask version) since this ID can always be retrieved in an implementation independent manner using a dedicated device management service. The device descriptor specifies the communication controller (Bus Access Unit, BAU) of the device in question.

Since different BAUs have their specific ways to retrieve and modify device configuration data, the next refinement step is to select an appropriate driver on the basis of this device descriptor. This driver is then able to retrieve the manufacturer ID and application version, which it registers alongside with its representation of a configurable device. Configurable devices provide high-level

methods to load a new application and to access the address and association table, handling access protection where needed.

Using this information, the Device Manager can select a matching application driver. This driver possesses semantic information about the application loaded on the EIB device which may include a standardized description of its function (switch, push button, staircase light, ...) or whether assumptions about the state of communication partners are made. A configurable device may be split up into multiple logical devices (for example representing multiple channels).

Application drivers also maintain the association between group addresses and their group objects. This means it would theoretically be possible to connect representations of group objects (for example, by way of the Wire Admin service), with the application drivers automatically modifying the configuration of the participating devices. Incompatible assignments would be prevented thanks to the semantic information held by the driver services. It is also conceivable that the configuration of any routers present on the network is updated automatically as well. Such a solution would actually encapsulate the expert knowledge currently provided by the system integrator within the driver services.

6 A High-Level Control Abstraction

In Section 5.1.1, the need for representing the application functionality of EIB devices which becomes accessible through an interface component in a way suitable for the use of other bundles was mentioned. Obviously, it would be desirable to use a wide-spread, standard representation to maximise the choice of components which can be combined with the provided abstraction. This would be in the interest of service providers, gateway operators and end users alike. Yet, there is no comprehensive, widely accepted standard. Actually, there even seems to be only a handful of potential candidates.

The functional blocks defined by the KNX interworking profiles certainly are most closely related to EIB technology-wise. Although they look promising—Konnex Association even has entered a cooperation with the European Committee of Manufacturers of Domestic Equipment (CECED) with the goal of defining interworking profiles for white goods—, they are still in the making. The same holds for the standardised UPnP Device Control Protocols (DCPs) [45], of which only a handful are published yet—with merely two ("HVAC" and "Lighting Controls") being relevant to home automation. Notwithstanding that, UPnP seems to have considerable support. Specifically, UPnP was declared the successor for the "Home Plug&Play" set of interworking profiles [12] defined by the United States-based CEBus Industry Council, none of which however seem to have ever been implemented by any significant number of products. Also, Echelon presented a UPnP bridge for LonMark[1] devices. LonMark in itself provides a considerable source of functional profiles as well, which are albeit strongly oriented towards LonWorks.

In the field of building automation, the BACnet standard [1] is already well-established. It was specifically designed to bring diverse "islands" of control together on the automation level to provide unified monitoring and command capabilities from a single workstation or control room master display. Yet, BACnet imposes a necessarily very technical point of view, dealing with alarms,

[1] cf. Section 3.1.4

events and object access (analogue input, loop controller) on an abstract level, limiting its usefulness for home automation purposes.

So, none of the available candidates seems a perfect match. Furthermore, all these standards deal with functional blocks. On reflection, this would actually be a limitation within an EIB system, where data points can be bound individually. Here, every group address could be considered a device in its own right.

Taking this idea one step further, one could expose them as individual services. This opens up interesting possibilities. Permissions can actually be assigned at the level of individual data points in this case. Also, they can quite easily be registered as individual Producer or Consumer services, allowing them to be connected using the Wire Admin service.

Actually, there is no reason to confine these data points to EIB shared variables. Data points (or even entire functional blocks, after breaking them up into their individual elements) from entirely different technological contexts can be accommodated as well. Thus, a technology-neutral representation is achieved.

6.1 Architecture Overview

The remainder of this chapter will present an interface to be used by OSGi services presenting added value to the end user during regular operation. Its task is to supply them with status information of and accept control messages for FAN devices. Its aim is to present the available control points in a uniform, technology-agnostic way, so that client services need not deal with specifics of a particular FAN. Although state-based in its nature, it also provides an event-based update notification service. It is explicitly not intended to provide a generic way of addressing set-up and configuration issues, as their intimate relation to the specifics of a field bus design makes this a highly involved task. The FAN (or possibly multiple FANs or other automation systems) whose functionality is exposed through this interface are assumed to be properly pre-configured. This also includes properly setting up the gateway as a communicating party to enable it to actually receive all information to be exposed and access functionality of other nodes without unwanted side effects.

The binding of client services to these data points is expected to be accomplished manually by an operator or the end user him/herself. For this purpose,

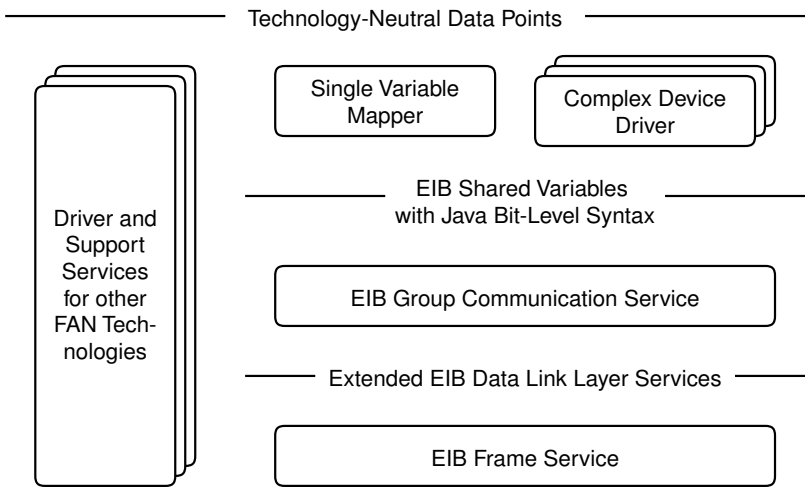


Figure 6.1: System Architecture

a scheme for enriching them with *semantic information* is proposed, which allows to present data points clearly arranged for easy identification.

As shown in Figure 6.1, components specific to their particular field bus designs will implement this interface, translating the network-specific to the common representation and addressing device-specific issues.

For EIB, these *driver components*[2] will build upon the protocol stack components presented in the previous chapter. Since the application level communication paradigm of EIB is state-based as well, it will in many cases be sufficient to maintain a 1-to-1 relationship between a data point service and an EIB shared variable. A single bundle will be sufficient to perform this translation for any number of such values, like temperature sensor readings or wall switches. Nevertheless, a driver can also perform arbitrary additional processing, like a moving average.

More complex devices like dimming or drive actuators are controlled through multiple group addresses and will require more effort regarding conversion into the common representation. They are therefore likely to be covered by device-specific drivers. By providing them with access to exactly the same shared variables as the device they represent (as well as internal parameters), such drivers can internally reproduce its behaviour. This way, they could provide the permanent status feedback demanded by the top-layer abstraction even if

[2]The use of the term "driver" should not obscure the fact that the OSGi Device Access specification is not involved in this concept in any way—except for aspects of nomenclature.

it is not available from the device (as it is for example the case with drive actuators).

All components shown in Figure 6.1 are services within the OSGi environment. No assumption is made concerning their combination into bundles for deployment. The following sections will present the key concepts of the abstraction proposed—breaking functionality up into a pool of uniform data points and reapplying structure by arranging these in tree-like manner according to the effect perceived.

6.2 A Pool of Data Points

The abstraction proposed aims to provide a simple model for representing the functionality available within a building automation network in a bus technology independent way. It is specifically targeted at the residential application domain and smaller-sized functional buildings and confines itself to handling control information only. An important goal is to support the construction of a clearly structured and immediately comprehensible user interface for monitoring and control purposes as well as binding data points to client services.

The abstraction provided can be used by any OSGi service. This may be an HTTP server which presents a control panel for local or remote use, a service which will send an SMS when the washing machine is detected to be spilling water or one implementing any of the technologies discussed in [40]. It should be noted that these services need not be IP-based. As an example, consider an SMS service which operates using a GSM modem connected directly to the OSGi platform host.

Besides remote services, purely local applications (as for instance presence simulation) can easily be implemented as well on top of this abstraction. Although the host becomes a single point of failure in this case (which suggests that elementary functionality should not rely on logical connections made this way), this is a perfectly viable approach for comfort functions.

A key benefit of a technology independent abstraction is to ease the work of service providers and gateway operators, who are spared of having to provide and administrate multiple versions of a service with otherwise identical value to the end user. Moreover, it also plays an important role for local integration. Although one will initially select a single FAN technology to cover all demands, one may simply not have the choice for later additions. This may be due to the fact that the chosen technology does not support some new requirement,

like device discovery, which is indispensable for the integration of loose goods. Such an abstraction also allows to cleanly integrate OSGi services as data point providers. For instance, the presence simulation component may allow its activation and deactivation this way.

The approach chosen is a *state-based* one, breaking down system functionality into values of primitive type. Every of these data points is represented in a uniform way. Technology independence is achieved by using real-world objects and properties as entities in the representation of the process image. This high-level semantic information is presented in textual form to the user only, however. Behavioural aspects are explicitly not addressed by this model.

As a consequence, complex devices providing multiple points of control are exposed as a set of unrelated data points. This is not a problem regarding interaction with other software components, but certainly one for human users. To enable them to pick the right data point from this pool, another main constituent of the approach proposed is a concept for associating data points with additional semantic information pertaining to their effect location and purpose. This concept is not limited to data points belonging to the same node, but has universal applicability.

6.3 Functional Aspects

Every data point is registered as a separate service object. In addition to a common service interface, a set of required property keys is specified, which enables client services to query the Service Registry for data points meeting specific criteria.

Although the state-based approach used may be reminiscent of the EIB application-level communication model, there are some major differences. First, the semantic information associated with data points is available for perusal by examining their service properties during regular operation. Secondly, this information is not related to field bus nodes, but associated with real-world entities. Third, the state-based paradigm is applied with full consequence. In an EIB system, data sinks are explicitly free to silently change application related status associated with a group variable. For example, an actuator controlling a stairway light usually switches off after a pre-set time-out without announcing the state change on the network. In contrast, data points are required to accurately reflect the status of their associated real-world entity at any time. This has to be ensured by the respective driver component, by

- Persistent unique identifier (PID)
- Object affected (air, door lock, water, glass, HVAC control, ...)
- Property of object (temperature, state, presence, integrity, ...)
- Read/only (sensor) or read/write (actuator)
- Data type descriptor (constant with specified mapping)
  - Boolean, long int, float, string, ...
  - Time stamp, day of week, time of day
  - OSGi Measurement
  - State Set
  - Event (memory store/recall)
- Physical unit or state labels
- minimum/maximum value

Figure 6.2: Data Point Properties

dead reckoning if necessary. That way, the entire state of the system is always available.

Despite this obvious benefit, a purely state-based model cannot accommodate actions like the sending of an alarm message, which do not possess state by their very nature. To be able to include such functionality in the data point model as well—for example, to have a panic button trigger it—, a special type of data point representing an event is provided.

Furthermore, one has to be aware of certain implications of a state based model with respect to consistency. For example, it is necessary to always separate set points from data points reflecting actual values. While this may be obvious when regarding the room temperature, dimmers capable of smooth transitions between light values will give rise to the same problem. Only when the effects of a control interaction are instantaneous and unconditional (for example, switching a simple light), set point and actual value can be folded into the same data point.

Also, special care has to be exercised whenever data points affect or imply the state of others. For example, a data point exposing a "house mode" control with the states "away" (all lights off) and "festive" (all lights on) will have to be able to assume at least a third state (probably "custom") to reflect the very likely condition that some lights will be turned on, while others are not. Again, not all situations where this rule applies are this obvious.

The properties of a data point are enumerated in Figure 6.2. The "Property" property holds the aspect sensed or controlled of the real-world object described by the "Object" property (e. g. "speed"). This "Object" is defined in a way independent of the specific use this information is put to (i. e. "wind", not "thunderstorm warning"). Both the "Object" and the "Property" property can hold free-form text.

In addition to the standard set of native Java data types, a means of specifying a certain day of week or time of day is included for the use with timer programmes. Also, data points which can enter a number of discrete, mutually exclusive states can be described properly. If present on the platform, the OSGi Measurement utility class can be leveraged as well.

Each of these types is associated with a specific constant value, which allows a client component to automatically generate appropriate user interface elements. For the same purpose, Boolean and multi-state types are accompanied by a set of state labels. The physical unit of numerical values can be provided as free text.

Concerning interaction with other services, it is obvious that multiple clients have to be supported. To ensure consistency, no read accesses must occur during the value being written. Implementing this requirement using the Java programming language is straightforward by declaring read and write access methods as synchronized.

For notification on update events, the Whiteboard approach again offers an elegant solution. By registering a single listener service, an interested party can receive update events from all data points. To identify the event source, its PID will accompany every notification.

The concept of a *pool of data points* integrates perfectly with the OSGi Wiring scheme. Given a suitable user interface, being able to "wire" data points can offer the end user a powerful, yet reasonably easy-to-handle tool for customizing platform functionality. Also, logging functionality based on time intervals as well as value thresholds can be implemented in a straightforward fashion. Yet, such criteria can only be associated with a single data point.

Concerning security and privacy, the OSGi framework already provides the necessary mechanisms to ensure that only trusted bundles are able to use data point or even low-level services unless they are given the proper permissions. Finer granularity of access control can be achieved by requiring these services to keep track of their client bundles by implementing a Service Factory. This could then be used to grant write access to a certain data point to trusted components while limiting others to read access.

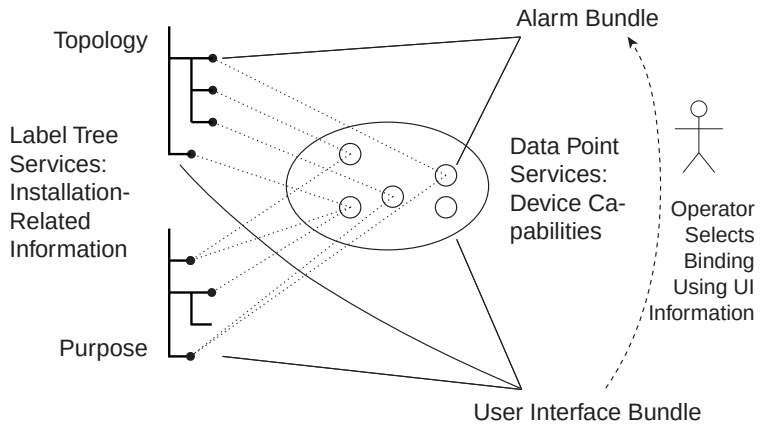


Figure 6.3: Data Points: Semantics Distribution and Binding

If a client bundle provides an entry point for user authentication (like the HTTP service), this information can be passed to the data point service addressed. Another extension would be to limit the frequency of access for certain bundles or users as a privacy measure. For example, remote meter reading could be allowed once a month only.

All these approaches benefit from having data points available as separate services for more fine-grained control. Although all these efforts are void as soon as an attacker gains access to the EIB physical medium, they raise the barrier for attacks via a remote connection.

6.4 Presentation

As has been detailed already, individual data points hold only information associated with the capabilities of a FAN device. Yet, this alone is insufficient for the end user, for whom the actual use delivered by such a device within his or her living environment is relevant.

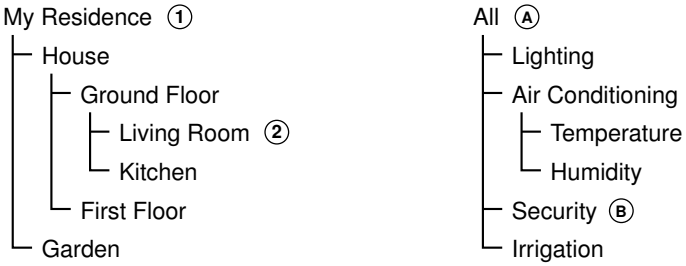
Therefore, the concept proposed includes two persistent *tree data structures* holding information pertaining to the location of effect and purpose of data points. Every tree node is labelled with a free-text description and references a list of data point via their PIDs. Client components still access functionality associated with data points using the respective service interfaces of the latter. They will only be concerned with the tree data when the need arises to identify a data point they are dealing with towards human users. This concept is illustrated in Figure 6.3.

Regarding the topological structure, storing the *location of effect* was chosen over referring to the physical location of the associated node for the reason of three specific characteristics of the envisioned field of application. First, nodes are frequently installed in distribution cabinets, with passive cabling leading to the—often drastically different—actual location of effect, which is the one relevant to the end user. Secondly, an approach strictly aligned with physical topology cannot easily accommodate higher-level functions, such as one that allows to switch off all power outlets on the ground floor reachable by small children. Last, but not least, EIB as a wide-spread representative uses a communication model which renders such an approach plainly impossible.

Therefore, an approach is adopted which from a technical point of view can be considered *function-oriented* (as opposed to structure-oriented) [31, 40] in that it entirely disregards the underlying FAN topology. Yet, far from ignoring structure, it imposes it on the functional attributes of data points. Maintaining it in tree structure helps to ensure consistency better than having every data point hold this information individually (consider renaming "mezzanine" to "first floor").

While a data point will only appear once in the tree structure describing the location of effect (so it is fully qualified by its path plus its object and aspect), it may appear multiple times in the one describing its purposes. Window blinds, for example, keep out the sun as well as potential intruders.

Although the actual design of the user interface is not prescribed, Figure 6.4 shows a possible example. Part of the available locations of effect and purpose are displayed on the left-hand and right-hand sides respectively. The list of data points is filtered according to the tree nodes selected. No restrictions are made at first. Next, the selection is narrowed to data points related to the living room. Finally, the scope of view is further limited to security-related data points within this room.



①
Ⓐ

House - Ground Floor - Living Room	Air	Temperature	22 ˚C
House - Ground Floor - Kitchen	Dishwasher	Remaining Time	50 min
House	House	Automation Mode	Short-Term Away
Garden - Front Flower Bed	Irrigation Pump	Activity	Off

②
Ⓐ

	Heating/AC	Temp. set point	23.5 ˚C
	Air	Temperature	22 ˚C
South Window	Blinds	Position(down)	30%
Patio Door	Door	Lock	Locked

②
Ⓑ

South Window	Blinds	Position (down)	30%
Patio Door	Door	Lock	Locked
Patio Door	Glass	Integrity	Undamaged

Figure 6.4: Data Point Presentation: Possible User Interface

7 Conclusion and Outlook

Extensive control networking lies at the heart of ‘smart” homes and buildings. EIB is a popular and reasonably powerful field bus technology aimed explicitly at this purpose. Besides a unified approach towards configuration, one of its key assets is the wide range of components available. This is further multiplied by the fact that bindings of device data points can be determined individually. This flexibility however entails a certain complexity, requiring special training from integrators.

Yet, especially its high-level communication model, which is based on a combination of value-based semantics and overlapping groups, can prove troublesome. The design of the behavioural EIS types in particular can be considered questionable, as the state-based nature of the EIB application-level communication model was found not to be consequently implemented.

By connecting control networks with one another and the outside world, gateways undoubtedly have a central role in leveraging the potential of the former. OSGi offers a powerful framework to dynamically configure their functionality. Nevertheless, the platform is open for various uses and not fixed on a particular business model. It allows the run-time combination of software components, while still remaining suitable for resource-limited devices. It supports a service-oriented infrastructure as well as a matching programming model. Its advantages, however, come at the price of subtle dependencies providing pitfalls to the unwary software engineer. Taking into account that a gateway platform should operate continuously, a defensive style of programming seems definitely recommended.

Concerning the integration of these two worlds, it has become apparent that EIB cannot fully exploit the dynamic capabilities of an OSGi environment due to their static nature. Moreover, bringing them together is not straightforward owing to the fact that, concerning regular operation, the functionality of devices can only be accessed by addressing groups of nodes. Specifically, it is necessary to consider the functionality to be exposed by the gateway when configuring an EIB network. In spite of these difficulties, a solution was proposed which provides the optimum support possible for device driver components. In that, it

surpasses solutions already on the market, like the EIB driver bundle available for the ProSyst "mBedded Server" [39, 46]. This product only offers very basic assistance for group communication (for example, no translation of EIS types) and will not serve multiple clients. Therefore, it will not support fine-grained decomposition of the EIB protocol stack.

For technology-neutral representation of FAN functionality, a consequently state-based, data point driven solution was adopted. Data points are associated with extensive meta information, allowing the automatic construction of user interfaces. The concept also includes update notifications for clients and is designed for immediate integration with OSGi Wiring. In addition, basic directions for ensuring security and privacy were given. Regarding presentation, the drawbacks of an approach oriented on the FAN physical topology were discussed. Consequently, a function-oriented approach was taken, which relates data point functionality to real-world entities meaningful to the end user. Nevertheless, it imposes a clear structure on the functional properties of a data point, replacing the physical topology by a tree of locations of effect. The ProSyst "mBedded Server" product also includes a control unit abstraction, which however has an obviously different focus than the approach presented here.

Concerning future directions, switching over to an open-source OSGi implementation would prove valuable to better analyse potential deadlock situations. Also, the EIB Frame Service could be designed even more robust by implementing indirection to the service object.

Further, the new possibilities introduced by KNX should be investigated. This includes the integration of the new "plug-and-play"-related features (A-Mode) introduced by the KNX standard, which allow to exploit the dynamic nature of an OSGi platform more fully. Regarding E-Modes, the gateway could also serve as a configuration controller. In addition, the proprietary Internet remote service architecture announced by EIBA and Konnex Association should be examined for possible integration as soon as it is actually published.

Also, support for further FANs technologies needs to be implemented. Here, LonWorks comes to mind specifically as it is widely used in building automation and a comprehensive solution for device access is readily available [10]. Considering spontaneous networking protocols, the integration of the approach presented here with Jini and UPnP-enabled networks will further increase its usefulness.

Finally, the work of the ISO/IEC Home Electronics Standards working group (briefly described in [47]), especially its HomeGate standard for residential gateways, should be taken into account as well.

The most promising aspect of this work seems to be the generic control abstraction. An article based on the present thesis [26] was accepted for presentation, which strongly suggests that this concept should be pursued further, for example by providing for frequency-based access permissions.

List of Abbreviations

N. B. *Only abbreviations not in current use which appear frequently in the present thesis are included.*

API	Application Programmer's Interface
BAU	Bus Attachment Unit
BCU	Bus Coupling Unit
BPSU	Bus Power Supply Unit
EIB	European Installation Bus
EIBA	EIB Association
EIS	EIB Interworking Standard
ETS	EIB Tool Software, Engineering Tool Software
FAN	Field Area Network
HVAC	Heating, Ventilation, Air Conditioning
KNX	KNX (Konnex)
LSB	Least Significant Bit
MSB	Most Significant Bit
OSGi	Open Services Gateway Initiative
PDU	Protocol Data Unit
PEI	Physical External Interface
PL	Power Line
PLC	Programmable Logic Controller
RF	Radio Frequency
SAP	Service Access Point
TP	Twisted Pair

Bibliography

N. B. *Many of these documents are published in printed form as well as on the World Wide Web. Owing to the volatile nature of the Web, the electronic version is not referenced in these cases.*

[1] ANSI/ASHRAE 135 (2001): BACnet – A data communication protocol for building automation and control networks.

[2] ANSI/ASHRAE 135 Addendum d (2001): BACnet – A data communication protocol for building automation and control networks.

[3] ANSI/EIA/CEA-709.1 (1999): Control networking standard.

[4] Keith Bennett, Paul Layzell, David Budgen, Pearl Brereton, Linda Macaulay, and Malcolm Munro. Service-based software: The future for flexible software. In *Proc. 7th Asia-Pacific Software Engineering Conference (APSEC 2000)*, pages 214–221, 2000.

[5] Guy Bieber and Jeff Carpenter. Introduction to Service-Oriented Programming (Rev 2.1). Available at `http://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf`, April 2001.

[6] Grady Booch, Ivar Jacobson, and James Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1998.

[7] Humberto Cervantes. The concept of service. Available at `http://www-adele.imag.fr/BEANOME/serviceconcept.htm`, March 2003.

[8] Humberto Cervantes and Jean-Marie Favre. Comparing JavaBeans and OSGi towards an integration of two complementary component models. In *Proc. 28th Euromicro Conference on Component Based Software Engineering*, pages 17–23, 2002.

[9] Humberto Cervantes and Richard S. Hall. Automating service dependency management in a service-oriented component model. In *Proc. 6th Workshop on Component-Based Software Engineering (CBSE)*, May 2003.

[10] Sergey Chemishkian. Building smart services for smart home. In *Proc. IEEE 4th International Workshop on Networked Appliances*, pages 215–224, 2002.

[11] Kirk Chen and Li Gong. *Programming Open Service Gateways with Java Embedded Server Technology*. Addison-Wesley, 2001.

[12] CEBus Industry Council. Home Plug & Play Specification 1.0. Available at `http://www.cebus.org/Files/hpnp10.zip`, 1998.

[13] Deggendorf University of Applied Sciences High-Tech Center for Modern Communication Systems. EIB Linux driver. Available at `http://www.hto.fh-deggendorf.de/komm/englisch/elinux.html`, 2003.

[14] Dietmar Dietrich, Wolfgang Kastner, and Thilo Sauter, editors. *EIB – Installation Bus System*. Publicis MCD, 2001.

[15] Pavlin Dobrev, David Famolari, Christian Kurzke, and Brent A. Miller. Device and service discovery in home networks with OSGi. *IEEE Communications Magazine*, 40(8):86–92, August 2002.

[16] EIB Association. *The EIB Handbook Series 3.0*, 1999.

[17] EN 50090: Home and building electronic systems (HBES).

[18] ENV 13154-2 (1998): Data communication for HVAC applications – Field net – Part 2: Protocols.

[19] ENV 13321-2 (1997): Data communication for HVAC applications – Automation net – Part 4: EIB.

[20] Ted Farrell. Service-oriented architecture. *Java Developer's Journal*, 9:12–14, April 2004.

[21] Philip Babcock Gove, editor. *Webster's third new international dictionary of the English language, unabridged*. G. & C. Merriam Company, 1981.

[22] Ethan Henry and Ed Lycklama. How do you plug Java memory leaks? *Dr. Dobb's Journal*, 25(2):115–119, February 2000.

[23] Bruce Horowitz, Nils Magnusson, and Niclas Klack. Telia's service delivery solution for the home. *IEEE Communications Magazine*, 40(4):120–125, April 2002.

[24] IEC/EN 60929 Annex E (2003): Control interface for controllable ballasts.

[25] ISO/IEC 7498-1 (1984): Information technology – Open systems interconnection – Basic reference model: The basic reference model.

[26] Wolfgang Kastner and Georg Neugschwandtner. Service interfaces for field-level home and building automation. Accepted for presentation at the 5th IEEE International Workshop on Factory Communication Systems (WFCS'2004).

[27] Konnex Association. *KNX Specifications, Version 1.1*, 2004.

[28] Peter Kriens and BJ Hargrave. Listener pattern considered harmful (draft). Available at `http://www.osgi.org/devzone` (White Papers section), 2001.

[29] Choonhwa Lee, David Nordstedt, and Sumi Helal. Enabling smart spaces with OSGi. *IEEE Pervasive Computing*, 2(3):89–94, July-September 2003.

[30] Hannes Leidenroth. *EIB-Anwenderhandbuch*. Verlag Technik, 2003.

[31] Maxim Lobashov, Gerhard Pratl, and Thilo Sauter. Applicability of Internet protocols for fieldbus access. In *Proc. 4th IEEE International Workshop on Factory Communication Systems*, pages 205–213, August 2002.

[32] Dave Marples and Peter Kriens. The Open Services Gateway Initiative: An introductory overview. *IEEE Communications Magazine*, 39(12):110–114, December 2001.

[33] Joel Nylund. Memory leaks in Java programs. *Java Report*, 4(11):22–30, November 1999.

[34] OSGi Alliance. *OSGi Service Platform Specification, Release 3*. IOS Press, 2003.

[35] Robert Ott and Heinrich Reiter. Connecting EIB components to distributed Java applications. In *Proc. 7th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '99)*, volume 1, pages 23–26, 1999.

[36] Michael P. Papazoglou and Dimitrios Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46:25–28, October 2003.

[37] Randall Perrey and Mark Lycett. Service-oriented architecture. In *Proc. 2003 Symposium on Applications and the Internet (SAINT 2003) Workshops*, pages 116–119, 2003.

[38] Gerhard Pratl, Maxim Lobachov, and Thilo Sauter. Highly modular gateway architecture for fieldbus/Internet connections. In *Proc. 4th IFAC Conference on Fieldbus Systems and Their Applications 2001 (FeT'2001)*, pages 293–299, 2002.

[39] ProSyst Software AG. ProSyst mBedded Server EIB Package API Documentation. Available at `http://dz.prosyst.com`, 2002.

[40] Thilo Sauter, Maksim Lobashov, and Gerhard Pratl. Lessons learnt from Internet access to fieldbus gateways. In *Proc. 28th Annual Conference of the IEEE*, volume 4, pages 2909–2914, 2002.

[41] Rainer Scherg. *EIB planen und installieren.* Vogel, 2002.

[42] A. Sillitti, T. Vernazza, and G. Succi. Service oriented programming: a new paradigm of software reuse. In *Proc. 7th International Conference on Software Reuse: Methods, Techniques, and Tools (ICSR 7)*, volume 2319 of *Lecture Notes in Computer Science*, pages 269–280. Springer, 2002.

[43] Clemens Szyperski. *Component Software – Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[44] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, third edition, 1996.

[45] UPnP Forum. Standardized device control protocols. Available at `http://www.upnp.org/standardizeddcps/`.

[46] Dimitar Valtchev and Ivailo Frankov. Service gateway architecture for a smart home. *IEEE Communications Magazine*, 40(4):126–132, April 2002.

[47] Kenneth Wacks. Home systems standards: achievements and challenges. *IEEE Communications Magazine*, 40(4):152–159, April 2002.