

# **EIB Handbook Series**

**Release 3.0**

## **Volume 2: Guide for Development**

### **Part 1: Cookbook**

#### *Chapter 1: EIB System Introduction*

21.05.1999

# Table of Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>GENERAL INTRODUCTION.....</b>                                 | <b>4</b>  |
| 1.1      | To this Cookbook .....   | 4         |
| 1.2      | Introduction to EIB Technology .....                             | 5         |
| 1.3      | EIB Product Development.....                                     | 6         |
| <b>2</b> | <b>INTRODUCTION TO THE EIB SYSTEM.....</b>                       | <b>9</b>  |
| 2.1      | Bus Access Units .....   | 9         |
| 2.2      | EIB System Structure.....  | 10        |
| 2.3      | Existing EIB BAU's .....   | 11        |
| 2.4      | Existing EIB Implementations.....                                | 13        |
| <b>3</b> | <b>COMMUNICATION ON THE EIB SYSTEM .....</b>                     | <b>14</b> |
| 3.1      | Data Exchange and Transmission.....                              | 15        |
| 3.1.1    | Group Addressing .....   | 15        |
| 3.1.2    | Transmission and Reception of a Communication Object Value ..... | 26        |
| 3.1.3    | Fast Polling Mode.....   | 31        |
| 3.1.4    | Broadcast Addressing.....  | 31        |
| 3.1.5    | Connectionless / Connection oriented addressing .....            | 31        |
| 3.2      | Communication Model and Telegram Structure .....                 | 32        |
| 3.2.1    | Physical Layer .....   | 33        |
| 3.2.2    | Data Link Layer.....   | 34        |
| 3.2.3    | Network Layer .....  | 38        |
| 3.2.4    | Transport Layer .....  | 39        |
| 3.2.5    | Application Layer.....   | 42        |
| <b>4</b> | <b>EIB INTERWORKING STANDARDS (EIS).....</b>                     | <b>45</b> |
| 4.1      | EIS 1: Switching .....   | 46        |
| 4.2      | EIS 2: Dimming .....   | 46        |
| 4.2.1    | EIB-Subfunction 'Position'.....                                  | 47        |
| 4.2.2    | EIB-Subfunction 'Control'.....                                   | 47        |
| 4.2.3    | EIB-Subfunction 'Value'.....                                     | 47        |
| 4.2.4    | Behaviour .....  | 48        |
| 4.3      | EIS 3: Time.....   | 50        |
| 4.4      | EIS 4: Date .....  | 51        |
| 4.5      | EIS 5: Value.....  | 52        |
| 4.6      | EIS 6: Scaling .....   | 52        |
| 4.7      | EIS 7: Drive Control .....                                       | 53        |
| 4.8      | EIS 8: Priority .....  | 55        |
| 4.9      | EIS 9: Float Value.....  | 55        |
| 4.10     | EIS 10: 16-bit Counter Value .....                               | 56        |
| 4.11     | EIS 11: 32-bit Counter Value .....                               | 56        |
| <b>5</b> | <b>EIB APPLICATION.....</b>                                      | <b>57</b> |

|       |  |     |
|-------|--|-----|
| 5.1   | Application program .....  | 57  |
| 5.1.1 | User Initialisation Program (U_Init) .....                         | 57  |
| 5.1.2 | User Save Program (U_Save).....                                    | 57  |
| 5.1.3 | User Main Program (U_Main) .....                                   | 58  |
| 5.2   | Software Services .....  | 58  |
| 5.2.1 | Communication Object Manipulation Functions .....                  | 58  |
| 5.2.2 | EEPROM Manipulation Support Functions.....                         | 59  |
| 5.2.3 | Application Support Functions.....                                 | 60  |
| 5.2.4 | PEI-support functions.....   | 61  |
| 5.2.5 | Timer Functions .....  | 62  |
| 5.2.6 | Message Handling Functions.....                                    | 63  |
| 5.2.7 | Arithmetic functions .....   | 64  |
| 5.2.8 | Miscellaneous Functions .....                                      | 64  |
| 6     | DEVELOPMENT TOOLS .....  | 67  |
| 6.1   | Reference Installation .....                                       | 67  |
| 6.2   | Assembler, Cross Compilers .....                                   | 70  |
| 6.2.1 | Assembler.....   | 70  |
| 6.2.2 | ByteCraft C-compiler .....   | 70  |
| 6.3   | EIB Interoperability Test Tool (EITT) .....                        | 75  |
| 6.3.1 | Sending Telegrams .....  | 75  |
| 6.3.2 | Receiving Telegrams .....  | 76  |
| 6.4   | Busmon .....   | 77  |
| 6.4.1 | Device Mode .....  | 78  |
| 6.4.2 | Monitor Mode .....   | 79  |
| 6.4.3 | Send Mode .....  | 79  |
| 6.5   | Emulators and Simulators .....                                     | 79  |
| 6.6   | ETS, ETS+, IDE .....   | 81  |
| 6.6.1 | ETS (EIB Tool Software) .....                                      | 81  |
| 6.6.2 | Device Definition with the ETS2 ManufTool .....                    | 82  |
| 6.6.3 | Product Certification.....   | 92  |
| 6.6.4 | Product Distribution.....  | 93  |
| 7     | APPENDICES.....  | 94  |
| 7.1   | Appendix A: Header Files for ByteCraft C-Compiler .....            | 94  |
| 7.1.1 | eiba0.05h .....  | 94  |
| 7.1.2 | eiba1.05h .....  | 97  |
| 7.1.3 | eiba2.05h .....  | 103 |
| 7.2   | Appendix B: Manufacturers providing Tools for AP Development ..... | 108 |

# **1 General Introduction**

## **1.1 To this Cookbook**

This Cookbook is intended as a guideline to development of applications for the European Installation Bus, EIB. Though this is initially not thought as a reference for product development, it can however answer on what is or what is not possible with some or other implementation of EIB, when regarding the capacities of the media or the system components, and in this way guide you to the proper hardware selection.

Several implementations of the EIB protocol do already exist. As a consequence of this and the modularity of the technology as well, different approaches are possible. This openness is reflected in this Cookbook.

This chapter is a general introduction to the EIB system, independent of the medium or any hardware. It gives a description of the existing EIB system, the communication on EIB and a generalised model of an EIB product. Finally, it also describes the integration of an EIB product in the database and the certification procedure.

Each of the following chapters is dedicated to one medium. The medium specific communication is explained and the range of system components is discussed. A lot of attention is paid to examples.

Chapter 2 will introduce you to the Twisted Pair technology.

Chapter 3 will introduce you to the Power Line technology.

Chapter 4 ....

### 1.2 Introduction to EIB Technology

EIB is a home and building automation system. All systems that make a building to be a home or an office, irrespective of its size or complexity, can easily be controlled and supervised.

EIB still allows the user to control the classic tasks, such as lighting and heating control, but adds flexibility and simplicity to otherwise complex operations.

EIB will also take over “invisible” processes, like air-conditioning, access-control and power management, joining them into this one integrated intelligent system.

Essentially, the EIB system is a decentralised structured system.

The EIB as a structured decentralised bus system is built up from EIB products, that are linked via bus lines, which can be of one of the supported media. The EIB bus protocol is managed together by all products on the bus. It is running in the microprocessors in every individual component. There is no need for a specific central computer, managing the bus operation.

This system provides services to the application program that may run in the products and makes use of its facilities.

EIB products communicate directly with one another. When a sensor is triggered, its Application Program (AP) orders the underlying system software to send a telegram. This telegram is received by one or more actuators, that react in a prescribed way, as configured by their application parameters.

The system itself is contained in the products when the system component manufacturers sell them. The application programs from the EIBA manufacturers however, and the description of the product they belong to and their AP-parameters are mainly stored in a customer's database.

Designing an EIB installation thus stands for choosing the products in the database, choosing the appropriate AP and defining the combined action between the products. This can all be achieved by using single software: the EIB Tool Software, ETS.

An EIB product is nearly always made up of a Bus Access Unit (BAU)<sup>1</sup> and an Application Module (AM). These can be two separate modules to be connected to one another over the Physical External Interface (PEI), or they can constitute an inseparable entity as well.

The EIB is in a continuous evolution. Several implementations and mask-versions of the EIB system software are already created, depending on the medium or the implemented microcontroller.

These systems are described in the following chapters.

---

<sup>1</sup> For the definition of Bus Access Unit (BAU) and Bus Coupling Unit (BCU) please refer to paragraph 2.1.

## 1.3 EIB Product Development

Developing an EIB product thus stands for choosing an appropriate BAU, designing the hardware (Application Module, AM) and writing the software (= Application Program, AP) and having it certified, so that it can be used in the customer's ETS-database.

The application developer has to decide, based on the estimated required resources his application will need, what specific BAU he is going to use. The BAU's differ in:

- computing power
- available memory (RAM and EEPROM)
- functioning of the interfaces  
(possible serial interfaces towards external processors)
- available services, like system software services, implemented layer services and management services
- mechanical characteristics

Some knowledge (data book) of the used micro-controller and programming experience, assembler or high-level language is recommended.

While starting the hardware development, one has to consider certain limitations of the BAU and the system. These can be the possible pin-functions of the PEI or the supported software services, as well as the regulations for CE-marking and EIBA-certification. This is all described in the relevant sections of the EIBA Handbook Series.

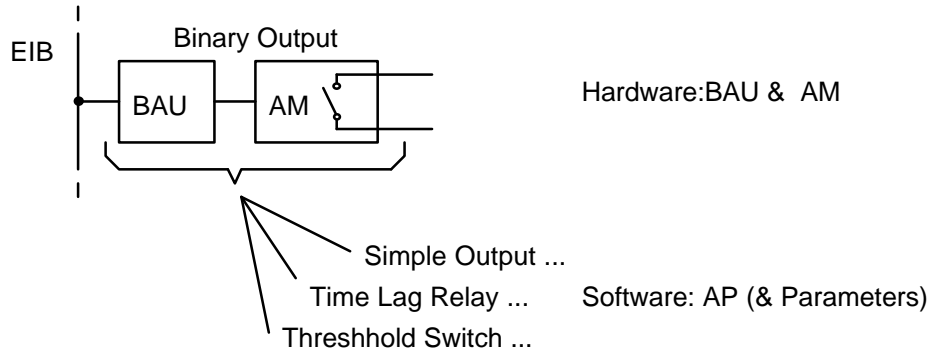
Software developers can find the necessary information in the different EIBA publications, such as this Cookbook, Volume 3 of the EIBA Handbook Series or during EIB Developer's courses given at the EIBA certified training centres, or at the EIBA General Directorate in Brussels.

In this chapter we will describe the general procedure of EIB product development. On this occasion we will also describe the hardware and the software tools required for product development. As an example is often the best way to show solutions of complex problems, we will illustrate in some essential steps the product development by means of simple examples in the following chapters.

We will frequently use the term 'EIB Product' or simply the term 'Product', since EIB software tools use this term. This is the right place to briefly explain this term.

An EIB product consists of a combination of the following components:

- A Bus Access Unit (BAU),
- an Application Module (AM) without computing power and
- an Application Program (AP).

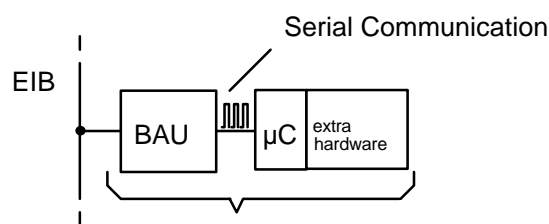


**Fig. 2/1/1-1: Example for EIB-Products**

Fig. 2/1/1-1 shows an example of hardware (BAU & AM) representing a binary output. Depending on the assigned AP you may obtain different products from this hardware, such as a binary output, a time lag relay or a threshold switch.

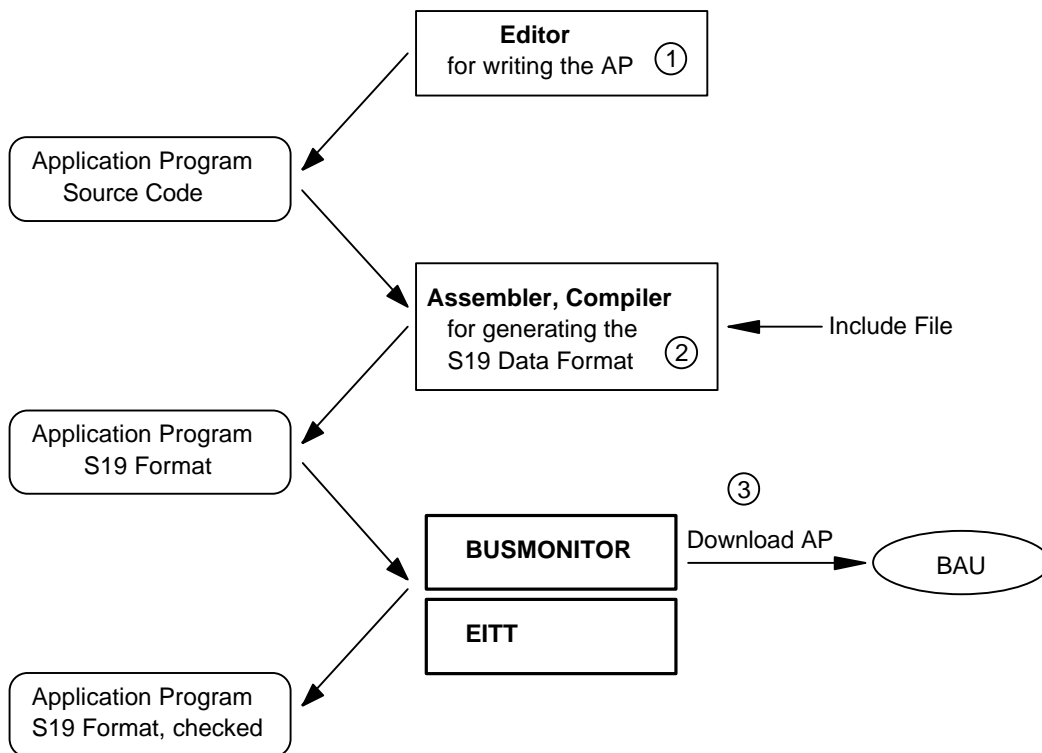
Another possible combination could be:

- a Bus Access Unit
- an Application Module or additional hardware with microcontroller
- an application software embarked on the additional microcontroller



**Fig. 2/1/1-2: EIB Product with external Microcontroller**

Fig. 2/1/1-3 shows the procedure of AP development. The software tools are represented by rectangles and the generated data format is shown in the round edged fields on the left.



**Fig. 2/1/1-3: Events during Application Development**

In order to develop your AP you have to carry out the following steps:

- ① Writing the source code of the AP. For this you may take any editor.
- ② Conversion of the source code into machine code, using the Motorola S19 data format. Any assembler or compiler, able to produce the format required for the applied processor in the BAU can do this. In order to facilitate programming, you may store program parts frequently used (e.g. the names of the software services) in an include file. These program parts will be added to the AP during the generation of the machine code.
- ③ Loading the machine code of the AP into a BAU by means of the *Busmonitor*. The Busmonitor contains several software modules that you may use in order to check the function of your product. Test of the functionality and interworking of the application with EITT.

This procedure will be demonstrated by an example in the medium specific chapters.



## **2 Introduction to the EIB system**

### **2.1 Bus Access Units**

A Bus Access Unit (BAU) is a basic system component, just like the repeater and the line coupler. A BAU consists mainly of two parts: a medium access module and a communication controller.

The medium access module:

- acts as a data-interface, i.e. separates/modulates the data signals from/on the medium specific carrier signal.
- optionally provides the power supply voltage(s) to the other parts of the device (communication controller and application module) if the power is fed via the medium
- generates some control signals for the communication controller, such as reset and save-signals.

The communication controller is mostly a microcontroller, with RAM, ROM and EEPROM.

- the ROM contains the pre-loaded fixed EIB system software, which can not be overwritten.
- the EEPROM contains the system parameters, for setting some system behaviour, and the application specific data, mostly the application program itself and its parameters.
- the RAM is used partly by the system software and by the application.

A BCU is a particular BAU that also provides the EIB standard PEI (Physical External Interface). This is a 10- or 12-pins connector to which an Application Module can be plugged and that can be configured in many ways (from output channels to serial communication interface).

A BAU will implement the OSI-communication model as a whole or only partially, or may extend it with new functions.

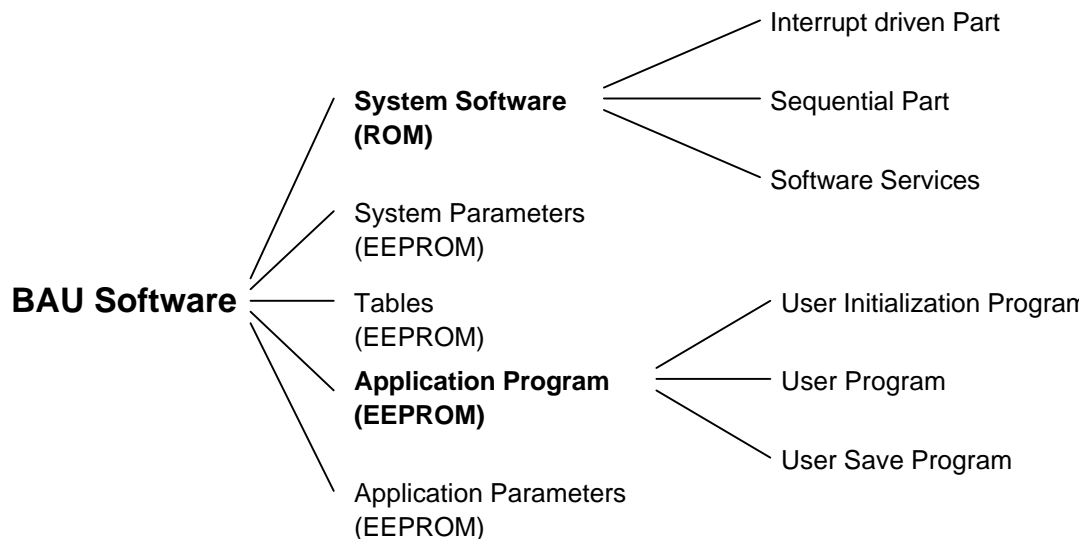
## 2.2 EIB System Structure

The system software structure depends on the BAU-type.

In general, it contains (at least) the following parts (functions):

- a part for handling communication
  - \* functions managing the communication
  - \* interaction with the user-program via communication object flags and values.
- a part for checking internal integrity (check routines) and managing the device.

In the following chapters we will give some more detailed information about the structure and tasks of the BAU's software. Fig. 2/1/1-4 shows a general view of the software components.



**Fig. 2/1/1-4: Structure of the BAU-Software**

We have already mentioned, that the BAU contains two kinds of software: the system software and the application program with data. The system software is made by the manufacturer of the BAU and does not vary with the application, whereas the application program is always adapted to its respective task.



In BCU 1 the application program is called as a subroutine from the system software. In BIM M112 and BCU 2 however, processor time is split up in slices, one of which is assigned to the application program.

By means of the *system parameters* the application program writer can adjust the basic behaviour of the BAU. The system parameters should be well distinguished from the *application parameters*. Whereas the same system parameters exist for all applications, the application program writer may define application parameters, as desired. In contrast to the system parameters, the values of application parameters are adjusted by the electrician during installation of an EIB system. The introduction of application parameters extends the range of application of a given application program. As an example we may think of a bus device realising a time-lag relay. In this case the time interval can be adjusted by an application parameter. Without an application parameter for each time interval, a separate AP would be necessary.

### 2.3 Existing EIB BAU's

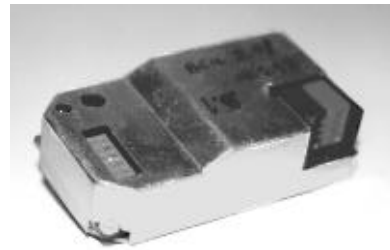
Bus Access Units are offered in various solutions differing from one another in their degree of integration. BAU's therefore range from fully-fledged BCU's - a completely finished unit, where an application module can be snapped onto - over BIM's - that bear less hardware and allow a closer access to the processor - up to chip sets as well, providing only the basic access to the bus.

This modularity in Bus Access Units is available for all the current communication media.

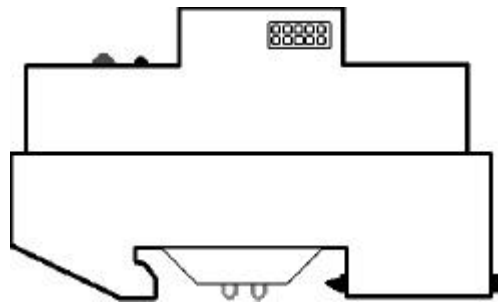
These BAU's are available in different housings, suited for the device they are intended for.



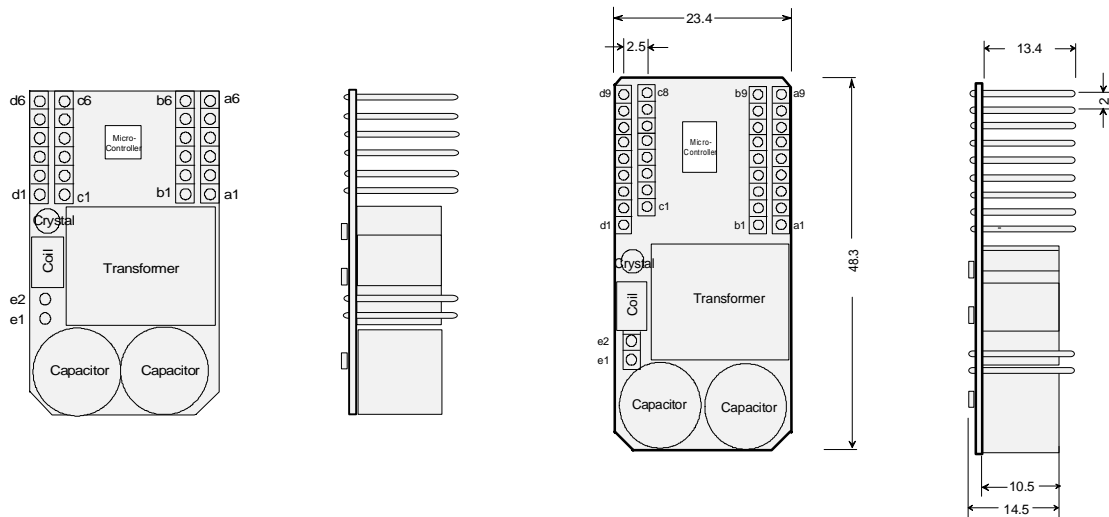
**Fig. 2/1/1-5: Flush mounted BCU, PL-BA**



**Fig. 2/1/1-6: Surface mounted BCU**



**Fig. 2/1/1-7: DIN-Rail BCU**



**Fig. 2/1/1-8: BIM M111**

**Fig. 2/1/1-9: BIM M112**

The following table compares the main differences between the current BAU's.

|                                      | <b>mask<br/>001x</b>      | <b>mask<br/>002x</b> | <b>mask<br/>070x</b> | <b>mask<br/>101x</b> |
|--------------------------------------|---------------------------|----------------------|----------------------|----------------------|
|                                      | <b>BCU 1<br/>BIM M111</b> | <b>BCU 2</b>         | <b>BIM M112</b>      | <b>PL-BA</b>         |
| micro-controller                     | 68HC05B6                  | 68HC05BE12           | 68HC11E9             | 68HC05B16            |
| Available RAM [bytes]                | 18                        | 40 - 90              | 7k or 15k            | 18                   |
| Available EEPROM [bytes]             | 230                       | 850                  | 8k or 31k            | 230                  |
| Max. Nr. of Group<br>Addresses       | 64                        | 64                   | 254                  | 64                   |
| Max. Nr. of Communication<br>Objects | 12                        | 28 - 48              | 255                  | 12                   |
| Access Protection                    | no                        | yes                  | yes                  | no                   |

**Fig. 2/1/1-10: Overview of existing BAU's**

## 2.4 Existing EIB Implementations

The system software versions (masks) are closely connected to the versions of the BAU from paragraph 2.3.

| MEDIUM   | TP  |   | PL   |
|----------|---|---|--|
|          | mask type 00  | mask type 07                              | mask type 10                                 |
| BAU-type | <b>BCU 1</b><br><b>BIM M111</b><br>mask 1.0<br>1.1<br>1.2 | <b>BCU 2</b><br>mask: 0.0<br><br><br><br> | <b>BIM M112</b><br>mask: 1.0<br><br><br><br> |
|          |   |   | <b>PL-BA</b><br>mask 1.1                     |

**Fig. 2/1/1-11: EIB Mask Versions**

The differences between these mask versions result from the requirements of the implementation of the communication protocol, as imposed by the specific media, for the used processor, together with the corrections and enhancements towards previous versions.

So far, downwards compatibility within the same mask type has been guaranteed: this means that an application program written for BCU 1 mask 1.0 can easily be downloaded by ®ETS in a BCU 1 mask 1.2<sup>2</sup>. Also compatibilities between mask versions of different mask types are made: mask 1.1 of the PL mask-type is compatible with mask 1.2 of the BCU 1 series at the application program level.

---

1. <sup>2</sup> Provided a fix-up table is delivered along with the application program.

### 3 Communication on the EIB System

Every bus device is addressable in two ways: by *physical addressing* and by *group addressing*.

The *physical address* is a unique identifier for naming the device throughout an EIB installation. It corresponds to the location of the device in the topology of the installation: it is composed of the area, line and number of the device in the line. The physical address is assigned during installation. Once it is programmed, the physical address is primarily used for downloading EIB application programs into the bus device, to download **group addresses** to each bus device and for maintenance purposes.

Bus devices, fulfilling a common task, are summarised in a *group*. They communicate by means of *group telegrams*, addressed by group addresses. In contrast to physical addressing, the address of a group does not reflect the subdivision of the EIB in zones and lines. The members of a certain group may be located anywhere inside the bus system. The advantage of group addressing is that a group telegram, sent by a member of a certain group, can be received and processed simultaneously by all other members of this group. For that reason group addressing simplifies the communication between bus devices and reduces the intensity of traffic. For example, if you want to set (adjust) all clocks installed in a building to the actual time, it is not necessary to transmit the time to all clocks one after the other. A single group telegram may adjust all the clocks simultaneously.

In normal operation communication happens exclusively by the use of group addresses. Neither the physical address of a bus device nor its group addresses are changed during normal operation.

3.1 Data Exchange and Transmission

3.1.1 Group Addressing

A group address is built up of two parts: a 4-bit main group and a 11-bit sub group. You may define up to main groups 0 to 13<sup>3</sup>, each of them containing up to 2048 sub groups. Fig. 2/1/1-12 shows the structure of a group address.

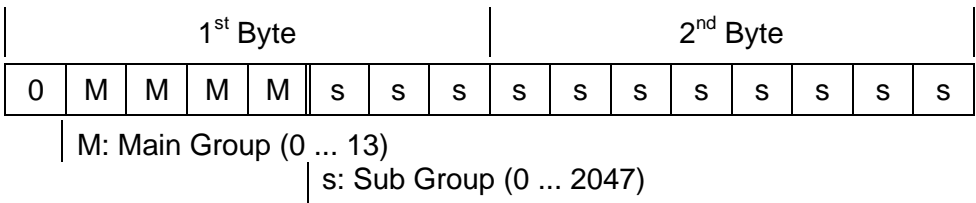



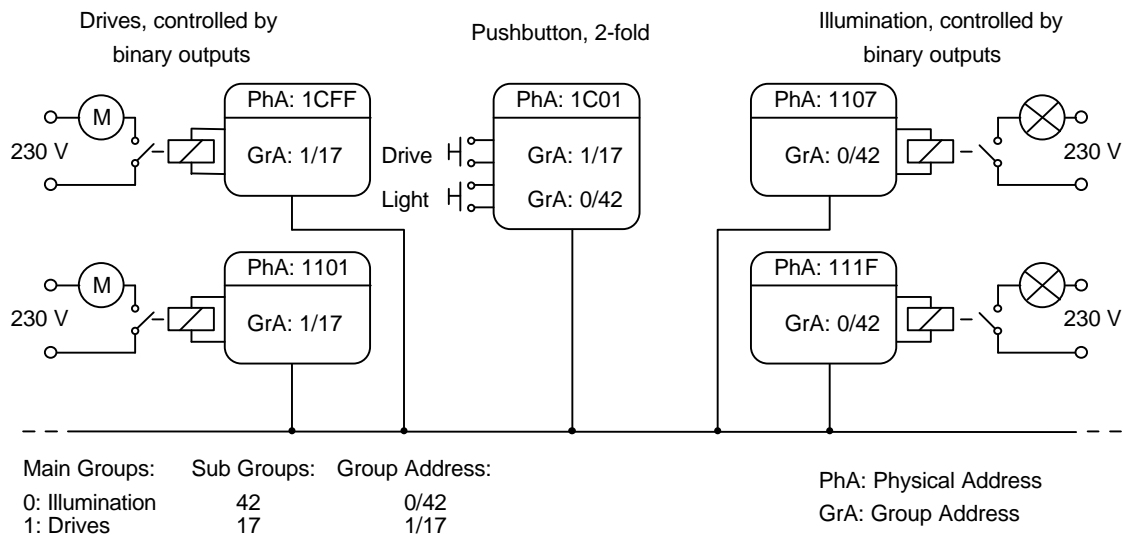
Fig. 2/1/1-12: Structure of a Group Address

 In the EIB Tool Software ®ETS2, the group addresses can also be presented to the user in a 3-level structure. (using the sub-group’s 3 most significant bits as "middle group").

The planner of an EIB installation may use group addresses as it suits him. It is recommended to attach groups, fulfilling similar functions, to a common main group.

Up to now we have assumed for simplification that each device performs just one single function. However, it is possible to realise more than one, maybe completely different functions with a single bus device. In this case the bus device has to be a member of several groups. Every bus device contains an address table, including the group addresses of all the groups to which this bus device belongs.

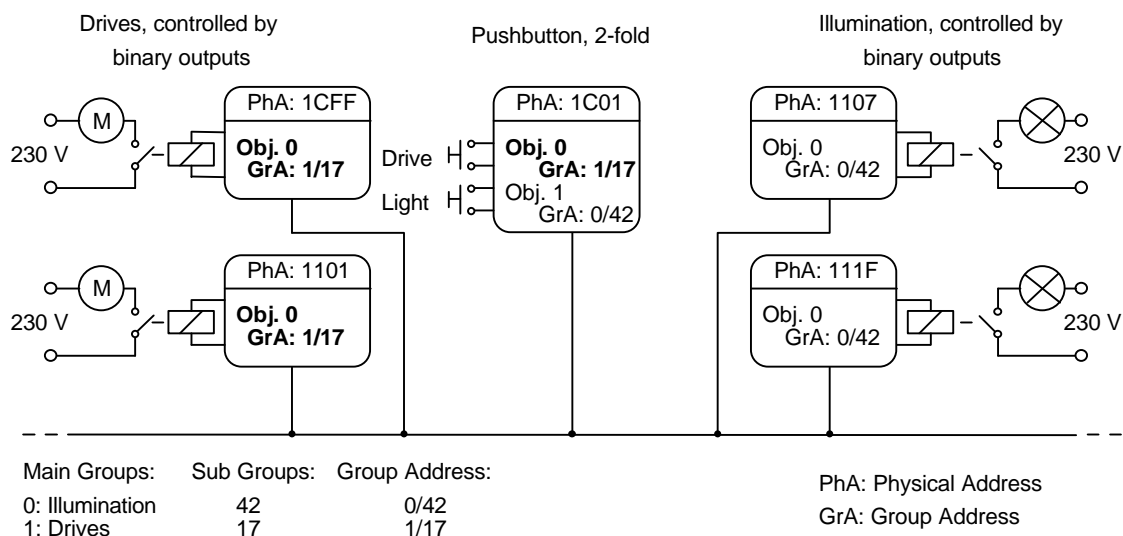
<sup>3</sup> The Main Groups 14 and 15 are not used.



**Fig. 2/1/1-13: Communication by Group Addresses**

As an example Fig. 2/1/1-13 represents two groups with group addresses 1/17 and 0/42. The bus device 1C01 represents a *sensor device* containing two push buttons, while the other four bus devices are *actuator devices* containing connectors. The bus device 1C01 is a member of both groups. By means of group address 1/17 one push button controls the drives connected to binary outputs 1CFF and 1101, while the second push button, by means of the group address 0/42, controls the lamps connected to binary outputs 1107 and 111F.

An application program communicates on EIB via communication objects. These are data-structures inside the BAU whose values can be sent or updated over EIB. This is done by assigning one or more group addresses to them. Communication objects, in different devices, that have at least one common group address are connected to each other.



**Fig. 2/1/1-14: Group Addresses and Communication Objects**



In the above figure, Obj. 0 of device 1C01, Obj. 0 of device 1CFF and Obj. 0 of device 1101 are members of the same group 1/17. This means if the device 1C01 sends a telegram on the GA 1/17, the device 1CFF and 1101 will update the value of their communication object 0. The drives will then start or stop.

## 3.1.1.1 Communication Objects

**Communication objects** are stored in RAM and EEPROM of the BAU. Each object used by an application is defined by its data structure. Among other things this data structure contains information about the state of the object. This information is called the **communication object value**.

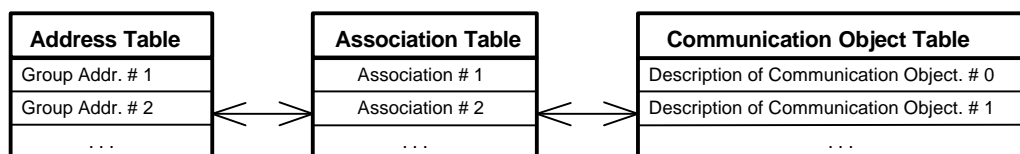
A possible communication object value could be:

- the state of a switch,
- a measured value (e.g. temperature, brightness),
- the actual time or the date, ...

We have to distinguish between **sending communication objects** and **receiving ones**. The value of a sending communication object is transmitted by means of a group telegram from the sending device to all devices belonging to the same group. In all devices having received this group telegram, the corresponding communication object value will be updated.

The characteristics of all communication objects used by a BAU are described in a **communication object table**. The number of required communication objects is determined by the needs of the attached application module. For example, a 2-fold push button requires 2 communication objects to store its state.

In order to send a communication object value by a group telegram, the system software has to know which group address belongs to the communication object. For that reason two other tables are used: the **address table** and the **association table**. The address table contains all addresses used by the BAU, whereas the association table links the group addresses with the communication objects.



**Fig. 2/1/1-15: Link between Group Addresses and Communication Objects**

In the following paragraphs we will describe structure and function of the tables.

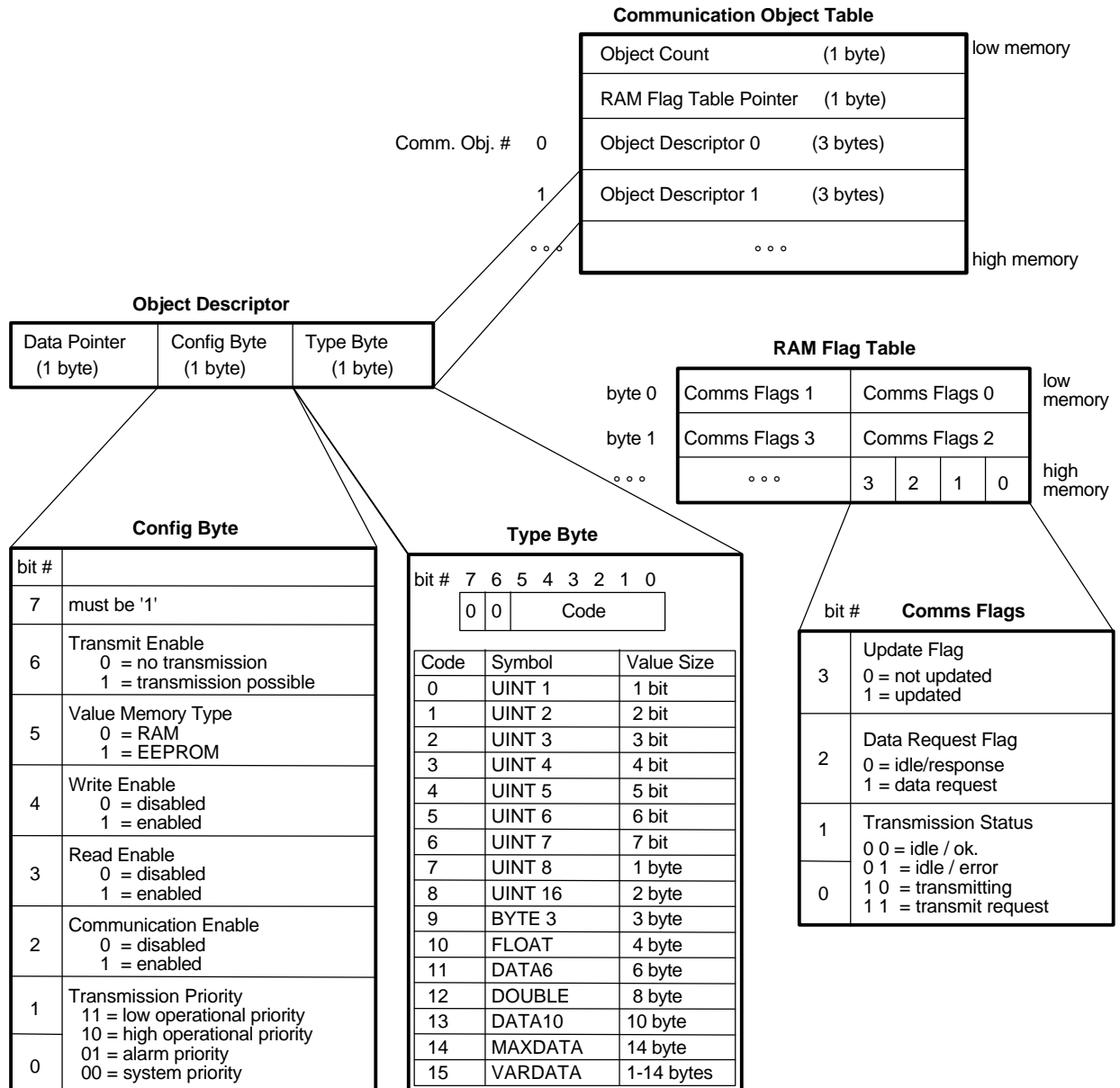
- ☞ A communication object can receive its data from more than one group address, but its value can only be sent through one single group address.  
More than one communication object in a device may be connected to the same group address.

☞ The following description of the functioning is made up for the BCU 1 BAU-types. In the BCU 1, the communication objects are part of the application layer and managed by both the system software and the application program. In later BAU-types however, the communication objects are part of the application program. So, for these, the application program writer has to fix his own communication object table, which must have the same structure as here described. So, the following description of the *functioning* is valid for both BAU-types.

### 3.1.1.2 The Communication Object Table

The communication object table contains, in its first byte, the number of included communication object descriptors. The second byte contains the pointer to the ***RAM flag table***. The succeeding bytes are the ***object descriptors*** themselves. The object descriptors are numbered consecutive, starting with 0 for the first object.

The communication object table is stored in the EEPROM of the BAU. Although the RAM flag table is stored at a different memory location it has to be regarded as a part of the communication object table. We will discuss the RAM flag table at first, because knowledge of its working method will facilitate our comprehension of the object descriptor.

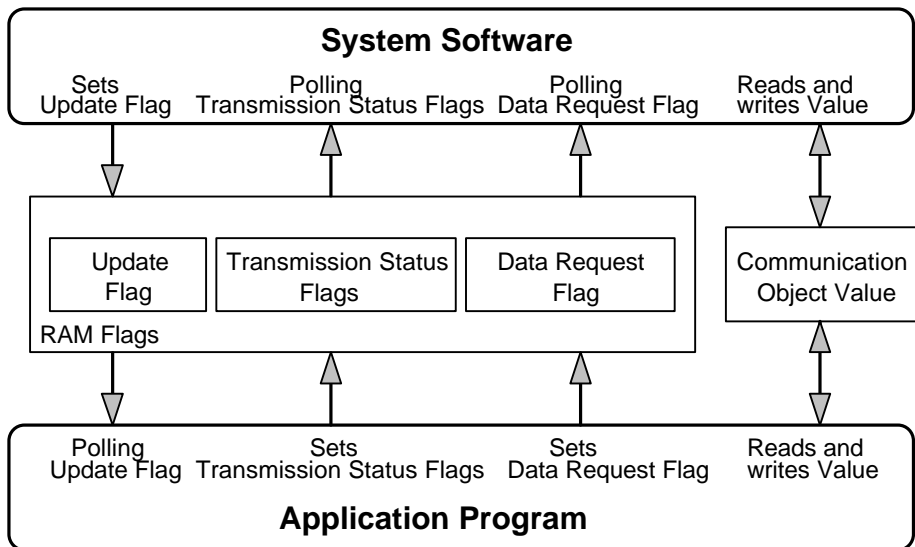


**Fig. 2/1/1-16: Communication Object Table and the RAM Flag Table**

## The RAM Flag Table

A group of four **RAM flags** belongs to each communication object. For that reason you need half a byte of RAM space for each communication object. The start of the RAM flag table is fixed by the RAM flag table pointer, stored in the communication object table. The end of the RAM flag table however, results from the number of used communication objects, i.e. the value of the object count byte.

The RAM flags, along with the communication object value, establish a software interface between the system software and the application program. You may set and check the flags by usage of certain system software services, or by direct memory access. In this context 'direct memory access' means that you may set, clear and check the flags manually (e.g. by the instructions 'bset flagnumber, flagaddress', 'bclr flagnumber, flagaddress', 'sta flagaddress' or 'lda flagaddress' ).



**Fig. 2/1/1-17: Interface between System and Application Program**

- The **update flag** is set by the system software in order to inform the application program that the corresponding communication object value has been updated due to reception of a group telegram.
- The **data request flag** is set together with the transmit status flags by the application program, in order to request the current state of a certain group. The transmit status flags will be discussed in the next paragraph. As a result, the system software sends an 'empty' group telegram to the addressed group, that means a group telegram without a communication object value. As an answer the group members will send group telegrams containing the requested communication object value.
- In both cases, the **transmit status flags** are set to the transmit request state by the application program in order to inform the system software that a communication object value has to be transmitted. This is allowed only if the previous transmission has been completed, i.e. the transmit status flags must not be in transmitting status.

## The Object Descriptor

The **object descriptor** describes the characteristics of its communication object. It is built up by three parts: The data pointer, the config byte and the type byte.

### *The Data Pointer*

The **data pointer** indicates the storage position of the communication object value. The value itself may be stored in RAM or in EEPROM. If the value is stored in RAM, the data pointer represents its address directly. But if the communication object value is stored in EEPROM, its address is formed by the sum of data pointer and an offset of 100h. All communication object values that are stored in EEPROM have to be located outside the EEPROM checkrange!

### *The Config Byte*

The **config byte**, by seven of its eight bits, configures the communication behaviour of the communication object in descending value:

The MSB is to be set to '1'. This bit is used merely in BCU 1 mask version 1.1 as a **transmit enable** bit. In previous system software versions it is to be set to '1'. If this bit is set in system software version 1.1, the communication object becomes a sending object. In this case, it gets the ability to send its value. If the bit is cleared, no transmission of the communication object value will be possible.

- **Value memory type:** If this bit is set, the communication object value is stored in EEPROM, otherwise in RAM.
- **Write enable:** If this bit is set, the communication object becomes a receiving object. In this case, a received group telegram may update the concerning communication object value. If this bit is cleared, an update will be impossible.
- **Read enable:** This bit controls the response behaviour of the communication object. If the bit is set, the communication object may send its value due to a data request. If the bit is cleared, a data request will not be taken care of.
- **Communication enable:** This bit controls the general communication ability of the communication object. If set, the communication capability depends on the state of the transmit enable bit, the write enable bit and the read enable bit. On the other hand, if this bit is cleared, communication with the concerning communication object is completely inhibited.
- **Transmission priority:** If several bus devices are sending a telegram at the same time, the telegrams will be sent in the order of the transmission priority of the included communication object values. A high priority is represented by a low number in these two bits.

## The Type Byte

The **type byte** determines the size of the communication object value. The value may range between 1 bit and 14 bytes.



By setting the transmit enable bit together with the write enable bit the communication object becomes a so-called bi-directional communication object. This kind of communication object may send and receive. However, it is recommended not to use bi-directional communication objects, because of the following grounds:

1. For an actuator, such a communication object would both serve for *writing* its output state (set value) via the bus as for *reading* its current state (actual value). If this is written and immediately read out, the answer will be the set value, and not the actual value at the time of reading (e.g. if the device has a switch on/off delay).
2. The EIB communication protocol does not differ between reception of a ValueWrite and a ValueResponse telegram. If a device receives the status of another device via a certain group address, its communication objects assigned to that group address will update their value. If one such communication object serves for receiving the set value, the device will react unexpectedly, as a side-effect of reading out the status of another device.

### 3.1.1.3 The Address Table

The first byte in the address table contains the number of all addresses included, that means, the sum of all group addresses and the single physical address. For example, if the 'length' is '4', it means that the address table contains the physical address and three group addresses.

The next entries in the address table are the physical address, followed by all group addresses involved. All addresses are marked in ascending order by their **connection number**, starting with connection 1 for the first group address. The connection numbers are required by the association table.

| Address Table |     |                            |
|---------------|-----|----------------------------|
| Connection #  | 0   | Length (1 byte)            |
|               | 1   | Physical Address (2 bytes) |
|               | 2   | Group Address (2 bytes)    |
|               | 3   | Group Address (2 bytes)    |
|               | ... | ...                        |
|               |     | Low Memory                 |
|               |     | High Memory                |

**Fig. 2/1/1-18: Structure of the Address Table**

When a group telegram is received the system software compares the received group address with all the group addresses in the address table in ascending order, beginning with connection 1. The search is stopped either if the received group address is found, or if the search algorithm reaches an address in the table that is higher than the received one. For that reason all group addresses have to be placed in ascending order. Otherwise a low address entered behind a higher one will not be found and the corresponding group telegram will be falsely ignored.

3.1.1.4 The Association Table

The association table contains in its first byte the number of included *associations*, followed by the associations themselves. Each association consists of two parts:

- The connection number coming from the address table and
- the communication object number coming from the object table.

In this way associations are established between group addresses and communication objects. Fig. 2/1/1-19 shows the structure of the association table.

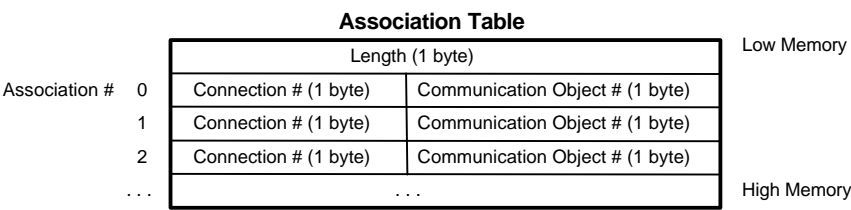


Fig. 2/1/1-19: Structure of the Association Table

The associations are numbered continuously, starting with 0 for the first one. Fig. 2/1/1-20 shows an example for a more complex linkage between group addresses and communication objects.

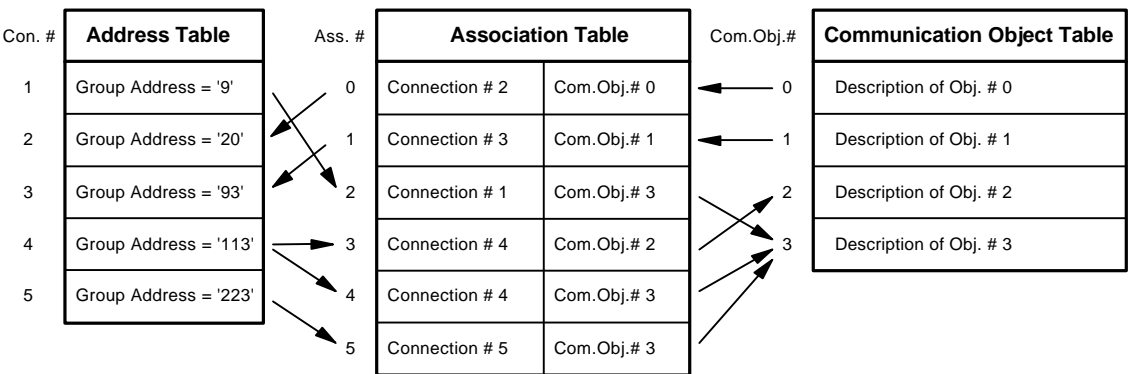


Fig. 2/1/1-20: Linkage among Group Addresses and Communication Objects

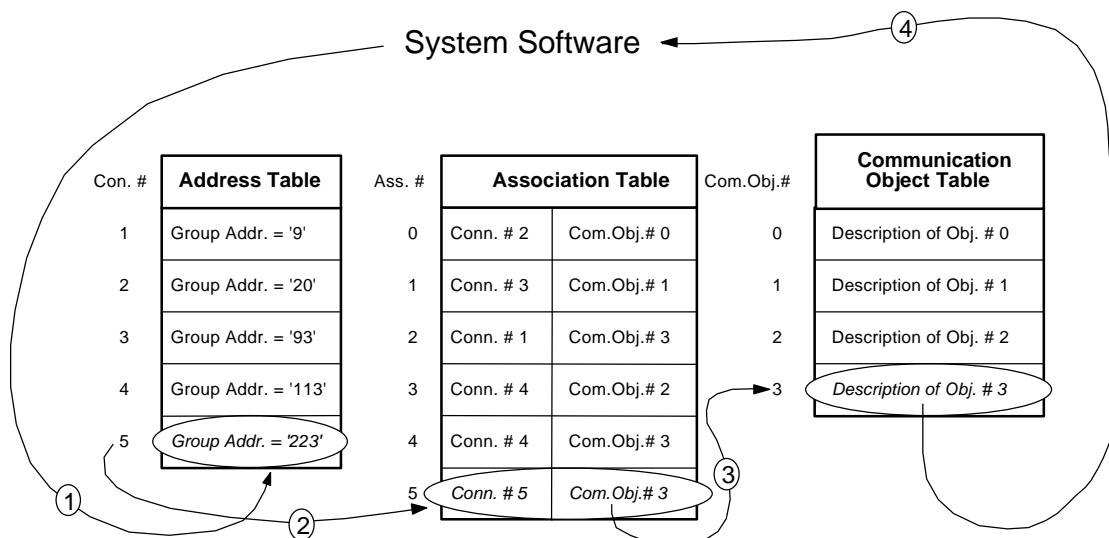
The arrows between the tables indicate whether the communication objects are sending - or receiving objects. The following kinds of linkage between group addresses and communication objects are allowed:

- A sending communication object using a single group address, e.g. communication objects 0 or 1,
- a receiving communication object, influenced by a single group address, e.g. communication object 2,
- a group address, influencing several communication objects, e.g. group address '113', and
- a receiving communication object, influenced by several group addresses, e.g. Object 3.

However, a sending communication object can belong only to one single group address.

The usage of the association table depends on whether a communication object value is received or sent. We will shortly discuss both cases. For the following text see Fig. 2/1/1-21.

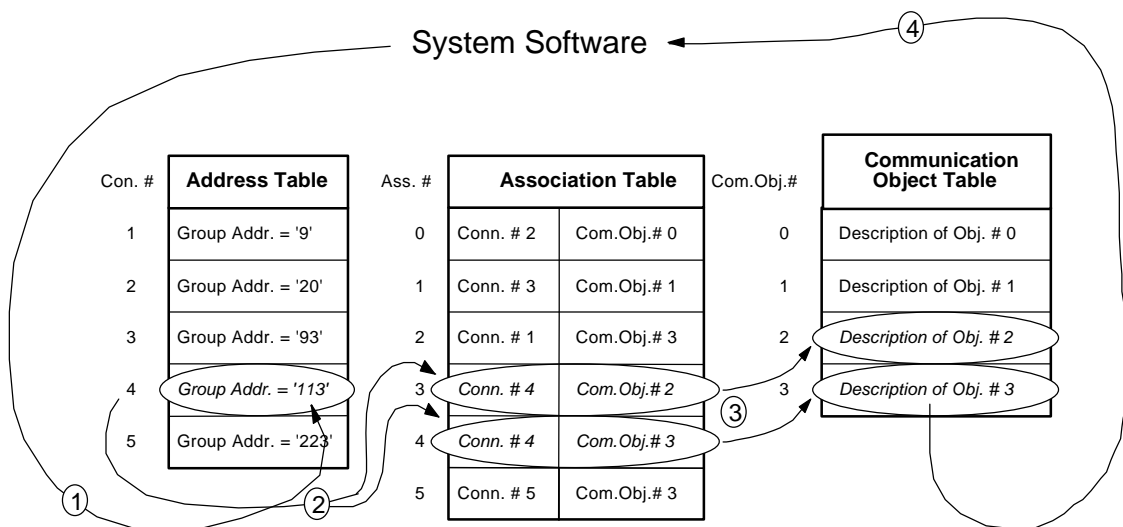
- ① When a bus device receives a group telegram, the system software looks in the address table for the received group address.
- ② If the group address is found, in a second step the system software looks for the concerning connection in the association table.
- ③ The communication object, associated with this connection determines, which communication object value has to be updated by the received group telegram.
- ④ The corresponding update flag is set and the communication object value is overwritten with the received value, the corresponding memory location of the communication object value coming from the object table.



**Fig. 2/1/1-21: Function of the Tables for Telegram Reception**



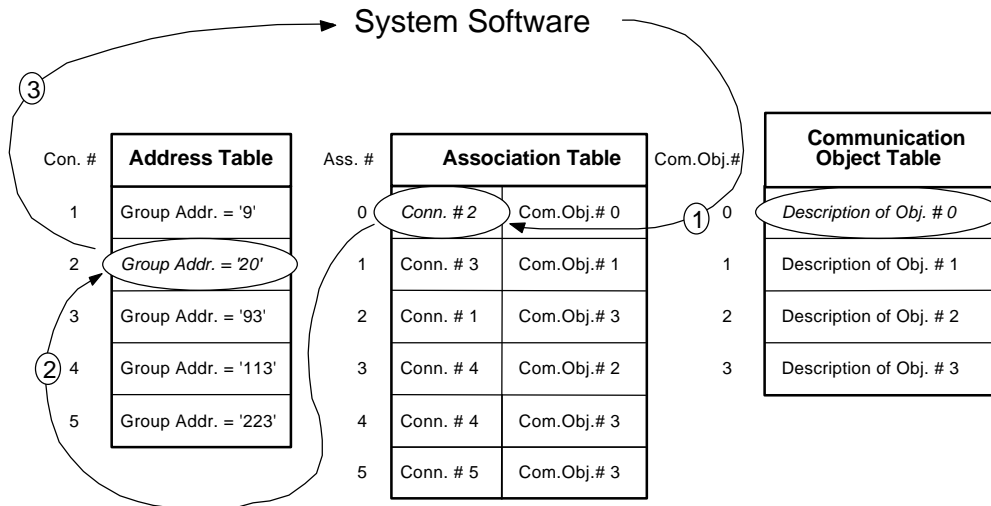
- ① If the connection is found in the association table several times (see Fig. 2/1/1-22), all associated communication objects are going to be updated. This case is shown in Fig. 2/1/1-22 by the example of the received group address '113'.
- ② In this case the system software finds the corresponding connection in two different lines of the association table.
- ③ Consequently the system software takes/gets two different communication object numbers from the association table. In this example the object numbers are 2 and 3.
- ④ The corresponding memory locations of both communication object values come again from the object table.



**Fig. 2/1/1-22: Telegram Reception, two Objects being updated**

The reverse procedure of group telegram transmission is much easier. For the following text see Fig. 2/1/1-23.

- ① When a communication object value has to be transmitted, the system software uses the association of the very same number as the sending communication object has. That means e.g. that the communication object 0 uses always the association 0, and so on.
- ② The connection of the used association determines the required group address. You may notice that in this case it is not necessary to search for the required data neither in the association table, nor in the address table.



**Fig. 2/1/1-23: Function of the Tables for Telegram Transmission**

### 3.1.2 Transmission and Reception of a Communication Object Value

The co-operation of communication objects and group addresses now having been shown, this paragraph will discuss in detail the processes taking place in the BAU during reception or transmission of a communication object value.

#### 3.1.2.1 Transmission of a Communication Object Value

Fig. 2/1/1-24 shows the example of a BAU configured as a sensor, which is transmitting a communication object value. The application module is realised by four switches, each belonging to an own communication object. The encircled numbers are indicating the order of events.

#### Activities of the Application Program (AP):

- ① The AP supervises the PEI's state. When the state does not change because no switch has operated, the AP proceeds without processing any relevant communication object.
- ② Otherwise, if one of the switches has been operated, the concerning communication object value - in this example value 1- will be updated by the AP. The value represents the state of the switch ('0000 0000' = 'off', '0000 0001' = 'on').
- ③ In order to demand a transmission of the updated communication object value, the corresponding transmit request flags are set by the AP.

At the end of the AP given by the RTS instruction the BAU continues with the system software.

### Activities of the System Software:

However, if a transmit request flag has been set the system software has to generate a group telegram.

- ④ Firstly the system software checks all RAM-flags. If no flag has been set, no group telegram will be sent.
- ⑤ In order to be able to build up a complete group telegram, the system software needs the communication object description, the current communication object value and the corresponding group address. The required information is fetched out of the tables by means of the system software. See Fig. 2/1/1-24.
- ⑥ After that the system software transmits the group telegram.
- ⑦ To be able to recognise the next transmission request, the system software clears the transmit request flags.

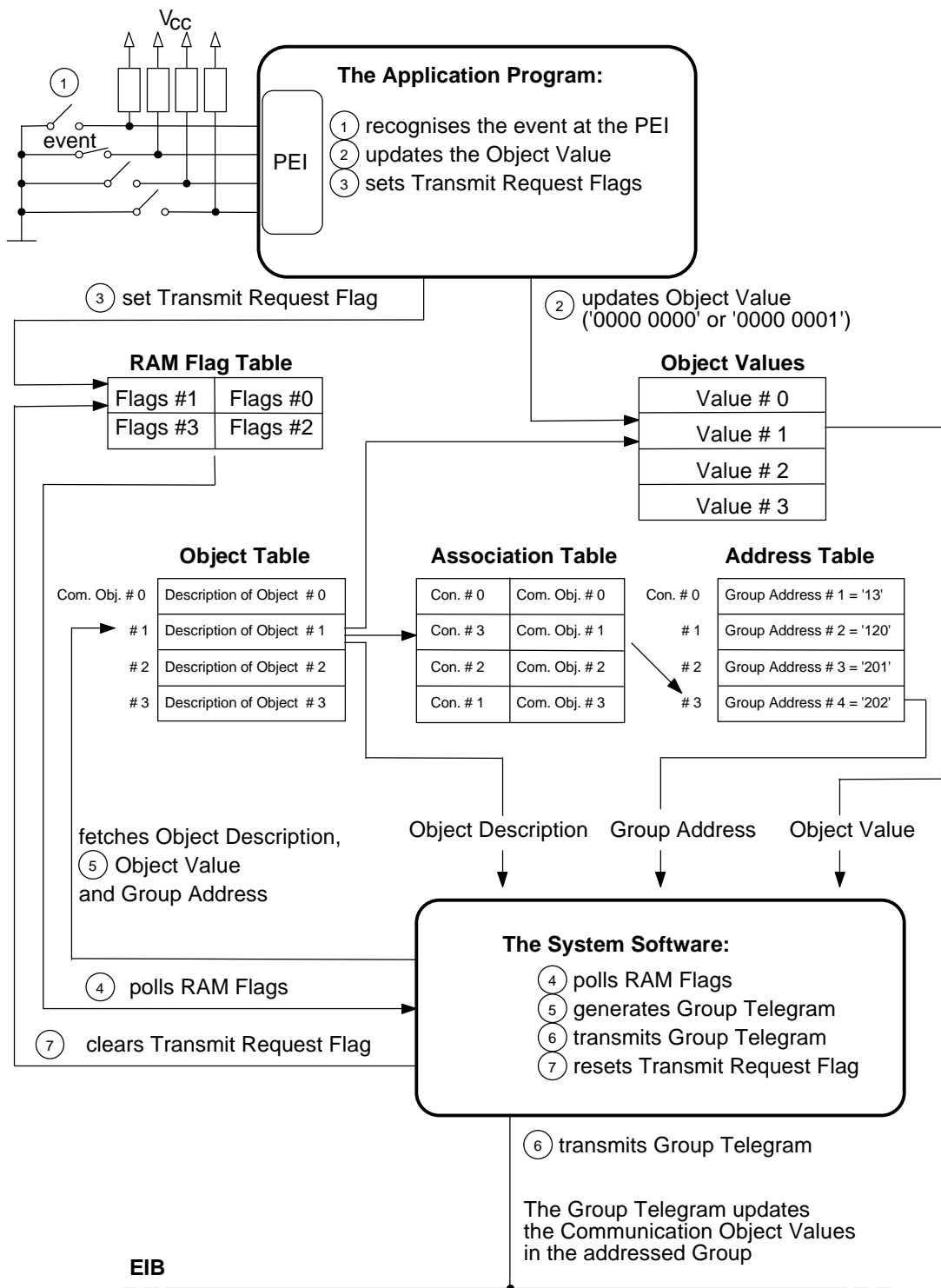


Fig. 2/1/1-24: Transmission of a Communication Object Value

### 3.1.2.2 Reception of a Communication Object Value

Fig. 2/1/1-25 shows the example of a BAU configured as an actuator, receiving a communication object value, transported by a group telegram. In this case the application module consists of two contactors. First of all the received group telegram is processed by the system software.

#### **Activities of the System Software:**

- ① The system software extracts the group address and the communication object value from the group telegram.
- ② After that the system software determines by means of the group address and the communication object descriptions which communication objects have to be updated with the received communication object value.
- ③ By this information, the system software is able to update the communication objects and to set the proper update flags. Since, in this example, the group address 3 is connected to two communication objects, both of them will be updated and both update flags will be set.

The application program checks these flags and performs the necessary task to correctly process the received values, depending its intended functioning.

#### **Activities of the Application Program (AP):**

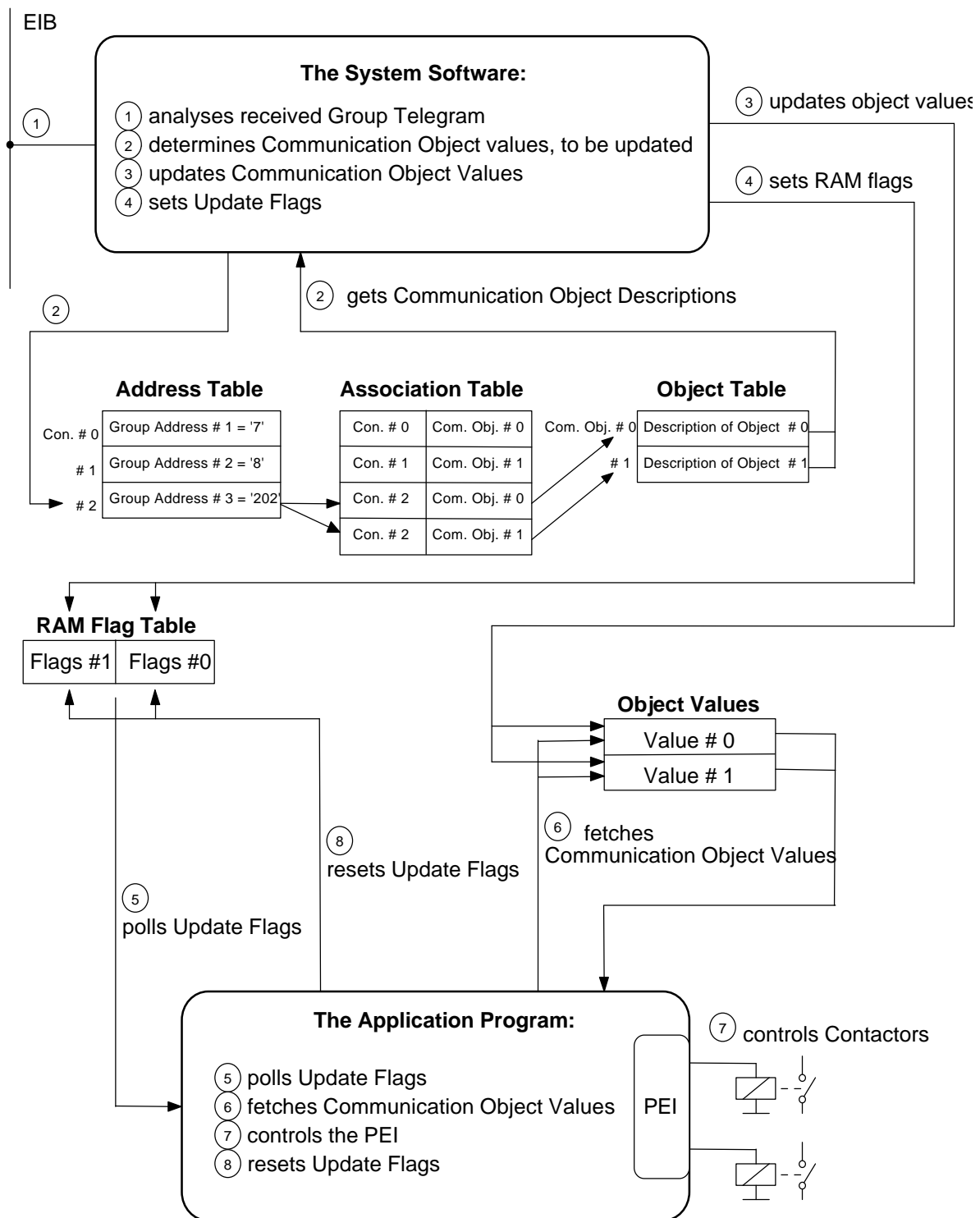
- ④ To notice changes of communication object values, the AP checks the update flags. If no flag has been set, the AP goes on with the rest of the program.
- ⑤ In Fig. 2/1/1-25 both update flags have been set.
- ⑥ Therefore the AP fetches the corresponding communication object values ...
- ⑦ ... and controls the contactors connected to the PEI, corresponding to these values.

To be able to recognise the next update, the update flags have to be cleared by the AP.

In either case, after the end of the AP, the system software continues operation.



In the last two paragraphs we have seen merely the basic tasks of the AP. Of course, in case of more complex applications the AP may realise much more extensive functions.



**Fig. 2/1/1-25: Reception of a Group Telegram**

### **3.1.3 Fast Polling Mode**

In the fast polling addressing mode, one product is defined as Polling Master and up to 16 products are Polling Slaves for a given Polling Group. The polling master starts a polling telegram by transmission of the control field, source- and destination address and a check-byte over these first 5 bytes. Then the subsequent polling slaves complete the telegram by adding one byte, each in a time-slot indicated by its polling number.

This technique allows the Polling Master to quickly obtain status information from a number of products using one single telegram, instead of having to address them one by one. This may be required for security applications.

### **3.1.4 Broadcast Addressing**

Broadcast addressing is an exception to normal group addressing. The destination address is 0x0000. Every device in the installation will accept telegrams sent via broadcasting. Broadcast addressing does not require an entry in the address table.

This kind of communication is thought specially for programming and reading of the physical address of a device.

This way, only the devices that are in programming mode, for example as their programming button is pressed, will process the telegram that writes the physical address.

### **3.1.5 Connectionless / Connection oriented addressing**

Normal group addressing is a connectionless communication way. There is no private connection between the sender and the receiver(s) of a group telegram. Using group addressing, the sender can reach multiple receivers without having to contact them one at a time. However, this does not guarantee that all devices with this group address will receive the information in case of various problems on the line or in the devices themselves.

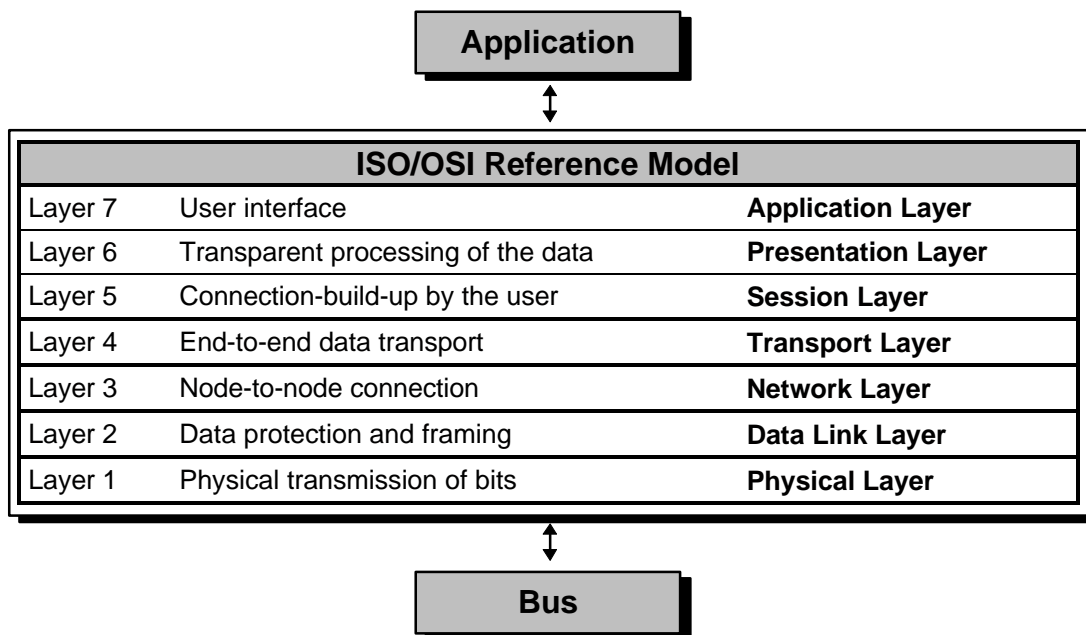
Connection-oriented communication allows building up a private connection between two devices. The transported telegrams are numbered, and by using numbered acknowledge, it can be ensured that all data is correctly received at destination.

## 3.2 Communication Model and Telegram Structure

The communication model used within the EIB is based on the ISO/OSI-Reference Model (*International Standards Organisation / Open Systems Interconnection*). Using this communication model you may extend your product's communication power to the application you have to realise.

👉 Knowledge of this OSI model and its implementation is not always required for basic application programming. The information of chapter 1 may be sufficient. So, you may also skip this paragraph and proceed with paragraph 4 (Interworking Standards)

The EIB communication system can be split into different organisational layers. Because of these separated organisational layers there are many advantages as far as interconnection and communication with other systems are concerned. So you may omit the upper layers (e.g. the application layer) of the EIB communication system and substitute them with your own (PC-based) application. On the other hand you may replace the lower layer(s) to alter the transmission media (e.g. radio instead of 2-wire-Bus).



**Fig. 2/1/1-26: ISO/OSI-Reference Model**

If you have an own application controller (AC) behind your Bus Access Unit (BAU) the application controller can handle the upper layers. You can easily switch on and off the layers you want to realise outside the Bus Access Unit.



The separation of the layers is performed within software modules. Only the physical layer is realised by hardware. Some layers are medium dependent<sup>4</sup>.

The layers are communicating with the neighboured layers through a messaging system . Because this communication takes place in particular within the operating system, the program doesn't have any direct access to that communication.

A message on the bus is forwarded through all layers beginning at the physical layer and ending at the application layer. The other way round a message of the application program is passed through all layers down to the physical layer.

Not all of the seven OSI-layers are completely implemented on the EIB. Particularly the session layer and the presentation layer are not put into action here. In the following paragraphs the remaining layers are explained. By means of the telegram structure, you can see how the data to be sent are modified or rather data are added by each layer.

### 3.2.1 Physical Layer

The physical layer controls the physical mechanisms of data transmission. It is responsible for the physical transmission of bits. Therefore these tasks are taken over by hardware.

On Twisted Pair, EIB uses the CSMA/CA access mechanism (Carrier Sense Multiple Access with Collision Avoidance). CSMA prevents a BAU to access the medium if it has detected that another one has already started emitting. Collision Avoidance is ensured due to the fact that there is one symbol that is overwritten by the other one. In case there is a strict simultaneous access of two devices, during the emission of the message of one device, one symbol which will be overwritten by a symbol of the message of the other device. This will be recognised by the first device, that will immediately stop sending, letting the other device transmit the rest of its message. On EIB the 0-bit overwrites the 1-bit.

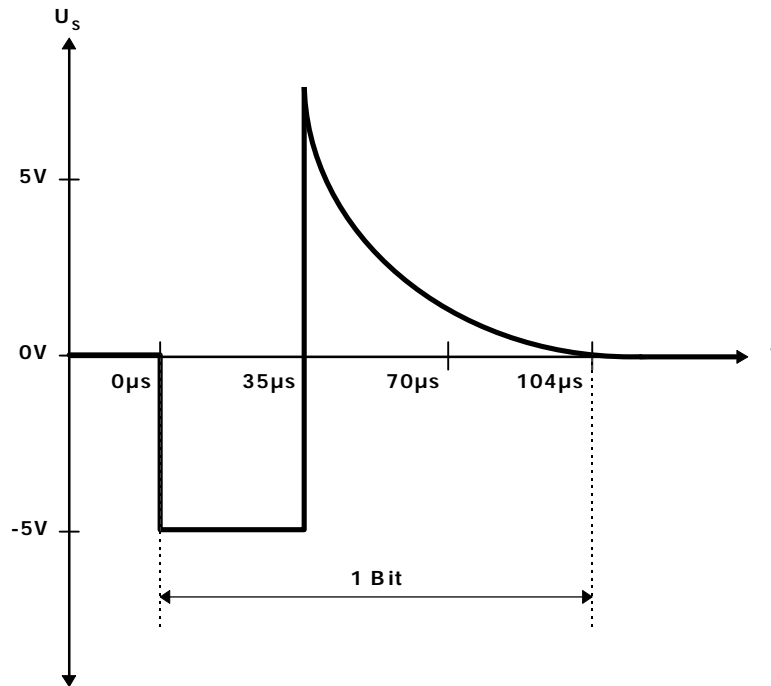
This is why telegrams with high operational priority have priority bits set to '0'. This way, they overwrite the lower level telegrams. Concerning the operational priority remember the Config Byte on page 21.

During transmission of a 1-bit the signal level remains on 0 volts. The voltage on the bus will only change, if a 0-bit is transmitted. The node releases a 35 µs pulse on the 2-wire bus. The inductances at the bus (transformers in the bus coupling units and chokes in the power supply units) are generating a compensating pulse. Therefore the duration of the transferred bit increases to 104 µs. The bit is transferred symmetrically without any steady component. The dc supply voltage on the bus is superposed with these data signals.

In the following figure you can see the behaviour of the voltage on the positive wire while sending a 0-bit.

---

<sup>4</sup> So far, only the description for the Twisted Pair medium will be given. Layer descriptions for other media will follow in future releases.



**Fig. 2/1/1-27: Signal Behaviour of a 0-bit.**  
(The DC Supply Voltage is superposed.)

### 3.2.2 Data Link Layer

The binary coded data are framed to blocks, start- and stop bits are added. By attaching a checksum data transmission will be secured. The data link layer also handles acknowledging of received data packets.

The characters are transmitted character by character. One character consists of a single start bit (0-bit), eight data bits, one parity bit and one stop bit. Between two characters there is a break of 2 bit times. The data begins with the least significant bit. Please keep that in mind while looking at the following pictures of the telegram structure. These illustrations are showing the data beginning with the most significant bit. This fact is especially important to understand the telegram priorities.

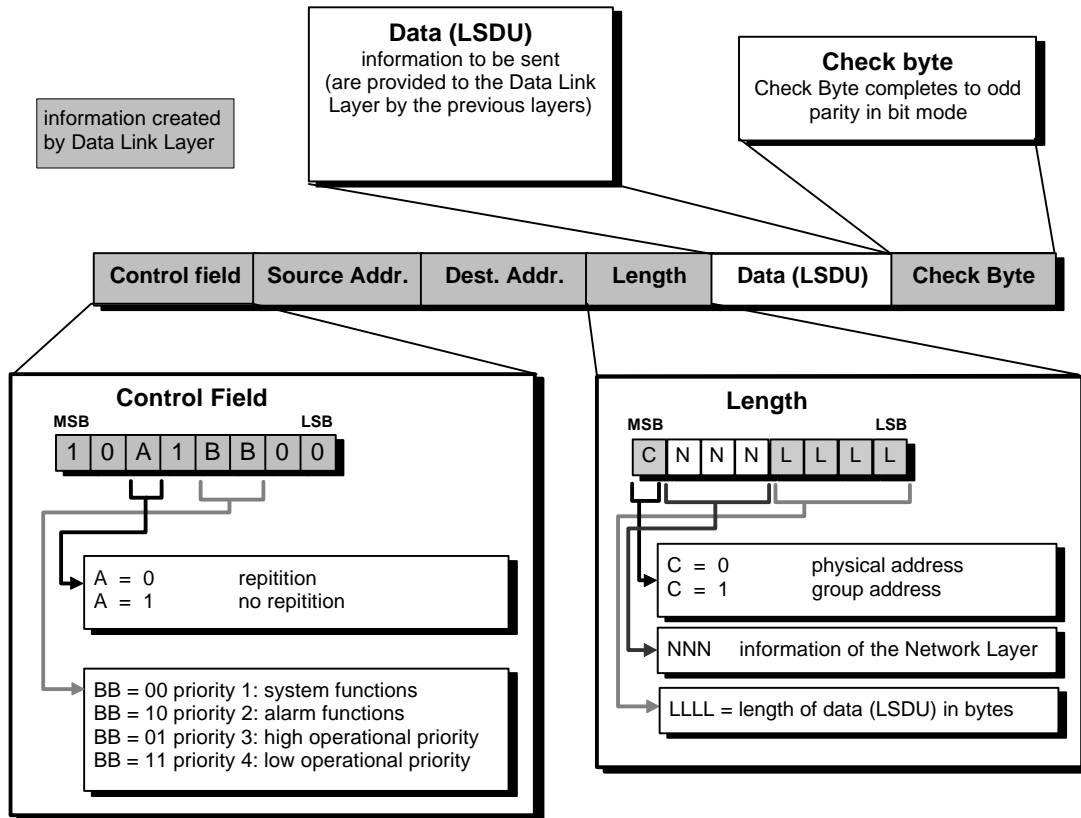
A data packet consists of several characters. The last character is a check byte completing the transmitted characters in bit mode to odd parity.





### 3.2.2.2 Message Telegram

A message telegram contains the data to be transferred over the bus. The data link layer gets the data to be sent (LSDU) from one of the previous layers. Usually they are provided by the Network Layer. The data link layer then completes these data with a control field, a source address, a destination address, a length byte and a concluding check byte. See Fig. 2/1/1-30.



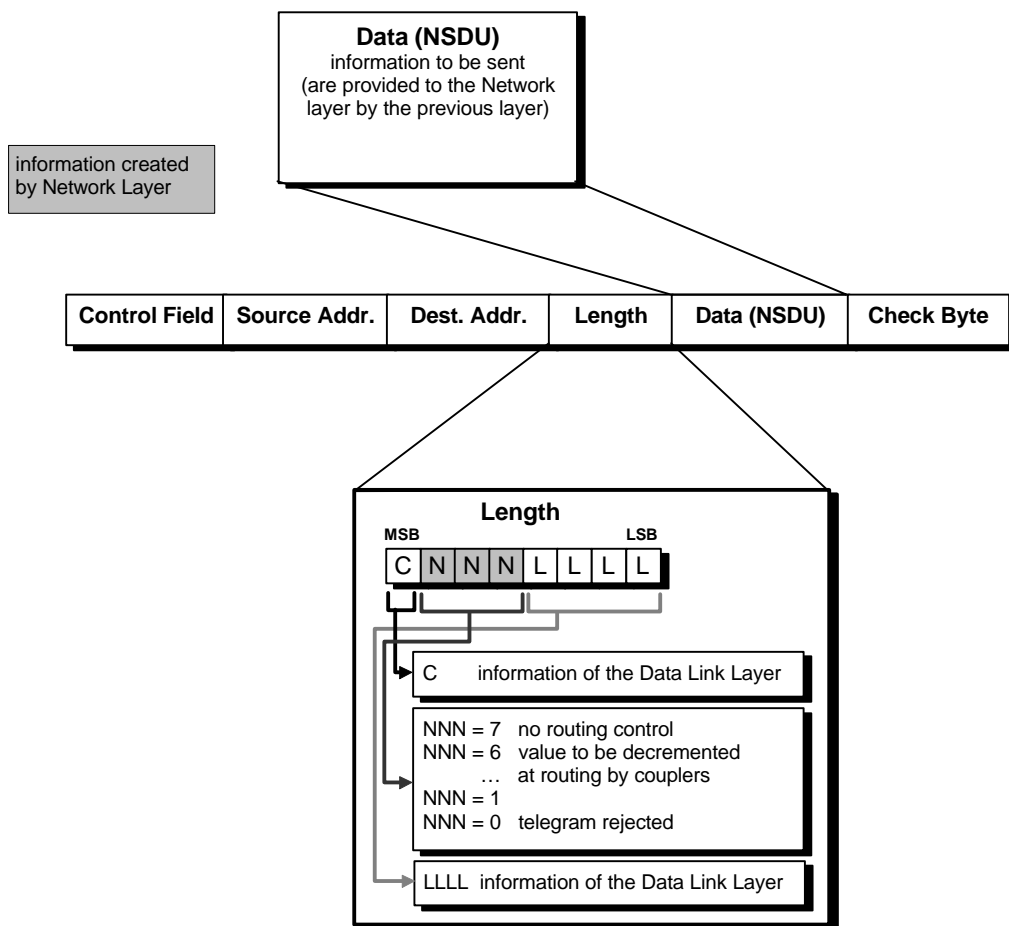
**Fig. 2/1/1-30: Telegram Structure LPDU (Data Link Layer)**

- **Control Field:**  
This field contains two bits indicating the data telegram priority for bus access. Another flag shows whether the telegram is a repetition telegram (because of INAK or BUSY). At the beginning of the control field there are two preamble bits.
- **Source Address:**  
The source address is the physical address of the sending bus device.
- **Destination Address:**  
The destination address contains the physical address or the group address of the bus device(s) the telegram is addressed to.
- **Length:**  
This byte contains the length of the following data (LSDU) and the Address Type Bit.
- **Address Type Bit:**  
This bit indicates the destination address type (physical address PA or group address GA).

- **Data (LSDU):**  
The LSDU contains the data to be transferred over the bus. These data are provided by the previous layer (network layer).
- **Check Byte:**  
The check byte completes all bytes to odd parity in bit mode.

### 3.2.3 Network Layer

Telegrams sent from one line to another one are transmitted via line- or backbone couplers. All these coupling devices have a filter table. This table contains information how to route a telegram through the whole EIB. Closed loops including line-, backbone couplers or repeaters are not allowed in principle, because they can lead to continuously circulating telegrams.



**Fig. 2/1/1-31: Telegram Structure (Network Layer)**

To avoid these circulating telegrams (because of inadvertently created loops) the network layer adds three bits to the data provided by the previous layer (NSDU). These three bits are called the **Routing Counter**. Creating a new telegram the routing counter is initially set to a start value. This start value can be initialised to a value in the range of 1 to 7. With every transfer via a line- or a backbone coupler the routing counter within the telegram decreases by one. The routing counter having reached the value zero, the telegram will be rejected by further line- and backbone couplers.

Consequently the number of couplers and repeaters in the transmission path may not exceed six. A telegram transmitted via more than 6 couplers and repeaters is rejected irretrievably. Defining the topology you should make sure that no telegram has to pass more than 6 couplers and repeaters.

The routing counter will not be decremented in the case of initialisation with the value 7. So you can switch off the routing control simply by initialising the routing counter value to 7.

### 3.2.4 Transport Layer

The Transport Layer is responsible for the end-to-end data transmission. The layer converts the internal connection number into an address on the bus and vice versa.

There are two different kinds of communication between bus devices:

#### **Connectionless communication:**

By a connectionless communication a telegram will be sent to one or more bus devices using the common group address. You don't have to build-up or breakdown a connection with the desired bus devices. Because of this fact it is a very effective kind of communication. If a bus device wants to send a telegram to several bus devices, the bus will be loaded with only one single group telegram followed by a number of acknowledgements simultaneously sent afterwards. Therefore time consumption for this point-to-many-point communication is very small.

Using the connectionless communication you cannot guarantee that every bus device addressed by the telegram has really received the telegram. If the sender receives at least one positive acknowledgement (IACK) and there is no negative acknowledgement (INAK), the telegram will seem to be sent successfully. A bus device cut off from the EIB-installation of course will not receive the telegram and will not acknowledge. However, a positive acknowledgement of another addressed bus device is enough to satisfy the sender. The sender itself cannot detect whether all addressed bus devices have received the message.

#### **Connection-oriented communication:**

The connection-oriented communication facilitates point-to-point communication between two bus devices. This communication needs a build-up and breakdown of the connection. Telegrams sent in this communication mode are numbered. So the receiver gets information about the order of telegrams sent and it is able to detect the loss of a telegram. The numbering is particularly important for transferring greater data blocks divided into many telegrams.

The reception of a numbered data telegram is confirmed by sending a numbered acknowledge telegram. This acknowledge telegram is a complete telegram and opposite to the acknowledge character mentioned before, it contains all information added by the data link layer (control field, addresses and check byte). Thus the assignment of data telegram and acknowledge telegram is secured.

If the reply of one of the devices is missing, the connection will be broken down after a certain time (time-out).

As the connection-oriented communication permits to transfer greater data amounts in a very secure way, it is used by system- or service functions. E.g., you might perform a complete memory dump for diagnostic purposes.

The transport layer provides four different protocol data units (TPDU Transport Layer Protocol Data Unit). See Fig. 2/1/1-32. These TPDU's are handed over as NSDU's. The network layer then adds its data (routing counter). For the rest the NSDU (provided by the transport layer) remains unchanged by the network layer because the routing counter is inserted into the length byte provided by the data link layer. Because of this fact the NSDU is identical to the LSDU.

The LSDU is completed by the data link layer with a control field, the addresses and a check byte. The generated LPDU is sent out to the bus by the physical layer. Please note that the NSDU and the LSDU are identical and correspond to the data provided by the transport layer (TPDU).

The different protocol data units of the transport layer (TPDU) and their purpose are described in the following section:

- Data Packet (UDT):

The Data Packet is exclusively used within the connectionless communication (group addressing). This TPDU consists of two leading 0-bits, 4 following don't-care bits and the data provided by the application.

- Numbered Data Packet (NDT):

With the help of Numbered Data Packets the user data are transferred within a connection-oriented communication (point-to-point connection). Beside the two indicating bits and the data provided by the application, the TPDU contains a sequence number of the data packet.

- Control Packet:

The Control Packet is used at build-up (OPEN) and breakdown (CLOSE) of a point-to-point connection. Because there is no need of user data the TSDU is omitted. In this stage of connection, numbering of packets makes no sense. So the sequence number bits are don't-care bits.







All the services can be divided into two different groups. The services of the first group are used for service- and programming purposes. The other ones are used within normal operation in connection with group addressing.

For the development of application programs the services mentioned last are more important. Therefore they are emphasised by italics in the service table. Using the suitable calls in your application program you can generate telegrams containing these services.

| Service   | APCI | Data   |
|---|------|--|
| <i>Read object value<br/>(Value Read)</i>         | 0000 | <i>no Data</i>   |
| <i>Response object value<br/>(Value Response)</i> | 0001 | <i>value of the object (1 bit - 14 byte)</i>   |
| <i>Write object value<br/>(ValueWrite)</i>        | 0010 | <i>value of the object (1 bit - 14 byte)</i>   |
| <i>Write phys. address<br/>(PhysAddress)</i>      | 0011 | <i>physical address (2 byte)</i>   |
| Read phys. address<br>(AddrRequest)               | 0100 | no data  |
| Response phys. address<br>(AddrResponse)          | 0101 | no data (PA is contained in the source address)  |
| Read AD-converter<br>(AdcRead)                    | 0110 | AD-port-number (6 bit), number of repetitions (1 byte)                                 |
| Response AD-value<br>(AdcResponse)                | 0111 | AD-port-number (6 bit), number of repetitions (1 byte),<br>AD-converter-value (2 byte) |
| Read memory<br>(MemoryRead)                       | 1000 | number of bytes (4 bit), offset address (2 byte)                                       |
| Response memory<br>(MemoryResponse)               | 1001 | number of bytes (4 bit), offset address (2 byte), memory<br>data (1 byte - 12 byte)    |
| Write Memory<br>(MemoryWrite)                     | 1010 | number of bytes (4 bit), offset address (2 byte), memory<br>data (1 byte - 12 byte)    |
| Message to user<br>(UserMsg)                      | 1011 | message (at the most 14 bytes + 6 bit)   |
| Read BAU-maskversion<br>(MaskVersionRead)         | 1100 | no data  |
| Response BAU-maskver.<br>(MaskVersionResponse)    | 1101 | mask type byte (1 byte), version byte (1 byte)   |
| Restart<br>(Restart)                              | 1110 | no data  |
| <escape>  | 1111 | used for management services   |

**Fig. 2/1/1-34: Description of the Application Layer Services (APCI)**

Normal communication on the EIB primarily takes place via communication objects. Because of this fact telegrams with the APCI's 0000, 0001, 0010 are most frequently used.

Within the application program there are two ways to communicate using objects:

- Writing object values:

If you want to update the object value of a group on the bus, you either have to call the operating system function "U\_transRequest" or you have to set the two transmit-request-flags of the object. The operating system will then generate a telegram with APCI = 0010 ("Write object value") and the object value will be transferred.

- Reading object values:

To read an object value of another bus device you have to set the two transmit-request-flags and the request-object-value-flag. The operating system will then send out a telegram with APCI = 0000 ("Read object value").

Response to a "Read object value"-request of another bus device happens by sending a telegram with APCI = 0001 ("Response object value"). The operating system has the unity of command to answer an object-value-request. The application system is not concerned.

Within the application program you only have to call operating system functions. The data and parameters are handed over to the operating system by setting a value at a certain memory address. The operating system handles the following tasks:

- Communication with other bus devices,
- generating telegrams,
- detection of object value changes,
- some I/O-functions,
- checking the integrity of the memory (by checksum),
- and other.

The application program you have to develop does not have to be occupied by these tasks. You have merely to initiate those functions by calling a suitable system software service.

## **4 EIB Interworking Standards (EIS)**

EIB components are produced and offered by many different manufacturers. In order to achieve a successful communication among different bus devices, it is necessary that all manufacturers implement the same 'language' in their bus devices.

For that reason standardised **EIB-functions** were defined, managing fundamental problems of installation in buildings. These EIB-functions are described in **EIB Interworking Standards (EIS)**.

If, for a certain problem, a standard already exists, the application provider shall use this standard in his applications. Beside this, he has to declare for each kind of device, which EIS (standards) are implemented.

If a suitable standard is not yet available, the application provider may fix his own functions and data format. However, in this case a detailed description of this non standard format has to be delivered with the application.

### **General Requirements and Recommendations for Interworking**

General requirements:

- For manual operations the inputs of an application which trigger telegrams should be separated at least by approximately 200 ms.
- For automatic operations the telegram repetition rate should be in the range of seconds or minutes.
- The transmission priority of each object has to be determined very carefully. Low operation priority is intended for ordinary functions.
- Each function has to be represented completely by a single telegram.

Recommendations:

- Communication objects should not be used bi-directionally.
- Status information of a device should be handled in separate communication objects.
- Time delays ought to be realised in actuators, not in sensors.

In the following paragraphs we will look more particularly at the present interworking standards.

## 4.1 EIS 1: Switching

The one bit EIB-function 'Switching' is intended to switch a load attached to the actuator. A group telegram containing the value '1' in its value field is going to switch on the concerning load, while a group telegram containing the value '0' is going to switch off the load.

|             |  |             |       |   |   |   |   |   |   |            |                        |
|-------------|--|-------------|-------|---|---|---|---|---|---|------------|------------------------|
| Function:   | EIB_switching  | Value size: | 1 bit |   |   |   |   |   |   |            |                        |
| Definition: | <table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>X</td></tr></table> | 0           | 0     | 0 | 0 | 0 | 0 | 0 | X | Behaviour: | X: '0' = off, '1' = on |
| 0           | 0  | 0           | 0     | 0 | 0 | 0 | X |   |   |            |                        |

**Fig. 2/1/1-35: EIS 1**

## 4.2 EIS 2: Dimming

The EIB-function 'Dimming' consists of three EIB-subfunctions, called

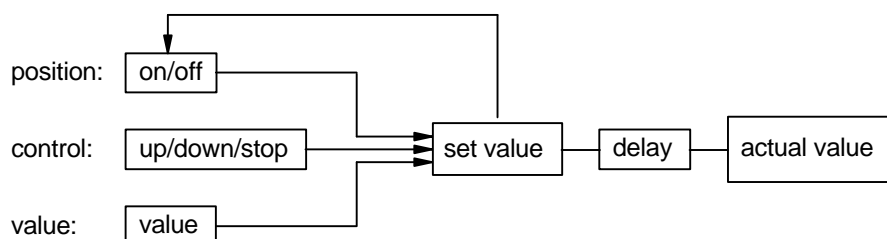
- 'Position' (switching on/off and status)
- 'Control' (relative dimming)
- 'Value' (absolute dimming).

The EIB-subfunction 'Position' supports the switching condition of the dimming actuator.

The EIB-subfunction 'Control' is used to increase or decrease the set value in the dimming actuator (relative dimming). Via this EIB-subfunction it is possible to switch the dimming actuator on, but not to switch it off.

The EIB-subfunction 'Value' directly effects the set value (absolute dimming). Via this EIB-subfunction, it is possible to switch the dimming actuator on and off.

For a dimming actuator, all of the three EIB-subfunctions shall be implemented, for a dimming sensor this is not required. Regardless of the other EIB-subfunctions in a dimming actuator, the last EIB-subfunction received shall be executed.



**Fig. 2/1/1-36: EIS 2**

4.2.1 EIB-Subfunction ‘Position’

With the EIB-subfunction ‘Position’, it is possible to switch a dimming actuator on or off with a simple EIB switch. This EIB-subfunction reflects the on/off status of the actuator.

On every change of the on/off status, irrespective of the trigger, and on every write to the EIB-subfunction ‘Position’ via the bus, a send-request for this EIB-subfunction will be given.

☞ In a group of dimming actuators, only one of them may send back its status on the same group address. By choosing related communication parameters in the BCU it is possible to prevent the sending of the status. ‘Send request’ should be disabled by default.

4.2.2 EIB-Subfunction ‘Control’

With the help of the EIB-subfunction ‘Control’, it is possible to increase or decrease the set value in steps, or stop the movement. The stepcode is also transmitted by this EIB-subfunction ‘Control’. ‘Step-break’ denotes, that the actuator stops at the given value.

It is possible to switch on an actuator by using the EIB-sub function ‘Control’. The stepcode indicates the amount of intervals into which the range of value 0...100% is subdivided.

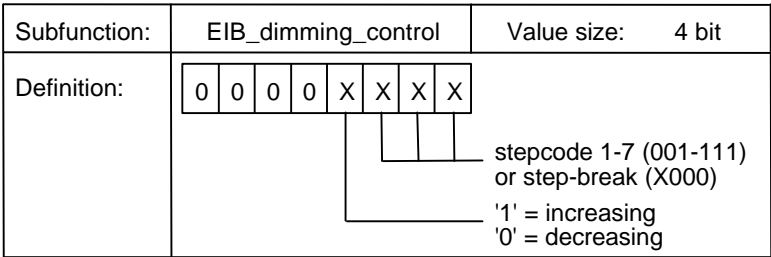


Fig. 2/1/1-37: EIS 2 'Control'

4.2.3 EIB-Subfunction ‘Value’

With the EIB-subfunction ‘Value’, a position within the range between ‘value low’ and 100% can be set directly. By using the EIB-subfunction ‘Value’, it is possible to switch on an actuator by writing a value <> ‘0’ and switch off by writing a ‘0’.

|              |                   |             |       |
|--------------|-------------------|-------------|-------|
| Subfunction: | EIB_dimming_value | Value size: | 8 bit |
|--------------|-------------------|-------------|-------|

|             |   |   |   |   |   |   |   |               |   |   |   |   |   |   |   |   |   |            |   |   |   |   |   |   |   |   |               |  |  |  |   |   |   |  |  |  |   |   |   |   |   |   |   |   |         |
|-------------|---|---|---|---|---|---|---|---------------|---|---|---|---|---|---|---|---|---|------------|---|---|---|---|---|---|---|---|---------------|--|--|--|---|---|---|--|--|--|---|---|---|---|---|---|---|---|---------|
| Definition: | <table border="1"><tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr></table><br><table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>= reserved</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>= 'value low'</td></tr><tr><td></td><td></td><td></td><td>.</td><td>.</td><td>.</td><td></td><td></td><td></td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>= 100 %</td></tr></table> | X | X | X | X | X | X | X             | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = reserved | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | = 'value low' |  |  |  | . | . | . |  |  |  | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | = 100 % |
| X           | X   | X | X | X | X | X | X |               |   |   |   |   |   |   |   |   |   |            |   |   |   |   |   |   |   |   |               |  |  |  |   |   |   |  |  |  |   |   |   |   |   |   |   |   |         |
| 0           | 0   | 0 | 0 | 0 | 0 | 0 | 0 | = reserved    |   |   |   |   |   |   |   |   |   |            |   |   |   |   |   |   |   |   |               |  |  |  |   |   |   |  |  |  |   |   |   |   |   |   |   |   |         |
| 0           | 0   | 0 | 0 | 0 | 0 | 0 | 1 | = 'value low' |   |   |   |   |   |   |   |   |   |            |   |   |   |   |   |   |   |   |               |  |  |  |   |   |   |  |  |  |   |   |   |   |   |   |   |   |         |
|             |   |   | . | . | . |   |   |               |   |   |   |   |   |   |   |   |   |            |   |   |   |   |   |   |   |   |               |  |  |  |   |   |   |  |  |  |   |   |   |   |   |   |   |   |         |
| 1           | 1   | 1 | 1 | 1 | 1 | 1 | 1 | = 100 %       |   |   |   |   |   |   |   |   |   |            |   |   |   |   |   |   |   |   |               |  |  |  |   |   |   |  |  |  |   |   |   |   |   |   |   |   |         |

**Fig. 2/1/1-38: EIS 2 'Value'**

This EIB-subfunction does not reflect the status of the dimming actuator. It can be different from the set- or the actual value.

## 4.2.4 Behaviour

### 4.2.4.1 Status

|         |   |
|---------|---|
| off     | dimming actuator switched off   |
| on      | dimming actuator switched on, constant brightness,<br>at least minimal brightness dimming |
| dimming | actuator switched on, moving from actual value<br>in direction of set value               |

### 4.2.4.2 Events

|                   |                                 |
|-------------------|---------------------------------|
| position = 0      | off command                     |
| position = 1      | on command                      |
| control = up dX   | command, dX more bright dimming |
| control = down dX | command, dX less bright dimming |
| control = stop    | command                         |
| value = 0         | dimming value = off             |
| value = X %       | dimming value = X % (not zero)  |
| value_reached     | actual value reached set value  |

The step size dX for up and down dimming may be 1/1, 1/2, 1/4, 1/8, 1/16, 1/32 and 1/64 of the full dimming range (0 - FFh).



#### 4.2.4.3 Parameters

The standard function requires no parameters at all. Standard behaviour of a dimming actuator:

| DESCRIPTION                              | STANDARD VALUES   |
|--|---|
| dimming speed                            | a sweep from 0 to 100 % in 4 sec shall be possible              |
| turn-on condition                        | set value = FFh   |
| reaction on absolute dimming values      | jmp = jump to value   |
| reaction on bus at power-fail and return | off = switch off when power fails, no action when power returns |

If parameters are implemented, the above mentioned behaviour should be included.

#### 4.2.4.4 Data

The dimming actuator works internally with the set- and the actual value. The actual value follows the set value. By dimming the follow up is delayed.

#### 4.2.4.5 Truth Table

In the truth table, only the standard transitions are given. Parameter dependent transitions are not given. Everywhere in the truth table where 'send status' is written, it is recommended to give the send-request during the event processing.

| STATUS  | EVENT             | ACTION  | FOLLOWING STATUS |
|---------|-------------------|---|------------------|
| off     | position = 0      | switch off;<br>send status;   | off              |
| off     | position = 1      | switch on;<br>send status;<br>actual value = set value = FFh  | on               |
| off     | control = up dX   | switch on;<br>send status;<br>actual value = min. value<br>set value = min. (actual value + dX, max. value) | dimming          |
| off     | control = down dX | none  | off              |
| off     | control stop      | none  | off              |
| off     | value = 0         | none  | off              |
| off     | value = X %       | switch on;<br>send status;<br>actual value = X %  | on               |
| off     | value_reached     | not possible  | off              |
| on      | position = 0      | switch off;<br>send status  | off              |
| on      | position = 1      | actual value = set value = FFh;<br>send status  | on               |
| on      | control = up dX   | set value = min. (actual value + dX, min. value)  | dimming          |
| on      | control = down dX | set value = min. (actual value - dX, min. value)  | dimming          |
| on      | control stop      | set value = actual value  | on               |
| on      | value = 0         | switch off;<br>send status  | off              |
| on      | value = X %       | actual value = X %  | on               |
| on      | value_reached     | not possible  | on               |
| dimming | position = 0      | switch off;<br>send status  | off              |
| dimming | position = 1      | actual value = set value = FFh;<br>send status  | on               |
| dimming | control = up dX   | set value = min. (set value - dX, min. value)   | dimming          |
| dimming | control = down dX | set value = max. (set value - dX, min. value)   | dimming          |
| dimming | control = stop    | set value = actual value  | on               |
| dimming | value = 0         | switch off;<br>send status  | off              |
| dimming | value = X %       | actual value = X %  | on               |
| dimming | value_reached     | none  | on               |

### 4.3 EIS 3: Time

The EIB-function 'Time' gives the current time to all EIB devices which process this information. In order to apply this function it is assumed that a 'system clock' in form of an AM is accessible by the bus.

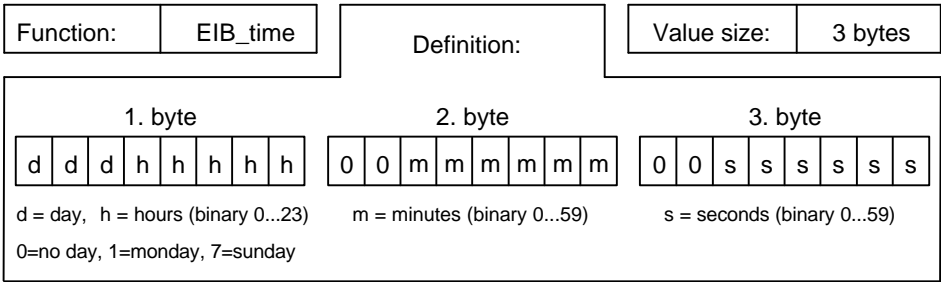


Fig. 2/1/1-39: EIS 3

The EIB-function 'Time' is sent cyclically by the system clock. It also may be possible to fetch the 'EIB\_time' by means of a data request group telegram. It should be possible to adjust the transmission rate of a system clock device by a parameter in the EEPROM.

4.4 EIS 4: Date

The EIB-function 'Date' gives the current date to all EIB devices which process this information. In order to apply this function it is assumed that a 'system clock' in form of an AM is accessible by the bus.

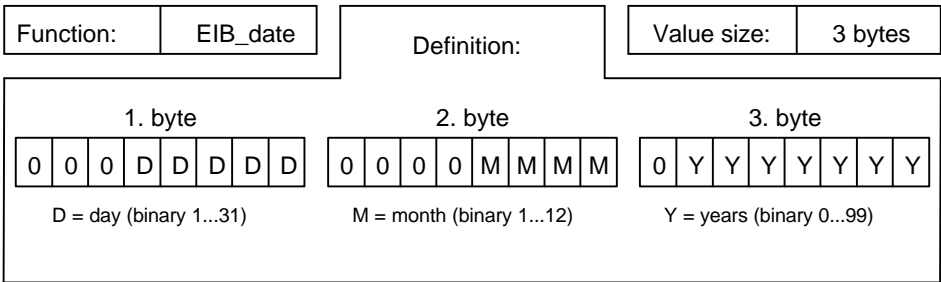
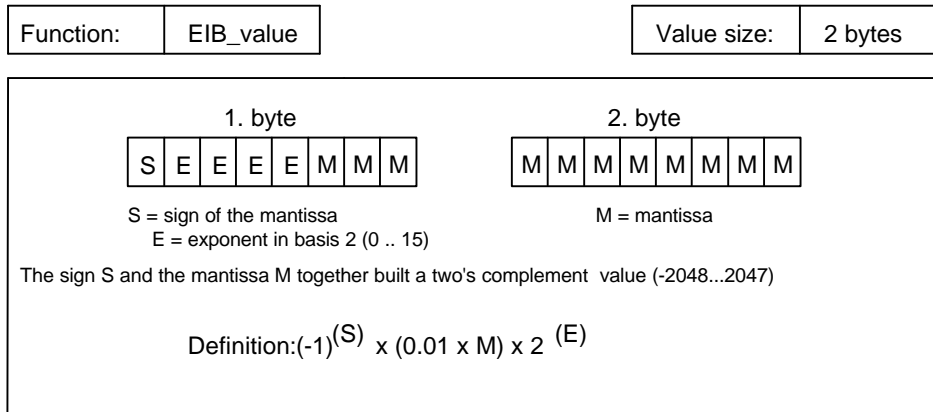


Fig. 2/1/1-40: EIS 4

The EIB-function 'Date' is sent cyclically by a system clock. It also is possible to fetch the 'EIB\_date' by means of a data request group telegram. It should be possible to adjust the transmission rate of the system clock device by a parameter in the EEPROM.

## 4.5 EIS 5: Value

The EIB-function 'Value' is used to transmit data representing physical values. However, physical units are not transmitted.



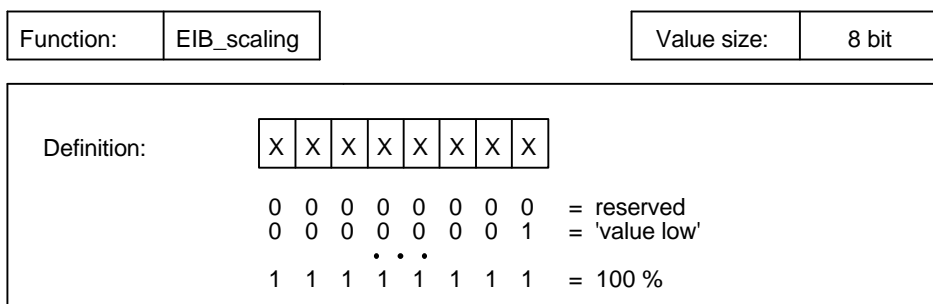
**Fig. 2/1/1-41: EIS 5**



Of course it is possible to transmit larger values or values of higher accuracy on the EIB bus. The transfer of up to 14 bytes of data in a single telegram is supported.

## 4.6 EIS 6: Scaling

The EIB-function 'Scaling' is used to transmit relative values with a resolution of 8 bit. This EIB-function represents e.g. the relative brightness in the range from 'value low' and 100%.



**Fig. 2/1/1-42: EIS 6**

The present units of measurement are relative brightness, relative humidity and wind direction.

## 4.7 EIS 7: Drive Control

The EIB-function ‘Drive Control’ consists of two EIB-subfunctions called ‘Move’ and ‘Step’.

- With help of the EIB-subfunction ‘Move’, it is possible to set the drive in motion or to change the direction of the movement. In addition, a gradual movement is possible.
- With help of the EIB-subfunction ‘Step’, it is possible to stop the drive in motion or to step the movement.

|              |  |                |   |   |   |   |   |   |   |  |   |   |   |   |   |   |   |   |
|--------------|--|----------------|---|---|---|---|---|---|---|--|---|---|---|---|---|---|---|---|
| Subfunction: | EIB_drive_move   | EIB_drive_step |   |   |   |   |   |   |   |  |   |   |   |   |   |   |   |   |
| Value size:  | 1 bit  | 1 bit          |   |   |   |   |   |   |   |  |   |   |   |   |   |   |   |   |
| Definition:  | <table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>M</td></tr></table> | 0              | 0 | 0 | 0 | 0 | 0 | 0 | M | <table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>S</td></tr></table> | 0 | 0 | 0 | 0 | 0 | 0 | 0 | S |
| 0            | 0  | 0              | 0 | 0 | 0 | 0 | M |   |   |  |   |   |   |   |   |   |   |   |
| 0            | 0  | 0              | 0 | 0 | 0 | 0 | S |   |   |  |   |   |   |   |   |   |   |   |

**Fig. 2/1/1-43: EIS 7**

The behaviour of the single subfunctions is shown below:

M = ‘1’ → move down (close)

M = ‘0’ → move up (open)

S = ‘1’ → stop or step down

S = ‘0’ → stop or step up



In case of the EIB-subfunction ‘Move’, no member of a group shall answer to a read access from the communication system, as an answer may cause a stopped drive to be set in motion.

In case of the EIB-subfunction ‘Step’, no member of a group should answer on a read access from the communication system, as an answer may cause a moved drive in steps to stop or a stopped drive move in steps.

The standard version has no parameters at all. Standard behaviour of an actuator is:

| DESCRIPTION  | STANDARD VALUES                       |
|--|---------------------------------------|
| move time specifies the duration of moving the drive | no limit or at least ‘very long time’ |
| step time specifies the time of one step             | no step or at least ‘very small time’ |

If parameters are implemented, the above mentioned behaviours should be included.

The actual behaviour of the EIB-function ‘Drive Control’ depends on the previous state of this EIB-function (i.e. the state of the drive). The complete behaviour is shown in the following table.

| STATUS        | EVENT    | ACTION  | FOLLOWING STATUS         |
|---------------|----------|---|--------------------------|
| stopped       | move = 1 | moving down;<br>optional: time-out = movetime;        | moving (down)            |
| stopped       | move = 0 | moving up;<br>optional: time-out = movetime;          | moving (up)              |
| stopped       | step = 1 | none;<br>optional: step down;<br>time-out = steptime; | stopped<br>stepping down |
| stopped       | step = 0 | none;<br>optional: step up;<br>time-out = steptime;   | stopped<br>stepping up   |
| stopped       | time-out | none;   | stopped                  |
| moving        | move = 1 | moving down;<br>optional: time-out = movetime;        | moving down              |
| moving        | move = 0 | moving up;<br>optional: time-out = movetime;          | moving up                |
| moving        | step = 1 | stop;   | stopped                  |
| moving        | step = 0 | stop;   | stopped                  |
| moving        | time-out | stop;   | stopped                  |
| stepping up   | move = 1 | moving down;<br>optional: time-out = movetime;        | moving (down)            |
| stepping up   | move = 0 | moving up;<br>optional: time-out = movetime;          | moving (up)              |
| stepping up   | step = 1 | step down;<br>time-out = steptime;                    | stepping down            |
| stepping up   | step = 0 | step up;<br>time-out = steptime;                      | stepping up              |
| stepping up   | time-out | stop;   | stopped                  |
| stepping down | move = 1 | moving down;<br>optional: time-out = movetime;        | moving (down)            |
| stepping down | move = 0 | moving up;<br>optional: time-out = movetime;          | moving (up)              |
| stepping down | step = 1 | step down;<br>time-out = steptime;                    | stepping down            |
| stepping down | step = 0 | step up;<br>time-out = steptime;                      | stepping up              |
| stepping down | time-out | stop;   | stopped                  |

## 4.8 EIS 8: Priority

The EIB-function 'Priority' consists of two EIB-subfunctions. The function's output value is formed by a combination of the output values of the EIB-subfunctions EIB\_priority\_position and EIB\_priority\_control:

- EIB\_priority\_position:

This subfunction corresponds to the function EIB\_switch. It is possible to switch an actuator under control of EIB\_priority\_control.

- EIB\_priority\_control:

This subfunction, when active, forces the output status of the priority-function to a certain value. When EIB\_priority\_control is not active, the output of 'Priority' is determined by the output state of EIB\_priority\_position.

|              |  |                      |   |   |   |                |                |   |   |  |   |   |   |   |   |   |                |                |
|--------------|--|----------------------|---|---|---|----------------|----------------|---|---|--|---|---|---|---|---|---|----------------|----------------|
| Subfunction: | EIB_priority_position  | EIB_priority_control |   |   |   |                |                |   |   |  |   |   |   |   |   |   |                |                |
| Value size:  | 1 bit  | 2 bit                |   |   |   |                |                |   |   |  |   |   |   |   |   |   |                |                |
| Definition:  | <table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>P</td></tr></table> | 0                    | 0 | 0 | 0 | 0              | 0              | 0 | P | <table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>C<sub>1</sub></td><td>C<sub>0</sub></td></tr></table> | 0 | 0 | 0 | 0 | 0 | 0 | C <sub>1</sub> | C <sub>0</sub> |
| 0            | 0  | 0                    | 0 | 0 | 0 | 0              | P              |   |   |  |   |   |   |   |   |   |                |                |
| 0            | 0  | 0                    | 0 | 0 | 0 | C <sub>1</sub> | C <sub>0</sub> |   |   |  |   |   |   |   |   |   |                |                |

**Fig. 2/1/1-44: EIS 8**

The behaviour of EIB\_priority is shown in the following truth table:

| EIB_priority_control<br>C <sub>1</sub> C <sub>0</sub> |            | EIB_priority_position<br>P | Output of EIB_priority |
|---|------------|----------------------------|------------------------|
| 0   | don't care | 0                          | 0                      |
| 0   | don't care | 1                          | 1                      |
| 1   | 0          | don't care                 | 0                      |
| 1   | 1          | don't care                 | 1                      |

**Fig. 2/1/1-45: EIS 8 Truth Table**

C<sub>1</sub> decides, whether EIB\_priority\_control is active (C<sub>1</sub>=1') or not (C<sub>1</sub>=0'). If active, the value of C<sub>0</sub> appears at the output of EIB\_priority, otherwise the value of P is delivered.

## 4.9 EIS 9: Float Value

The EIB-function "EIB\_Float\_Value" is used to transmit data representing physical values, like electric current (A), pressure (Pa), power (W), ... in IEEE floating point format.

|             |                 |             |                                       |
|-------------|-----------------|-------------|---------------------------------------|
| Function:   | EIB_Float_Value | Value size: | 4 bytes                               |
| Definition: | Bit             | 31          | 30    23    22                      0 |
|             |                 | Sign        | exponent      mantissa                |
|             | Byte            | 1           | 2      3      4                       |

**Fig. 2/1/1-46: EIS 9 'Float Value'**



The units of measurement are not transmitted over the bus.

## 4.10 EIS 10: 16-bit Counter Value

The EIB-function “EIB\_Counter 16-bit” is used to transmit data representing 16-bit counter values. The negative values of the signed counter values are encoded in the two’s complement.

The counter values are encoded as follows:

Signed counter value:

|      |      |               |   |
|------|------|---------------|---|
| Bit  | 15   | 14            | 0 |
|      | Sign | binary number |   |
| Byte | 1    | 2             |   |

Unsigned counter value:

|      |               |   |
|------|---------------|---|
| Bit  | 15            | 0 |
|      | binary number |   |
| Byte | 1             | 2 |

## 4.11 EIS 11: 32-bit Counter Value

The EIB-function “EIB\_Counter 16-bit” is used to transmit data representing 16-bit counter values. The negative values of the signed counter values are encoded in the two’s complement.

The counter values are encoded as follows:

Signed long counter value:

|      |      |               |   |  |   |   |
|------|------|---------------|---|--|---|---|
| Bit  | 31   | 30            |   |  |   | 0 |
|      | Sign | binary number |   |  |   |   |
| Byte | 1    |               | 2 |  | 3 | 4 |

Unsigned long counter value:

|      |                                       |   |   |   |
|------|---------------------------------------|---|---|---|
| Bit  | 31 <span style="float:right">0</span> |   |   |   |
|      | binary number                         |   |   |   |
| Byte | 1                                     | 2 | 3 | 4 |



## 5 EIB Application

### 5.1 Application program

A conventional application program is built up from 3 routines:

- U\_Init: the user initialisation program
- U\_Main: the user main program
- U\_Save: the user save program

All 3 routines are called as subroutines for the system software and are therefore to be terminated by “RTS” (Return from Subroutine). These routines do not necessarily need to be implemented, but at least an entry with “RTS” is required.

#### 5.1.1 User Initialisation Program (U\_Init)

The BAU system software calls the initialisation routine once at start-up, e.g. after a reset or on power on.

Starts at: 100H + [0x0113]

The application programmer writes the start of the user initialisation program minus an offset of 100H at the EEPROM address 0x0113 for BCU 1. This way, the system software is able to calculate the start address for the initialisation program.

This routine helps to bring the product in a certain state. For instance, it can open or close relay contacts (for safety conditions when bus voltage turns back), immediately send certain sensor values or restore some data back from EEPROM.

#### 5.1.2 User Save Program (U\_Save)

The BAU system software calls this save program on power supply breakdown. After return from this routine, the BAU is reset.

Starts at: 100H + [0x0115]

Sixty milliseconds are guaranteed for this routine for certain actions, like switching off a load, switch on emergency lighting or save at maximum two RAM bytes to EEPROM, such as current communication object values, that can be read back by the U\_Init routine.

### 5.1.3 User Main Program (U\_Main)

This is the actual application program that is periodically called by the system software.

Starts at: 100H + [0x0114]

Before the application program is called, the processor's registers (REGA-REGN) are cleared. This routine may last 7 ms at maximum, or exceptionally 10 ms. The length of the intervals in between two following calls is strongly dependent on the bus communication and/or a possible serial communication through the PEI. Make sure to reset the watch-dog timer every 1.5 ms, though most applications take less time.

## 5.2 Software Services

The implementation and use of these services is strongly dependent on the BAU-type and version. Therefore, in the following only an overview of the currently available services in the various BAU's will be given. See how these services can be found in the application examples in the medium dependent chapters of this Cookbook. For a detailed description you should refer to the BAU's data sheets in Volume 2 / Part 3 of the EIBA Handbook Series.

### 5.2.1 Communication Object Manipulation Functions

#### 5.2.1.1 Reading RAM-Flags

##### U\_FlagsGet

|                        |   |                 |          |                      |       |
|------------------------|---|-----------------|----------|----------------------|-------|
| <u>Description:</u>    | fetches the RAM-flags of a specified communication object, without resetting the update-flag. |                 |          |                      |       |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u> | mask 1.0 | <u>Call address:</u> | 0C8CH |
|                        | <b>BIM M111</b>   |                 | mask 1.1 |                      | 0C9DH |
|                        |   |                 | mask 1.2 |                      | 0C9DH |

#### 5.2.1.2 Testing RAM-Flags

##### U\_testObj

|                        |   |                 |          |                      |       |
|------------------------|---|-----------------|----------|----------------------|-------|
| <u>Description:</u>    | fetches the RAM-flags of a specified communication object, <b>and</b> resets the update-flag. |                 |          |                      |       |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u> | mask 1.1 | <u>Call address:</u> | 0CA5H |
|                        | <b>BIM M111</b>   |                 | mask 1.2 |                      | 0CA5H |

### 5.2.1.3 Writing RAM-Flags

#### U\_flagsSet

|                        |   |                 |          |                            |
|------------------------|---|-----------------|----------|----------------------------|
| <u>Description:</u>    | sets the RAM-flags of a specified communication object. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u> | mask 1.0 | <u>Call address:</u> 0C94H |
|                        | <b>BIM M111</b>   |                 | mask 1.1 | 0CB3H                      |
|                        |   |                 | mask 1.2 | 0CB3H                      |

### 5.2.1.4 Setting Transmit-request

#### U\_transRequest

|                        |   |                 |          |                            |
|------------------------|---|-----------------|----------|----------------------------|
| <u>Description:</u>    | sets a transmit request for the specified communication object. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u> | mask 1.0 | <u>Call address:</u> 0D91H |
|                        | <b>BIM M111</b>   |                 | mask 1.1 | 0DB9H                      |
|                        |   |                 | mask 1.2 | 0DB9H                      |

## 5.2.2 EEPROM Manipulation Support Functions

### 5.2.2.1 Writing to EEPROM

#### EEwrite

|                        |   |                 |          |                            |
|------------------------|---|-----------------|----------|----------------------------|
| <u>Description:</u>    | writes a byte to the specified location in EEPROM-memory.<br>Attention: update checksum if necessary. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u> | mask 1.0 | <u>Call address:</u> 0C2DH |
|                        | <b>BIM M111</b>   |                 | mask 1.1 | 0C38H                      |
|                        |   |                 | mask 1.2 | 0C38H                      |

### 5.2.2.2 Updating Checksum

#### EEsetChecksum

|                        |   |                 |          |                            |
|------------------------|---|-----------------|----------|----------------------------|
| <u>Description:</u>    | updates the checksum byte of the EEPROM. This function must be called if any byte inside the check-range is modified by the USER. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u> | mask 1.0 | <u>Call address:</u> 0C5DH |
|                        | <b>BIM M111</b>   |                 | mask 1.1 | 0C68H                      |
|                        |   |                 | mask 1.2 | 0C68H                      |

### 5.2.3 Application Support Functions

#### 5.2.3.1 Debouncing

##### U\_debounce

|                        |   |                 |                      |
|------------------------|---|-----------------|----------------------|
| <u>Description:</u>    | debounces a complete byte. As long as the debounce time is not yet expired, or the value is changing, the latest debounced value is returned. |                 |                      |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u> | <u>Call address:</u> |
|                        | <b>BIM M111</b>   | mask 1.0        | 0C64H                |
|                        |   | mask 1.1        | 0C75H                |
|                        |   | mask 1.2        | 0C75H                |

##### U\_deb10

|                        |  |                 |                      |
|------------------------|--|-----------------|----------------------|
| <u>Description:</u>    | This function is the same as U_debounce, except that a fixed debounce time of 10 ms is used. |                 |                      |
| <u>Implemented in:</u> | <b>BCU 1</b>   | <u>Version:</u> | <u>Call address:</u> |
|                        | <b>BIM M111</b>  | mask 1.1        | 0C73H                |
|                        |  | mask 1.2        | 0C73H                |

##### U\_deb30

|                        |  |                 |                      |
|------------------------|--|-----------------|----------------------|
| <u>Description:</u>    | This function is the same as U_debounce, except that a fixed debounce time of 30 ms is used. |                 |                      |
| <u>Implemented in:</u> | <b>BCU 1</b>   | <u>Version:</u> | <u>Call address:</u> |
|                        | <b>BIM M111</b>  | mask 1.1        | 0C6FH                |
|                        |  | mask 1.2        | 0C6FH                |

#### 5.2.3.2 Ignoring Messages to the User

##### U\_delMsgs

|                        |   |                 |                      |
|------------------------|---|-----------------|----------------------|
| <u>Description:</u>    | removes any messages addressed to the user program. Call this function in the user main-routine if you do not expect any messages. Otherwise, the buffer resources in the BCU may get exhausted due to external faults. |                 |                      |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u> | <u>Call address:</u> |
|                        | <b>BIM M111</b>   | mask 1.0        | 0C82H                |
|                        |   | mask 1.1        | 0C93H                |
|                        |   | mask 1.2        | 0C93H                |

#### 5.2.3.3 Characterisation Function

##### U\_map

|                        |  |                 |                      |
|------------------------|--|-----------------|----------------------|
| <u>Description:</u>    | maps the input value by means of a conversion table. |                 |                      |
| <u>Implemented in:</u> | <b>BCU 1</b>   | <u>Version:</u> | <u>Call address:</u> |
|                        | <b>BIM M111</b>                                      | mask 1.0        | 0C9BH                |
|                        |  | mask 1.1        | 0CBAH                |
|                        |  | mask 1.2        | 0CBAH                |

### 5.2.3.4 Doing AD-Conversion

#### U\_readAD

|                        |   |                 |          |                            |
|------------------------|---|-----------------|----------|----------------------------|
| <u>Description:</u>    | starts the AD-conversion for the specified port and reads the result. This operation is repeated a specified number of times. The returned value is the sum of all read values. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u> | mask 1.0 | <u>Call address:</u> 0D35H |
|                        | <b>BIM M111</b>   |                 | mask 1.1 | 0D54H                      |
|                        |   |                 | mask 1.2 | 0D54H                      |

## 5.2.4 PEI-support functions

### 5.2.4.1 Binary-Port Access

#### U\_ioAST

|                        |   |                 |          |                            |
|------------------------|---|-----------------|----------|----------------------------|
| <u>Description:</u>    | allows reading & writing from/to pins of the PEI. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>                                      | <u>Version:</u> | mask 1.0 | <u>Call address:</u> 0DA7H |
|                        | <b>BIM M111</b>                                   |                 | mask 1.1 | 0DCFH                      |
|                        |   |                 | mask 1.2 | 0DCFH                      |

### 5.2.4.2 Serial Interface via PEI

#### S\_AstShift

|                        |  |                 |          |                            |
|------------------------|--|-----------------|----------|----------------------------|
| <u>Description:</u>    | the specified data-block is exchanged via the serial PEI, using a ca. 130 ms time-out. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>   | <u>Version:</u> | mask 1.0 | <u>Call address:</u> 1103H |
|                        | <b>BIM M111</b>  |                 | mask 1.1 | 1117H                      |
|                        |  |                 | mask 1.2 | 1117H                      |

#### S\_LAstShift

|                        |  |                 |          |                            |
|------------------------|--|-----------------|----------|----------------------------|
| <u>Description:</u>    | the specified data-block is exchanged via the serial PEI, using a ca. 1s time-out. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>   | <u>Version:</u> | mask 1.0 | <u>Call address:</u> 1101H |
|                        | <b>BIM M111</b>  |                 | mask 1.1 | 1115H                      |
|                        |  |                 | mask 1.2 | 1115H                      |

#### U\_SerialShift

|                        |  |                 |          |                            |
|------------------------|--|-----------------|----------|----------------------------|
| <u>Description:</u>    | the specified byte is exchanged via the serial PEI, using a ca. 130 ms time-out. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>   | <u>Version:</u> | mask 1.2 | <u>Call address:</u> 0C90H |
|                        | <b>BIM M111</b>  |                 |          |                            |

## **5.2.5 Timer Functions**

### **5.2.5.1 System Timer**

#### **5.2.5.1.1 Starting Timer**

##### **TM\_Load**

|                        |  |                 |          |                      |       |
|------------------------|--|-----------------|----------|----------------------|-------|
| <u>Description:</u>    | the specified timer is initialised with the indicated operation mode and time-range. In addition, in operation mode 0, the run-time is set and the timer is started. |                 |          |                      |       |
| <u>Implemented in:</u> | <b>BCU 1</b>   | <u>Version:</u> | mask 1.0 | <u>Call address:</u> | 0E0CH |
|                        | <b>BIM M111</b>  |                 | mask 1.1 |                      | 0E2BH |
|                        |  |                 | mask 1.2 |                      | 0E2BH |

#### **5.2.5.1.2 Reading Timer Status**

##### **TM\_GetFlg**

|                        |  |                 |          |                      |       |
|------------------------|--|-----------------|----------|----------------------|-------|
| <u>Description:</u>    | the timer-status of the specified timer is returned. In operation mode 0 it returns if the run-time has expired. In operation mode 1, the time since the last call to this function is returned. |                 |          |                      |       |
| <u>Implemented in:</u> | <b>BCU 1</b>   | <u>Version:</u> | mask 1.0 | <u>Call address:</u> | 0E2AH |
|                        | <b>BIM M111</b>  |                 | mask 1.1 |                      | 0E49H |
|                        |  |                 | mask 1.2 |                      | 0E49H |

### **5.2.5.2 User Timer**

#### **5.2.5.2.1 Starting User Timer**

##### **U\_SetTM**

|                        |                    |                 |          |                      |       |
|------------------------|--------------------|-----------------|----------|----------------------|-------|
| <u>Description:</u>    | loads a user-timer |                 |          |                      |       |
| <u>Implemented in:</u> | <b>BCU 1</b>       | <u>Version:</u> | mask 1.0 | <u>Call address:</u> | 0D8AH |
|                        | <b>BIM M111</b>    |                 | mask 1.1 |                      | 0DB3H |
|                        |                    |                 | mask 1.2 |                      | 0DB3H |

##### **U\_SetTMx**

|                        |  |                 |          |                      |       |
|------------------------|--|-----------------|----------|----------------------|-------|
| <u>Description:</u>    | this function is the same as U_SetTM except that the pointer to the EEPROM description block is fetched from the byte directly before the user main program. |                 |          |                      |       |
| <u>Implemented in:</u> | <b>BCU 1</b>   | <u>Version:</u> | mask 1.1 | <u>Call address:</u> | 0DAFH |
|                        | <b>BIM M111</b>  |                 | mask 1.2 |                      | 0DAFH |

### 5.2.5.2.2 Reading User Timer Status

#### U\_GetTM


|                        |   |                 |          |                            |
|------------------------|---|-----------------|----------|----------------------------|
| <u>Description:</u>    | gets the status of a user-timer. This is the only function which updates the specified timer. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u> | mask 1.0 | <u>Call address:</u> 0D4DH |
|                        | <b>BIM M111</b>   |                 | mask 1.1 | 0D71H                      |
|                        |   |                 | mask 1.2 | 0D71H                      |

#### U\_GetTMx

|                        |   |                 |          |                            |
|------------------------|---|-----------------|----------|----------------------------|
| <u>Description:</u>    | this function is the same as U_GetTM, except that the pointer to the EEPROM description block is fetched from the byte directly before the user main program. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u> | mask 1.1 | <u>Call address:</u> 0D6CH |
|                        | <b>BIM M111</b>   |                 | mask 1.2 | 0D6CH                      |

### 5.2.5.2.3 Delay

#### U\_Delay

|                        |  |                 |          |                            |
|------------------------|--|-----------------|----------|----------------------------|
| <u>Description:</u>    | waits for the specified amount of time. The delay is based upon the internal hardware timer.<br> No delay-times above 15 ms should be used. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>   | <u>Version:</u> | mask 1.0 | <u>Call address:</u> 0DDBH |
|                        | <b>BIM M111</b>  |                 | mask 1.1 | 0DFAH                      |
|                        |  |                 | mask 1.2 | 0DFAH                      |

## 5.2.6 Message Handling Functions

### 5.2.6.1 Buffer Allocation

#### AllocBuf

|                        |                             |                 |          |                            |
|------------------------|-----------------------------|-----------------|----------|----------------------------|
| <u>Description:</u>    | allocates a message buffer. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>                | <u>Version:</u> | mask 1.0 | <u>Call address:</u> 116AH |
|                        | <b>BIM M111</b>             |                 | mask 1.1 | 117EH                      |
|                        |                             |                 | mask 1.2 | 117EH                      |

### 5.2.6.2 Buffer Release

#### FreeBuf

|                        |   |                 |          |                            |
|------------------------|---|-----------------|----------|----------------------------|
| <u>Description:</u>    | releases a previously allocated buffer. |                 |          |                            |
| <u>Implemented in:</u> | <b>BCU 1</b>                            | <u>Version:</u> | mask 1.0 | <u>Call address:</u> 118CH |
|                        | <b>BIM M111</b>                         |                 | mask 1.1 | 11A0H                      |
|                        |   |                 | mask 1.2 | 11A0H                      |

### 5.2.6.3 Message Request

#### PopBuf

|                        |                                      |                      |          |
|------------------------|--------------------------------------|----------------------|----------|
| <u>Description:</u>    | searches for a certain message type. |                      |          |
| <u>Implemented in:</u> | <b>BCU 1</b>                         | <u>Version:</u>      | mask 1.0 |
|                        | <b>BIM M111</b>                      |                      | mask 1.1 |
|                        |                                      |                      | mask 1.2 |
|                        |                                      | <u>Call address:</u> | 11ACH    |
|                        |                                      |                      | 11C0H    |
|                        |                                      |                      | 11C0H    |

## 5.2.7 Arithmetic functions

### 5.2.7.1 Unsigned Integer Multiplication

#### multDE\_FG

|                        |   |                      |          |
|------------------------|---|----------------------|----------|
| <u>Description:</u>    | multiplies the unsigned integer values in the registers RegD:RegE and RegF:RegG |                      |          |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u>      | mask 1.0 |
|                        | <b>BIM M111</b>   |                      | mask 1.1 |
|                        |   |                      | mask 1.2 |
|                        |   | <u>Call address:</u> | 0B3CH    |
|                        |   |                      | 0B4BH    |
|                        |   |                      | 0B4BH    |

### 5.2.7.2 Unsigned Integer Division

#### divDE\_BC

|                        |  |                      |          |
|------------------------|--|----------------------|----------|
| <u>Description:</u>    | divides the unsigned integer value in the registers RegD:RegE by the unsigned integer value RegB:RegC. |                      |          |
| <u>Implemented in:</u> | <b>BCU 1</b>   | <u>Version:</u>      | mask 1.0 |
|                        | <b>BIM M111</b>  |                      | mask 1.1 |
|                        |  |                      | mask 1.2 |
|                        |  | <u>Call address:</u> | 0AFCH    |
|                        |  |                      | 0B0BH    |
|                        |  |                      | 0B0BH    |

## 5.2.8 Miscellaneous Functions

### 5.2.8.1 Accu Shift / Rotate

#### 5.2.8.1.1 AccuShift Left

#### shlAn

|                        |  |                      |             |
|------------------------|--|----------------------|-------------|
| <u>Description:</u>    | shifts the accu left by n bits. The possible values for n are 4, 5, 6 and 7. The new bits are set to zero. |                      |             |
| <u>Implemented in:</u> | <b>BCU 1</b>   | <u>Version:</u>      | mask 1.0    |
|                        | <b>BIM M111</b>  |                      | mask 1.1    |
|                        |  |                      | mask 1.2    |
|                        |  | <u>Call address:</u> | 0B9AH (n=4) |
|                        |  |                      | 0B99H (n=5) |
|                        |  |                      | 0B98H (n=6) |
|                        |  |                      | 0B97H (n=7) |
|                        |  |                      | 0BA9H (n=4) |
|                        |  |                      | 0BA8H (n=5) |
|                        |  |                      | 0BA7H (n=6) |
|                        |  |                      | 0BA6H (n=7) |



## 5.2.8.1.2 AccuShift Right

### shrAn

|                        |   |                 |          |                      |             |
|------------------------|---|-----------------|----------|----------------------|-------------|
| <u>Description:</u>    | shifts the accu right by n bits. The possible values for n are 4, 5, 6 and 7. The new bits are set to zero. |                 |          |                      |             |
| <u>Implemented in:</u> | <b>BCU 1</b>  | <u>Version:</u> | mask 1.0 | <u>Call address:</u> | 0BDAH (n=4) |
|                        | <b>BIM M111</b>   |                 | mask 1.1 |                      | 0BD9H (n=5) |
|                        |   |                 | mask 1.2 |                      | 0BD8H (n=6) |
|                        |   |                 |          |                      | 0BD7H (n=7) |
|                        |   |                 |          |                      | 0BE9H (n=4) |
|                        |   |                 |          |                      | 0BE8H (n=5) |
|                        |   |                 |          |                      | 0BE7H (n=6) |
|                        |   |                 |          |                      | 0BE6H (n=7) |

## 5.2.8.1.3 Accu Rotate Left

### rolAn

|                        |                                  |                 |          |                      |             |
|------------------------|----------------------------------|-----------------|----------|----------------------|-------------|
| <u>Description:</u>    | rotates the accu left by n bits. |                 |          |                      |             |
| <u>Implemented in:</u> | <b>BCU 1</b>                     | <u>Version:</u> | mask 1.2 | <u>Call address:</u> | 0AF4H (n=1) |
|                        | <b>BIM M111</b>                  |                 |          |                      | 0AF2H (n=2) |
|                        |                                  |                 |          |                      | 0AF0H (n=3) |
|                        |                                  |                 |          |                      | 0AEEH (n=4) |
|                        |                                  |                 |          |                      | 0AECB (n=5) |

## 5.2.8.2 Bit Manipulation

### 5.2.8.2.1 BitWrite

#### U\_SetBit

|                        |  |                 |          |                      |       |
|------------------------|--|-----------------|----------|----------------------|-------|
| <u>Description:</u>    | sets the specified bit in register RegH. |                 |          |                      |       |
| <u>Implemented in:</u> | <b>BCU 1</b>                             | <u>Version:</u> | mask 1.0 | <u>Call address:</u> | 0DF9H |
|                        | <b>BIM M111</b>                          |                 | mask 1.1 |                      | 0E18H |
|                        |  |                 | mask 1.2 |                      | 0E18H |

### 5.2.8.2.2 BitRead

#### U\_GetBit

|                        |   |                 |          |                      |       |
|------------------------|---|-----------------|----------|----------------------|-------|
| <u>Description:</u>    | reads the specified bit in register RegH. |                 |          |                      |       |
| <u>Implemented in:</u> | <b>BCU 1</b>                              | <u>Version:</u> | mask 1.0 | <u>Call address:</u> | 0DEDH |
|                        | <b>BIM M111</b>                           |                 | mask 1.1 |                      | 0E0CH |
|                        |   |                 | mask 1.2 |                      | 0E0CH |

### 5.2.8.3 Tables

These tables can be helpful for mask-operations.

#### 5.2.8.3.1 OR-TAB

**OR\_TAB**

Description:

| Addr. (HEX) | Value (binary) |
|-------------|----------------|
| 28          | 1111 1110      |
| 29          | 1111 1101      |
| 2A          | 1111 1011      |
| 2B          | 1111 0111      |
| 2C          | 1110 1111      |
| 2D          | 1101 1111      |
| 2E          | 1011 1111      |
| 2F          | 0111 1111      |

Implemented in:

BCU 1

BIM M111

Version:

mask 1.0

mask 1.1

mask 1.2

Call address:

0020H

0020H

0020H

#### 5.2.8.3.2 AND-TAB

**AND\_TAB**

Description:

| Addr. (HEX) | Value (binary) |
|-------------|----------------|
| 20          | 0000 0001      |
| 21          | 0000 0010      |
| 22          | 0000 0100      |
| 23          | 0000 1000      |
| 24          | 0001 0000      |
| 25          | 0010 0000      |
| 26          | 0100 0000      |
| 27          | 1000 0000      |

Implemented in:

BCU 1

BIM M111

Version:

mask 1.0

mask 1.1

mask 1.2

Call address:

0028H

0028H

0028H

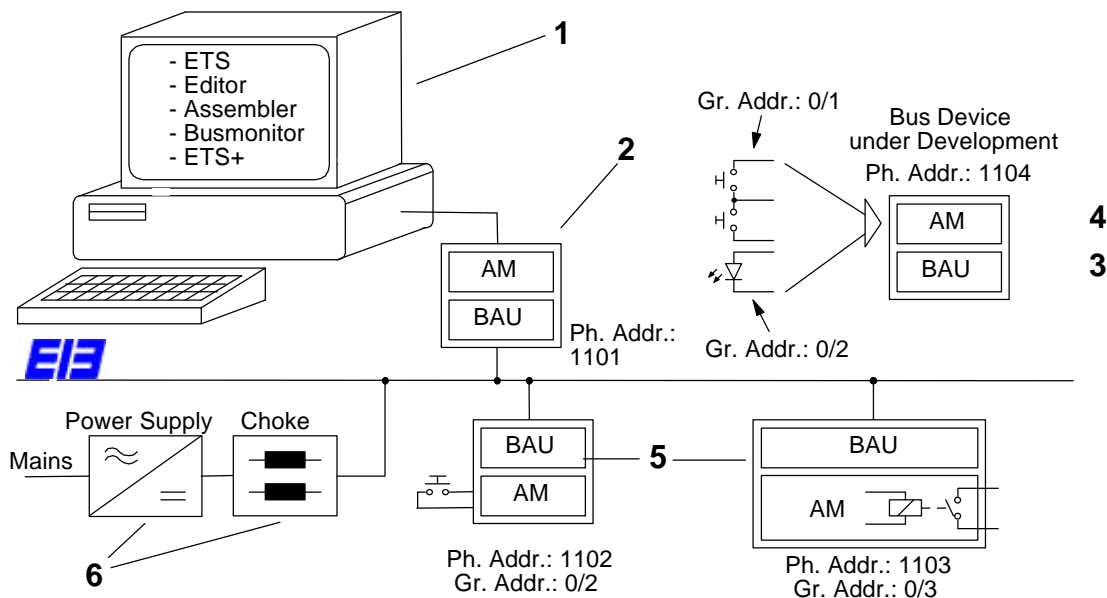
## 6 Development Tools

### 6.1 Reference Installation

Before you write your first AP, you should install at least one bus line, henceforth called the *development bus line*, consisting of the following devices:

1. A PC, with the development software installed,
2. a serial interface device,
3. a BAU for the new AP,
4. the Application Module, matching to the AP,
5. reference bus devices and
6. a power supply unit and a choke, in case of development for TP

Fig. 2/1/1-47 shows this configuration of a development bus line. It can be regarded as a minimum configuration though the reference devices are not absolutely necessary.



**Fig. 2/1/1-47: The development bus line**

Now we will briefly discuss the tasks of these devices.

#### 6.1.1.1 Personal Computer

All activities concerning EIB are carried out by means of a 'normal' compatible personal computer (PC). The PC may be used

- for the AP development,
- to check-up the correct functioning of the AP,
- to wrap up the AP into a product database, and
- for installation of EIB systems.

These activities are supported by various software tools that will be described further on. Since some of these tools handle extensive relational databases an 80486DX processor with 8 Mb or more main memory is recommended.

#### 6.1.1.2 Serial Interface

The serial interface is used to connect the PC to the EIB. The AP of the serial interface realises direct communication between the PC and the EIB.

- ☞ For the use of the serial interface, please refer to the manufacturer's specific documentation and database.

#### 6.1.1.3 BAU

This is the BAU-type that you want to develop your product for. BAU's do not only differ in the medium that they are intended for, but also in available memory for the application program, computing power and the system implementation and mask-version (indicated on the BAU's housing). The variations in housing (flush mounted, surface mounted and DIN rail mounted) are not relevant to their functionalities.

#### 6.1.1.4 Application Module (AM)

The AM must be available at least as a prototype for AP development. It is not absolutely necessary that the AM already realises all features at this stage of development. Quite the reverse, at this stage it may even be more advantageous if some functions of the AM are merely simulated.

Thus, for example, in order to develop an AP for an electronic thermometer you might temporarily substitute the (analogue) temperature sensor by a potentiometer.

#### 6.1.1.5 Power Supply Unit and choke

The power for TP devices is normally provided by one single Power Supply. TP bus devices may sink up to 100 mW, but normal power consumption should be considerably lower.

The choke connects the PSU with the bus, thus decoupling power feeding and data signals. It also plays a major role in the creation of the data signals. As a special feature the choke has a reset switch. When this switch is operated the bus voltage drops and in result all BAU's connected to this line enter the reset state. You may use this feature to find out how your product behaves under a supply voltage break down.

### 6.1.1.6 Reference Devices

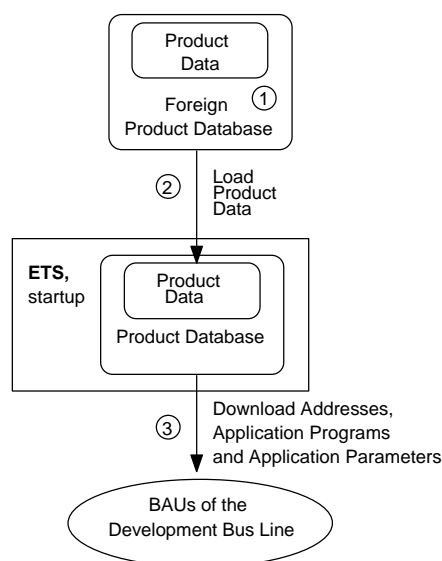
The reference devices are EIB devices already successfully developed. They may be made by your competitors or by your own company. In order to check the function of your new product (under development) they should fulfil a similar task.

Should you for instance want to develop a push button device (with status indication), suitable reference devices could be a binary input device and a binary output device (see Fig. 2/1/1-47).

### 6.1.1.7 Installation of the Development Bus Line

The development bus line has to be installed by means of the 'Commissioning/Test' part of the ETS (EIB Tool Software). To do so, you have to carry out the following steps (see Fig. 2/1/1-48):

- ① Get hold of a product database from the manufacturer of your reference devices.
- ② Import the product data into ETS database.
- ③ Assign physical addresses to the devices, select between the available application programs for that device, connect the communication objects to group addresses and set the parameters according to the desired functionality of the device. This can then be downloaded via the bus into the EEPROM of the BAU.



**Fig. 2/1/1-48: Installation of the development bus line**

## **6.2 Assembler, Cross Compilers**

### **6.2.1 Assembler**

There are no restrictions for the assembler. Any assembler that can write code for the processor in the selected BAU (in the Motorola S1 format) can be used.

### **6.2.2 ByteCraft C-compiler**

#### **6.2.2.1 The C6805 Compiler “IDE”**

This C6805-compiler can be used in 3 different ways:

1. Using the integrated development environment (IDE),
2. Using “Prompted Output”,
3. or using a lot of parameters, from the DOS-prompt

of which the IDE is most convenient.

The main menu of the IDE has only four items. Some extra menu-items could be added:

- “Read Error File” so one can locate and read the first error(s)
- “Read List File” so one can see how the C6805-compiler has compiled your source code into which machine code.
- “Read Hex-File” HEX- (S19-) files can be downloaded with BusMon or imported in the ETS+.



**Fig. 2/1/1-49: The IDE main window**

It is described in the user manual how to do this.

### 6.2.2.2 Conditions for Writing Code

#### 6.2.2.2.1 Generating compact code

In general, compact C-code results in compact assembly-code. So, rather write ++COMPACT then COMPACT = COMPACT + 1.

- `RegI = RegD & 0B1111;` 12 bytes  
`RegI = RegI >> 3;`
- `RegI = RegD >> 3;` 11 bytes  
`RegI = RegI & 0B1111;`
- `RegI = (RegD >> 3) & 0B1111;` 9 bytes
- `if (Sign=1)...` 6 bytes  
`if (Sign)` 4 bytes (Sign is 0 or 1)
- `Control = (D>T_D_H);` 19 bytes  
`if (D<T_D_H) Control=0;` 18 bytes

It is necessary to consider the data-type that is used for a constant or variable as this has serious impact on the (length of) the generated code.

- `Update = 1` 4 bytes  
`Status.1 = 1` 3 bytes (bits Status)
- `if (RegD2 < 5)` 8 bytes (RegD2 is signed char)  
`if (newCH < 5)` 6 bytes (newCH is char)

In some exceptional cases, one can include own, shorter algorithms. For e.g. a division by 8 of a two-byte value, the compiler uses a lot of code, where as 3 simple shift operations give the same result.

#### 6.2.2.2.2 Arrays and Pointers

EIBA makes include files available for code generation with the C-compiler. The use of these include files oblige EEPROM constants to be declared as arrays. This is only the case if you want to declare them as application parameter later on, or at least if you really want to have them in memory. Otherwise, just write

```
const int D = 10;
```

Pointers can be declared as follows:

```
char near * charptr;
```

Now, the following expressions are possible (useful ?):

```
charptr = &newCH;
charptr = DN (if DN is declared as an array), and even
charptr = DN + 10;
charptr = &DN[0]
```

And, of course, it's possible to pass arguments to functions via pointers. However, as the pointer as argument value is passed through the accumulator, this is less useful for small data.

Example:

```
void main(void)
{
    Standard (&Control) ; /* This takes 5 bytes */
}
```

```
void Standard(char near * paramptr)
{
    RegD = *paramptr; /* This takes 6 bytes */
}
```

☞ Stack overflow:

Take care of the available stack space when calling routines inside the application program. As the return address is stored in 2 bytes on the stack, calling a subroutine decrements the stack size. Also the system software services (see 5.2) use the stack. In general, one should not cascade more than 2 sub-routines.

### 6.2.2.3 Code Development

In the appenices of this book, a sample file DEMO.05C is included. Also the include files EIBAx.05H (x=0, 1 or 2) are listed there.

Code development can initially be done by changing the contents of the DEMO.05C according to the following steps:

▣ **STEP 1:** Change the contents of the header to your own program name and company identifications.

#pragma option f88; sets the number of lines on each page of the listing file (.LST) to 88. The default is 66. 0 means continuous listing without page headers or form feeds.

#pragma regcc cc; gives the programmer read-only access to the condition code register, using the name cc. With regac and regix, one can also access the accumulator and index register. But only cc is bitwise usable.



- ➡ **STEP 2:** Replace BAU\_VER\_11 with BAU\_VER\_10, depending on the BAU-version you are going to use.
- ➡ **STEP 3:** Change the product definition, communication oriented and hardware oriented settings to fit the needs of your application, or leave them unchanged. The description of these concepts can be found elsewhere in this Cookbook.
- ➡ **STEP 4:** Include the file <EIBA0.05H>
- This file must reside in the same directory as the C-compiler. Otherwise, the complete path has to be given:  
<D:\PROGS\C6805\PROGS\EIBA0.05H>
- This file makes some definitions for the compiler concerning the 68HC05B6 processor and the location of software services for the specified BAU-version.
- ➡ **STEP 5:** After #define MOREOBJ you enter the number of extra entries you would like to have preserved for group addresses in address table and association table.
- Principally, one has to decide on the number and type of communication objects that will be required for one's application.
- After #define obj. you can specify the name for this communication object in your program.
- E.g.: #define obj01 TempOut
- For objects with a datatype smaller than 1 byte (1 till 7 bit), you may only enter size 1. It is after the type-field that the relevant value field type is written.
- After the following entry "g<sub>xxx</sub>" you have to assign one single default group address (in hex format). Moreover, the group addresses for the successive objects shall be assigned in ascending order. This is due to the search algorithm of the BAU system software.
- Last but not least, the memory type has to be declared, as given in the example. Note that for EEPROM types, you have to give a default type.
- ➡ **STEP 6:** Include the file <EIBA1.05H>
- This file actually defines each communication object, as an array of unsigned integer. Mind that this gives some consequences for the way code is produced.
- Example: Now, TempOut[0] is the high (or only) byte of object TempOut.
- Further on in this include file, EEPROM memory is configured and size and contents of the different tables are calculated.
- ➡ **STEP 7:** Parameters and constants are now to be defined. They have to be declared as arrays. Normally, they are placed before the user program, but you have control over it with the @-operator.

**⇒ STEP 8:** Declare RAM-variables.

Filling the RAM space 0x00CE to 0x00DF is started in EIBA1.05H by allocating 4 bits per communication object for its flags. Calculation of the used RAM space is required to prevent user-variables from being defined in the area 0x00E0 to 0x00FF, which is reserved for BAU system software and stack space.

The BAU temporary registers (RegB to RegN) were renamed with more understandable names and types. These locations are cleared by the system software just before the user main program is started, thus saving memory for the user program, as you do not have to clear them every time. When selecting such a register, be aware that also system software routines that you call change them.

**⇒ STEP 9:** If you intend to use telegram rate limitation, the system software requires this parameter to be set just before the user code (mostly `usr_init`).**⇒ STEP 10:** Write the code for `usr_init` and `usr_save`. Even if you don't put any statements here, declare these procedures anyhow, because EIBA2.05H makes use of them. Moreover, the system software expects at least a RTS.**⇒ STEP 11:** If you use the user timer with the functions `U_SetTMx` and `U_GetTMx`, the pointer to the pointer (!) to the EEPROM description block is fetched from this byte, which is directly before the user main program.**⇒ STEP 12:** Now, the main program must follow.

As it is used in EIBA2.05H, it has to be named "main". The compiler ends it with RTS, as demanded for EIB-applications in BAU's.

**⇒ STEP 13:** Include the file <EIBA2.05H>

This file generates the association table, communication object table and BAU constants and fills up the unused EEPROM space with the opcode for a software interrupt SWI (83H), which leads to a high probability of resets in case of errors.

Save your source code when finished and leave your text-editor. Back in the IDE, move to "Assemble / Compile" and click to start compilation. Few seconds later, the number of errors and warnings is shown. Press a key to return to the main menu.

After a program has successfully been compiled, you can test it by downloading it in the BAU (with the intended mask-version). The objects are assigned to the default group addresses that you have written at their declaration.

## **6.3 EIB Interoperability Test Tool (EITT)**

The EITT EIB Interoperability Test Tool, or Testtool, is a program supporting testing of EIB-devices and preparation for certification. It is available to EIBA member companies and test labs only.

First, the Bus Device Under test (BDUT) shall be projected and installed with the ETS+ in a small test set-up. The device information, such as the physical address, the group addresses for the objects and the object types can then be retrieved by EITT from this ETS-project via the command “ETS Import”.

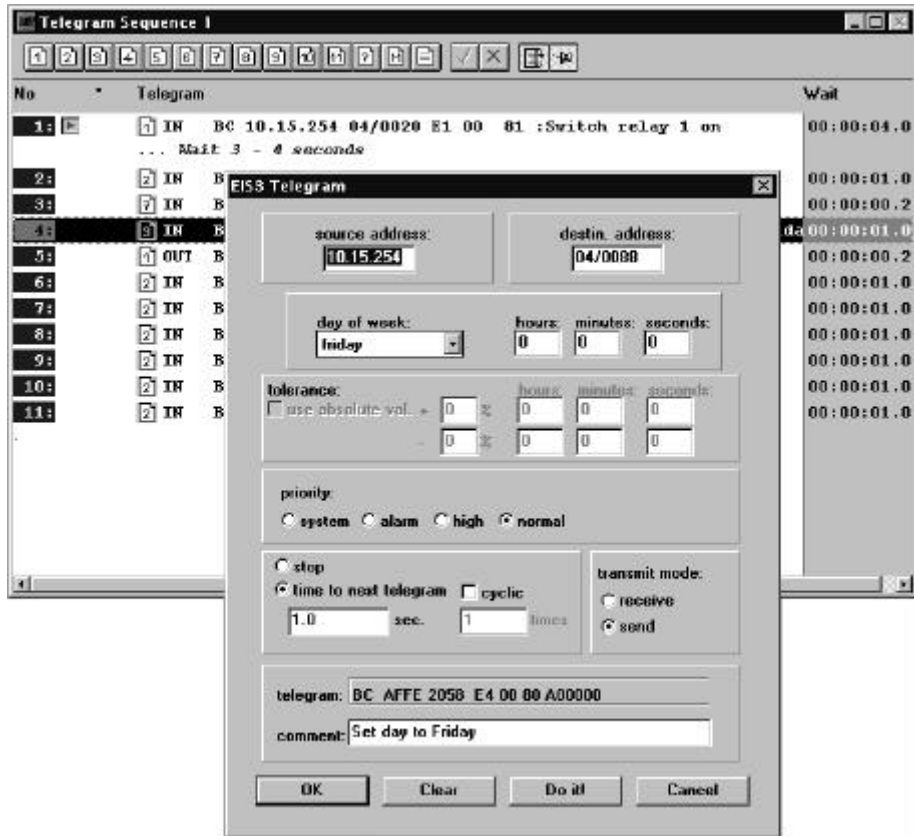
The EITT works with its own projects. These projects contain the device details and all sent and received telegram sequences that can be saved to disk as separate files.

### **6.3.1 Sending Telegrams**

The EITT allows the user to compile telegram sequences. Such sequences may contain self compiled telegrams to be sent on the bus, as well as telegrams that are to be expected, e.g. from the BDUT. It is also possible to copy one or more telegrams and alter existing telegrams by means of the “Replace command”. This is convenient e.g. when writing lots of telegrams that differ only in small details.

The transmission and reception process can be controlled with the Play and Stop button in the main window and the check boxes in the Telegram Sequence Window.

When creating telegrams, the user can select from the available EIS-formats, or may as well write non-EIS telegrams and comments.



**Fig. 2/1/1-50: Telegram Sequence and Telegram Detail**

In the EISx Telegram Window, the source and destination address and the priority can be set. The actual data can be entered very easily according to the selected EIS standard. The resulting telegram is displayed too.

For each telegram, the time to the next telegram to be sent or received can be entered.

### 6.3.2 Receiving Telegrams

The Trace Buffer window lists all telegrams that ran on the bus, both telegrams sent by EITT as well all other telegrams. The telegrams are preceded by the direction (sent = “<” / received = “>”), the indication whether the telegram passed a test or not and the time of reception. It is possible to define a filter for displaying the received telegrams.

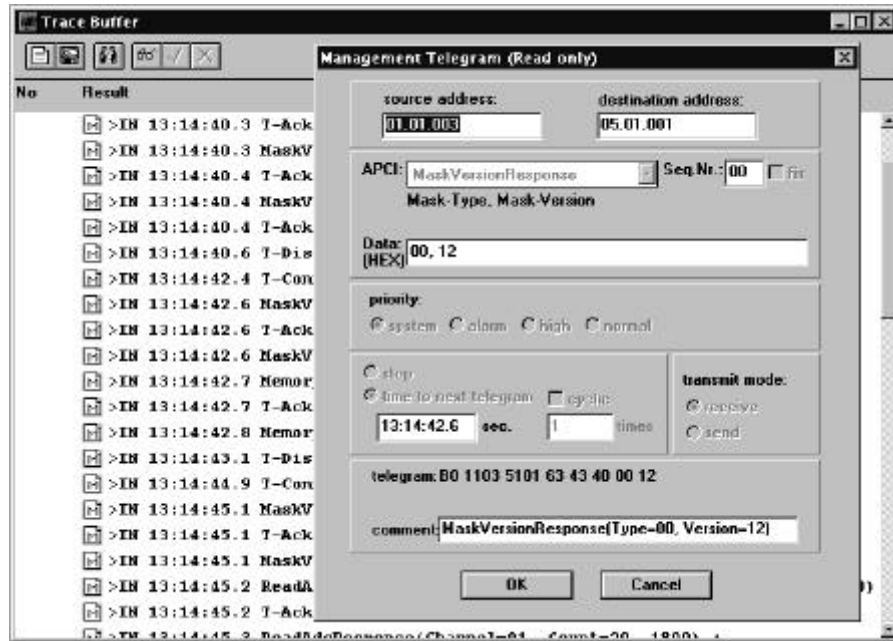


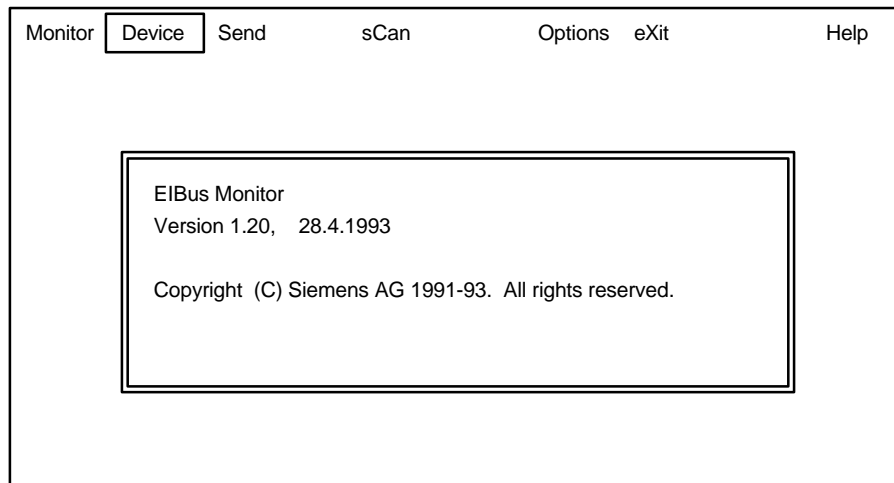
Fig. 2/1/1-51: Trace Buffer and Telegram Detail

It is possible to analyse the recorded telegrams. The Show Telegram - Value window shows source and destination address, the telegram priority and the APCI-field and the value.

## 6.4 Busmon

The Busmonitor software is mainly used to download new APs into BAU's in order to check their function. The Busmonitor is a DOS program. Its application requires profound comprehension of the EIB system. The following three modes (of operation) are supported:

1. The *device mode*,
2. the *monitor mode* and
3. the *send mode*.



**Fig. 2/1/1-52: Main Menu of the Busmonitor**

### **6.4.1 Device Mode**

The Device Mode summarises the following functions:

- Loading a physical address into a selected BAU. The BAU is selected by pushing its program button.
- Reading the physical address from a selected BAU.
- Download of an AP into a BAU. The AP must be available on your hard disk or on a diskette in S19 format.
- Upload of the AP of a BAU. The AP will be stored in S19 format on your hard disk or on a diskette.
- Display the BAU's memory.
- Modification of memory locations in the EEPROM or the RAM area of the BAU.
- Check-up of the state of a BAU.
- Reset of the BAU's runtime error flags.
- Reading the input values of the A/D converter.
- Switching the BAU's program LED on or off, respectively, in order to locate a BAU within a building by means of its physical address.
- Listing of all BAU's (detected physical addresses) of a line.
- Listing of all BAU's and couplers of an EIB system.
- Entry of group addresses into the filter tables of couplers.

### **6.4.2 Monitor Mode**

The Busmonitor is able to keep a record of all telegrams transmitted via the line to which your PC is connected. In monitor mode the Busmonitor may

- display the telegrams on your monitor,
- write them into a file or
- print them on your printer..

You may thus apply the monitor mode in order to check the function of (sensor) devices.

### **6.4.3 Send Mode**

In send mode you may generate and send different telegrams onto the bus. This feature may be applied in order to check the function of actuator devices. Besides this, you may check the function of sensor devices that do not send their group telegrams by themselves. In this case a data request group telegram has to be sent. Following a read request the sensor device sends a response group telegram which is displayed on your monitor.

## **6.5 Emulators and Simulators**

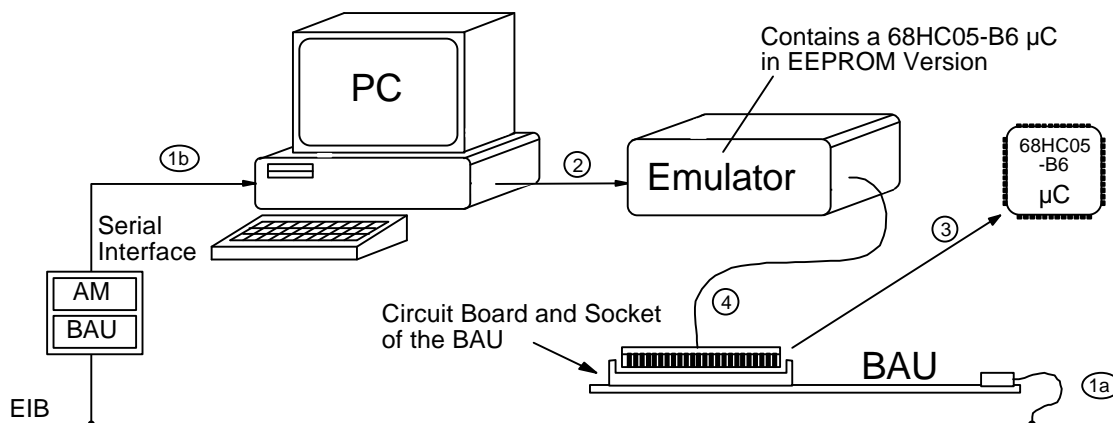
In the previous chapter we have described the Busmonitor, being a simple and valuable AP development tool. Especially for the development of simple APs it is most suitable. However, with increasing complexity of the AP the Busmonitor becomes less efficient. In this case the use of an *in circuit emulator* (or shortly emulator) is strongly recommended.

An emulator is a development tool that can completely reproduce all logical functions of a microcontroller. As an advantage the emulator is able to trace and display various internal and external signals of the microcontroller. Especially for debugging of the AP this ability results in a considerable time saving effect.

A suitable emulator realises at least the following features:

- Single step operation of the AP. This is realised by an interruption of the AP emulation after each instruction. While the microcontroller is stopped, you may take a look at the
  - contents of Registers and memory locations and
  - the state of the ports.

- Running the AP until a certain break condition is fulfilled. Some examples for break conditions are:
  - The program counter reaches a certain address.
  - Appearance of a certain value in a certain memory location.
  - Appearance of a certain bit pattern at a port, generated either by the microcontroller or by the external hardware (e.g. the AM).
  - Combinations of the conditions listed above.
- Storage of any (internal and external) signals into the additional memory of the emulator while the AP is running (trace mode). The signals recorded may be displayed independently of the AP execution.
- Measurement of the runtime of the AP.



**Fig. 2/1/1-53: Application of an Emulator for AP Development**

In order to use an emulator for EIB AP development you have to carry out the following steps (Fig. 2/1/1-53):

- ①<sub>a,b</sub> Upload of the system software from the BAU. For this purpose you may use the Busmonitor.
  - ② Download of the system software into the microcontroller of your emulator. For this purpose the internal memory of the emulator's microcontroller has to be EEPROM.
  - ③ Opening of your BAU's cabinet and removal of the original BAU's microcontroller.
  - ④ Connection of the emulator to the BAU via the microcontroller socket. Therefore the manufacturer of the emulator provides a plug matching the socket.
- ☞ In recent BAU's the microcontroller is soldered immediately to the BAU's circuit boards without a socket. In this case you have to secure a reliable connection between the emulator and the BAU. This task may be sophisticated.



If the manufacturer of your emulator does not provide a 68HC05 microcontroller with an EEPROM, you may equally insert the original microcontroller from the BAU into the emulator.

## **6.6 ETS, ETS+, IDE**

### **6.6.1 ETS (EIB Tool Software)**

The **ETS** is a windows software, mainly used by the electrician for projecting and for installation of EIB systems. Although the ETS does not directly support the product development, you will need it for both of the following development steps:

- Installation of your development bus line, in particular of the reference devices, and
- experimental installation of your new products before you start the certification process.

ETS consists of several modules, as shown in Fig. 2/1/1-54.



**Fig. 2/1/1-54: The ETS main window**

#### **6.6.1.1 Module Project Design**

The 'Project Design' module is not absolutely necessary for the product development.

By means of this module, the electrician plans EIB projects, using EIB devices imported into his database from databases provided to him by the manufacturers. Multiple views allow the electrician to structure his design in many ways.

In the topology view, the electrician puts devices in the lines, thus giving physical addresses, and assigns common group addresses to the device's objects, this way defining their working together. He also chooses every device's parameters, for setting its behaviour.

Devices can be grouped logically in functions in the 'Function View' and their location in the building is found in the 'Building View'.

### 6.6.1.2 Module Commissioning & Test

The 'Commissioning & Test' module allows the electrician to install this EIB project. He can do this locally, but may also have to export the project from the design PC onto floppies, in order to re-import it e.g. onto a laptop used on the building-site.

Small changes to the project can still be done. The physical addresses are written and the application programs are then downloaded.

Some diagnostic functions are available as well. The electrician may check the presence of devices in the installation, transmit and read telegrams from the bus, read group values or read memory contents of devices.



**Fig. 2/1/1-55: Selection of the Company Name**

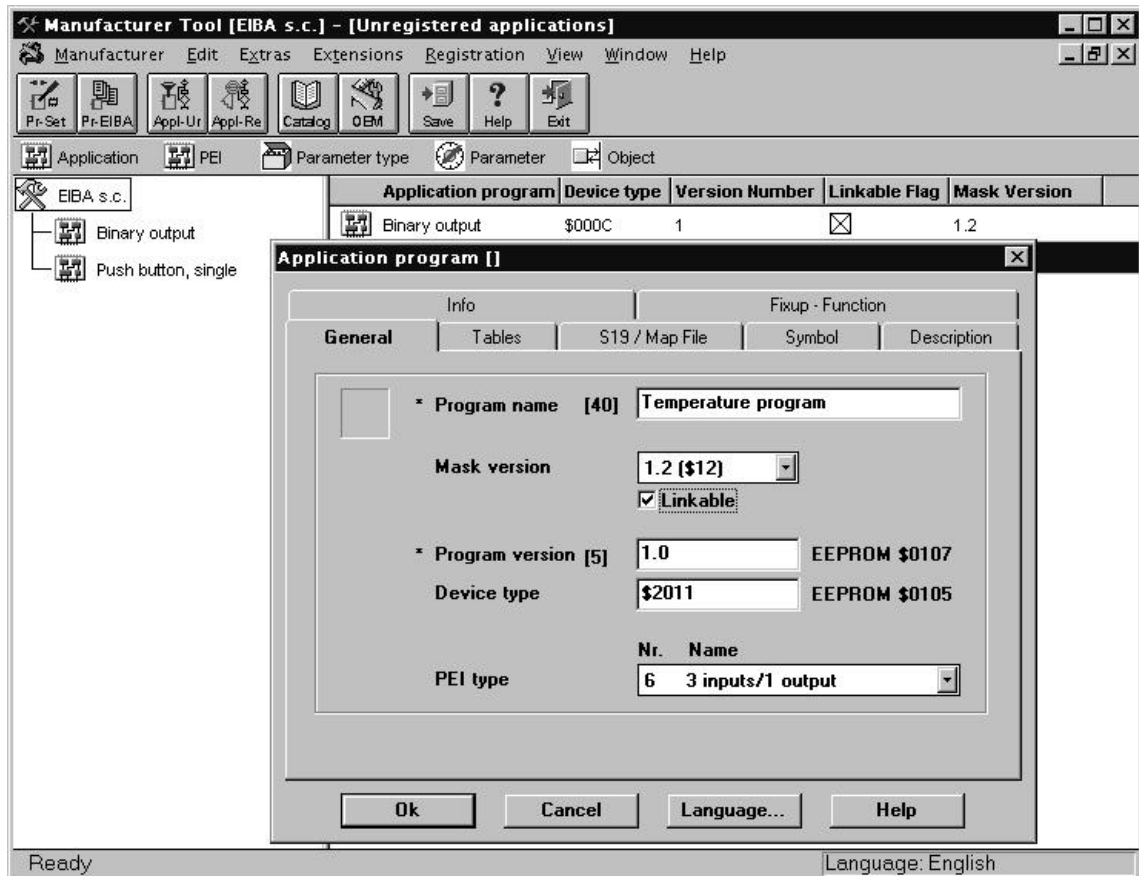
### 6.6.2 Device Definition with the ETS2 ManufTool

The following description is only intended to present a general working procedure for entering application- and product descriptions in the ETS2 databases. Please refer to the tool's manuals for a complete and more detailed reading.

The ETS2 manufacturer module is started by single clicking the 'ManTool'-button in the ETS2 main Window Fig. 2/1/1-54. The manufacturer tool main window is opened, and the user is asked to select his company, as displayed in Fig. 2/1/1-55.

#### 6.6.2.1 Description of the Application Program

Clicking the 'Appl-Ur'-button opens the 'Unregistered Applications' window (Fig. 2/1/1-56).



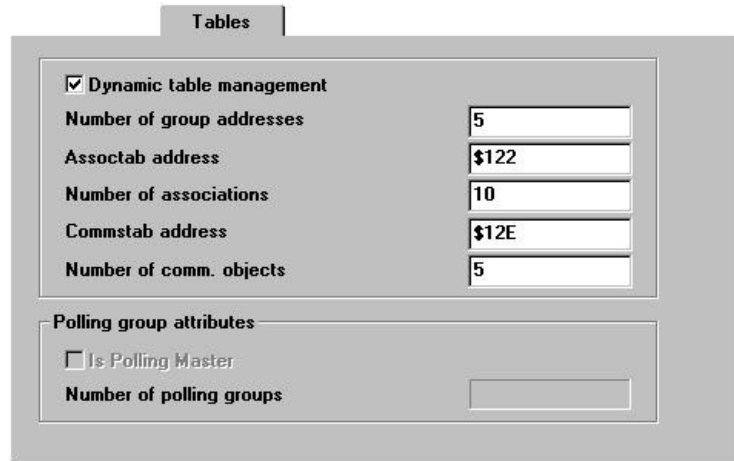
**Fig. 2/1/1-56: Describing a new Application**

### *Reading the Application*

To create a new application, drag the 'Application'-symbol from the toolbar to the manufacturer icon. As a result, the ETS 2 opens the 'Application Program' detail window (Fig. 2/1/1-56). Information on the application program, grouped together on several tab pages, can be entered here.

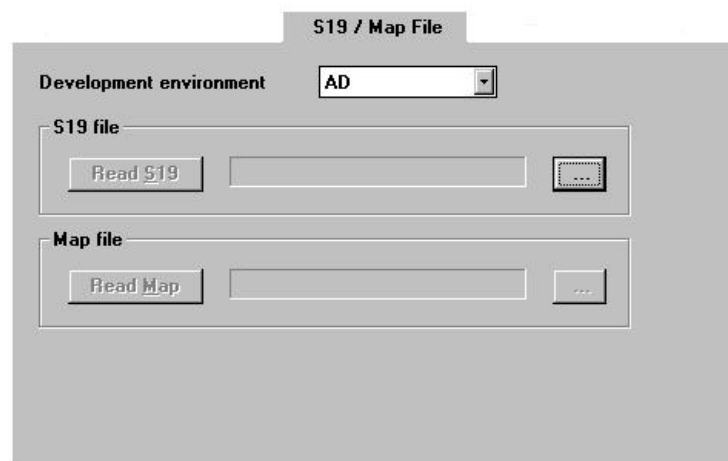
On the 'General'-page, the name of the application program has to be entered. Mask version, device type and PEI-type must be selected.

On the 'Tables'-page, the (maximum) number of group addresses, associations and communication objects, as well as the location of the association table and the communication object table must be filled in.



**Fig. 2/1/1-57: Definition of Tables**

☞ Checking 'Dynamic Table Management' will allow the ETS2 to automatically calculate the size of association and address table. This could be useful if the user e.g. makes a lot of associations to one object, but none to the other. Unused address table space can then be used for extra associations.



**Fig. 2/1/1-58: Reading the S19- and Map file**

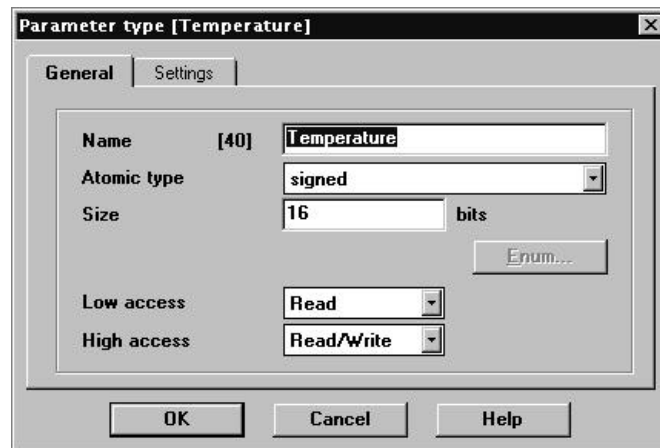
On the 'S19 / Map File'-page, you can read in your assembled application program (in Motorola-format) and its Map File.

### ***Defining Application Parameters***

When the program detail information is filled in, the parameters for this program may be defined. For this, the parameter types have to be created first.

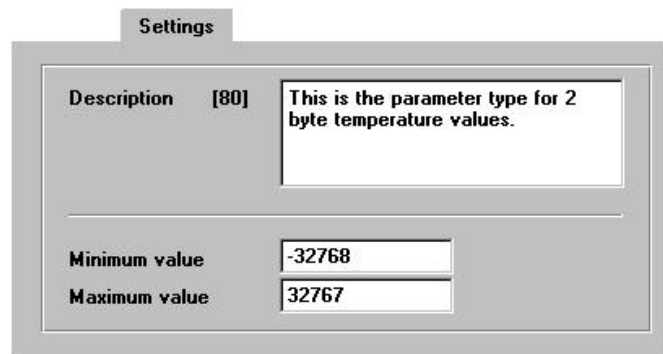
This can be done by dragging the 'Parameter Type'-icon from the toolbar and dropping it onto the application, but one may as well copy an existing parameter type from a previous application. ETS 2 will now ask to define the parameter. This is shown in Fig. 2/1/1-59.

On the 'General'-page you fill in the name of this parameter type. You select the 'Atomic Type' from the drop down box (enum, floating point, signed, string, ...) and pre-set the low and high access rights for parameters of this type.



**Fig. 2/1/1-59: Parameter Type Definition Window**

On the 'Settings'-page, you can fill in a description for this parameter type, for your own information. You may also alter the allowed range for the parameter type. ETS2 bases its proposal on the Atomic Type and the Size.



**Fig. 2/1/1-60: The Parameter Type Settings**

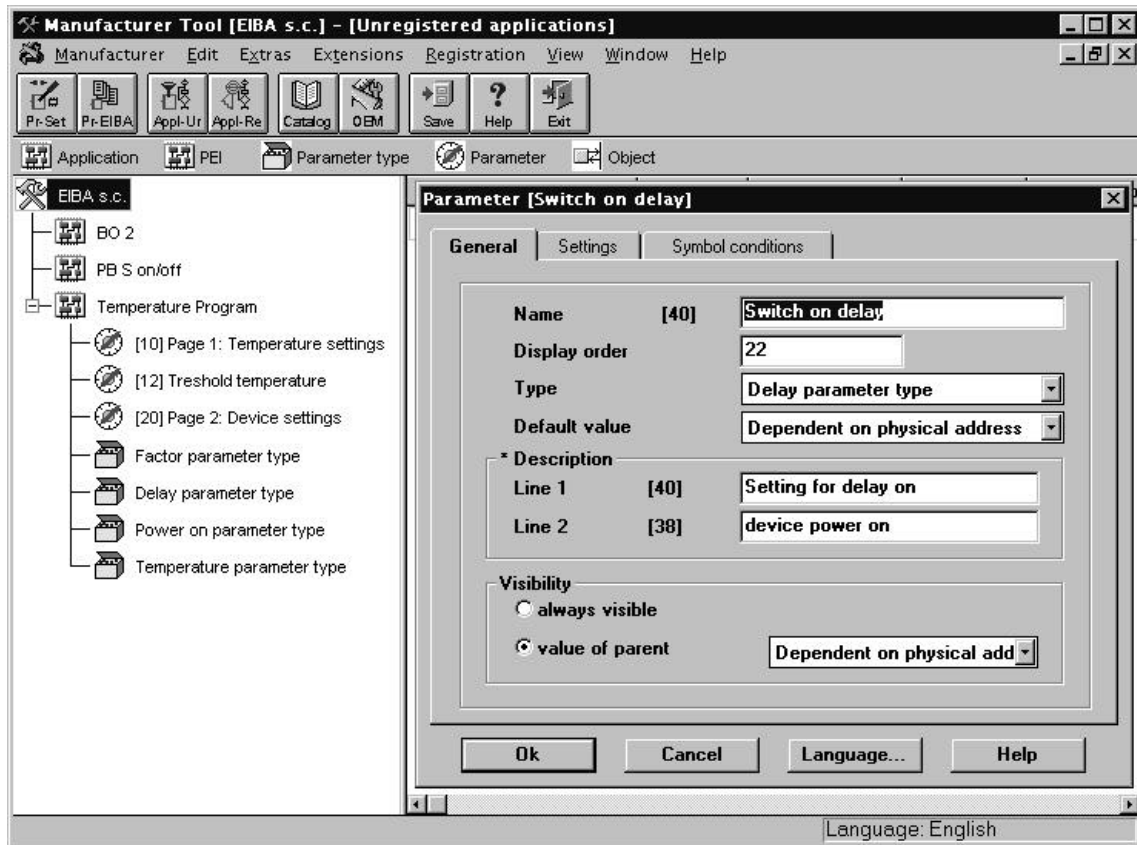
We can now start the definition of the parameters as such. See Fig. 2/1/1-61.

All parameters are added by drag & drop from the toolbar.

- ➡ If a parameter is dropped onto the application icon, it will be a normal parameter.
- ➡ If a parameter is dropped on another parameter, than we can make the visibility of this parameter depend on the underlying parameter. This parameter is a 'Child Parameter'; the underlying parameter is the 'Parent Parameter'. This allows the application programmer to hide parameters that are only relevant for certain values of other parameters, or that are to be interpreted differently depending on the value of another parameter.
- ➡ If a parameter is not given an address, a 'Dummy Parameter', the contents of its 'Description'-field will be used as title of a new tab-page.

- If a parameter is given the address 0, it does not relate to a location in the BAU's memory, but can be used for structuring the parameters.

The order of the parameters, and in this way their relative dependency and grouping on tab-pages is set by means of the 'Display order'-field.



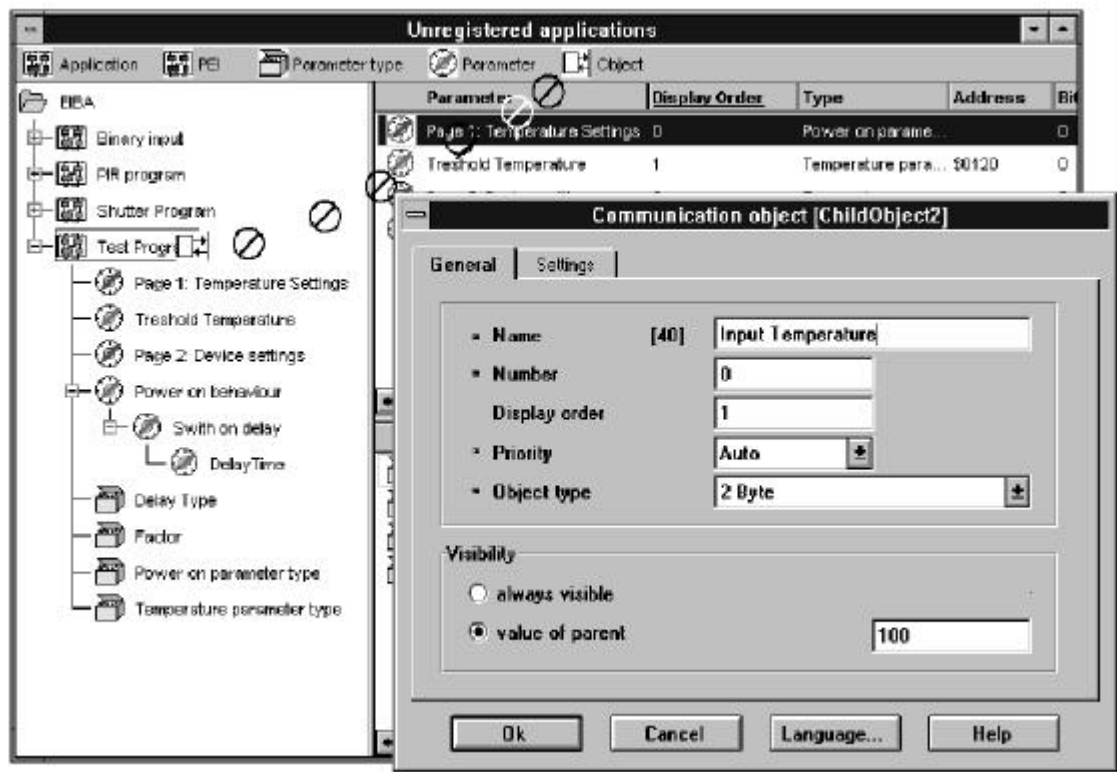
**Fig. 2/1/1-61: Definition of a new Parameter**

- ✎ In this way, the parameter 'Switch on delay' is set as a child parameter to 'Power on behaviour' parameter, since the delay time is irrelevant when 'No action' is selected for 'Switch on behaviour'. In the same way, 'DelayTime' is a made a child parameter to this parameter.

### *Defining Communication Objects for the Application Program*

We can also assign communication objects to our application program, see Fig. 2/1/1-62.

This is done by dragging the 'Object' symbol from the toolbar and dropping it on the program icon. The 'Communication object' detail window will open and you can define the communication object. On the 'Settings'-page, the function, description and default flags can be filled in.



**Fig. 2/1/1-62: Object Insertion and Definition**

Just like child parameters, communication objects can be made dependent of parent parameters. As the chosen value of the parameter defines the visibility of an object, the dimming object of an application can be hidden when the user should select only the on-off function of the application. In the 'Settings'-page, the dependency of the parameter can be set (See Fig. 2/1/1-63).

- ✎ The example program can either process fetch the time by reception when receiving it via the bus, or it can read it from a clock, connected to the device. This source can be selected by the 'Time retrieval-parameters, whose value is 0 if the user selects 'Local (Connected Clock)' or 1 if the user selects 'Via EIB (External Clock)'. Only in this second case, the 'Object for time polling' will be shown.

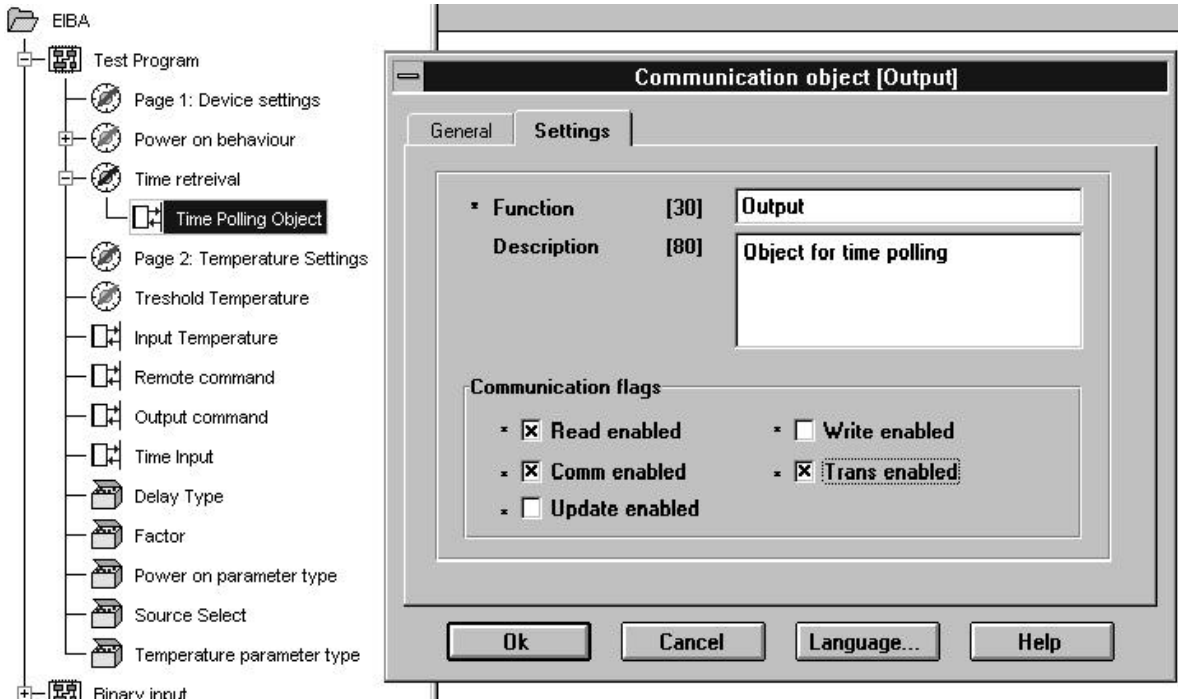


Fig. 2/1/1-63: Parameter dependent Objects

### 6.6.2.2 Assignment of AP to Products

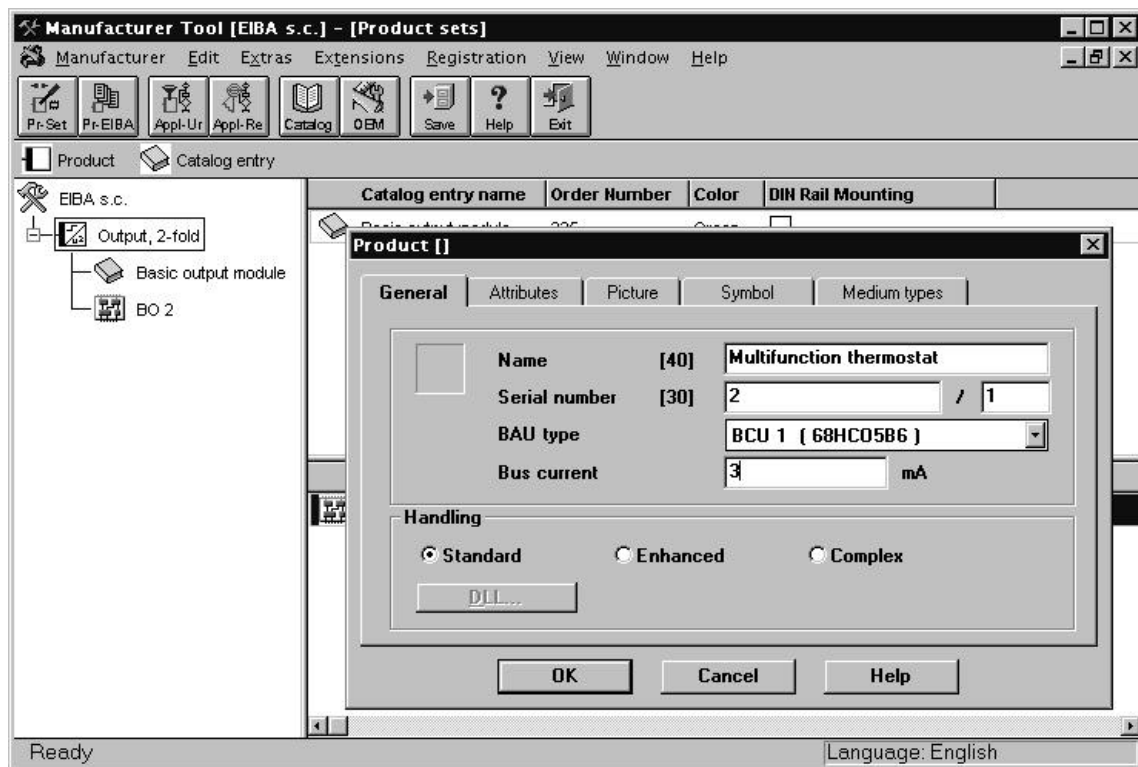
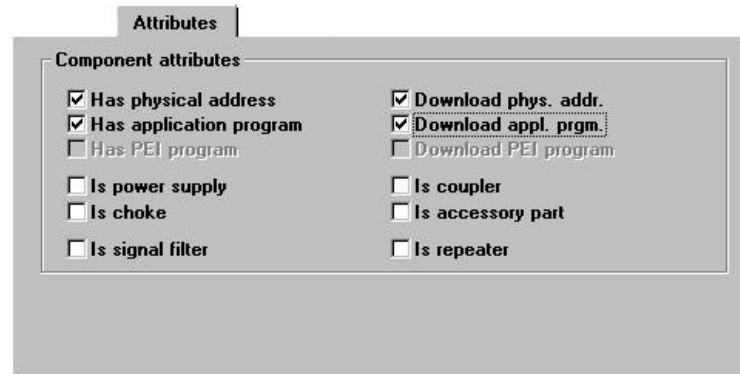


Fig. 2/1/1-64: Definition of the new Product



First of all, a new product has to be described. For this purpose, the 'Product Set'-window is opened by single-clicking the 'Pr-Set' button. The 'Device'-symbol dragged from the toolbar and dropped onto the manufacturer icon. As a result, the 'Product' detail-window opens, in which the description of the new product can be entered. The 'Attributes' page (Fig. 2/1/1-65) lets you define the components and desired actions of the product.



The screenshot shows a window titled 'Attributes' with a tab labeled 'Attributes'. Inside the window, there is a section titled 'Component attributes' containing two columns of checkboxes. The first column includes 'Has physical address' (checked), 'Has application program' (checked), 'Has PEI program' (unchecked), 'Is power supply' (unchecked), 'Is choke' (unchecked), and 'Is signal filter' (unchecked). The second column includes 'Download phys. addr.' (checked), 'Download appl. prgm.' (checked), 'Download PEI program' (unchecked), 'Is coupler' (unchecked), 'Is accessory part' (unchecked), and 'Is repeater' (unchecked).

**Fig. 2/1/1-65: Product, 'Attributes'**

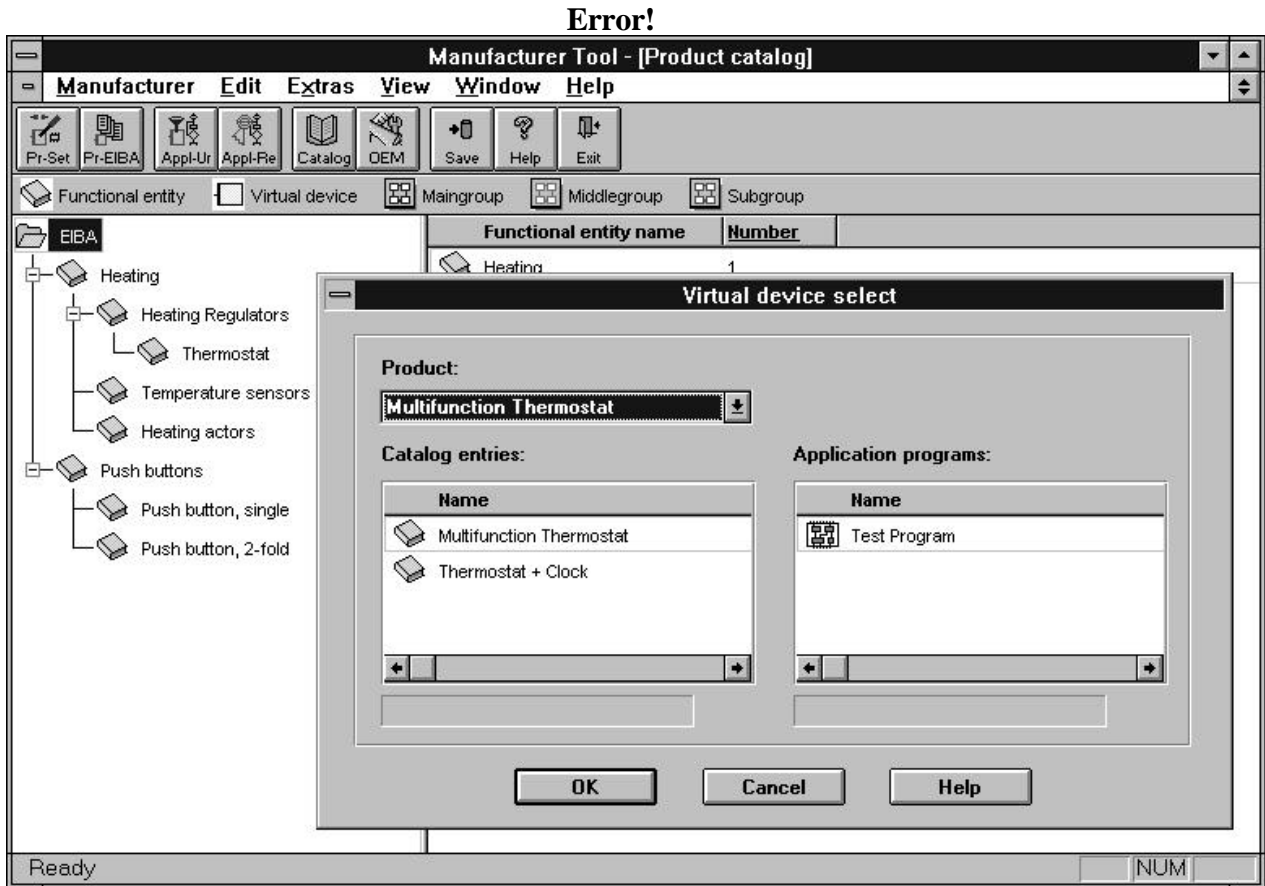
To assign an application to this product, we also open the 'Unregistered Application' window, from which one or more applications can be selected, dragged and dropped onto the product symbol in the 'Product Sets'-window.

One or more catalog entries for a product may be described, by dragging the 'Catalog entry' symbol from the toolbar and dropping it on the product symbol. When making up the 'Manufacturer Catalog' later on, it will be these catalog entries that one refers to.

## 6.6.2.3 Adding the Product to the Manufacturer Catalog

The manufacturer's catalog is hidden under the 'Catalog'-button. A manufacturer may structure his catalog by means of 'Functional Entities', in several levels. A device, with an application program, is put in this catalog by dragging the 'Virtual Device' symbol from the toolbar, dropping it onto the desired functional entity and selecting product and application from the 'Virtual Device'-window that pops up (Fig. 2/1/1-66).

When the 'OK'-button is pressed, the virtual device detail window opens (Fig. 2/1/1-67).



**Fig. 2/1/1-66: Insertion of a Virtual Device**

On the first page the 'Product Family' and 'Product Type' field must be selected from the drop down boxes. These refer to the upper 2 levels of the previously made functional entities in the product catalog. The end-user must select the device according to this structure as well.

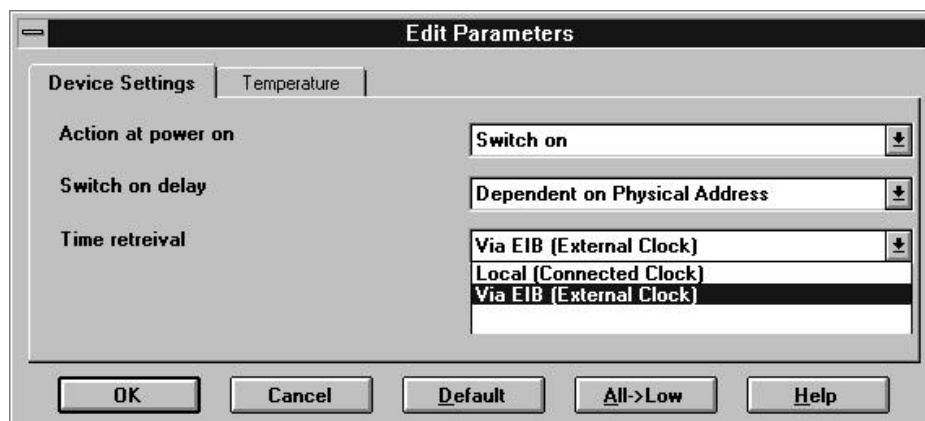


**Fig. 2/1/1-67: Virtual Device, 'General'**



**Fig. 2/1/1-68: Virtual Device, 'Extras'**

On the second page (Fig. 2/1/1-68) the catalog entry and the application program may be altered. The 'Read Application'-button serves for refreshing the application image, should one have corrected something in the application in the 'Unregistered Applications'-window.



**Fig. 2/1/1-69: Preview of the Parameters**

Useful as well in the 'Device Detail'-window is the 'Parameter'-button. Clicking this button will show the parameter tab-pages as they will be seen by the end-user (Fig. 2/1/1-69). If something has changed to your parameter structure of the application program, e.g. when new parameters are added or their structuring has been modified, the result can be seen by pressing 'Read Application' and 'Parameters' consecutively.

#### 6.6.2.4 Experimental Installation of the Product

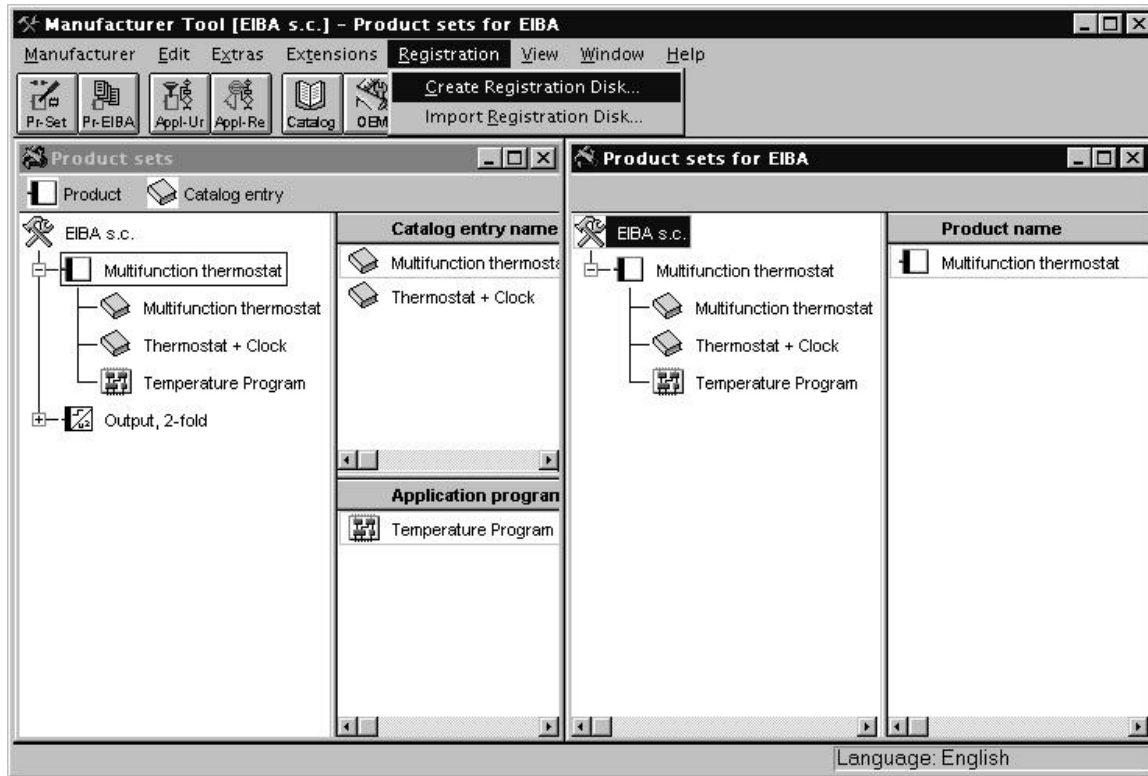
The product and its application have now been completely described. As a manufacturer, you can use it in a test project in the same database.

- ☞ For this purpose, the 'Single Window' approach is useful. You only use the ETS2 'Building View' and do not create any building structure. Just insert the BDUT (Bus Device Under Test) by dropping the Product-icon on the project icon.

### 6.6.3 Product Certification

EIBA requires manufacturers to have their EIB products certified. The software part of this procedure consists of an export/import procedure of the product description with ETS+ from/to the manufacturer's database.

This is started by copying (over drag & drop) the desired product(s) from the "Product sets"-window to the "Product sets for EIBA"-window.



**Fig. 2/1/1-70: Preparing Products for Export**

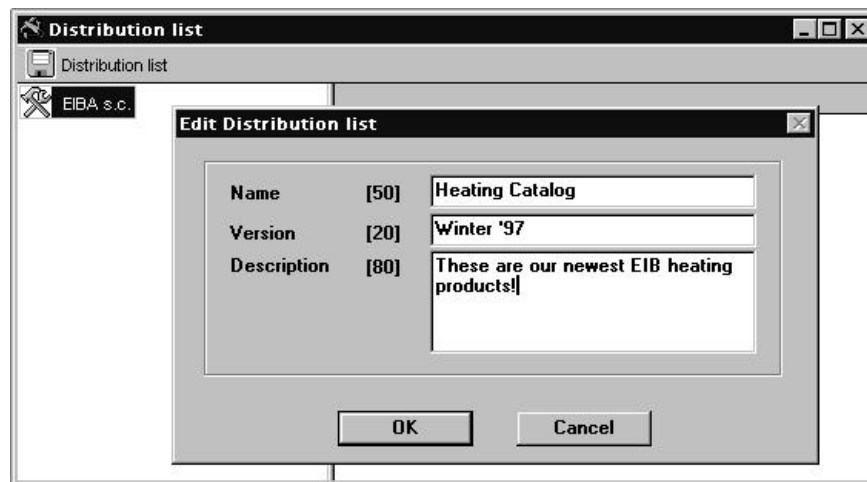
The next step consists of creating an export file. This can be achieved by means of the menu-items "Registration"- "Create Registration" disk. The registration relevant information of your product is saved in a file. Together with the documents for product registration (see Volume 5 of the EIB Handbook Series), this file is sent to the EIBA Certification Department for registration.

When this file returns from EIBA, it can be imported by means of the command "Import Registration Disk". The product is now ready for distribution to the end-user.

#### 6.6.4 Product Distribution

Via the menu “Manufacturer” and the item “Distribution list”, the Distribution list window is opened. Distribution lists are files that contain (part of) the products catalog and can be imported by the end-user.

To create a distribution list, the “Distribution list”-icon is dragged from the toolbar and dropped on the manufacturer name. After the attributes have been defined for it (see Fig. 2/1/1-71), you may add products to this distribution list by dragging them from the “product Catalog “-view and dropping them on the distribution list icon. When dropping a functional entity, all attached products will be added to the distribution list.



**Fig. 2/1/1-71: Definition of a "Distribution list"**

Once the desired products have been selected, the distribution list may be exported by selecting “Export” from the “Edit”-menu or from the toolbar. The distribution list is saved as a file with the extension “vd1”. These files can be imported by the ETS end-user for use of your product in real EIB installations.

## 7 Appendices

### 7.1 Appendix A: Header Files for ByteCraft C-Compiler

#### 7.1.1 eiba0.05h

```
#define uchar unsigned char
#define uint unsigned int

/*----- */
/*   Register assignments for the 68HC805B6 microcontroller   */
/*----- */

#pragma vector __RESET          @ 0x1ffe ;
#pragma has MUL ;

/*----- */
/*   BCU FIRMWARE DECLARATIONS                               */
/*----- */

uchar    RegB          @ 0x50;
uchar    RegC          @ 0x51;
uchar    RegD          @ 0x52;
uchar    RegE          @ 0x53;
uchar    RegF          @ 0x54;
uchar    RegG          @ 0x55;
uchar    RegH          @ 0x56;
uchar    RegI          @ 0x57;
uchar    RegJ          @ 0x58;
uchar    RegK          @ 0x59;
uchar    RegL          @ 0x5A;
uchar    RegM          @ 0x5B;
uchar    RegN          @ 0x5C;
uchar    RegRes[ 113 ] @ 0x5D;

#pragma option -l
#ifdef BCU_VER_10
#pragma option +l
void      U_FlagsGet( uchar com_obj_num )      @ 0x0c8c;
void      U_FlagsSet( uchar com_obj_num )      @ 0x0c94;
void      U_TransRequest( uchar com_obj_num )  @ 0x0d91;
void      EEWrite( uchar address, uchar value ) @ 0x0c2d;
void      EESetChecksum( void )                @ 0x0c5d;
uchar     U_Debounce( uchar deb_time, uchar deb_value ) @ 0x0c64;
void      U_DeIMsgs( void )                    @ 0x0c82;
void      U_Map( registerx tab_ptr )           @ 0x0c9b;
void      U_ReadAD( uchar num_reads, uchar port_num ) @ 0x0d35;
```

```

void      U_ioAST( uchar flags )           @ 0x0da7;
void      S_AstShift( registerx ptr_blk )  @ 0x1103;
void      S_LAstShift( registerx ptr_blk ) @ 0x1101;
void      TM_Load( uchar time, uchar tim_mode ) @ 0x0e0c;
uchar     TM_GetFlg( uchar tim_num )       @ 0x0e2a;
void      U_SetTM( uchar ptr_blk, uchar tim_num ) @ 0x0d8a;
void      U_GetTM( uchar ptr_blk, uchar tim_num ) @ 0x0d4d;
void      U_Delay( uchar delay )           @ 0x0ddb;
void      AllocBuf( void )                 @ 0x116a;
void      FreeBuf( registerx ptr_buffer )   @ 0x118c;
void      PopBuf( uchar msg_buf_type )     @ 0x11ac;
void      MultDE_FG( void )                @ 0x0b3c;
void      DivDE_BC( void )                 @ 0x0afc;
uchar     Shift4Left( registera accu )     @ 0x0b9a;
uchar     Shift5Left( registera accu )     @ 0x0b99;
uchar     Shift6Left( registera accu )     @ 0x0b98;
uchar     Shift7Left( registera accu )     @ 0x0b97;
uchar     Shift4Right( registera accu )    @ 0x0bda;
uchar     Shift5Right( registera accu )    @ 0x0bd9;
uchar     Shift6Right( registera accu )    @ 0x0bd8;
uchar     Shift7Right( registera accu )    @ 0x0bd7;
void      U_SetBitH( uchar bit_num )       @ 0x0df9;
void      U_GetBitH( uchar bit_num )       @ 0x0ded;
#pragma option -l
#endif

#ifdef BCU_VER_11
#pragma option +l
void      U_FlagsGet( uchar com_obj_num )   @ 0x0c9d;
void      U_FlagsSet( uchar com_obj_num )   @ 0x0cb3;
void      U_testObj( uchar com_obj_num )    @ 0x0CA5;
void      U_TransRequest( uchar com_obj_num ) @ 0x0db9;
void      EEWrite( uchar address, uchar value ) @ 0x0c38;
void      EESetChecksum( void )              @ 0x0c68;
uchar     U_Debounce( uchar deb_time, uchar deb_value ) @ 0x0c75;
uchar     U_Deb10( uchar deb_value )        @ 0x0c73;
uchar     U_Deb30( uchar deb_value )        @ 0x0c6f;
void      U_DelMsgs( void )                  @ 0x0c93;
void      U_Map( registerx tab_ptr )         @ 0x0cba;
void      U_ReadAD( uchar num_reads, uchar port_num ) @ 0x0d54;
void      U_ioAST( uchar flags )            @ 0x0dcf;
void      S_AstShift( registerx ptr_blk )    @ 0x1117;
void      S_LAstShift( registerx ptr_blk )    @ 0x1115;
void      TM_Load( uchar time, uchar tim_mode ) @ 0x0e2b;
void      TM_GetFlg( uchar tim_num )         @ 0x0e49;
void      U_SetTM( uchar ptr_blk, uchar tim_num ) @ 0x0db3;
void      U_SetTMx( uchar tim_num )          @ 0x0daf;
void      U_GetTM( uchar ptr_blk, uchar tim_num ) @ 0x0d71;
void      U_GetTMx( uchar tim_num )          @ 0x0d6c;
void      U_Delay( uchar delay )             @ 0x0dfa;
void      AllocBuf( void )                   @ 0x117e;
void      FreeBuf( registerx ptr_buffer )     @ 0x11a0;
void      PopBuf( uchar msg_buf_type )       @ 0x11c0;
void      MultDE_FG( void )                  @ 0x0b4b;
void      DivDE_BC( void )                   @ 0x0b0b;
uchar     Shift4Left( registera accu )       @ 0x0ba9;
uchar     Shift5Left( registera accu )       @ 0x0ba8;
uchar     Shift6Left( registera accu )       @ 0x0ba7;
uchar     Shift7Left( registera accu )       @ 0x0ba6;

```

```
uchar      Shift4Right( registera accu )      @ 0x0be9;
uchar      Shift5Right( registera accu )      @ 0x0be8;
uchar      Shift6Right( registera accu )      @ 0x0be7;
uchar      Shift7Right( registera accu )      @ 0x0be6;
void       U_SetBitH( uchar bit_num )         @ 0x0e18;
void       U_GetBitH( uchar bit_num )         @ 0x0e0c;
#endif
#pragma option +l

#ifdef BCU_VER_12
#pragma option +l
void       U_FlagsGet( uchar com_obj_num )    @ 0x0c9d;
void       U_FlagsSet( uchar com_obj_num )    @ 0x0cb3;
void       U_testObj( uchar com_obj_num )     @ 0x0CA5;
void       U_TransRequest( uchar com_obj_num ) @ 0x0db9;
void       EEWrite( uchar address, uchar value ) @ 0x0c38;
void       EESetChecksum( void )              @ 0x0c68;
uchar      U_Debounce( uchar deb_time, uchar deb_value ) @ 0x0c75;
uchar      U_Deb10( uchar deb_value )         @ 0x0c73;
uchar      U_Deb30( uchar deb_value )         @ 0x0c6f;
void       U_DelMsgs( void )                  @ 0x0c93;
void       U_Map( registerx tab_ptr )         @ 0x0cba;
void       U_ReadAD( uchar num_reads, uchar port_num ) @ 0x0d54;
void       U_ioAST( uchar flags )             @ 0x0dcf;
void       S_AstShift( registerx ptr_blk )    @ 0x1117;
void       S_LAstShift( registerx ptr_blk )   @ 0x1115;
void       U_SerialShift                     @ 0x0C90;
void       TM_Load( uchar time, uchar tim_mode ) @ 0x0e2b;
void       TM_GetFlg( uchar tim_num )         @ 0x0e49;
void       U_SetTM( uchar ptr_blk, uchar tim_num ) @ 0x0db3;
void       U_SetTMx( uchar tim_num )          @ 0x0daf;
void       U_GetTM( uchar ptr_blk, uchar tim_num ) @ 0x0d71;
void       U_GetTMx( uchar tim_num )          @ 0x0d6c;
void       U_Delay( uchar delay )             @ 0x0dfa;
void       AllocBuf( void )                   @ 0x117e;
void       FreeBuf( registerx ptr_buffer )    @ 0x11a0;
void       PopBuf( uchar msg_buf_type )       @ 0x11c0;
void       MultDE_FG( void )                  @ 0x0b4b;
void       DivDE_BC( void )                   @ 0x0b0b;
uchar      Shift4Left( registera accu )       @ 0x0ba9;
uchar      Shift5Left( registera accu )       @ 0x0ba8;
uchar      Shift6Left( registera accu )       @ 0x0ba7;
uchar      Shift7Left( registera accu )       @ 0x0ba6;
uchar      Shift4Right( registera accu )      @ 0x0be9;
uchar      Shift5Right( registera accu )      @ 0x0be8;
uchar      Shift6Right( registera accu )      @ 0x0be7;
uchar      Shift7Right( registera accu )      @ 0x0be6;
uchar      rolA1( registera accu )            @ 0x0AF4;
uchar      rolA2( registera accu )            @ 0x0AF2;
uchar      rolA3( registera accu )            @ 0x0AF0;
uchar      rolA4( registera accu )            @ 0x0AEE;
uchar      rolA7( registera accu )            @ 0x0AEC;
void       U_SetBitH( uchar bit_num )         @ 0x0e18;
void       U_GetBitH( uchar bit_num )         @ 0x0e0c;
#endif
#pragma option +l
```



**7.1.2 eiba1.05h**

```
#pragma option -l

/*----- */
/*      Preparation of Communication Object data structure      */
/*----- */

#ifdef obj10
#define NUMOBJ 10
#else
#ifdef obj09
#define NUMOBJ 9
#else
#ifdef obj08
#define NUMOBJ 8
#else
#ifdef obj07
#define NUMOBJ 7
#else
#ifdef obj06
#define NUMOBJ 6
#else
#ifdef obj05
#define NUMOBJ 5
#else
#ifdef obj04
#define NUMOBJ 4
#else
#ifdef obj03
#define NUMOBJ 3
#else
#ifdef obj02
#define NUMOBJ 2
#else
#ifdef obj01
#define NUMOBJ 1
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif

#define BEG_TAB 0x0116
#define SIZE_TAB ( 6+NUMOBJ*7+MOREOBJ*4 )
#define FREE_SIZE ( 0x200 - ( BEG_TAB + SIZE_TAB ) )

/*----- */
/*      Program and Data space definition      */
/*----- */

#pragma memory ROMPROG [ FREE_SIZE ] @ ( BEG_TAB + SIZE_TAB );
#pragma memory RAMPAGE0 [18] @ 0x00CE;
```

```
#pragma option +l
/*----- */
/*      Communication Object : Flags Declaration      */
/*----- */
```

```
bits CommFlags[ ( NUMOBJ + 1 ) / 2 ];
```

```
/*----- */
/*      Communication Object : Value Declarations */
/*----- */
```

```
#pragma option -l
/*----- */
/*      Communication Object # 1 Definition      */
/*----- */
```

```
#define end_check 0
```

```
#ifndef obj01
#ifndef ram01
#define off01 0x01ff
#pragma option +l
```

```
uchar obj01[ size01 ];
```

```
#pragma option -l
#else
#define off01 0x01ff-size01
#undef end_check
```

```
#define end_check 0x01ff-off01
#pragma option +l
```

```
const uchar obj01[ size01 ] @ off01 = { dfv01 };
```

```
#pragma option -l
#endif
#endif
```

```
/*----- */
/*      Communication Object # 2 Definition      */
/*----- */
```

```
#ifndef obj02
#ifndef ram02
#define off02 off01
#pragma option +l
```

```
uchar obj02[ size02 ];
```

```
#pragma option -l
#else
#define off02 off01-size02
#undef end_check
```

```
#define end_check 0x01ff-off02
#pragma option +l
```

```
const uchar obj02[ size02 ] @ off02 = { dfv02 };
```

```
#pragma option -l
#endif
#endif

/*----- */
/*      Communication Object # 3 Definition      */
/*----- */

#ifdef obj03
#ifdef ram03
#define off03 off02
#pragma option +l

uchar obj03[ size03 ];

#pragma option -l
#else
#define off03 off02-size03
#undef end_check

#define end_check 0x01ff-off03
#pragma option +l

const uchar obj03[ size03 ] @ off03 = { dfv03 };
```

```
#pragma option -l
#endif
#endif

/*----- */
/*      Communication Object # 4 Definition      */
/*----- */

#ifdef obj04
#ifdef ram04
#define off04 off03
#pragma option +l

uchar obj04[ size04 ];

#pragma option -l
#else
#define off04 off03-size04
#undef end_check

#define end_check 0x01ff-off04
#pragma option +l

const uchar obj04[ size04 ] @ off04 = { dfv04 };
```

```
#pragma option -l
#endif
#endif
```

```
/*----- */
/*      Communication Object # 5 Definition      */
/*----- */
```

```
#ifdef obj05
#ifdef ram05
#define off05 off04
#pragma option +l

uchar obj05[ size05 ];

#pragma option -l
#else
#define off05 off04-size05
#undef end_check

#define end_check 0x01ff-off05
#pragma option +l

const uchar obj05[ size05 ] @ off05 = { dfv05 };

#pragma option -l
#endif
#endif
```

```
/*----- */
/*      Communication Object # 6 Definition      */
/*----- */
```

```
#ifdef obj06
#ifdef ram06
#define off06 off05
#pragma option +l

uchar obj06[ size06 ];

#pragma option -l
#else
#define off06 off05-size06
#undef end_check

#define end_check 0x01ff-off06
#pragma option +l

const uchar obj06[ size06 ] @ off06 = { dfv06 };

#pragma option -l
#endif
#endif
```

```
/*----- */
/*      Communication Object # 7 Definition      */
/*----- */
```

```
#ifdef obj07
#ifdef ram07
#define off07 off06
#pragma option +l
```

```
uchar obj07[ size07 ];

#pragma option -l
#else
#define off07 off06-size07
#undef end_check

#define end_check 0x01ff-off07
#pragma option +l

const uchar obj07[ size07 ] @ off07 = { dfv07 };

#pragma option -l
#endif
#endif

/*----- */
/*      Communication Object # 8 Definition      */
/*----- */

#ifdef obj08
#ifdef ram08
#define off08 off07
#pragma option +l

uchar obj08[ size08 ];

#pragma option -l
#else
#define off08 off07-size08
#undef end_check

#define end_check 0x01ff-off08
#pragma option +l

const uchar obj08[ size08 ] @ off08 = { dfv08 };

#pragma option -l
#endif
#endif

/*----- */
/*      Communication Object # 9 Definition      */
/*----- */

#ifdef obj09
#ifdef ram09
#define off09 off08
#pragma option +l

uchar obj09[ size09 ];

#pragma option -l
#else
#define off09 off08-size09
#undef end_check

#define end_check 0x01ff-off09
#pragma option +l
```

```
const uchar obj09[ size09 ] @ off09 = { dfv09 };

#pragma option -l
#endif
#endif

/*----- */
/*      Communication Object # 10 Definition      */
/*----- */

#ifdef obj10
#ifdef ram10
#define off10 off09
#pragma option +l

uchar obj10[ size10 ];

#pragma option -l
#else
#define off10 off09-size10
#undef end_check

#define end_check 0x01ff-off10
#pragma option +l

const uchar obj10[ size10 ] @ off10 = { dfv10 };

#pragma option -l
#endif
#endif

/*----- */
/*      Preparation of Group Address Table and Association Table      */
/*----- */

#ifdef obj10
#define atl ga01,ga02,ga03,ga04,ga05,ga06,ga07,ga08,ga09,ga10
#define asl 1,0,2,1,3,2,4,3,5,4,6,5,7,6,8,7,9,8,10,9
#else
#ifdef obj09
#define atl ga01,ga02,ga03,ga04,ga05,ga06,ga07,ga08,ga09
#define asl 1,0,2,1,3,2,4,3,5,4,6,5,7,6,8,7,9,8
#else
#ifdef obj08
#define atl ga01,ga02,ga03,ga04,ga05,ga06,ga07,ga08
#define asl 1,0,2,1,3,2,4,3,5,4,6,5,7,6,8,7
#else
#ifdef obj07
#define atl ga01,ga02,ga03,ga04,ga05,ga06,ga07
#define asl 1,0,2,1,3,2,4,3,5,4,6,5,7,6
#else
#ifdef obj06
#define atl ga01,ga02,ga03,ga04,ga05,ga06
#define asl 1,0,2,1,3,2,4,3,5,4,6,5
#else
#ifdef obj05
#define atl ga01,ga02,ga03,ga04,ga05
#define asl 1,0,2,1,3,2,4,3,5,4
```

```
#else
#ifdef obj04
#define atl ga01,ga02,ga03,ga04
#define asl 1,0,2,1,3,2,4,3
#else
#ifdef obj03
#define atl ga01,ga02,ga03
#define asl 1,0,2,1,3,2
#else
#ifdef obj02
#define atl ga01,ga02
#define asl 1,0,2,1
#else
#define atl ga01
#define asl 1,0
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
```

```
#pragma option +l
```

### **7.1.3 eiba2.05h**

```
#pragma option -l
```

```
/*----- */
/*      Fill-up BCU EEPROM till end of unused space      */
/*----- */
```

```
const char end_prog[] = { 0xAA };
```

```
#define qta ( 0x01fe - end_check ) - end_prog
```

```
#if (qta>128)
```

```
const char fill_mem_128[ 128 ] = {\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83};
```

```
#undef qta
```

```
#define qta (( 0x01fe - end_check ) - end_prog) - 128
```

```
#endif
```

```
#if (qta>64)
const char fill_mem_064[ 64 ] = {\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83};

#undef qta
#define qta (( 0x01fe - end_check ) - end_prog) - 64
#endif

#if (qta>32)
const char fill_mem_032[ 32 ] = {\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83};

#undef qta
#define qta (( 0x01fe - end_check ) - end_prog) - 32
#endif

#if (qta>16)
const char fill_mem_016[ 16 ] = {\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83,\
    0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83};

#undef qta
#define qta (( 0x01fe - end_check ) - end_prog) - 16
#endif

#if (qta>8)
const char fill_mem_008[ 8 ] = { 0x83,0x83,0x83,0x83,0x83,0x83,0x83,0x83 };
#undef qta
#define qta (( 0x01fe - end_check ) - end_prog) - 8
#endif

#if (qta>4)
const char fill_mem_004[ 4 ] = { 0x83,0x83,0x83,0x83 };
#undef qta
#define qta (( 0x01fe - end_check ) - end_prog) - 4
#endif

#if (qta>2)
const char fill_mem_002[ 2 ] = { 0x83,0x83 };
#undef qta
#define qta (( 0x01fe - end_check ) - end_prog) - 2
#endif

#if (qta>0)
const char fill_mem_000[ 1 ] = { 0x83 };
```



```
#undef qta
#define qta (( 0x01fe - end_check ) - end_prog) - 1
#endif

#pragma option +l

#pragma option +l
/*----- */
/*      Generation of Group Address Table*/
/*----- */

const uchar AddrTable[ (NUMOBJ+MOREOBJ) * 2 + 3 ] @ 0x0116 =
{
    NUMOBJ+1, 0xff, 0xff,
    atl
};

/*----- */
/*      Generation of Association Table      */
/*----- */

const uchar AssocTab[ (NUMOBJ+MOREOBJ) * 2 + 1 ] @
    ( 0x0116 + (NUMOBJ+MOREOBJ)*2 + 3 ) =
{
    NUMOBJ,
    asl
};
#pragma option -l

/*----- */
/*      Preparation of Communication Object Table */
/*----- */

#ifdef obj10
#define line10 ,obj10&0xFF,0xdf+((obj10&0x0100)>>3),type10
#else
#define line10
#endif

#ifdef obj09
#define line09 ,obj09&0xFF,0xdf+((obj09&0x0100)>>3),type09
#else
#define line09
#endif

#ifdef obj08
#define line08 ,obj08&0xFF,0xdf+((obj08&0x0100)>>3),type08
#else
#define line08
#endif

#ifdef obj07
#define line07 ,obj07&0xFF,0xdf+((obj07&0x0100)>>3),type07
#else
#define line07
#endif

#ifdef obj06
```

```
#define line06 ,obj06&0xFF,0xdf+((obj06&0x0100)>>3),type06
#else
#define line06
#endif

#ifdef obj05
#define line05 ,obj05&0xFF,0xdf+((obj05&0x0100)>>3),type05
#else
#define line05
#endif

#ifdef obj04
#define line04 ,obj04&0xFF,0xdf+((obj04&0x0100)>>3),type04
#else
#define line04
#endif

#ifdef obj03
#define line03 ,obj03&0xFF,0xdf+((obj03&0x0100)>>3),type03
#else
#define line03
#endif

#ifdef obj02
#define line02 ,obj02&0xFF,0xdf+((obj02&0x0100)>>3),type02
#else
#define line02
#endif

#ifdef obj01
#define line01 obj01&0xFF,0xdf+((obj01&0x0100)>>3),type01
#else
#define line01
#endif

#pragma option +l

/*----- */
/*      Generation of Communication Object Table      */
/*----- */

const uchar CommTab[ NUMOBJ * 3 + 2 ] @
    ( 0x116 + (NUMOBJ+MOREOBJ) * 4 + 4 ) =
{
    NUMOBJ,
    &CommFlags,
    line01
    line02
    line03
    line04
    line05
    line06
    line07
    line08
    line09
    line10
};

/*----- */
```

```

/* Generation of BCU constants                                     */
/*-----*/

const uchar    EE_OptionReg[] @ 0x0100 = {
    0xFC +
    ( OPTION_EE1P << 1 ) +
    OPTION_SEC };

const uchar    EE_Manufact[] @ 0x0104 = { COMPANY_ID };
const uchar    EE_DevTyp1[] @ 0x0105 = { DEVICE_TYPE / 0x0100 };
const uchar    EE_DevTyp2[] @ 0x0106 = { DEVICE_TYPE & 0xFF };
const uchar    EE_Version[] @ 0x0107 = { (SW_VER << 4) + SW_SUBVER };
const uchar    EE_CheckLim[] @ 0x0108 = { end_prog - 0x0100 };
const uchar    EE_PEL_Type[] @ 0x0109 = { PEL_TYPE };
const uchar    EE_SincRate[] @ 0x010A = { SYNC_RATE };
const uchar    EE_PortCDDR[] @ 0x010B = { PORTC_DDR };
const uchar    EE_PortADDR[] @ 0x010C = { PORTA_DDR };
const uchar    EE_RunError[] @ 0x010D = { 0xFF };
const uchar    EE_RouteCnt[] @ 0x010E = { ROUTE_COUNT << 4 };
const uchar    EE_MxRstCnt[] @ 0x010F = {
    ( BUSY_RETX_LIMIT << 5 ) +
    INAK_RETX_LIMIT };

const uchar    EE_ConfigDes[] @ 0x0110 = {
    ( PLMB_SPEED << 7 ) +
    ( A_EVENT_MSG << 6 ) +
    ( APPL_PROC << 5 ) +
    ( RUN_TIME_OBJ << 4 ) +
    ( TELEGRAM_RATE << 3 ) +
    ( CPOL << 2 ) +
    ( CPHA << 1 ) +
    ( PLMA ) };

const uchar    EE_AssocTabPtr[] @ 0x0111 = {( &AssocTab - 0x0100 )};
const uchar    EE_CommsTabPtr[] @ 0x0112 = {( &CommTab - 0x0100 )};
const uchar    EE_UsrInitPtr[] @ 0x0113 = {( &usr_init - 0x0100 )};
const uchar    EE_UsrPrgPtr[] @ 0x0114 = {( &main - 0x0100 )};
const uchar    EE_UsrSavPtr[] @ 0x0115 = {( &usr_save - 0x0100 )};

```

## **7.2 Appendix B: Manufacturers providing Tools for AP Development**

### **Ashling Microsystems Limited**

Butler House  
19-23 Market Street  
Maidenhead  
Berks SL6 8AA - United Kingdom  
Tel: 01628 / 773070  
(C-compiler, Assembler, Emulator)

### **iSystem GmbH**

Einsteinstraße 5  
85221 Dachau  
Germany  
Tel: 00 49 / 8131 / 25083  
(C-Compiler, Assembler, Emulator for 68HC05)

### **Lauterbach Datentechnik GmbH**

Fichtenstraße 27  
85649 Hofolding  
Tel: 00 49 / 8104 / 8943 - 0  
(Emulator)

### **Pentica Systems GmbH**

Besigheimer Weg 117  
74343 Sachsenheim  
Germany  
Tel: 00 49 / 7147 / 3085  
(C-Compiler, Assembler, Emulator for 68HC05)

### **Roth Hard + Software**

Waldstraße 16  
82284 Grafrath  
Germany  
Tel: 00 49 / 8144 / 1536  
(C-Compiler, Assembler for 68HC05)