Free Development Environment for Bus Coupling Units of the European Installation Bus

BCU SDK Edition

Martin Kögler (mkoegler@auto.tuwien.ac.at)

20th December 2005

Copyright

BCU SDK Documentation

Copyright (C) 2005 Martin Kögler <mkoegler@auto.tuwien.ac.at>

You can redistribute and/or modify this document under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this document; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Where the GNU General Public License mentions ”source code”, these LaTeX files or another editable file format, if it became the preferred format of modification, shall be referred to. Any not ”source code” form derived from this ”source code” are regarded as ”binary form”.

Modified versions, if the modification is beyond correcting errors or reformatting, must be marked as such.

Acknowledgments

This text is based on my diploma thesis.

Wolfgang Kastner and Georg Neugschwandtner, Automations Systems Group, Technical University Vienna, have helped me a lot.

Abstract

The European Installation Bus (EIB) is a field bus system for home and building automation. Bus Coupling Units (BCUs) provide a standardized platform for embedded bus devices. They can be programmed and configured via the bus. BCUs are based on the Freescale (Motorola) M68HC05 microcontroller family and provide a few tens of bytes of RAM and less than 1 KB of EEPROM. A common integration tool (called ETS) is used for the planning and installation of EIB systems.

Several problems exist for non commercial development projects. Although a free SDK for the BCU 1 is available, there is no free C compiler. Additionally, only certified programs can be processed by ETS. ETS as well as standard libraries for PC based bus access are only available for Windows.

During the course of the present project, a set of free tools for developing programs for BCU 1 and BCU 2 (twisted pair version) as well as loading them into the respective BCU were created. A RAD (Rapid Application Development) like approach is used for programming. Properties and event handlers of the used objects are described using a special specification language. Necessary code elements are written in C (inline assembler is also supported). An interface to an integration tool is also available.

A multi-user and network-capable Linux daemon to access the EIB was developed, which provides access to the transport layer as well as complex device management functions. Different interfaces for bus access are supported (PEI 16, FT 1.2, EIBnet/IP Routing + Tunneling and TPUART).

The tool chain is based on the GNU tool chain. The hardware limitations of the target system were a key point of the porting activities. It is described how GCC was ported to an accumulator architecture with only two 8 bit registers – but with 16 bit address space – and only one call stack.

Small code size is a primary requirement. Therefore, integers of 3, 5, 6 and 7 bytes are supported for situations, where a two byte integer is too small, but 4 or 8 byte integer types would be unnecessarily large. As the architecture uses variable length instruction formats, a mechanism which selects the smallest variant was implemented into the linker.

A mechanism is shown which makes GCC distribute variables over non contiguous segments automatically. This feature is required by the BCU 2 architecture. Additionally, transparent access to the EEPROM was added. Its concept is related to ISO/IEC TR 18037 named address spaces.

Kurzfassung

Der European Installation Bus (EIB) ist ein Feldbus für die Heim- und Gebäudeautomation. Die Standardplattform für Embedded-Teilnehmer bilden Bus Coupling Units (BCUs). Diese können über den Bus mit Anwendungen und Konfiguration versehen werden. BCUs basieren auf der Freescale (Motorola) M68HC05 Mikrocontroller-Familie. Sie bieten einige dutzend Bytes RAM und weniger als 1 KB EEPROM an. Für die Planung und Installation von EIB-Systemen steht eine einheitliche Integrationssoftware (ETS) zur Verfügung.

Für nicht-kommerzielle Entwicklungen stellen sich mehrere Probleme. Zwar existiert für die BCU 1 ein freies SDK, es steht aber kein freier C-Compiler zur Verfügung. Außerdem können standardmäßig verfügbare ETS-Versionen nur zertifizierte Anwendungen verarbeiten. Darüber hinaus sind ETS und Standardbibliotheken für den PC-basierten Buszugriff nur für Windows verfügbar.

Im Rahmen der vorliegenden Arbeit wurde ein Set von frei verfügbaren Tools geschaffen, das sowohl die Entwicklung von Programmen für BCU 1 und BCU 2 (Twisted-Pair-Version) als auch deren Übertragung auf die entsprechende BCU ermöglicht. Für die Programmierung wird eine RAD (Rapid Application Development)-ähnliche Vorgehensweise unterstützt. Eigenschaften und Eventhandler der benutzten Objekte werden mittels einer eigenen Spezifikationssprache festgelegt. Nötige Codeelemente werden in C (mit Unterstützung für Inline Assembler) ergänzt. Eine Schnittstelle für ein Integrationswerkzeug wird ebenfalls bereitgestellt.

Für den Zugriff auf EIB wurde ein Multi-User- und netzwerk-fähiger Linux-Daemon entwickelt, der einerseits direkten Zugriff auf die Transportschicht, andererseits auch komplexe Gerätemanagementfunktionen anbietet. Für den Buszugriff werden verschiedene Schnittstellen unterstützt (PEI 16, FT 1.2, EIBnet/IP Routing + Tunneling und TPUART).

Das Entwicklungswerkzeug basiert auf der GNU Toolchain. Bei der Portierung standen die Hardwarebeschränkungen der Zielplattform im Mittelpunkt. Es wird gezeigt, wie GCC auf eine Akkumulator-Architektur mit nur zwei 8-Bit Registern – aber 16 Bit Addressraum – und reinem Call-Stack portiert werden kann.

Geringe Codegröße ist eine zentrale Anforderung. Daher werden Integervariablen mit Größen von 3, 5, und 7 Bytes für Situationen unterstützt, in denen der Wertebereich eines 2-Byte-Integer nicht ausreicht, 4- oder 8-Byte-Typen aber zu groß sind. Da die Architektur verschieden lange Instruktionsformate bereitstellt, wurde auch in den Linker ein Mechanismus implementiert, der das kleinstmögliche Format wählt.

Es wird ein Mechanismus gezeigt, der es erlaubt, mit GCC automatisch Variablen auf nicht zusammenhängende Segmente zu verteilen. Diese Möglichkeit ist aufgrund der speziellen Anforderungen der BCU 2 notwendig. Weiters wird transparenter Zugriff auf das EEPROM bereitgestellt. Der dabei gewählte Mechanismus orientiert sich am Konzept der “named address spaces” aus ISO/IEC TR 18037.

Contents

1. Introduction **19**

1.1. The European Installation Bus . . . 19
1.2. The GNU project . . . 20
1.3. Goal of the present project . . . 20
1.4. Features and limitations . . . 20
1.4.1. Licence . . . 21
1.5. Place of the BCU SDK in the development and deployment work flow . . 22
1.5.1. Development work flow . . . 22
1.5.2. Deployment work flow . . . 22
1.6. Course of the project . . . 25
1.7. Future work . . . 26
1.8. Structure of the document . . . 26

I. M68HC05 **27**

2. M68HC05 architecture **29**

2.1. Register . . . 29
2.2. Addressing modes . . . 30
2.3. Instruction set . . . 31

3. GNU utilities **35**

3.1. Overview of the GNU utilities . . . 35
3.2. Configuration . . . 36
3.3. *Opcode* library . . . 37
3.4. *Bfd* library . . . 37
3.4.1. Relaxation . . . 39
3.5. Binutils . . . 40
3.6. GNU assembler . . . 40
3.6.1. Assembler syntax . . . 41
3.7. GNU linker . . . 42
3.8. Sim . . . 43
3.9. GNU debugger . . . 44
3.10. Newlib . . . 45

3.11. Libgloss 45

4. GCC 47

4.1. Structure of GCC 47
4.2. RTL 48
4.3. Machine description 51
4.3.1. Normal named instruction 51
4.3.2. Normal anonymous instruction 52
4.3.3. Definition of an expander 52
4.3.4. Definition of constants 53
4.3.5. Definition of attributes 54
4.3.6. Definition of a combination of instruction and splitter 54
4.3.7. Peephole optimization 55
4.4. Libgcc 55
4.5. Target description 55
4.6. Overview of the M68HC05 port 56
4.7. Details 58
4.7.1. Type layout 58
4.7.2. Register 58
4.7.3. Register classes 59
4.7.4. Pointer 59
4.7.5. Calling convention 60
4.7.6. Stack frame 60
4.7.7. Frame pointer elimination 60
4.7.8. Sections 60
4.7.9. Constraints 61
4.7.10. Operands 61
4.7.11. RTL split helper functions 63
4.7.12. RTL patterns 64
4.7.13. Predicates 65
4.7.14. Cost functions 66
4.7.15. The *eeprom* attribute 66
4.7.16. The *loram* attribute 67

II. BCU/EIB 69

5. BCU operating system 71

5.1. Modes of communication 71
5.2. BCU 1 71
5.2.1. Accessing the PEI 73
5.2.2. Timer Subsystem 73
5.2.3. BCU 1 API 74
5.3. BCU2 79

5.3.1. BCU 2 API 80

6. BCU SDK **83**
6.1. Common files 83
6.2. XML related programs 83
6.3. Build system 84
6.4. Configuration file parser 87
6.5. Bcugen1 and bcugen2 88
6.6. Overview of the generated code 89
6.7. Memory layout 89

7. EIB bus access **93**
7.1. Overview 93
7.2. Architecture 94
7.3. Back ends 96
7.3.1. EMI2 96
7.3.2. EMI1 96
7.3.3. KNX USB interface 97
7.3.4. EIBnet/IP Routing 98
7.3.5. EIBnet/IP Tunneling 98
7.3.6. TPUART kernel driver 99
7.3.7. TPUART user mode driver 99
7.4. Core 100
7.4.1. Layer 3 100
7.4.2. Layer 4 100
7.5. Layer 7 101
7.6. EIBnet/IP server front end 102
7.7. EIBD front end 102
7.7.1. Protocol 103

III. Using the BCU SDK **109**

8. Input format **111**
8.1. BCU configuration 111
8.1.1. Device block 112
8.1.2. FunctionalBlock block 114
8.1.3. Interface block 114
8.1.4. IntParameter block 116
8.1.5. FloatParameter block 116
8.1.6. ListParameter block 117
8.1.7. StringParameter block 117
8.1.8. GroupObject block 118
8.1.9. Object block 119

8.1.10. Property block . . . 119
8.1.11. Debounce block . . . 121
8.1.12. Timer block . . . 122
8.1.13. PollingMaster block . . . 124
8.1.14. PollingSlave block . . . 124
8.2. C files . . . 124
8.3. API functions . . . 126

9. File format for data exchange with integration tools 129
9.1. Configuration process . . . 130
9.2. Basic definitions . . . 131
9.3. Application information . . . 131
9.3.1. Functional block . . . 132
9.3.2. Interface . . . 132
9.3.3. Group objects . . . 134
9.3.4. Properties . . . 135
9.3.5. Polling master . . . 135
9.3.6. Polling slave . . . 135
9.3.7. Parameter . . . 136
9.4. Configuration description . . . 137
9.4.1. Group objects . . . 138
9.4.2. Property . . . 138
9.4.3. Polling master . . . 139
9.4.4. Polling slave . . . 139
9.4.5. Parameter . . . 139
9.5. Limitations . . . 139

10. Usage/Examples 141
10.1. Installation . . . 141
10.1.1. Installation in a home directory . . . 141
10.1.2. Prerequisites . . . 141
10.1.3. Getting the source . . . 142
10.1.4. Installing GCC . . . 142
10.1.5. Installing pthsem . . . 142
10.1.6. Installing the BCU SDK . . . 142
10.1.7. Granting EIB access to normal users . . . 142
10.1.8. Development version . . . 143
10.1.9. Building install packages . . . 144
10.2. Using eibd . . . 144
10.2.1. Command line interface . . . 144
10.2.2. USB backend . . . 145
10.2.3. EIBnet/IP server . . . 146
10.2.4. Example programs . . . 147
10.2.5. Usage examples . . . 148

10.2.6. *eibd* utilities 149
10.3. Recovering from errors 149
10.4. Developing BCU applications 150
10.5. Generating BCU applications 151
10.6. Example program 151
10.6.1. A negation which can be disabled 151
10.6.2. Cyclic switching 154

IV. Appendix 157

A. Image format 159
A.1. Streams 159
A.1.1. L_BCU_TYPE 159
A.1.2. L_CODE 159
A.1.3. L_STRING_PAR 160
A.1.4. L_INT_PAR 160
A.1.5. L_FLOAT_PAR 160
A.1.6. L_LIST_PAR 160
A.1.7. L_GROUP_OBJECT 160
A.1.8. L_BCU1_SIZE 160
A.1.9. L_BCU2_SIZE 161
A.1.10. L_BCU2_INIT 161
A.1.11. L_BCU2_KEY 162
A.2. Valid images 162

B. Tables 163
B.1. Available DP Types 163
B.2. Available property IDs 171

List of Figures

1.1. BCU SDK work flow 23
1.2. BCU SDK data flow 24

4.1. Comparison of ISO/IEC TR 18037 named address spaces with m68hc05-gcc address spaces 66

6.1. *build.ai* operational sequence and data flow 85
6.2. *build.img* operational sequence and data flow 86
6.3. Memory map of a BCU 1 90
6.4. Memory map of a BCU 2 91

7.1. Structure of *eibd* 95

List of Tables

4.1. Type sizes . . . 58
4.2. Register classes . . . 59
4.3. Constraints . . . 62

5.1. PEI types . . . 72

8.1. Group object types . . . 118
8.2. Property types . . . 120
8.3. Timer resolutions . . . 123

1. Introduction

1.1. The European Installation Bus

The European Installation Bus *EIB* is a home and building automation bus system. It is optimized for low-speed control applications like lighting and blinds control.

EIB devices can be configured remotely via the network. The EIB protocol follows a collapsed OSI architecture with layers 1, 2, 3, 4 and 7 implemented. Different transmission media are available.

EIB was absorbed in the KNX specification ([KNX]), which is maintained by Konnex Association. References to the KNX Specification ([KNX]) in this document will be of the form [KNX] *Document number-section number*.

For planning an EIB installation and configuring the individual devices, a special MS Windows based software, called *ETS*, is used. This software is maintained by EIBA (EIB Association). It will handle every device which has passed compliance certification. For PC based EIB nodes, a Windows library for EIB access is available, which is also used by ETS.

BCUs (*Bus Coupling Units)* are standardized, generic platforms for embedded EIB devices. They include the entire physical layer network interface, power supply and a microcontroller with an implementation of the EIB protocol stack stored in the ROM. As processor core, the Freescale (Motorola) M68HC05 architecture is used.[1] The application program can be downloaded into the EEPROM via the bus.

Currently, two BCU families exist: the older BCU 1 and the new BCU 2 family. Within both, there are different revisions which can be distinguished by the mask version. For this project, only the mask versions 1.2 and 2.0 have been used.

BCUs are available for EIB twisted-pair (TP) and power-line (PL) media. Within KNX, these are referred to as TP1 and PL110. In this project, only the twisted pair medium (KNX TP1) is supported. This medium consists of two dedicated low voltage wires and supports bus powered devices.

Every BCU includes an asynchronous serial interface which provides access to the EIB protocol stack. The BCU 1 supports a protocol with RTS/CTS handshake ("PEI 16"), the BCU 2 additionally a FT1.2 based protocol (for details, see [KNX] 3/6/2-6.3 and 3/6/2-6.4).

An alternate way to access an EIB TP network is the Siemens TPUART IC. This IC implements OSI layer 1 and parts of layer 2 only, instead of the entire stack as BCUs do.

[1] One rare variant uses the M68HC11. It is however only available without housing and thus referred to as BIM (Bus Interface Module).

Finally, EIBnet/IP allows to access EIB via IP based networks (see [KNX] 3/8). It provides tunneling of EIB frames in both a point-to-point mode (referred to as Tunneling) and a point-to-multipoint mode (referred to as Routing).

1.2. The GNU project

The GNU project was founded by Richard Stallman. It has the goal to produce a complete, free operating system. It includes, for example, the GNU Compiler Collection (GCC), the GNU Debugger (GDB) and Emacs.

Most GNU projects are available under the GNU General Public License, which guarantees free access to the source code. As a legal organization, the Free Software Foundation (FSF) was founded. It holds the copyright for most projects.

1.3. Goal of the present project

The Automation Systems Group at the Institute of Computer Aided Automation has home and building automation as one research topic. Partly based on work carried out at the Fachhochschule Deggendorf, different tools for EIB, like bus access drivers and installation tools, were created.

For programming the BCU 1, the Fachhochschule Deggendorf released the *Free EIB IDE* [EIBIDE], which is a free clone of the normal SDK for the BCUs. It basically consists of a syntax compatible assembler, compatible header files and a simple GUI, where all functions can be accessed. The Free EIB IDE includes a loader which allows downloading of images via the bus. For bus access, it uses the FT1.2 protocol and thus requires a BCU 2. Additionally, it includes a bus monitor, which supports a TPUART based interface as well as the FT1.2 protocol. This bus monitor will not decode the frame contents.

The primary requirement of the present project was to create a programming environment which supports the next generation of BCUs, the BCU 2. During the course of the project, a complete set of development tools for both BCU 1 and BCU 2 has been developed. The entire set will in the following be referred to as *BCU SDK*. The programming tools are based on GNU utilities like GCC and Binutils.

1.4. Features and limitations

The *BCU SDK* has the following features:

- It includes its own specification language to describe the configuration of the BCU environment. Its concept is RAD-like, requiring the programmer to specify properties and event handlers only.

- It uses C for the code fragments (inline assembler is supported).

- To access the bus, FT1.2, the BCU 1 kernel driver, the KNX USB protocol (according to [KNX] AN037, only EMI1 and EMI2), the TPUART kernel and user mode driver and EIBnet/IP Routing + Tunneling are supported. Additionally (more or less working) experimental access to a BCU 1 without a kernel driver is supported. Either way, all management tasks are supported as well. Further details (including a description of these interfaces) can be found in Chapter 7.
- It can act as a limited EIBnet/IP server.
- It provides an API to provide EIB access in other programs. Several utility programs, which also illustrate the use of this API, are included.
- It includes a standard bus monitor, which optionally can decode EIB frames. Additionally, a special monitor mode (called *vBusmonitor*) even allows some traffic to be traced without switching to bus monitor mode.
- The programs are compiled after all configuration settings are known to reduce image size.

The following limitations are present:

- No data exchange with the ETS is possible.
- No graphical interface has been written. Input files can be edited with any text editor. The BCU SDK programs can be invoked from the command line or from a makefile.
- The compiler output will in most cases be larger than well optimized, hand written assembler code.
- It is not compatible with the original, commercial BCU SDK.
- Only C code (with inline assembler) is supported.
- If the bus is accessed via a BCU 1 or BCU 2, this BCU is inaccessible to the *BCU SDK* (using the local mode is not supported).

1.4.1. Licence

As the used tool chain is released under open source licences, the new parts of the SDK should also be freely available as open source.

As there are many definitions of free and open source software, the Debian definition of freedom, which is described in the Debian Free Software Guidelines (DFSG) ([DFSG]), was used. According to these guidelines, a BSD style licence ([BSD]) or the GPL ([GPL]) are reasonable choices.

Finally, the whole project was released under the GPL. The *eibd* client library (see Chapter 7) and the libraries for the BCU contain a linkage exception (like libgcc) so that they may be used for non-open source software. Modified versions of a XML Schema definition and XML DTDs must use a different namespace and version number.

1.5. Place of the BCU SDK in the development and deployment work flow

The part the BCU SDK has in the complete work flow of developing BCU applications and deploying them in installations is shown in Figure 1.1. At the bottom, the EIB network with the BCUs to be programmed is shown. A bus interface which is supported by the BCU SDK is connected to the network.

The entire development process is hidden from the project engineer by the integration tool. This tool assists the selection and download of application programs, parameters and communication relationships to build a working installation. The design of such a tool is not within the scope of the present project.

1.5.1. Development work flow

The developer writes the *BCU configuration* file (see Section 8.1) and the necessary C code fragments. Then he runs the *build.ai* program, which creates the *application information* (see Section 9.3). This file contains the program text[2] as well as meta information, like parameters.

With the help of a program, a skeleton for the *configuration description* (see Section 9.4) can be generated out of the *application information*. This skeleton has to be edited manually to reflect the required configuration (e.g. group addresses). This step only affects the meta data, the program text is not touched. Informally, the *application information* could be considered a questionnaire which has to be answered by the *configuration description* (see Figure 1.2).

Then the *build.img* program is invoked which creates a binary image from the program text and meta data contained in the *configuration description*. This image can be downloaded to the BCU using *eibd*. Additionally, *eibd* allows some management functions to be invoked directly.

When running *build.ai*, the program text can also be stored in an extra file. This file can be used by *build.img* instead of the program text contained in the *configuration description*. This is useful during development, as it avoids having to recreate the *configuration description* for code changes which do not affect the *BCU configuration*.

1.5.2. Deployment work flow

When the development process is finished, the final *application information* is passed to the integration tool. Such a tool can group and store many of these files in product databases.

Then the project engineer develops the structure and configuration of an EIB installation. In this process, the integration tool generates the *configuration description* meta

[2]This can be either a binary image or preprocessed code. In the BCU SDK, it contains a slightly modified collection of all program sources.

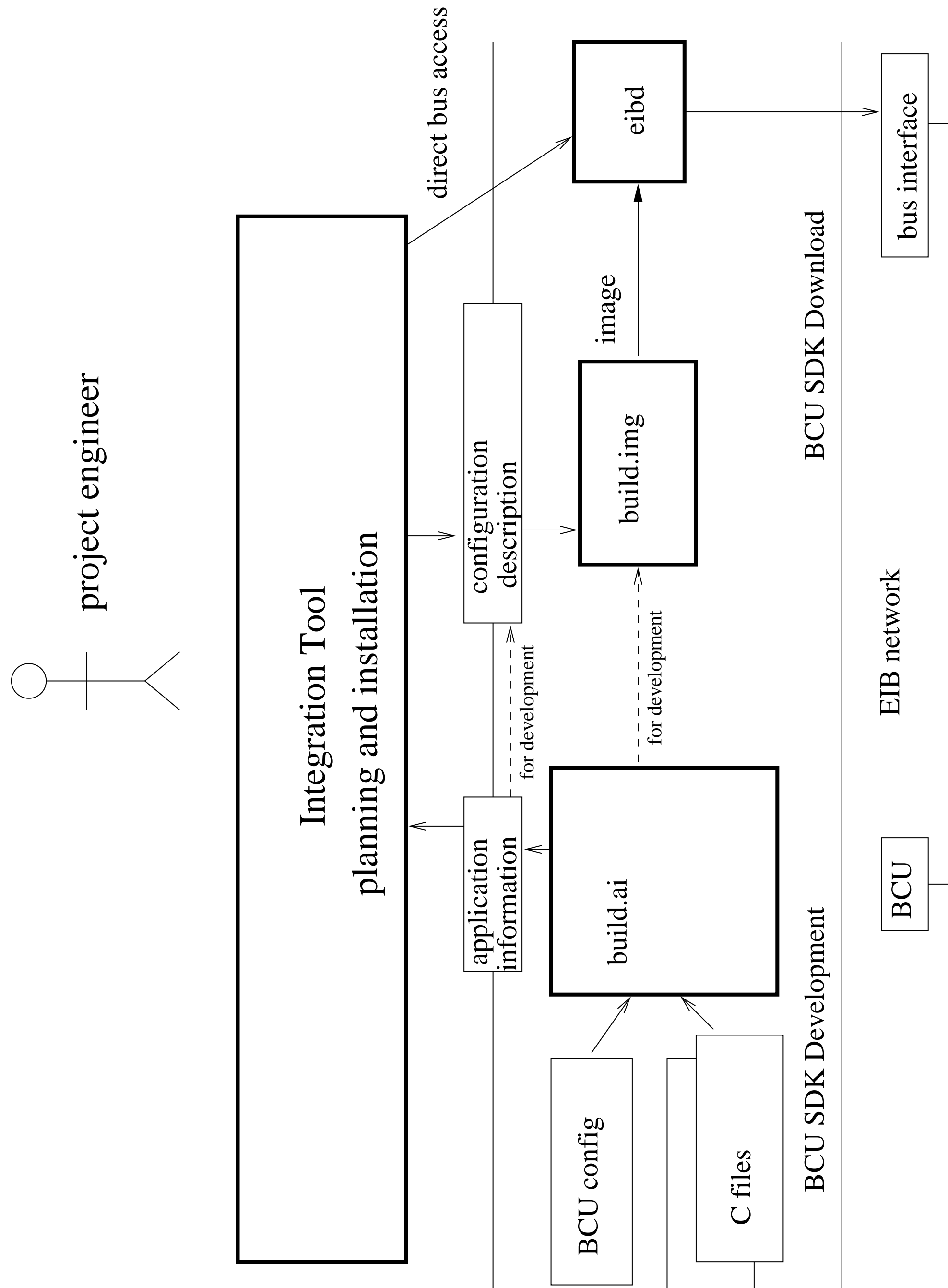

Figure 1.1.: BCU SDK work flow

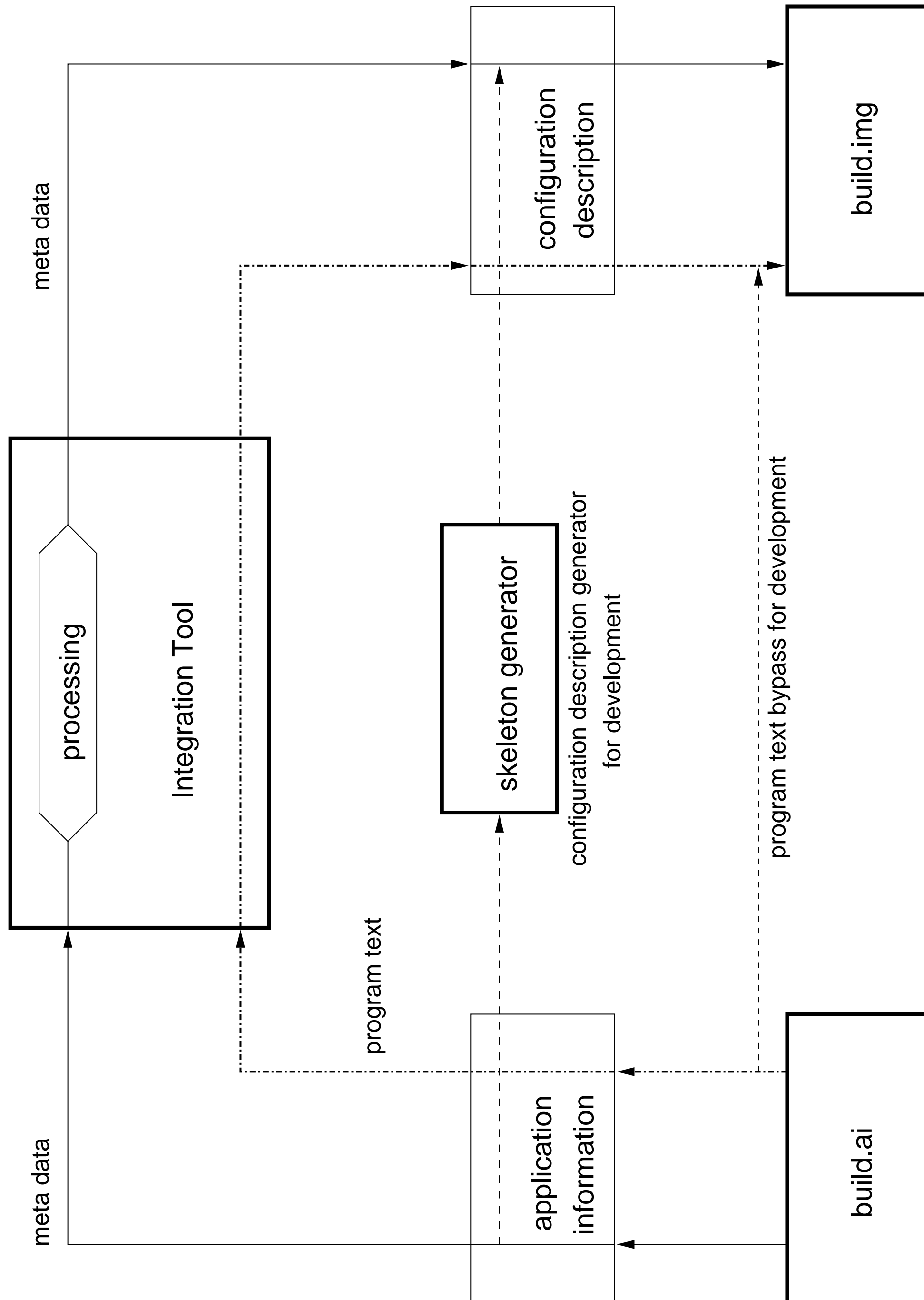

Figure 1.2.: BCU SDK data flow

data based on the *application information* meta data. The way this is done is entirely open to the integration tool. Again, the program text is not touched.

Finally, the project engineer starts the download process within the integration tool. To create the image to be loaded, the *configuration description* for each BCU is passed through the *build.img* program. The resulting image is passed to eibd for download. If necessary, the integration tool can perform physical address assignment beforehand using the management functions of *eibd*.

1.6. Course of the project

The project started with porting the binutils (and later, GCC). Additionally, device management using the FT1.2 protocol was explored. Later on, accessing the EIB bus via the BCU 1 driver and TPUART driver were added to a first prototype of a EIB bus access utility. The BCU 1 driver proved to work not very reliably on the test systems.

A first prototype of a BCU downloader and a BCU 1 image building tool was written. It supported the use of a BCU 1 to create new group telegrams based on the values of other group objects. Here all information (including all addresses) was specified in the input files and no support for interaction with an integration tool was considered.

As the BCU 1 kernel driver was not available on every development machine, experimental access to the BCU 1 without such a driver was implemented. When the new user mode driver was run on a Linux 2.6 system the first time, the system stopped responding. After several tries, a deadlock in the low latency mode of the serial driver was found. A workaround for this problem was finally included in Linux 2.6.11.

Later, an EIBnet/IP router became available, for which an EIBnet/IP Tunneling mode was added to the program. This access mechanism did not work well, as the IP router stopped responding after about one second for yet unknown reasons. The same program worked without any problem, when it was retried a few month later. Next, preliminary Linux 2.6 versions of the EIB kernel drivers became available. To make them usable, some bugs had to be fixed.

The old structure of the EIB bus access utility, which was single threaded and expected the data from the bus to come in a strict request/reply form, proved to be unreliable. Also, this concept was not suited for handling bus access devices where the communication process as well as the timing is not entirely controlled by the host.

As now the limitations and problems were known, the a complete redesign was made and development was started from scratch. As a replacement for the EIB bus access utility, *eibd* was written. It is based on GNU pth ([PTH]), a non preemptive multi threading library. Because the existing synchronization primitives did not provide the features needed, semaphore support was added. A patched version is distributed under the name *pthsem* ([PTHSEM]). Finally, the image building utility was rewritten as well to support most features of the BCU 2.

1.7. Future work

- The most important future work will be an integration tool.
- Another useful project would be the creation of a graphical front end for the BCU SDK.
- There are still some features missing, which could be added to the BCU SDK (e.g. user PEI handler or user application callbacks).
- The BCU SDK could be ported to the power line BCUs.
- Additionally, the BCU kernel drivers need to be improved, especially the timing behavior of the BCU 1 driver.
- The GCC offers lots of optimization possibilities. Also, the automatic generation of bit operations is still missing.
- The back ends of eibd for BCU 1 and BCU 2 do not yet support all features the BCU provides (e.g. additional group addresses).
- The TPUART kernel driver and back end could be extended to deliver more telegrams in the vBusmonitor mode.
- Based on the current GDB release, a BCU simulator could be created which supports the debugging of BCU applications.

1.8. Structure of the document

This document is divided into three parts:

M68HC05 This part describes the M68HC05 architecture and covers key issues concerning the porting process of the GNU utilities.

EIB/BCU This part gives an overview of the internals of the BCU SDK.

Using the BCU SDK This part contains information concerning the use of the BCU SDK, including installation, operation and file formats.

Part I.

M68HC05

2. M68HC05 architecture

The Freescale (formerly Motorola) M68HC05 is a family of 8 bit, low cost microcontrollers. They are based on an accumulator architecture with their IO registers mapped at the start of the memory. The members of the family differ in the amount of RAM, ROM and EEPROM as well as the different IO interfaces available.

The models MC68HC05B6 and MC68HC705BE12 are used in BCU 1 and BCU 2, respectively. While the MC68HC05B6 is a generally available model, the MC68HC705BE12 contains KNX specific on-chip peripherals and is only used in KNX. As the peripherals are managed by the BCU operating system, only the processor core is described.

The processor core is a von Neumann architecture with a linear 16 bit address space. The different memory types are mapped at different addresses. For read accesses, there is no difference for all memory types.

The opcodes have a length of one byte with null to two bytes of address information. There are no alignment constraints for instructions as well as for data.

2.1. Register

A The 8 bit accumulator, which is used in nearly every arithmetic operation. It is used as a source operand as well as the destination.

X The 8 bit index register. It can be used as temporary storage, as operand for the multiplication, as well as the only way to access data at memory locations which are not fixed.

PC The 16 bit instruction pointer. Internally, some models fix the three high order bits to 0.

SP The stack pointer. Its value cannot be retrieved or set by any instruction. Only a reset of its value to the startup value is available. The high order bits are fixed, so that it can only store values between 0xC0 and 0xFF. An interrupt allocates 5 bytes, a normal subroutine call uses 2 bytes on the stack.

CCR The condition code register. It has five flags:

- **H** Half carry, which can be used for BCD arithmetic.
- **I** Interrupt mask, which controls the generation of external interrupts.
- **Z** Zero, which is set, when the result is zero.

C Carry, which is set in case of an overflow of an arithmetic operation or is used in shift and rotate operations.

N Negative, which is set if the result has the sign bit (bit 7) set.

The flags can be only checked by conditional jump operations. Nearly all operations change the condition code register.

2.2. Addressing modes

INH, implicit For some operations like *TAX* (transfer A to X), the operators are fixed. Such an operation only stores the opcode.

DIR, 8 bit address All logic and arithmetic operations, which support memory addresses, support this addressing mode. Here, after the opcode, a memory address within the first 256 bytes follows.

EXT, 16 bit address A limited set of operations also supports 16 bit addresses. Here, the opcode is followed by the 16 bit address in big endian format.

IX, indexed without offset The content of the X register is used as memory address within the first 256 bytes. Here only the opcode is present. In the Motorola assembler, such an addressing mode is written as *,X*.

IX1, indexed with 8 bit offset All logic and arithmetic operations which support memory addresses support this addressing mode. After the opcode, an 8 bit offset follows. As address, the content of the X register plus the offset is used. In the Motorola assembler, such an addressing mode is written as *offset, X*.

IX2, indexed with 16 bit offset A limited set of operations also supports a 16 bit offset as well as this addressing mode. As address, the content of the X register plus the offset is used. In memory, after the opcode, the 16 bit offset is stored in big endian format. In the Motorola assembler, such an addressing mode is written as *offset, X*.

IMM, immediate The 8 bit parameter of the operation is stored directly after the opcode. In the Motorola assembler syntax, an immediate value is prefixed by a #.

REL, PC relative This addressing mode is used for conditional jumps. Here, a signed offset relative to the end of the jump instruction is stored as an 8 bit value after the opcode.

8 bit address + PC relative Some jumps need a data address as well as a target address. Here, the data address and then the 8 bit PC relative address are stored after the opcode.

2.3. Instruction set

The opcodes of the instruction set are organized in a way that for most operations the first 4 bits determine the used addressing mode and the second 4 bits the operation.

The following instructions are supported:

- Instructions which support IMM, DIR, EXT, IX, IX1 and IX2:

  LDA loads an 8 bit operand into the A register.

  STA stores the content of the A register, has no IMM mode.

  LDX loads an 8 bit operand into the X register.

  STX stores the content of the X register, has no IMM mode.

  ADD adds the operand to A and stores the result in A.

  ADC adds the operand plus C to A and stores the result in A.

  SUB subtracts the operand from A and stores the result in A.

  SBC subtracts the operand plus C from A and stores the result in A.

  CMP sets the flags N, Z and C according to the result of A - operand.

  CPX sets the flags N, Z and C according to the result of X - operand.

  BIT sets the flags N and Z according to the result of A XOR operand.

  AND A = A AND operand

  ORA A = A OR operand

  EOR A = A XOR operand

  JMP jumps to the address pointed to by the operand, has no IMM mode.

  JSR pushes the address of the next instruction on the stack and jumps to the address pointed to by the operand. Instead of the IMM operation, the operation is called BSR and uses REL mode.

- The following operations use one operand as source and destination. They support the modes DIR, IX1 and IX. Additionally, there is one variant which uses the A register, whose name has an A appended, as well as a variant for the X register, with an X appended.

  CLR stores 0.

  INC increments the operand.

  DEC decrements the operand.

  NEG performs binary negation.

  COM calculates the two's complement.

  ROL rotates left through carry.

ROR rotates right through carry.

ASL / LSL shifts left (with carry as 9 bit).

LSR logic shift right (with carry as 9 bit), which replaces the sign bit with 0.

ASR arithmetic shift right (with carry as 9 bit), which keeps the sign bit.

TST set N and Z according to the operand.

- Operations, which only support the DIR mode:

BCLR clear a bit of the operand. In the Motorola assembler syntax, the bit number is written as the first parameter.

BSET set a bit of the operand. In the Motorola assembler syntax, the bit number is written as the first parameter.

BRCLR branch if a bit is cleared. In the Motorola assembler syntax, the bit number is written as the first parameter. As a third parameter, the PC relative target address follows.

BRSET branch if a bit is set. In the Motorola assembler syntax, the bit number is written as the first parameter. As a third parameter, the PC relative target address follows.

- The following conditional branches with the REL addressing mode are supported:

BRA branch always

BRN branch never

BHI branch if higher (C or Z =0)

BLS branch if lower same (C or Z =1)

BCC / BHS branch if carry is clear, branch higher same (C=0)

BCS / BLO branch if carry set, branch lower (C=1)

BNE branch if not equal (Z=0)

BEQ branch if equal (Z=1)

BHCC branch if half carry clear (H=0)

BHCS branch half carry set (H=1)

BPL branch if plus (N=0)

BMI branch if minus (N=1)

BMC branch if interrupt mask clear (I=0)

BMS branch if interrupt mask set (I=1)

BIL branch if IRQ pin is low

BIH branch if IRQ pin is high

- The following operations have no parameter:

  TAX stores the content of A in X.

  TXA stores the content of X in A.

  CLC clears the carry flag (C).

  SEC sets the carry flag (C).

  SEI sets the interrupt mask bit (I).

  CLI clears the interrupt mask bit (I).

  MUL stores the result of the unsigned multiplication of A and X in X (high byte) and A (low byte).

  RTI returns from interrupt, restores the processor state to the state before the interrupt.

  RTS returns from subroutine, pops the return address from the stack and sets the PC to it.

  SWI triggers a software interrupt.

  RSP resets the stack pointer to 0xFF.

  NOP does nothing.

  STOP enables interrupts and stop oscillator.

  WAIT enables interrupts and stop CPU clock.

The following operations modify the flags as side effect:

- AND, CLR, DEC, EOR, INC, LDA, LDX, ORA, STA and STX modify N and Z
- ASL, ASR, COM, LSL, LSR, NEG, ROL, ROR, SUB and SBC modify N, Z and C
- ADD and ADC modify H, N, Z and C
- BRCLR and BRSET modify C
- MUL modifies C and H

3. GNU utilities

For the BCU SDK an assembler, compiler and linker were needed. As there was no free tool chain available, a new one needed to be created.

First ideas of writing a compiler which directly generates binary code from scratch were abandoned, because writing a complete C parser with type checking would have been too much work for the scope of this project.

Therefore, the decision was to port an existing C compiler to the M68HC05 architecture. In addition to GCC, other free compilers were searched for. Possible candidates were *anyc* ([ANYC]) (for the compiler front end) or *SDCC* ([SDCC]), which supports different architectures and has some optimizations. Compared to GCC, *SDCC* is much simpler and has less features.

Finally, GCC was selected, because the front end and optimizations part has definitely proven to work, as it is the standard compiler on most free operating systems. In addition the core parts of GCC are maintained by a large community. As GCC was selected, it was clear to use the binutils as assembler and linker.

Porting GCC to the MC68HC05 architecture caused some problems, which were solved with various tricks. At the moment, the GCC port generates usable code, but there is much room left for target specific optimizations. If *SDCC* had been the first choice, the structure would have been much simpler, but some parts, like the automatic removal of unused static variables, would have had to be implemented from scratch.

Large parts of the porting activity consists of finding the right code in other architectures and copying it into the new one. In many situations, the code needs small adaptations, but only a very small part is totally new code.

3.1. Overview of the GNU utilities

The FSF (*Free Software Foundation*) distributes a wide range of software. The following list covers the parts relevant to the present project (as names, the names of the directory in the main building directory are used):

libiberty A common library, which is used in many GNU programs.

intl The GNU translation library, which provides *gettext*.

bfd The *bfd* library is used to create, read and write executables and object files in various formats.

opcode The *opcode* library implements a disassembler for various architectures.

gas The GNU assembler, which can support a wide range of architectures.

ld The GNU linker.

binutils Various tools to work with executables and object files, like *strip*, *ar* and *ranlib*. The distribution package *binutils* also includes *gas* and *ld*.

sim Normally distributed as a part of *gdb*. It contains a simulator for some architectures.

gdb The GNU debugger. It also supports remote debugging as well as debugging of programs running in the simulator.

GCC The GNU C compiler (or GNU compiler collection). It is presented in detail in Chapter 4.

libstdc++ The GNU C++ runtime library. As it is too big, it is not used here.

dejagnu A regression testing framework.

newlib Used as a C runtime library on smaller platforms (not a GNU project).

libgloss Part of newlib. It should contain the platform specific interfaces to an operating system for newlib.

For this project bfd, opcode, gas, ld, binutils and GCC were ported to the M68HC05 architecture. Since finding all errors in GCC only by reviewing the output is impossible, a simulator for the M68HC05 architecture based on the M68HC11 version of sim was created.

Because the regression tests for GCC need dejagnu and a C runtime library, dejagnu, libgloss and newlib were also ported. As finding bugs in the GCC output without a debugger turned out to be very difficult, a working gdb based on the simulator was created. Functions which were too difficult to implement and are not really needed were left out (see section 3.9).

The rest of this chapter covers the important points of the port of the GNU utilities (except GCC).

3.2. Configuration

GNU programs as well as other software use *autoconf* to adapt the software for a specific target. If a software has to be compiled, first *configure* must be run, which determines what kind of operating system is installed, which header files, libraries and compilers are present, to which location the program should be installed, and other settings.

configure describes machines by a triplet, e.g. *i686-pc-linux-gnu*. Its first part is the processor, then the machine type follows and finally comes the operating system.

Up to three triplets are used:

build The machine, on which the build process is executed.

host The machine, on which the built program should run.

target The machine, for which libraries should be built or for which the program should generate output.

Under normal conditions, the correct values are guessed automatically. In the case of a cross tool chain, as the m68hc05 port, the target has to be specified.

As the name of the processor *m68hc05* was chosen, so that running *configure* with the parameter *–target=m68hc05* creates the correct Makefile for the m68hc05 port.

Internally, the triplet *m68hc05-unknown-bcu* is used. As the operating system, *bcu* is used because the whole port was designed with the features and limits of the BCU in mind.

3.3. *Opcode* library

The new architecture was added to the disassembler library.

As a first step, an instruction list in a generally usable format was created. An example entry looks as follows:

```
M68HC05_INS("adc",M68HC05_IMM,0xa9)
```

Every instruction is represented by the definition *M68HC05_INS*. The application program, which uses this list, can convert it to the needed form by defining *M68HC05_INS* and including the list after that. The first parameter contains the opcode name as a string, the second the supported addressing mode and the third the opcode number.

The only change to the original instruction set is that for bit operations (BSET, BCLR, BRSET, BRCLR), the bit number is appended after a period. Therefore such instructions have the syntax *bset.1 test* instead of *bset 1,test*. All names are written in lower case.

Then the files for the m68hc05 architecture were added and hooked into the general code. Finally, a small function, which reads one byte, searches the opcode in a table, determines its addressing mode, reads the parameters and prints everything in a human readable format, was written.

3.4. *Bfd* library

The bfd library [BFD, BFDINT] provides an abstract layer to access different object formats in the same way. Additionally, it contains a part of the linker functions.

For the port of the bfd library, ELF was chosen as the default object and executable format, because all new bfd ports use it and it provides all necessary features.

References to symbols which are not resolved yet are stored as relocations in ELF. A relocation states that at a specific location in the code, the address of a symbol should be stored in a specific way. Each architecture defines its own set of relocations. For the

M68HC05 architecture, there was only the definition of the architecture number, but not for relocations. So they had to be defined from scratch.

The implemented set includes:

R_M68HC05_NONE A relocation, which does nothing. This is needed to turn a relocation off.

R_M68HC05_8 stores the address as 8 bit value.

R_M68HC05_16 stores the address as 16 bit value.

R_M68HC05_32 stores the address as 32 bit value.

R_M68HC05_HI8 stores bits 8–15 of the address.

R_M68HC05_LO8 stores bits 0–7 of the address.

R_M68HC05_HLO8 stores bits 16–23 of the address.

R_M68HC05_HHI8 stores bits 24–31 of the address.

R_M68HC05_HI16 stores bits 16–31 of the address.

R_M68HC05_LO16 stores bits 0–15 of the address.

R_M68HC05_PCREL_8 stores a PC relative address as 8 bit value, as used for relative branches.

R_M68HC05_PCREL_16 stores a PC relative address as 16 bit value, this relocation is currently unused.

R_M68HC05_PCREL_32 stores a PC relative address as 16 bit value, this relocation is currently unused.

R_M68HC05_RELAX_8 stores the address as 8 bit value. Additionally it marks a relaxable instruction (see section 3.4.1).

R_M68HC05_RELAX_16 stores the address as 16 bit value. Additionally it marks a relaxable instruction.

R_M68HC05_RELAX_GROUP this relocation does not change the code. It only marks, that an expanded branch starts.

R_M68HC05_SECTION_OFFSET8 stores the offset in the current section of the address as 8 bit value.

R_M68HC05_SECTION_OFFSET16 stores the offset in the current section of the address as 16 bit value.

As the 16 bit addresses are stored in big endian format, big endian is used as default format.

Bfd uses other names for the relocations for internal purposes. Commonly used relocations have default names, architecture specific relocations must be added to this list. For managing this, a clever way is used: A file contains a list of their names and their documentation. Out of this list, the C definitions are generated by running *make headers* in the bfd directory.

In addition to the list of relocations, bfd keeps a list of the architectures and target vectors. An architecture describes the name, byte and word size as well as subtypes. As an example, for i386, a CPU with 64 bit extension is represented as a subtype.

The target vector contains the list of concrete functions to handle object files. For most parts, the default values are used. Only a few are overwritten.

The important parts are:

- a function to map between ELF and bfd relocations
- a function to process architecture specific relocations
- a function to get the relocated section content[1]
- functions to relax a section

3.4.1. Relaxation

Relaxation (in the context of bfd) means that the size of code sections is shrunk at link time by replacing instructions with other instructions.

For various instructions, the M68HC05 architecture offers multiple variants with different instruction lengths, but otherwise same function. Often, the final value of a symbol is still unknown at assembler time, so the largest variant must be chosen. If the address of the jump target is unknown or too far away, relative jumps are automatically negated and an absolute jump to the final location is added. As the generation of small code is a key requirement for the tool chain, relaxation is really needed.

For the relaxation, three relocations have been defined. The code is based on the M68HC11 port ([GNU11]). It checks every relocation, if it is one of these three special relocations.

The location of the opcode is known from the relocation type. If the parameter of the instruction can fit in a smaller variant, the opcode and the type of the relaxation are changed and the free bytes are deleted. For expanded jumps, the process is reversed, if possible. This is repeated, until no more conversions are possible.

The code which is processing the opcodes takes advantage of their schema, so only a few rules are needed. For example, every opcode whose upper 4 bits are 0xC can be converted to 0xB if the address fits in 1 byte.

[1] This is the same function as bfd normally performs, but uses the relocation function for this architecture.

There are two special rules: JMP is turned into BRA and JSR into BSR, if possible. In assembler code, only JMP and JSR should be used, because BRA will be expanded as a conditional jump and BSR will not be expanded. The relaxation will generate the optimal code.

The delete byte routine is simpler than the M68HC11 variant, because everything which uses a symbol or a PC relative address is stored as a relocation. So it is only necessary to adjust the values of the relocations. Bfd ensures that the correct value will be stored. For the M68HC11, the routine needs to check every relative jump and adjust it manually.

The heavy use of this technique (typically about $\frac{1}{3}$ of the assembler code size can be eliminated for GCC code) causes the stabs debugging line symbols to lose their synchronization with the code. This makes source level debugging nearly impossible.

So a routine which adjusts the stab information was written. As it turned out that using DWARF2 was better suited as debugging information, DWARF2 become the default debugging information for GCC. Therefore the stabs adjustment code is not tested very well. It is kept, because stabs is kept as an alternative for objdump and similar utilities which do not support DWARF2.

3.5. Binutils

The binutils directory contains various programs to process object files such as *ar*, *strip* and *strings*. As they use bfd, no porting activity was needed. Only within the *readelf* program, which can decode elf headers, the new relocations were added.

3.6. GNU assembler

The GNU assembler (*gas*) [GAS, GASINT] is a collection of different assemblers for different architectures. Most of them use the bfd library to write object files. For outdated ports, which do not use *bfd*, there was recently a call to remove them ([BIN01]).

The target specific part of the assembler is kept in a header and a C file. The header file contains some common definitions, like the architecture name, and various hooks for architecture specific functions. The C file contains the code.

gas processes a file the following way:

1. Each line is parsed and the temporary assembler code is stored in a fragment. If the length of an opcode cannot be determined at this time, the free space for the longest possible code is left and the next opcode is stored in a new fragment.

   For references to symbols, a fixup is created. This is the same as a relocation, only that it can store a complex assembler expression.

2. The assembler asks the back end for the size of each variable fragment and adjusts the symbols according to it.

3. The assembler asks the back end if the size of a fragment has changed because of the new symbol values. This is repeated until no further change occurs.
4. With the complete symbol information, the back end converts the variable fragments to fixed length and creates more fixups, if necessary.
5. In the output pass, fixups with constant values are embedded in the code. Then the code is stored. For the remaining fixups, relocations are stored in the object file.

The back end code consists of:

- An expression parser wrapper, which can also parse the target specific functions, like *lo8*.
- A function which divides the assembler instruction into its parts, finds the corresponding opcodes, parses the parameters, checks everything and emits the corresponding fragment and fixups.
- A size estimation function, which returns the worst case size of a fragment.
- A conversion function, which converts a variable fragment into a fixed length fragment.
- A fixup patcher, which stores resolved values in the output segments.
- Two functions to support the target specific functions, like *lo8*, for data storage pseudo operations.
- A function to replace parts of section names with unique values.

3.6.1. Assembler syntax

For the pseudo operations, the normal *gas* syntax is used. Additionally it supports the following functions as top level functions in expressions:

lo8 stores bits 0–7 of the expression result.

hi8 stores bits 8–15 of the expression result.

hlo8 stores bits 16–23 of the expression result.

hhi8 stores bits 24–31 of the expression result.

lo16 stores bits 0–15 of the expression result.

hi16 stores bits 16–31 of the expression result.

offset8 stores the offset of the expression result relatively to its section start.

offset16 stores the offset of the expression result relatively to its section start.

Functions ending with *8* produce an 8 bit result and may only be used in places where 8 bit values can be stored; 16 bit functions may only be used at places where a 16 bit value can be stored. It is not possible to store a *lo8* in a 16 bit word.

The syntax of the assembler operations is not the Motorola syntax:

- The name of the instruction must be written in lower case.
- For bit operations, the bit number is appended after a period to the instruction name.
- The X register is written as *%X* to avoid confusion with a symbol called *X*.
- The *%X* is written in front of the offset, instead of behind.
- An immediate value is indicated by a *$*.
- Hexadecimal numbers are written as *0xABCD*, as usual in *gas*.

Conditional jumps are automatically expanded if they do not fit a PC relative relocation or the target is unknown.

BSR, BRA and BRN should not be used (use JSR, JMP and NOP instead).

In section names ending in *!!!!!*, this part is replaced by a unique number for the assembler run. This is used to put symbols in different sections to aid the section movement code of the linker.

Some examples:

```
.section test.!!!!!
test:
lda %X,2
lda %X
sta test
lda $2
lda $0x10
brset.0 test1, test
add $offset8(test)
.byte lo8(test)
.hword offset16(test)
```

3.7. GNU linker

The GNU linker (*ld*) [LD, LDINT] is based on *bfd*, which is used to process all executables. Normally, no or very little architecture specific code is needed.

The linker is controlled by a linker script, which determines in which order and at which addresses the various parts of the object files should be placed.

For the M68HC05 target, a generic linker script is not possible. The default script works for the simulator. BCU specific scripts were not placed in the *ld* distribution because they depend closely on the BCU runtime environment.

The script is basically a default elf linker script. It creates an additional section named *.data.lo* for storing the pseudo register of GCC at an address lower than 0x100.

The only C code added to *ld* is the one necessary for section movement. This code can be easily dropped. It is needed for the BCU SDK to support the distribution of variables over different data segments.

The section movement code is activated by the *--move-sections* command line switch of *ld*. Its operation is controlled by a number of *SYSLIB* directives.

A clean solution would have been to introduce a new syntax for it in the parser. However, this would have the disadvantage that merging the patches from upstream would become more complicated.

Therefore an unused hook, named *SYSLIB*, was used. A command has the following syntax:

```
SYSLIB([from-section]:[to-section]:[current-symbol]:[maximum-symbol])
```

For each SYSLIB directive, the section movement code performs the following (before the relaxation):

- The values of the current and maximum size symbols are evaluated.
- If the current size is smaller than the maximum size, the processing of this directive is finished.
- If the from-section is empty, the processing is finished.
- A section of the from-section is selected (at the moment the biggest section is selected).
- The section is moved to the to-section and the process is repeated.

A better implementation of this code, which examines all directives at the same time, would generate better results. As the typical variable size used in BCU programs will probably be around 2 bytes and therefore mostly 1–2 byte sections will be created, the results should not be too bad.

3.8. Sim

GDB contains a collection of simulators for different architectures, which can be either used as stand alone programs or inside gdb.

They consist of a common core and target specific supplements. The core provides a framework to load programs, register and access virtual devices as well as some virtual devices like RAM. Additionally, it is possible to configure the number and parameters of virtual devices with special command line parameters.

m68hc05-sim is based on the m68hc11 port. Because it is intended for regression tests, the simulation is not complete:

- All opcodes, except STOP, WAIT, SWI, RTI, BIH, BIL are simulated. The cycles counter is also implemented.
- The interrupt subsystem is not implemented (only the stubs exist).
- The CPU device only supports the reset command.
- If an ELF executable is loaded, the simulator starts at the entry point of the ELF file, otherwise at 0xFFFE.
- The unused opcode 0xA7 is used as breakpoint.
- No IO device and IO port of the m68hc05 processor is implemented.
- The default memory configuration consists of 64k of RAM.

By default, a minimal operating system is active, which uses the SWI instruction as entry point. If a SWI is executed, the content of the X register determines the action:

0 halts the simulator using the content of register A as exit code.

1 writes the content of register A to *stdout*.

2 reads a byte from *stdin* and stores it in register A. If an error occurs, the C flag is not set, otherwise it is set.

The core new part is the instruction interpreter. It uses the opcode list and generates the interpretation function. This function is a huge switch statement. For each opcode a case is generated. For each case, all addresses are generated and the parameters are fetched, according to the addressing mode. Then, the C statements to execute the instruction are generated according to the instruction name.

3.9. GNU debugger

The GNU debugger (*GDB*) [GDB, GDBINT] port is intended to step through programs for the simulator. As such, not all functionality is available.

The real registers can be accessed as *$a*, *$x*, *$pc*, *$ccr* and *$hwsp*. The virtual registers used by GCC are accessible as *$RegB* to *$RegN*, *$SP* and *$FP*.[2]

To find a virtual register, gdb needs the symbol table of the program. For *$SP* and *$FP* the complete address of the stack location instead of the actual 1 byte value.

The largest part of the gdb port is the stack frame handling. In gdb each stack frame on the call stack has an identification, created from two numbers: stack pointer and program counter.

[2]See Chapter 4 for details about virtual registers.

For each stack frame, a data structure is created which contains the value or address of all saved registers. With this information, it is possible to evaluate expressions in the context of one of the callers of the current function.

Most ports extract the necessary information by scanning the prologue of the function. For the m68hc05 this is not done because the prologue generated by GCC is generated as high level RTL code, which is converted into low level RTL code and passed to an optimizer. This makes the recognition of the prologue code very difficult and the determining the actions of the code is even more complicated. Because the prologue is not analyzed, evaluations in non-top stack frames might give wrong results.

There is no easy solution for this problem, since future optimizations might move the prologue code to another location. For example, the save of a callee save register can be moved just before its first usage. Likewise, the content of the SP register needs no change, if no called function uses the stack.

As the calling convention of GCC is supposed to be specific to a certain compiler version and therefore maybe incompatible between different versions, calling functions out of gdb is not supported.

A usage example:

```
m68hc05-gdb testprog
(gdb) target sim
(gdb) load testprog
(gdb) break main
(gdb) run
```

Note that the bss-section is not cleared and the data-section is not reloaded if *run* is executed a second time in a gdb process.

3.10. Newlib

The portable C runtime library newlib needs no big changes. Only the endianess of the m68hc05 architecture is declared in a header file and the compiler options are adjusted for the generation of small sized code.

3.11. Libgloss

Libgloss is a library which contains the glue code between newlib and the target operating system. An advantage of this concept is that for a slightly modified target (e.g. another syscall entry), only a small part has to be changed.

For the m68hc05 port, libgloss allocates the virtual register for GCC, contains the startup code and wraps the call to the emulated operating system of the simulator (read, write, exit). For other POSIX compatible functions, which are expected by newlib, a dummy implementation is included.

If a program for the simulator is linked, the command line options *-lsim -lc -lsim* must be added because of the circular dependencies between libgloss and newlib.

4. GCC

Porting GCC [GCC, GCCINT] to the M68HC05 architecture turned out to be a complicated task. GCC is designed to work with architectures with many registers, stack and unrestricted memory access. The M68HC05 has only two registers, the stack is unusable for user programs and has no 16 bit pointer. Thus, the missing features need to be emulated.

While designing the GCC port, the constraints of a BCU were used as the design driver. It can be used for any member of the M68HC05 microcontroller family, if enough memory for the stack and virtual registers is available in appropriate memory regions.

4.1. Structure of GCC

GCC consists of several target independent language front ends, a language and target independent optimizer and a code generation back end.

At the moment, GCC uses three different internal representations:

GENERIC For representation, GCC defines the C type *tree*. It is generated by the front end and passed to the new RTL independent optimizer. The details of this representation are only relevant for the creation of a front end.

GIMPLE This is a subset of the *GENERIC* representation. Using this representation, different optimizations are made. This is a new representation which was added for GCC 4.0.

RTL (*Register Transfer Language*) This representation is used to store the program in a more target specific form. For representation, GCC defines the C type *rtx*. The syntax used to display RTL statements is similar to LISP. RTL uses registers (and memory, if necessary) to store values.

If a file is compiled, the following happens:

1. The file is parsed by the language front end, the syntax and semantics are checked and a GENERIC representation is created.
2. The GENERIC tree is converted into a GIMPLE tree.
3. Some optimizations are done on the GIMPLE tree.
4. The GIMPLE tree is converted to RTL.

5. Some optimizations are done on the RTL.
6. The register allocator/reload pass runs on the RTL.
7. More optimizations and machine dependent passes are done on the RTL.
8. The RTL is converted to an assembler file.

The assembler file is run through an assembler and eventually linked, but these steps are done only by the GCC front end program.

The compilation normally is done separately for each global element, such as a function or global variable. If global optimizations are turned on, the global optimization at tree level collects the tree representation of the whole file, does its work and then runs the remaining compilation process for each element separately.

In the following, issues which are not relevant for the present project, such as general optimizations or language front ends, are not discussed. Details about them can be found in the GCC documentation [GCCINT] and in the source files. [ASU86] provides an introduction to compilers.

4.2. RTL

A function is represented as a list of RTL instructions. Examples are:

insn A normal instruction

jump_insn An instruction, which can be a jump or a conditional jump

call_insn A call to a function

code_label A label

Each instruction contains a description of the effect and some additional information. The effect is described by the following patterns:

(set LVAL VAL) sets LVAL to the content of VAL.

(clobber X) indicates that the content of X is destroyed.

(use X) indicates that X is used. Related calculations must not be removed by optimization.

(parallel list) states that each instruction in the list is calculated at the same time.

(call function args) represents a function call. If the return value is used, it is used as VAL in a *set* pattern.

(return) represents a return statement.

(unspec ...) is used to represent target specific patterns.

Each expression has a *mode*, which describes the size and type of the expression. Important ones are:

QI A 1 byte integer

HI A 2 byte integer

AI A 3 byte integer (port specific)

SI A 4 byte integer

FI A 5 byte integer (port specific)

CI A 6 byte integer (port specific)

EI A 7 byte integer (port specific)

DI A 8 byte integer

SF A 4 byte floating point value

DF A 8 byte floating point value

Expressions can be:

(const_int N) The integer constant N.

(const_double:MODE ...) Depending on the mode, this is either a floating point constant or a very long integer constant.

(symbol_ref:MODE SYMBOL) Used to represent the address of SYMBOL.

(label_ref LABEL) Reference to a code label.

(reg:MODE N) Represents a register. If N is smaller than FIRST_PSEUDO_REGISTER, a real register is described, else a pseudo register, which needs to be changed to a real register or to a stack location.

(subreg:MODE expr byte) Extracts a part of size *MODE* of *expr* starting at *byte*.

(cc0) Used to reference to condition codes.

(pc) Used to reference to the program counter.

(mem:MODE addr) Refers to the memory location *addr*.

(if_then_else COND THEN ELSE) returns depending on *COND THEN* or *ELSE*. *COND* can be e.g. *ne*, *ltu* and *eq*, which takes two expressions.

(sign_extend:MODE expr) Extends *expr* to the size of *MODE*, keeping the sign bit.

(zero_extend:MODE expr) Extends *expr* to the size of *MODE* by filling up the high order bits with 0.

Additionally there are arithmetic and logic operations like *plus*, *minus*, *mult*, *ior*, *and*, *compare* and *ashift*, which take two expressions. Unary operations like *neg* or *not* also exist.

Examples of RTL statements are[1]:

- Compare the content of register 42 with 0 and set the condition code register.

```
(insn 27 68 28 2 (set (cc0)
        (compare (reg:QI 42)
            (const_int 0 [0x0]))) -1 (nil)
    (nil))
```

- Jump to label 33, if the condition code register fulfills the condition lower or equal.

```
(jump_insn 28 27 69 2 (set (pc)
        (if_then_else (le (cc0)
                (const_int 0 [0x0]))
            (label_ref 33)
            (pc))) -1 (nil)
    (nil))
```

- Jump to label 26

```
(jump_insn 31 30 32 3 (set (pc)
        (label_ref 26)) -1 (nil)
    (nil))
```

- Extract the low byte of register 32 and store it in register 42.

  The [*r*.5] states the high level variable whose value is stored in the register (register 32). In this case, *r.5* is a variable generated by an optimizer, whose value is based on *r*.

```
(insn 25 24 26 1 (set (reg:QI 42)
        (subreg:QI (reg:HI 32 [ r.5 ]) 1)) -1 (nil)
    (nil))
```

- Store the content of the memory location at symbol *r* in register 32.

```
(insn 23 22 24 1 (set (reg:HI 32 [ r.5 ])
        (mem/i:HI (symbol_ref:HI ("r") <var_decl 0xb7e9c288 r>)
                                    [0 r+0 S2 A8])) -1 (nil)
    (nil))
```

[1] Details are not represented here. For the precise meaning of the various elements refer to [GCCINT]

- Add the constant 20 to register 28 and store the result in register 27.

```
(insn 52 51 53 4 (set (reg:HI 27 [ D.1137 ])
        (plus:HI (reg:HI 28 [ y.7 ])
            (const_int 20 [0x14]))) -1 (nil)
    (nil))
```

- State that the content of register 9 (which is called RegB and in this case not a pseudo register) is no longer valid.

```
(insn 61 67 62 5 (clobber (reg/i:HI 9 RegB)) -1 (nil)
    (nil))
```

- Call the function *eestore_HI*. As a side effect the contents of register 12 and 10 are used.

```
(call_insn 15 14 17 1 (call (mem:HI (symbol_ref:HI ("eestore_HI")
                                        [flags 0x41]) [0 S2 A8])
        (const_int 0 [0x0])) -1 (nil)
    (expr_list:REG_EH_REGION (const_int 0 [0x0])
        (nil))
    (expr_list:REG_DEP_TRUE (use (reg:HI 12 RegE))
        (expr_list:REG_DEP_TRUE (use (reg:HI 10 RegC))
            (nil))))
```

4.3. Machine description

Most of the work done with the RTL is done by pattern matching. The patterns and actions used are described in the *machine description*. This file uses also a LISP like syntax.

Many patterns have names. Unofficial names start with * and are only used when an RTL statement is printed. They are intended for debugging. Other names, like *movqi* are expected by GCC and must perform a certain function.

The most relevant patterns are described in the following.

4.3.1. Normal named instruction

```
(define_insn "jump"
 [(set (pc)
       (label_ref (match_operand 0 "" "")))]
"is_lowlevel()"
"jmp %0")
```

Define an instruction pattern named *jump*. The first element is the RTL pattern. The *match_operand* indicates a variable part. In most cases, it has a mode appended which must match the mode of the variable part. Then the number of the operand follows, which is used for referencing. The next element can contain the name of a predicate which the operand must match as a string. The last element contains constraints for the reload pass. If an operand must be exactly the same as another operand, *match_dup* is used instead of *match_operand*. If the pattern of the insn matches, the second element of the insn is evaluated. If it is true, the pattern matches. This expression can be empty. Then the output follows. This can either be a string or some C statements enclosed in { and }. In the output string, %modifier*number* (e.g. *%o0* or *%0*) stands for the textual representation of the operand with the specified number. The modifier is passed to the output function for operands. *%%* turns into *%*.

As a last optional part, the attributes of the instruction can be modified. Because the name does not start with ***, an expander is also generated.

4.3.2. Normal anonymous instruction

```
(define_insn "*cmpa"
[(set (cc0)
      (compare (reg:QI REG_A)
               (match_operand:QI 0 "regmemimmst_operand" "rB,Y,i")))
]
"is_lowlevel()"
"@
cmp %0
cmp %%X,(%o0+SPBASE)
cmp $%0"
[(set_attr "cc" "compare")])
```

In this example, the attribute *cc* is set to *compare*. The instruction specifies three alternatives in the *match_operand* part. The @ at the output string states that each of the following lines is the output pattern for an alternative.

4.3.3. Definition of an expander

```
(define_expand "movhi"
 [(set (match_operand:HI 0 "nonimmediate_operand" "")
       (match_operand:HI 1 "general_operand" ""))]
"!is_lowlevel()"
{
if(eeprom_operand(operand0,HImode))
  {
   rtx tmp=gen_reg_rtx(HImode);
   emit_insn(gen_movhi(tmp,operands[1]));
```

```
   emit_insn(gen_eestoreHI(operands[0],tmp));
   DONE;
  }
 if (memaddr_operand(operand0, HImode)
     && memaddr_operand(operand1, HImode))
    {
        rtx tmp=gen_reg_rtx(Pmode);
        emit_insn(gen_movhi(tmp,operands[1]));
        operands[1] = tmp;
    }
}
)
```

This defines an expander for movhi. An expander generates a function (in that case called *gen_movhi*), which takes all operands and returns the RTL. With that approach, each target has the possibility to return non standard RTL expressions for specific situations.

As for *define_insn*, the first element is the RTL pattern, then a condition can follow. The *match_operand* has the last element empty, because an expander cannot match an instruction and therefore is not used in the reload pass.

Then some C code (either as string or in enclosed in { }) follows, which can alter the RTL. In the array *operands* the matched operands are stored. Additionally they are accessible as *operand*number.

This example (out of the M68HC05 port) was chosen because it shows some ways to alter the result:

- If *operand0* matches the predicate *eeprom_operand*, a new temporary register is created. Then the expanders of *movhi* and *eestoreHI* are called and their result is appended to the output with *emit_insn*. Finally *DONE* says that we are finished and the default pattern may not be appended.

- If *operand0* and *operand1* match the predicate *memaddr_operand*, the content of *operand1* is moved to a temporary register and operand 1 is replaced with it. In this case the default pattern will be appended, but with a different operand.

- In any other case, the default pattern is appended.

4.3.4. Definition of constants

```
(define_constants [
(REG_A 0)
(REG_X 1)
(REG_AG 5)
(REG_SP 7)
(REG_FP 8)
])
```

Define_constants does what one would expect. It defines that e.g. REG_A can be used instead of 0.

4.3.5. Definition of attributes

```
(define_attr "cc" "none,set_czn,set_zn,set_n,compare,clobber"
  (const_string "none"))
```

Defines an attribute named *cc*, which can be one of the values listed in the second argument.

4.3.6. Definition of a combination of instruction and splitter

```
(define_insn_and_split "cmpqi"
  [(set (cc0)
        (compare (match_operand:QI 0 "regmempt_operand" "rAQT")
                 (match_operand:QI 1 "regmemimmst_operand" "rRBUi")))
   ]
"!is_lowlevel()"
"error"
"is_lowlevel()&&(!is_rega_rtx(operands[0]))"
[
(match_dup 2)
(set (reg:QI REG_A) (match_dup 3))
(match_dup 4)
(match_dup 5)
(set (cc0) (compare (reg:QI REG_A) (match_dup 6)))
]
{
operands[2]=genXLoad(operands[0],QImode,0);
operands[3]=genSUBREG(operands[0],QImode,0,QImode);
operands[4]=genXClobber(operands[0],QImode,0);
operands[5]=genXLoad(operands[1],QImode,0);
operands[6]=genSUBREG(operands[1],QImode,0,QImode);
}
[(set_attr "cc" "compare")])
```

This pattern is a combination of a *define_insn* and a split pattern. The first, second, third and last argument are the same as for a *define_insn*.

The fourth argument specifies a condition which must be fulfilled before the instruction can be split into other instructions. A split is only done at certain passes of the compiler.

The next argument specifies the output pattern of the split. *match_dup* indicates the places where an operand should be inserted. Then a preparation statement follows, which can modify existing operands or calculate new ones.

4.3.7. Peephole optimization

define_peephole2 is used to describe a peephole optimization. It basically does the same as a split pattern, but is called at a different time and for a different purpose and it works on multiple instructions.

4.4. Libgcc

For functions which GCC needs, but the target does not support directly, functions in the GCC library must be created. The default set consists of, among others, a floating point emulator, functions to calculate with 64 bit integers and exception handling functions. This library is automatically linked to a program when it is linked by GCC.

4.5. Target description

This part contains the rest of all definitions and functions to make GCC work. It consists of a header file and a C file. The header file almost exclusively consists of various *define* statements related to the target. The C file contains all functions which are used in the header file. It also contains target specific predicates. Additionally, the GCC core exposes hooks for certain functions, which are also set and implemented in the C file.

The target description includes:

- target specific compiler options
- defines for the compiler and flags to pass to the linker and assembler
- storage layout and type sizes
- available registers, usage restrictions as well as their assignment to register classes
- stack layout, handling and the usage of stack related registers
- calling convention
- output of data structures and assembler operands
- handling of trampolines (memory structures for calling nested functions via pointers)
- supported memory addressing modes
- handling of the condition code register
- instruction costs

4.6. Overview of the M68HC05 port

Porting GCC to the M68HC05 architecture was not a trivial task. A lot of different approaches were tried until the current solution was found.

First experiments started with GCC 3.3.3, later GCC 3.4.3 was used. Finally the whole code was adapted to the latest development (CVS) version. Maintaining the old branches was stopped, because the CVS version proved to be bug free enough. Thus, it was planned to release the final version of this port based on a working CVS version. Besides, keeping up with the old versions was too much work.

The C++ language front end of GCC for this port is compilable. It can be used to compile C programs with the better type checking of C++ (but with C++ features like exceptions and RTTI disabled). If used this way, it should work without any problems. The C++ runtime library is too big and therefore not available.

In addition to the physical register, RegB and RegN are added as virtual (or soft) registers. The frame and stack pointer are also defined as virtual registers. Because GCC expects its internal frame and argument pointer to point to specific positions in the stack frame, a virtual argument and frame pointer are also defined. They are replaced with the real stack and frame pointer as soon as possible. The locations of these registers must be allocated by the user. If multi-byte values are stored in registers, GCC allocates some contiguous 8 bit registers.

As the call stack is too small and the call stack pointer is inaccessible, a data stack is simulated by GCC. The segment where the data stack is located is described by the symbol *SPBASE*. This must be defined by the user. As the stack pointer for this stack is one byte large, *SPBASE* to *SPBASE+0xFE* are used for the stack. The initial value of the stack pointer determines the start of the stack. Theoretically, the stack could start at every location. This has the disadvantage that every conversion of the stack pointer into a normal pointer (and also the other way round) needs a 16 bit addition, which needs about 6 instructions. To avoid this, SPBASE must start at a 256 byte boundary. The stack grows downwards.

The first stack slot used can have any address in it, but if it is not 0xff, only a smaller stack is available. As a contiguous RAM section of the BCU is less than 128 bytes, this is no limitation. If a stack location is referenced, the stack pointer is loaded into the X register. Then the memory at *(SPBASE+constant offset)* is accessed, using X as the index register.

In the first versions, all RTL instructions directly output assembler code. To support all addressing modes for two operand expressions, often more than 30 alternatives were needed to support all combinations of operands. A second disadvantage of the concept was that e.g. an addition of two 4 byte values needed about 12 instructions in one output template. If the stack was used, the X register was used to temporarily hold the stack pointer. Because the output statement has no knowledge of the content of the X register, the stack pointer was reloaded too often. This is in contrast to the need for small code size.

So a different solution was developed. Initially, a high level RTL is generated, which only works on pseudo registers and virtual registers. Additionally, the addressing for the

stack is done relative to the stack pointer. After the register allocation/reload pass in the machine specific reorganization pass, the RTL is converted into the low level RTL.

In this representation, each RTL insn represents a real instruction. An addition of two 4 byte values is converted into the 12 statements needed plus all necessary reloads and clobber statements of the X register. Over this representation the optimizer is run again, which eliminates all unnecessary load and store operations.

GCC expects pointers which can cover the whole address space. With the M68HC05 architecture, the problem occurs that only an 8 bit index register is available. So the store, load and call operations with 16 bit pointers have to be emulated. The first solution was to use a table. The upper 8 bits of the pointer were used to jump to a statement in this table which loads the lower 8 bits into the X register. The statement then executes the operation with the hard coded pointer, which equals the upper part of the pointer. An advantage of this approach is that none (if the load operation is duplicated for each possible pointer location) or only up to three bytes of RAM are used. The disadvantage is that a lot of code is needed.

The current solution works with self modifying code. In the RAM, four bytes (plus one to save data for the store operation) are reserved. If a pointer operation happens, the opcode of the corresponding instruction with the IX2 addressing mode is stored at the first byte, then the content of the pointer and finally a RTS. A jump to the RAM region starts the operation.

If multi byte values are accessed with a pointer, the offset relative to the pointer is stored in the X register, so that no extra address calculations are needed.

The GCC floating point simulator for single and double precision values is compiled into libgcc. However, many of its functions need too much memory, which causes them to fail on a BCU. Some functions even need more stack size than the M68HC05 GCC port supports.

Multiplications and divisions not supported by the hardware are also emulated. In some situations GCC will expand an unsupported multiplication to a combination of supported instructions.

A feature that was completely left out is *setjmp*/*longjmp*, as it cannot be implemented in an efficient way. Already saving the virtual registers would need a lot of memory. The main problem however is the call stack, for which the stack pointer is inaccessible. An implementation of the *setjmp* function would need to issue a call (or a sequence of calls) from known locations and then search these values in the call stack area. The address of the first known address plus 2 would be the call stack pointer value before the call.

longjmp would need to save the used stack location (either the current stack pointer could be determined by the previous method or the whole possible stack would have to be saved), then the stack pointer needs to be reset. After that recursive calls are made until the saved stack pointer value is reached. Finally the content of the stack can be restored.

As a consequence, an exception handling system is also impossible to implement, because it requires mechanisms similar to *setjmp*/*longjmp*.

Another known limitation is that ignored overflows are possible even for signed compares. This is because internally a subtraction is made. As the M68HC05 has no overflow

type	size in bit
char	8
short	16
int	16
long	32
long long	64
single	32
double	64
void*	16

Table 4.1.: Type sizes

detection mechanism and its simulation would enlarge the code, it was decided to ignore such overflows. If GCC optimizes the compare in another way, a correct behavior is possible.

As an experimental option, *–mint8* is present. It changes the size of the type *int* to 8 bits, but keeps the size of the type *short* of 16 bit length. This is a clear violation of the C standard and may cause faults in GCC. It is a workaround to stop the automatic promotion of 8 bit values to 16 bit values.

The main problem, where the promotion cannot be prevented, are 8 bit return values. As this promotion is not done for library functions automatically, it would be impossible to write some of them in C. To solve the problem, the register normally holding the low byte of the return value is used instead of the normal return register if an 8 bit value is returned by a library function. A better solution would be to disable the promotion of the return value in *start_function*, which makes GCC incompatible with the C standard.

To use the non-standard integer sizes, define a new type with the specific mode:

```
typedef signed int sint7 __attribute__ ((mode (EI)));
typedef unsigned int uint3 __attribute__ ((mode (AI)));
```

which defines a 7 byte integer type named *sint7* and a 3 byte unsigned integer type named *uint3*.

4.7. Details

4.7.1. Type layout

All types are stored in big endian format. All values are stored packed with no additional alignment bytes. For the type sizes see Table 4.1.

4.7.2. Register

The general virtual general purpose 8 bit registers are called *RegB*, *RegC*, *RegD*, *RegE*, *RegF*, *RegG*, *RegH*, *RegI*, *RegJ*, *RegK*, *RegL*, *RegM* and *RegN*.

name	purpose	content	constraint
NO_REGS	required by GCC	no register	-
A_REGS		A	a
X_REGS		X	x
STACK_REGS	stack pointer	FP, SP	w
GENERAL_REGS	general purpose register	RegB – RegN	r
POINTER_REGS	usable pointer	RegB – RegN, FP, SP	b
REGS_B		RegB	z
REGS_C		RegC	c
REGS_D		RegD	d
REGS_E		RegE	e
REGS_F		RegF	f
REGS_H		RegH	h
REGS_BC		RegB, RegC	q
REGS_DE		RegD, RegE	t
REGS_FG		RegF, RegG	u
REGS_HI		RegH, RegI	v
REGS_KLMN		RegK – RegN	y
ALL_REGS	required by GCC	all register	-

Table 4.2.: Register classes

The name of the hardware registers are *A* and *X*. They (and therefore their register classes) should never be used as input or output registers in any user assembler statement. This is because the compiler does not know that these registers are implicitly used by the high level RTL while the register allocator runs. If a value is needed in one of these registers, tell the compiler to put the value in a general purpose register, and do the load/store from/to them in the user assembler code. All hardware registers may be clobbered by a function call.

The data stack pointer is called *SP*, the frame pointer *FP*. Besides QI mode, these registers can also be accessed in HI mode, although only one byte is reserved in memory for them. The HI references are eliminated in the low level RTL.

4.7.3. Register classes

GCC groups registers which have a common purpose in register classes. For a list of the available register classes see Table 4.2.

4.7.4. Pointer

For pointers, only addresses in a register in the register class *POINTER_REGS* are supported. Internally, they are implemented as GCC base registers. The GCC index

registers are not used.

A legitimate pointer expression is either a pointer register or a constant plus a pointer register. If a non-stack register is used as base, the offset must be 0 for the high level RTL and between 0 and 7 for the low level RTL. The offset in the low level RTL is used to access sub-bytes of multi byte values. As the largest supported type is 8 bytes large, the offset is limited to 7.

4.7.5. Calling convention

The return value starts at *RegB*. The first argument is stored in *RegB* and all small values are stored in the following registers. The last usable register for that purpose is *RegK*. Larger values are passed on the stack or by pointer.

RegM – *RegN* are used as static chain register, if it is needed. As this chain register is not implemented as a stack register, normal pointer operations are used, which results in bigger code size.

RegB – *RegE* are callee saved registers, *RegF* – *RegN* are caller saved registers.

A function with the *nosave* attribute does not save any register in its prologue.

Note: The calling convention may be changed in future versions of this GCC port.

4.7.6. Stack frame

When a function call is made, a stack frame contains the argument on its top. If the frame pointer is used by the function, it is saved at the next location. Then all used callee saved registers follow.

4.7.7. Frame pointer elimination

The port provides all necessary information to replace the use of the frame pointer with the stack pointer, if it is possible. The functions *initial_elimination_offset* calculates the needed offset.

4.7.8. Sections

Code is placed in the *.text* section, read-only data in the *.rodata* section.

Each global or static variable is placed in a section which starts with *.bss.* (initialized with 0) or *.data.* and ends with a unique number. Internally the section name ends with *!!!!!*, which is replaced with a unique number by the assembler.

For common symbols, the ELF COMMON section is not used, because multiple COMMON sections are not possible. Instead they are allocated as normal variables in a bss section.

4.7.9. Constraints

All supported constraints are listed in Table 4.3.

Constraints are used by the reload pass to determine if a variable fits a certain location. If an operand in the RTL is specified, it contains a constraint string, possibly with multiple alternatives, separated by a comma.

Each constraint can include some modifiers. The important ones are:

& for early clobber, which means that this operand will be (partially) written before all source operands are read. Therefore it may not overlap with another operand.

? means that this alternative is more expensive and should be avoided.

= means that a value is written to this operand.

If an instruction has multiple operands, they must have the same number of alternatives. The reload pass examines the first alternative of all operands, then the second, and so on. Finally, the cheapest one is selected and pseudo registers are either bound to a real register or to a stack location. This process is repeated until a sufficient solution is found.

A GCC port can define its own constraints. Apart from a general constraint, which is either fulfilled or not, address and memory constraints are possible. A memory constraint will also match, if the operand can be placed in memory.

For the M68HC05 port, some constraints exist as normal constraints as well as memory constraints (e.g. *A* and *B*). This is caused by the fact that a general memory operand is not supported in many situations. In these situations, the normal constraint is used.

If the stack is also supported, the *U* constraint is added. The *R* constraint is also added, because otherwise GCC will fail in some situations. The problem is that GCC keeps open the decision whether a pseudo register should be put on the stack in some cases. Therefore the stack constraint does not match. As the other constraints do not allow a value to be put on the stack (they are not memory constraints), GCC concludes that the value must be in a register. If it runs out of registers, the compilation fails.

The *R* constraint causes that, in such a case, the constraint matches if the value can be put on the stack only while the reload is in progress. At the end of the reload, the value is either put in a register, which causes another constraint to match, or is put on the stack, which causes the *U* constraint to match.

In situations, where pointers are possible (*Q* constraint), memory constraints are used, as this is a cleaner solution.

4.7.10. Operands

Operands in the assembler templates are specified as %modifier*number* (e.g. *%o0*), where number stands for the number of the operand.

This port defines two modifiers:

o Print only the offset for a memory location at x+offset or x (where x can be register SP, FP or X).

constraint	purpose
0, 1, 2, 3, 4, 5, 6, 7, 8, 9	the same as operand with the this number
a, b, c, d, e, f, h, q, t, u, v, w, x, y, z	register classes, see Table 4.2
i	immediate value
m	any memory operand
r	register of GENERAL_REGS
A	fixed memory location (memory constraint)
B	fixed memory location
C	loram memory location
D	lowlevel loram pointer (X or X+constant)
F	immediate floating point value
G	immediate double with value 0
M	immediate between 0 and 0xff
N	immediate value 1
Q	valid base pointer expression (memory constraint)
R	accept registers which can be reloaded on the stack in the reload process
S	stack pointer
T	memory on the stack (memory constraint)
U	memory on the stack
W	memory on the stack using the frame pointer
Y	memory at register X plus constant offset
X	anything
Z	frame pointer
<, >, g, n, o, p, s, E, H, I, J, K, L, O, P, V	not of interest or reserved by GCC

Table 4.3.: Constraints

A Print the second part of a double as a hexadecimal number.

The function is implemented in *print_operand*, which supports the following constructs:

- offset of memory at x+offset or x with code o (where x can be register SP, FP or X)
- a register as its name
- an integer constant
- a single floating point value, encoded as a hexadecimal number
- the upper or lower part of a double floating point value, encoded as a hexadecimal number
- a specific byte of a floating point constant
- an address

If an address is a register, its name is printed. If it is a register plus an offset, the register name is printed, appended with '_' and the offset. If a byte of an address is requested, the address is wrapped in a call of the corresponding byte access function of the assembler. The rest is printed in the normal GCC way.

4.7.11. RTL split helper functions

Three functions are used to split the high level RTL in low level RTL:

- genXLoad takes an operand and determines if it makes use of the stack. If this is true, a load of the X register with the stack (or frame) pointer is returned, else a NOP.
- genXClobber returns a clobber of register X, if the operand needs to call a library function which destroys the content of the X register, else it returns a NOP.
- genSUBREG takes an operand and returns an operand which corresponds to a selected byte of the operand. For constant addresses, special RTL code is returned, which is turned into a *lo8*, *hi8*, *hlo8* or *hhi8* function by the *print_operand* function.

  If the stack is referenced, the use of the stack pointer or frame pointer is replaced with the X register.

How an addition of a long value is split is shown in the following. For each byte (processing from the low to the high byte), the following output is produced:

- genXLoad(operand[1],byte)
- LDA of genSUBREG(operand[1],byte)

- genXClobber(operand[1],byte)
- genXLoad(operand[2],byte)
- ADD/ADC of genSUBREG(operand[2],byte)
- genXClobber(operand[2],byte)
- genXLoad(operand[0],byte)
- STA of genSUBREG(operand[0],byte)
- genXClobber(operand[0],byte)

The NOPs are eliminated immediately after the split. The resulting code may contain redundant reloads of the X register. They are eliminated by a GCC optimization pass, if optimizations are turned on.

4.7.12. RTL patterns

The machine description, as well as the source file for a part of libgcc, are generated by a PHP script. Every script language could have been used, PHP was chosen out of personal preference. For a normal GCC build, the generated files are present, so PHP is not needed. It is only needed if changes are made to the generation code.

The reason why a script language is used is that each pattern exists in many variants, which only differ in the supported mode and the number of instructions created in split definitions. They are also highly repetitive, so using a scripting language simplified their maintenance.

The following functions are defined:

genexpandshift generates an expander for a shift operation. It outputs a *for*-loop, which performs a one-bit shift for *shift count* times.

If the shift is a logic shift, and the shift count is a constant which is a multiple of 8, the shift will be turned into a sequence of move operations.

gensplit_move defines a splitter, which turns a move operation into a sequence of move operations for each byte.

gensplit_binop_nocc defines a splitter which performs a binary operation on each byte of the operands. It is intended for operations which do not need to pass information in the carry flag.

gensplit_binop_cc defines a splitter which performs a binary operation on each byte of the operands. It is intended for operations which must pass information in the carry flag.

gensplit_shift splits a shift/rotate operation in byte-wise shift and rotate operations.

asm_op outputs the definition of a low level instruction which uses A as a source and the destination operand.

asm_shift outputs the definition of a low level shift/rotate operation.

expand_neg outputs a splitter which turns a negation into byte-wise operations.

expand_zeroextend outputs a splitter for zero extension.

expand_extend outputs a splitter for a sign extension.

asm_jmp outputs an instruction pattern for a conditional jump.

expand_eestore outputs an expander which creates a library call to store a value in the EEPROM.

expand_move creates an expander which checks if the destination operand is in the EEPROM (*eeprom* attribute present). If this is true, an EEPROM store operation is created. Otherwise a normal move pattern is generated.

4.7.13. Predicates

The port specific predicates match:

regmem_operand A register or memory at a fixed location, which can be determined at least at link time.

regmemne_operand A register or memory at a fixed location, which can be determined at least at link time. This predicate rejects memory references if the pointer contains the *eeprom* attribute.

regmemimm_operand A register, an immediate value or a memory at a fixed location, which can be determined at least at link time.

regmempt_operand a register or memory reference supported by the back end.

regmemptne_operand A register or a memory reference supported by the back end. This predicate rejects memory references if the pointer contains the *eeprom* attribute.

regmemimmpt_operand A register, an immediate value or memory reference supported by the back end.

regmemimmst_operand A register, an immediate value or stack reference supported by the back end.

4.7.14. Cost functions

For the cost calculation of addresses, fixed addresses have no real cost. References to the stack are a little bit more expensive and the use of pointers has a significant higher cost. This causes that the use of pointers in addresses is avoided whenever possible.

For normal RTL expressions, the cost calculation is similar to the one for addresses: constants, registers and fixed memory locations have no costs. Stack locations are more expensive and the use of pointer registers is significantly more expensive. This cost is further multiplied by the size of the accessed value to represent that e.g. a move in HI mode needs more instructions than in QI mode.

4.7.15. The *eeprom* attribute

As a new experimental feature, the *eeprom* attribute was added. This attribute is intended to access memory in the EEPROM in a transparent way. If an pointer points to such an EEPROM location, it has the attribute *eepromt* added.

This attribute can be added to a variable declaration. If a value is stored to a memory location which has the *eeprom* attribute or the pointer type has the attribute *eepromt* (which means that the pointer target is in the EEPROM), a routine named *eestore*_mode is called. Here, *mode* can be QI, HI, SI, DI, CI, EI, FI, AI, SF or DF depending on the size of the value. As the first parameter, the pointer to the EEPROM location is passed and as the second the value to be written.

ISO/IEC TR 18037 named address spaces	m68hc05-gcc address spaces
	#define _eeprom __attribute__ ((eeprom)) #define _eepromt __attribute__ ((eepromt))
_eeprom char a;	_eeprom char a;
_eeprom char* b;	_eepromt char*b;
_eeprom int*_eeprom c;	_eepromt int*_eeprom c;

Figure 4.1.: Comparison of ISO/IEC TR 18037 named address spaces with m68hc05-gcc address spaces

The semantic is similar to the named address spaces of [CEXT], but uses two GCC attributes instead of one directive for each named address space. Using define statements for attributes, it can be used like in [CEXT], with the difference that the user must select if the variant for the target of a pointer or for a variable is to be used. This attribute does not care if a variable is actually allocated in an EEPROM section nor does it check if a pointer actually points to the EEPROM.

Implementing this function was not trivial. The first attempt was to make minimal changes to the GCC core. Only the *eeprom* attribute was used for all purposes. Mak-

ing the attribute known to GCC was simple, in fact it only has to be added to the attribute list. Then all internal GCC code infers the correct type for many *tree* based representations, keeping the *eeprom* attribute, if it is necessary.

A drawback was that this approach was not sufficient for structures (in conjunction with pointers). The attribute needed to be propagated to all elements of a type. Implementing this propagation was not trivial and caused a lot of problems.

Because GCC by default offers no way to implement back end functions on *tree* level, this must be done at RTL level. If the type points to a fixed location, GCC stores the type in the memory reference. In that case, a MEM-expression has only to be checked for the *eeprom* attribute. If this is present, the store operation is replaced by a function call. All other functions except the move instructions reject eeprom types, so only the move expanders have to be updated.

The problem was the pointer arithmetic. In such a case it is possible that GCC splits a pointer expression in many RTL statements while generating the RTL. Finally, the pointer is stored in a pseudo register, but without any type information. Therefore such references were missed. Thus, the code which stored the expression in some memory references was changed to always store the expression. After a regression test, this change seems to be compatible with the GCC core. Yet, it caused some assertions in the debug print functions to fail. This was fixed by printing the root node of the tree in these cases.

After presenting this solution on the GCC mailing list and some discussion, a new solution was written. Instead of storing the type (or the expression) in the RTL, *MEM_REF_FLAGS* were introduced. If a tree expression is associated with a MEM RTL expression, the back end calculates the flags in a target hook. The *eeprom_operand* predicate only needs to check if a bit is set in the *MEM_REF_FLAGS*.

Instead of propagating the *eeprom* attribute, a named address space attribute (*MEM_AREA*) was added to the tree type. For pointers, it stores the address space of the pointer target. For variables, it stores the address space where they are located. This attribute is propagated in expressions which can potentially refer to memory. Much of the propagation is done automatically, while a tree expression is created. Only for dereferencing pointers more work is necessary.

Additional target hooks for named address space compatibility checking as well as address space merging were added. In the back end code, the two C attributes were introduced, because GCC does not support to use one attribute for both purposes.

Named address spaces are also of interest for other targets, like the *AVR* microcontroller family.

4.7.16. The *loram* attribute

For accessing the low RAM section, a second named address space is used. It uses the attributes *loram* and *loramt* (similar to *eeprom* and *eepromt*). The use of this attribute results in smaller code in many situations, as normal load/store operations can be used instead of the self modifying load/store. In this case, the X register is used to cache the pointer.

Pointers to the low RAM are automatically converted into normal pointers, if necessary. A normal pointer may only be converted to a low RAM pointer (with an explicit conversion), if it really points to the appropriate memory region (0x000 – 0x0FF).

This attribute does not yet produce optimal code.

Part II.

BCU/EIB

5. BCU operating system

A BCU is a standardized device which is connected to the EIB bus and can load an application program. It contains the *physical external interface* (*PEI*), where application modules can be connected to.

As a second variant, *BIMs* exist, which only contain all electronic parts on a small circuit board (without housing). They offer the same possibilities as a BCU, but necessary elements like the programming button have to be connected separately. Concerning the software interface, a BIM is identical to a BCU with the same mask version. The main difference is that since BIMs are integrated into the housing of the final product, the PEI interface is typically not accessible. In the remainder of the text, the term BCU will also refer to compatible BIMs.

Although a BIM based on a M68HC11 is available, most BCUs (and BIMs) use the M68HC05 processor core. They have some RAM and EEPROM integrated. The EIB system software is contained in the ROM. In the EEPROM, the application program is loaded.

Each EIB device has two elements (apart from the bus connector): A programming mode button to turn programming mode on and off and a LED, which shows if the device currently is in programming mode.

5.1. Modes of communication

EIB supports both point-to-point and multicast communication. The respective addressing modes are referred to as *individual* or *physical* and *group* addressing. Individual addressing is only used for management and installation tasks. For this purpose, the protocol stack provides reliable connections as well as connectionless communication.

The exchange of process data is almost exclusively done via group addressing. Each group communication endpoint in a BCU is called *group object* or *communication object*. It can send and/or receive values on a certain set of *group addresses*. If a device sends a new value in a *A_GroupValue_Write* telegram, all other group objects which listen to this address update their state and may trigger some action.

In BCUs, all addresses are stored in an address table. It contains the single individual address and all group addresses used by this device.

5.2. BCU 1

Different mask versions of the BCU 1 exist. This section will cover the version *1.2*. Other mask versions share the same principles, but offer more or less features. This

PEI type	description
0	No PEI module expected
2	4 inputs, 1 output (LED)
4	2 inputs, 2 output + 1 output (LED)
6	3 inputs, 1 output + 1 output (LED)
8	5 inputs
12	serial synchronous interface message protocol
14	serial synchronous interface data block protocol
16	serial asynchronous interface message protocol
17	programmable I/O
19	4 output + 1 output (LED)
20	Download (reserved, may not be used in application programs)

Table 5.1.: PEI types

section will in most cases cover the BCU SDK interface rather than the plain assembler view of the interface.

The RAM and memory mapped IO is located between 0x000 and 0x100. For a user program, a block of 18 bytes is available. Additionally, the RAM contains the registers *RegB* – *RegN* and the *system state* at 0x060. If the application program is running, this location should contain a value of 0x2E. If a problem occurs (e.g. a checksum error), it can contain a different value. A stopped application program can be started by writing 0x2E to this location.

For the BCU SDK, there is no need to access any RAM location directly. If port A or port C are accessed directly, bit set/clear operations must be done using an inline assembler statement at the location 0x00 and 0x02. If only the standard PEI interface is used in a program, no direct port access is necessary.

The EEPROM has a size of 256 bytes and starts at 0x100. The first 22 bytes contain a header which describes the program and contains all entry points. Parts of the BCU EEPROM can be protected by a checksum. If a checksum error is detected, the user program is halted. This is controlled via the header.

A header byte of particular interest for the developer is 0x10D, the *RunError*. This value can be read during runtime. If an error condition happens, a bit is set to 0. Interpreting this value, some error conditions (e.g. a stack overflow, see [BCU1, BCU2]) can be detected. Another important value of the header is the expected PEI type. The user program only runs when the expected PEI type matches the PEI type of the currently connected application module. At the moment, 20 PEI types are defined (only useful values are listed in Table 5.1).

The header also describes:

- The manufacturer and version of a program
- The DDR settings for port A and C

- The routing count
- Retransmission limits
- Enabling of U_DELMSG, which causes that some messages are removed from the message queues of the operating system. Either this function is called by the application program or the flag has to be enabled; otherwise a BCU may stop responding. The BCU SDK selects a reasonable default, so that the user does not have to care about this.
- Settings for the serial synchronous interface
- The telegram rate limit

The header is followed by the address table. Internally, a BCU only maps each used EIB address to its number in the address table. The number of a group address is mapped to a set of corresponding group objects using the association table.

The group objects are stored in the *communication object table*. Each group object consists of a RAM or EEPROM location, which contains its value and a 4 bit set of flags (*RAM flags*). These flags store the current transmission/update status. In the table, a group object is described by three bytes. Of these, the first byte is the offset of the value location relative to the start of the EEPROM or RAM. The second byte describes which receive or send operations should be performed by the operating system on this group object. The third byte determines its size.

The processing of the application program is done inside a routine which is called periodically by the BCU. Additionally, an initialization routine and a save routine, which is called in case of a power failure, must be present inside the application program. For handling group objects as well as these three routines, the BCU SDK provides a high level interface.

5.2.1. Accessing the PEI

How the PEI is accessed depends on the used PEI type. For PEI Type 17, the value of *PortCDDR* is used for the DDR setting of Port C. The data port of port C is at the address 0x002. Beware that if you only want to set one bit, you must use inline assembler with bit set/clear operations, because GCC currently only writes whole bytes. Try to use the BCU API instead of direct port access, if possible.

For the PEI Types 2, 4, 6, 8 and 19 direct access to Port C is possible. The suggested API function for that is *U_ioAST*. If an analogue value should be read, *U_readAD* can be used. For the serial PEI types, *U_SerialShift*, *U_AstShift* and *U_LAstShift* can be used.

5.2.2. Timer Subsystem

The BCU 1 provides two kinds of timers: user timers and system timers. The count of user timers is only limited by the available memory. For each user timer, a byte in the

RAM is allocated. Additionally, the resolution is stored in the EEPROM. As resolution, values ranging from 133 ms to 72 minutes are supported. This timers can be loaded with a value between 0 and 127 and provide functions to check if they have expired.

Additionally, the BCU 1 provides up to two system timers. They can be used as count down timers (like a user timer) or as difference counters, where they return the elapsed time units since the last check. As resolution, they only provide a range between 0.5 ms and 33 seconds.

5.2.3. BCU 1 API

The BCU 1 provides a set of API functions. The original names are used to refer to the entry point of a function. The C wrapper functions have the same name, with _ as prefix. For nearly all functions a wrapper exists, but the usage of some functions may produce longer code than necessary because of the necessary glue code.

The wrapper functions are written as C inline functions and use the GCC assembler statement. They are written in a way that for most cases, the smallest solution is chosen by GCC. As a general rule, do not use an API function directly, if the BCU SDK provides an other interface for it.

The following documentation provides only an overview of the available functions. Details are available in the BCU documentation ([BCU1] and [BCU2]).

- `extern const uchar OR_TAB[8];`

  contains a bit mask, which can be used to set a bit.

- `extern const uchar AND_TAB[8];`

  contains a bit mask, which can be used to clear a bit.

- `uchar _U_flagsGet (uchar no);`

  returns the *RAM flags* of the group object *no*.

- `void _U_flagsSet (uchar no, uchar flag);`

  sets the *RAM flags* of the group object *no* to *flag*.

- `uchar _U_testObj (uchar no);`

  returns the *RAM flags* of group object *no* and resets the *update* flag. This function should not be used for group objects, which use the *on_update* function, because it may clear the update flag before the BCU SDK has processed it.

- `void _U_transRequest (uchar no);`

  instructs the BCU to transmit the content of group object *no*.

- `void _EEwrite (uchar offset, uchar value);`

  writes *value* to the EEPROM at location 0x100+*offset*.

- `void _EEsetChecksum ();`

  updates the EEPROM checksum. A call of this function should not be necessary, because all variables are allocated in a way that changeable EEPROM locations are not covered by the checksum.

- ```
  uchar _U_debounce (uchar value, uchar time);
  uchar _U_deb10 (uchar value);
  uchar _U_deb30 (uchar value);
  ```

  _U_debounce debounces a value with a time of *time* in 0.5 ms units. *_U_deb10* and *_U_deb30* do the same, but with a fixed time of 10 or 30 ms. These functions should not be used directly, because they use resources which can be used by the BCU SDK for another purpose.

- `void _U_delMsgs ();`

  removes any message for the user program.

- `short _U_readAD (uchar channel, uchar count);`

  reads the AD channel *channel count* times and returns the sum of all read values.

- ```
  typedef struct
  {
    signed short value;
    bool error;
  } U_map_Result;

  U_map_Result _U_map (signed short value, uchar ptr);
  signed short inline _U_map_NE (signed short value, uchar ptr);
  ```

  performs the map operation, as described in the KNX specification. If no error detection is needed, *_U_map_NE* can be used, which produces smaller code. *ptr* is the offset of the conversion table relative to the EEPROM start at 0x100.

- `uchar _U_ioAST (uchar val);`

  handles binary IO on the PEI port. The upper 4 bits of *val* determine if a read (0) or a write (1) operation has to be done on a specific pin of the PEI. If a bit is written, its new value is stored in the lower 4 bits, e.g. bit 0 and 4 are used for IO bit 0.

  The old state of the pins is stored in the upper 4 bits of the result. If a bit is read, its value is stored in the lower 4 bits. Here again, e.g. bit 0 and 4 correspond to IO bit 0.

- ```
  typedef struct
  {
    uchar pointer;
  ```
  ```

```
  bool error;
} S_xxShift_Result;

S_xxShift_Result _S_AstShift (uchar ptr);
uchar _S_AstShift_NE (uchar ptr);
S_xxShift_Result _S_LastShift (uchar ptr);
uchar _S_LastShift_NE (uchar ptr);
```

performs a data exchange over the SPI PEI interface (type 14). If no error detection is needed, the *NE* variant creates smaller code.

- ```
  typedef struct
  {
    uchar octet;
    bool error;
  } U_SerialShift_Result;

  U_SerialShift_Result _U_SerialShift (uchar octet);
  uchar _U_SerialShift_NE (uchar octet);
  ```

  exchanges a byte over the serial PEI interface. If no error detection is needed, the *NE* variant creates smaller code.

- `void _TM_Load (uchar setup, uchar runtime);`

  initializes a system timer.

- ```
  typedef struct
  {
    bool expired;
    uchar time;
  } TM_GetFlg_Result;

  TM_GetFlg_Result _TM_GetFlg (uchar timer);
  bool _TM_GetFlg_M0 (uchar timer);
  uchar _TM_GetFlg_M1 (uchar timer);
  ```

  returns the status of a system timer. If the timer is in mode 0 (count down timer), the *M0* variant should be used, in mode 1 (difference timer) the *M1* variant.

- ```
  void _U_SetTM (uchar timer, uchar pointer, uchar time);
  void _U_SetTMx (uchar timer, uchar time);
  ```

  initializes a user timer.

- ```
  bool _U_GetTM (uchar timer, uchar pointer);
  bool _U_GetTMx (uchar timer);
  ```

  updates a user timer and returns, if it is expired.

- `void _U_Delay (uchar delay);`

  wait *delay* (in 0.5 ms)

- ```
  typedef struct
  {
    uchar_loptr pointer;
    bool valid;
  } AllocBuf_Result;

  AllocBuf_Result _AllocBuf (bool longbuf);
  uchar_loptr _AllocBuf_NE (bool longbuf);
  ```

  allocates a buffer. If *longbuf* is true, a long buffer is allocated, else a short. Use the *NE* variant, if you do not check the *valid* bit of the return value.

  uchar_loptr is a pointer to an unsigned character with the *loram* attribute. It can be used like any normal pointer. If the pointer is assigned to other pointer variables, try to keep the *loram* attribute, as it allows the generation of smaller load and store operations.

- `void _FreeBuf (uchar_loptr pointer);`

  frees a buffer.

- ```
  typedef struct
  {
    uchar_loptr pointer;
    bool found;
  } PopBuf_Result;

  PopBuf_Result _PopBuf (uchar msg);
  uchar_loptr _PopBuf_NE (uchar msg);
  ```

  searches for a message of a certain type. Use the *NE* variant, if you do not check the returned *found* value.

- ```
  typedef struct
  {
    unsigned short product;
    bool overflow;
  } U_Mul_Result;

  U_Mul_Result _multDE_FG (unsigned short v1,
                           unsigned short v2);
  unsigned short _multDE_FG_NE (unsigned short v1,
                                unsigned short v2);
  ```
  ```

multiplies two unsigned 16 bit values. If you do not check for an overflow, use the *NE* variant. Do not expect that using this function has a specific effect on the resulting code size compared to a normal implementation in C.

- ```
  typedef struct
  {
    unsigned short quotient;
    unsigned short remainder;
    bool error;
  } U_Div_Result;

  U_Div_Result _divDE_BC (unsigned short dividend,
                          unsigned short divisor);
  unsigned short _divDE_BC_quotient (unsigned short dividend,
                                     unsigned short divisor);
  unsigned short _divDE_BC_remainder (unsigned short dividend,
                                      unsigned short divisor);
  ```

  divides 16 bit unsigned values. If you do not check for an overflow and only need the quotient or remainder, use the *quotient* or *remainder* variant. Do not expect that using these functions has a specific effect on the resulting code size compared to a normal implementation in C.

- ```
  uchar  _shlA4(uchar val)
  uchar  _shlA5(uchar val)
  uchar  _shlA6(uchar val)
  uchar  _shlA7(uchar val)
  uchar  _shrA4(uchar val)
  uchar  _shrA5(uchar val)
  uchar  _shrA6(uchar val)
  uchar  _shrA7(uchar val)
  uchar  _rolA1(uchar val)
  uchar  _rolA2(uchar val)
  uchar  _rolA3(uchar val)
  uchar  _rolA4(uchar val)
  uchar  _rolA7(uchar val)
  ```

  shifts or rotates a 8 bit value. Do not expect that using these functions has a specific effect on the resulting code size compared to a normal implementation in C.

- `uchar _U_SetBit (uchar octet, uchar bit, bool set);`

  sets the bit *bit* to *set* of *octet* and returns the value. Do not expect that using this function has a specific effect on the resulting code size compared to a normal implementation in C.

- `bool _U_GetBit (uchar octet, uchar bit);`

  returns the value of bit *bit* in *octet*. Do not expect that using this function has a specific effect on the resulting code size compared to a normal implementation in C.

5.3. BCU2

A BCU 2 provides the same features as the BCU 1. Therefore, the content of Section 5.2 is also valid for the BCU 2 if not otherwise noted.

The main new features are:

- More EEPROM and RAM
- Access Control
- Support for properties
- A new message system, which provides queues to send messages to specific layers
- New timer types
- Support for the FT1.2 Protocol and the PEI type 10, which supports a user PEI handler
- Support for own EIB telegram handlers

The BCU 2 has over 850 bytes of EEPROM available for the user application. Additionally a new RAM section is added, where 24 bytes are available for the user application.

The BCU 2 provides 4 protection levels. Properties as well as memory regions have access levels. An access is only permitted if the current access level is lower or same as the access level of the object. Connections which are not authenticated use access level 3.

The memory management of the BCU 2 has changed. A downloading tool must allocate a section of memory before it can be read or written. During allocation, the access levels as well as the presence of a checksum are specified.

Properties provide a clean interface to access internals of an EIB device. They are used in a BCU 2 for two purposes:

- They are used for the loading of applications as well as to query information about a BCU and change its state.
- An application program can also create its own properties to provide access to its state. Either a variable or an array can be exported by a property. Otherwise a handler function can be used.

In that case, the handler function is called every time when the property is read or written. The handler must do the processing of the incoming EIB message and return the response message.

The BCU 2 offers 4 builtin objects. When loading an application, the address table, association table and application program object are unloaded. Then, for each object a memory region is allocated and filled with data. More allocation of memory regions may follow. Then the pointers for this object are set and its state is changed to *loaded*.

As many pointers are now set using properties, they can store 16 bit values. Therefore many things need no longer be in the region between 0x100 and 0x1FF, while other values, like the telegram rate limit, must still be in these memory locations.

In a BCU 2, each OSI layer is a task, which has its own message queue. Using the BCU 2 API functions, it is possible to send messages to specific tasks.

In a BCU 2, a system timer can also be used to send a message when it expires. Additionally, it can be used as a periodic message timer, which periodically sends a message to the user application.

The PEI Type 10 is added. For this PEI Type, the user can write his own PEI handler.

The BCU 2 provides an application call back, where the application can provide its own mechanisms to handle EIB telegrams. The default handler can be called from the application call back, so that only specific cases must be handled by the application.

5.3.1. BCU 2 API

- `void _U_EE_WriteBlock (void *ptr, long data);`

  writes data to an address aligned at a 4 byte boundary in the EEPROM.

- `uchar _U_GetAccess ();`

  returns the current access level.

- `void _U_SetPollingRsp (uchar val);`

  sets the result for the polling slave.

- `void _U_Char_Out (uchar val);`

  sends a byte over the PEI interface using the SCI or SPI protocol.

- ```
  void _U_TS_Set (uchar timer, uchar mode, uchar scale,
                  uchar value, uchar param);
  ```

  sets a BCU 2 timer.

- `void _U_TS_Del (uchar val);`

  stops a BCU 2 timer.

- `void _U_MS_Post (uchar msgid, uchar pointer);`

  sends a message to a message queue.
  ```

- `void _U_MS_Switch (uchar msgid, uchar destination);`
  moves a message to a different message queue.
- `short _FP_Flt2Int (uchar ptr, uchar exponent);`
  converts a floating point value to an integer.
- `void _FP_Int2Flt (short val, uchar ptr, uchar exponent);`
  converts an integer into a floating point value.
- `void _U_FT12_Reset (uchar baudrate);`
  resets the FT1.2 protocol.
-
  ```
  typedef struct
  {
    bool newstate;
    uchar stateok;
  } FT12_GetStatus_Result;

  FT12_GetStatus_Result _U_FT12_GetStatus (bool force_reset);
  ```
  gets the FT1.2 protocol state.
- `void _U_SCI_Init (uchar baudrate);`
  initializes the SCI protocol.
- `void _U_SPI_Init ();`
  initializes the SPI protocol.

6. BCU SDK

The BCU SDK consists of the XML[1] DTD[2] and Schema files for the data exchange (in the *xml* directory), which are covered in Chapter 9; helper programs to extract and embed the program ID in XML files; eibd (in the *eibd* directory, see Chapter 7); BCU headers and libraries; a generation tool for all necessary glue code and tables; and some build scripts.

To access a XML file, the BCU SDK uses the libxml2 tree interface. With it, the content of a XML file can be loaded into a tree like structure. An XML tree can also be written to a file.

6.1. Common files

The *common* directory contains a collection of different files, which are used in different places. Besides the definition of commonly used types as well as classes for array, string and stack handling, it includes all functions to process images for the BCU SDK.

In the files *image.cpp* and *image.h*, the class *Image* for loading a BCU image is defined. It provides easy access to the different streams (see Appendix A for details about the image format). Additionally, such an image can be modified and turned into a byte stream again.

The file *loadimage.cpp* contains functions to check if an image is really loadable. Additionally, it extracts the important things and stores them in the class *BCUImage*. For the BCU 1, it stores the individual address of the destination BCU and the EEPROM content. For the BCU 2, it additionally stores all needed keys and a list of instructions of A_PropertyValue_Write and A_Memory_Write requests, which are needed to load the program. The real loading part only has to issue the commands as they are in the list, and check the results.

6.2. XML related programs

The *archive* directory contains the *embedprogid* and *extractprogid* programs. Both read an XML file and check if they are really an *application information* or a *configuration description*, respectively.

Depending on the program, either the hex dump in the *ProgramID* element is decoded and saved as a file, or the content of a file is encoded as a hexadecimal string and stored

[1]Extensible Markup Language
[2]Document Type Description

as *ProgramID* in the XML file.

6.3. Build system

The *build* directory contain the scripts which control the build process of images and the *application information*.

The *build.ai* shell script first calls the *bcugen1* program to create the *application information*, all headers, assembler and C files (see Figure 6.1).

Besides the XML file, it also builds an image. This is done for the following reasons:

- It should find every syntax error in the source program, which could happen during the execution of the *build.img* script.
- The image gives an upper limit of the needed code size, as by default all features are compiled in.
- The image could be used for an ETS like approach of image handling, where the parameters and address tables are changed in the executable.

The list of included files is run through the preprocessor, so that all source files of the user are in one C file. The original line numbers are kept in the *#line* directive, so that GCC will still use the original file names. This is stored in the file *c.inc*.

Together with the generated header file *c.h*, this is compiled into the object file *c.o*. The allocation of parameters, which are stored in the file *p.c*, is also compiled. The parameters are kept in an extra file to make it impossible for GCC to optimize them away.

The assembler header and glue code (in *p1.s*) is assembled. All files are linked with the necessary libraries and the linker script for the selected BCU into an ELF executable. The base name of the linker script and libraries is determined by the *bcugen1* program. It outputs the name of a variant (which includes the mask version as well as an user specified subtype). Then the linker is run again with another linker script to convert the ELF file into an image file.

Finally, all files which are needed for the *build.img* part, are packed into an archive, which is stored as *ProgramID* of the XML file. It should be noted that the *ProgramID* tag thus stores the program text as discussed in the introduction.

The image building shell script *build.img* takes a *configuration description* and extracts the *ProgramID*, unless a file was passed as a parameter explicitly for this purpose (see Figure 6.2). It extracts the parts of the archive and runs the *bcugen2* program. Then the image build process is run, similar as it is done by the *build.ai* script. However, the parameter C file is not used, because it is not needed in this step.

The *gencitemplate* transforms an *application information* into a *configuration description* skeleton using an XSLT transformation.

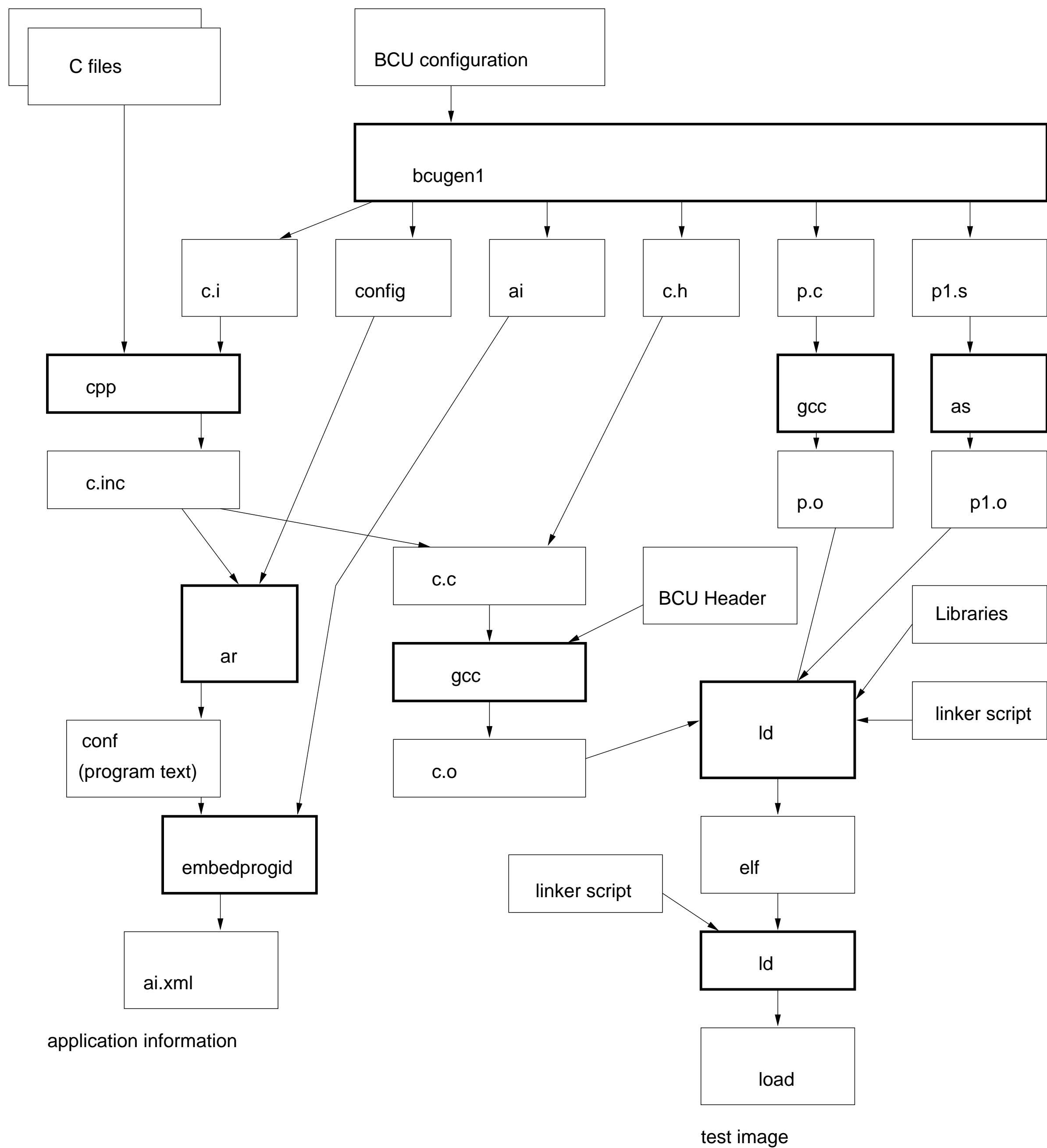

Figure 6.1.: *build.ai* operational sequence and data flow

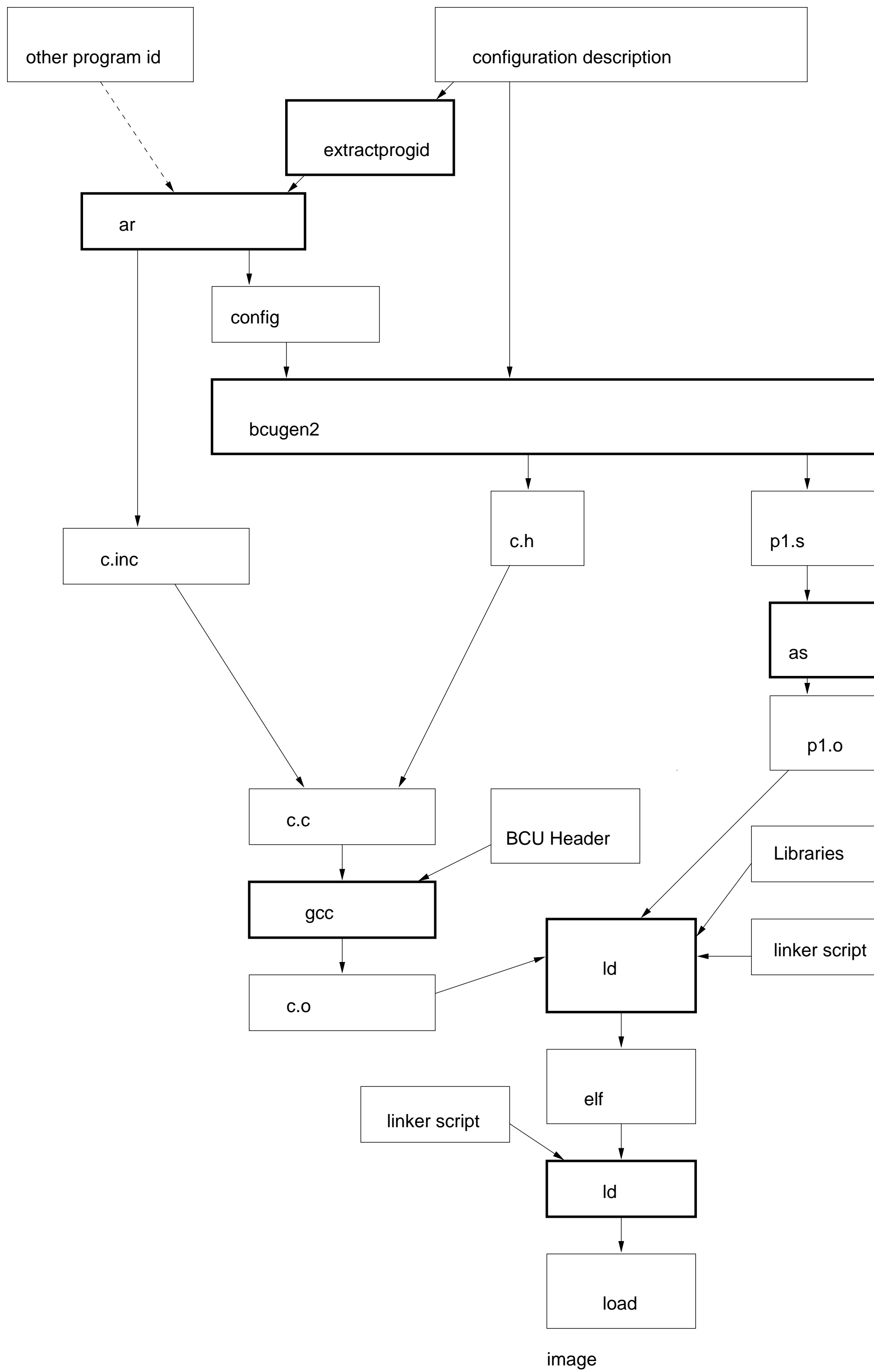

Figure 6.2.: *build.img* operational sequence and data flow

6.4. Configuration file parser

The configuration file parse is one of the core parts of the BCU SDK. It is used for the *BCU configuration* (see Section 8.1) as well as for the temporary data exchange format between the runs of *build.ai* and *build.img*.

A supported IO format is described by a list of objects and their attributes. Each object is mapped to a block in the configuration file, each attribute to an entry. Out of this description, all relevant classes as well as a configuration file writer are generated.

A definition for an object looks like this:

```
OBJECT(Debounce)

ATTRIB_IDENT(Name)
ATTRIB_FLOAT(Time)

END_OBJECT
```

The declaration of an object starts with *OBJECT* and ends with *END_OBJECT*. Between these, all attributes are listed:

PRIVATE_VAR adds a normal variable to the object.

ATTRIB_STRING defines an attribute which stores a string.

ATTRIB_IDENT defines an attribute which stores an identifier.

ATTRIB_INT defines an attribute which stores an integer.

ATTRIB_BOOL defines an attribute which stores a boolean.

ATTRIB_FLOAT defines an attribute which stores a float.

ATTRIB_ARRAY_OBJECT defines an attribute, which can store many objects of the specified type. The object must be declared using this specification before it can be used in this directive.

ATTRIB_ENUM defines an attribute which stores its value as an enumeration. In the configuration file, an identifier is used. To map between these, two mapping functions must be provided.

ATTRIB_INT_MAP defines an attribute which stores an integer. In the configuration file, an identifier may be used besides an integer. This identifier is converted using a mapping function.

ATTRIB_FLOAT_MAP defines an attribute which stores a floating point value. In the configuration file, an identifier may be used besides a floating point value. This identifier is converted using a mapping function.

ATTRIB_ENUM_MAP defines an attribute which stores an array of name/value pairs.

ATTRIB_IDENT_ARRAY defines an attribute which stores an array of identifiers.

ATTRIB_STRING_ARRAY defines an attribute which stores an array of strings.

ATTRIB_EXPR defines an attribute which stores an expression suitable for *InvisibleIf*.

Using different defines, the list of objects and attributes is included at different locations. Using this mechanism, the parsers, scanner, class definitions, initialization code and output code are generated out of a single list.

For mapping lists, the name/value pairs of one type are stored in one file. Each pair is written as *MAP(*Value,Name*)*. They are included with different definitions of *MAP* to generate all mapping functions for one type out of one list (sometimes even the definitions of an enumeration).

All attributes, except the object array, include a line number variable. This is automatically set when a value is read by the parser. The writer only exports variables which have the line number set. Using this mechanism, no reserved values for the attribute variable itself are needed. Identifiers and strings are stored in the class *String*. For arrays, the template class *Array* is used.

As the BCU SDK uses two formats (*BCU information* and the data exchange format between *build.ai* and *build.img*) which are very similar but have small differences, all attributes which are not appropriate in one of the two parsers are hidden using preprocessor commands.

6.5. Bcugen1 and bcugen2

The core work is done by the *bcugen1* and *bcugen2* programs. They share nearly all code, but use different parsers, different checks and code output functions.

bcugen1 reads the *BCU configuration* into a *Device* object. After that extensive checks are done, which set missing attributes to default values. Additionally, some values, like the XML Ids, are calculated.

The necessary parts of the *Device* object for *bcugen2* are written into a configuration file. Additionally, the *application information* is generated from it. As the last part, all code needed to compile a BCU image is exported (also using the information in the *Device* object).

bcugen2 reads the output of *bcugen1* into a *Device* object and merges the content of the *configuration description* with it. Then a slightly modified check routine is run, after which all needed code is exported.

The differences for the generated code are:

- In the output of *bcugen1* nothing is deactivated, so this is the worst case for the size estimation.

- *bcugen1* outputs only empty address and association tables of a selectable size.

- *bcugen1* puts parameters in a different file and adds the location of the parameters to the image.

The image created from the output of *bcugen1* is more like a BCU program as it is used by the ETS. As a proof of concept program, *imageedit* was written (included in the *BCU SDK*), which takes the output image of *build.ai* and a *configuration description*. It then stores the selected values into the image.

This program only supports group objects, parameters (except *FloatParameter*) and the access control keys. As the images of *bcugen2* provide better optimization possibilities, its development was not continued.

6.6. Overview of the generated code

The assembler code part only calls C stub routines without any parameters. If values must be passed, special memory locations are used. The stub function contains the *nosave* attribute, so that no register values of *RegB* – *RegN* are preserved.

Then the stub loads the parameters into local variables and calls the user function. The user function is declared as static, so that it can be integrated by GCC into the stub if this results in smaller code. Thereby the overhead of one function call is removed. Finally the stub stores the result in special memory locations. The assembler part passes the values to the BCU operating system.

For parameters, a static constant variable is defined. In many situations, GCC can propagate its value into the code and thereby remove it.

The three BCU entry functions (init, save, run) initialize the stack and then execute their stub. Before the stub is executed for *run*, all group objects and timers are checked for events. If an event has occurred, the event handler stub is called.

The generated code tries to move every element into a different section, so that the layout of the final image can be selected by the linker script.

The linker script for the BCU 2 also moves variables between the two RAM segments by using the move section feature of the linker.

6.7. Memory layout

The memory layout of a BCU 1 application is given in Figure 6.3. The *.ram* section is intended for variables, which must have an address below 0x100. Then the *.bss* and *.data* sections follow. At startup, the *.bss* section is cleared. In the *.data* section, the initializers are copied from the EEPROM. The user stack starts at the highest byte of the RAM area which is available to the user program. If too much stack is used, the stack grows into the *.data* or *.bss* segment, which will cause strange errors in most cases.

In the EEPROM, the application header is followed by the address and association tables. In a *build.img* image, all tables have exactly the required size with no space between them.

Address	Region	Contents
0x000	IO Space ROM	
0x050	Low RAM	RegB – RegN reserved
0x0CE		.ram .bss .data stack
0xE0		call stack
0x100	EEPROM	Header
		Addresstable Association table
		Group Objects Init Code Code Timer .loconst read only copy of .data .eeprom .parameter
0x1FF		checksum

Figure 6.3.: Memory map of a BCU 1

Address	Area	Contents
0x000	IO Space ROM	
0x050	Low RAM	RegB – RegN reserved
0x0CE		.ram .bss .data stack
0xE0		call stack
0x100	EEPROM	Header
		Addresstable .eeprom Timer .loconst
<=0X1FF		Group Objects Init Code Properties Code read only copy of .data copy of .data.hi Association table .parameter
0x4DF		
0x900	High RAM	
0x972		.bss.hi .data.hi stack
0x98A		
0x9D0		

Figure 6.4.: Memory map of a BCU 2

After that, the definitions of the group objects, the startup code, and the remaining code follow. Then the EEPROM structure of user timers is placed, if they are used. The *.loconst* section is intended for constants which must be located between 0x100 and 0x1FF. After that, normal constants follow. The next part is a copy of the *.data* segment for initialization. After this, the checksum protected EEPROM area ends.

The *.eeprom* section is intended for variables which may be changed during runtime but must be located in the EEPROM. The current version of the BCU SDK guarantees that this section is between 0x100 and 0x1FF, even for a BCU 2.

The *.parameter* section is unused for *build.img* images. It is intended for parameters, like they are used by the ETS. The BCU SDK *build.ai* script places the parameters in this section.

The memory layout of a BCU 2 application is given in Figure 6.4. The order of the sections differ from the normal order, because some sections must be below 0x1ff (or 0xff). Therefore, these sections are located before the others.

As the image format (see Section A) supports only a limited number of sections, it is impossible to split the text segment into parts to allow the association table to be placed before the program code. Because this order is not important for the BCU operating system, the association table is placed after the text segment.

The main difference to a BCU 1 image is that a second RAM region is available. This region contains a data and bss segment called *.data.hi* and *.bss.hi*. Using the section movement code, parts of the normal variables are moved to this location.

The default variant allocates the stack in the low RAM, the *histack* variant in the high RAM. The default variant as well as the *histack* variant use this high memory region only as an overflow location. As other variants may use totally different placement strategies, an application must not expect a normal variable to be placed at a specific location. If a variable is needed at a location lower than 0x100, place it in the *.ram* section. All other variables can be placed in the default data section.

A constraint of the BCU 2 image, is that address table, *.eeprom* section, user timers as well as the *.loconst* section together must be smaller than 0x100 bytes, so that they fit the memory region between 0x100 and 0x1FF. With the *viewimage* program, the used memory size as well as the free stack size can be displayed.

7. EIB bus access

7.1. Overview

To access the EIB bus, a daemon (called *eibd*) for Linux systems was developed. It provides a simple EIB protocol stack as well as some management functions. A big advantage of using a daemon is, that the applications can use a high level API. Another is that multiple clients, even on different computers, can connect to the bus simultaneously. A disadvantage of this concept is that some future extensions will need a modification of the daemon.

The daemon supports different ways to access the EIB bus:

- PEI10 protocol (FT1.2 protocol subset)
- PEI16 protocol (using the BCU 1 kernel driver)
- TPUART (using the TPUART kernel driver)
- EIBnet/IP Routing
- EIBnet/IP Tunneling
- TPUART user mode driver
- PEI16 user mode driver (not really usable)
- KNX USB protocol (only EMI1 and EMI2)

The daemon consists of a front end which accepts connections from applications over TCP/IP or Unix domain sockets, a protocol and management core and some back ends, which are the interface to the medium access devices.

The daemon is intended for the TP1 medium and uses the TP1 frame format as its internal representation. It does not support TP1 polling. Support for the extended frame format is present, but it requires support in the back end part for the selected bus access mechanism. Of the EMI based back ends, only CEMI frames support data areas which are large enough (therefore, only EIBnet/IP is possible). Both TPUART back ends support sending extended data frames. The kernel level driver based back end supports decoding such data frames, if they are delivered by the kernel driver (which does not support them at the moment). In the user mode TPUART back end, the decoding of extended data frames is unimplemented.

Because using the bus monitor mode prevents the sending of frames, a best effort bus monitor, called *vBusmonitor*, was introduced. This feature can be activated at any time.

If *eibd* runs in bus monitor mode, a *vBusmonitor* client will get the normal bus monitor services. Otherwise all telegrams which *eibd* receives will be delivered. Theoretically, there need not be a difference in the services between the two modes. For the current back ends, at least all ACKs are lost. Most back ends also only deliver frames to or from the bus access device.

eibd has no security mechanisms. If the operating system allows a connection, *eibd* will provide its services. Because dynamic buffer management is used, buffer overflows are not very likely to happen. The best security level can be achieved, if the daemon only listens on the Unix domain socket and the access to this socket is restricted to a certain group. Additionally, it should not be run as root. The user account which runs eibd then needs access privileges for the appropriate device node. For EIBnet/IP, no privileges are needed.

7.2. Architecture

The whole of *eibd* is based on the *GNU pth* user mode threading library ([PTH]).

GNU pth only supports non preemptive threading. Also, only one thread is running at a time. This makes locking superfluous in many situations and therefore the program code is simpler. However, it may be a performance problem for some application tasks. For *eibd*, this is not an issue, because nearly all the time, the tasks are waiting for an event.

Additionally, it provides a powerful event management, which supports waiting until one of a set of different events has occurred. For system calls which can block, *pth* provides wrappers which will switch to another task if the system call blocks. As an additional parameter, the pth versions of the system calls accept a list of events which will abort the system call if they occur before the system call is completed.

For inter-thread communication FIFO queues and semaphores are used. The semaphore support for *pth* was written as part of eibd and is available as a patch ([PTHSEM]).

Using pthreads was also considered, but the need for more locking to avoid race conditions as well as the missing support of a waiting construct for multiple events discouraged its use.

For storing arrays, the template class *Array* was written; for the use of strings, the class *String*. For the packing and unpacking of frames of the different layers, the classes *APDU* (Layer 7), *TPDU* (Layer 4) and *LPDU* (Layer 2) were introduced. Each subclass of them represents a specific type and implements associated functions.

The back end to be used can be selected over an URL. The first part selects the back end, then a colon and back end specific data follow.

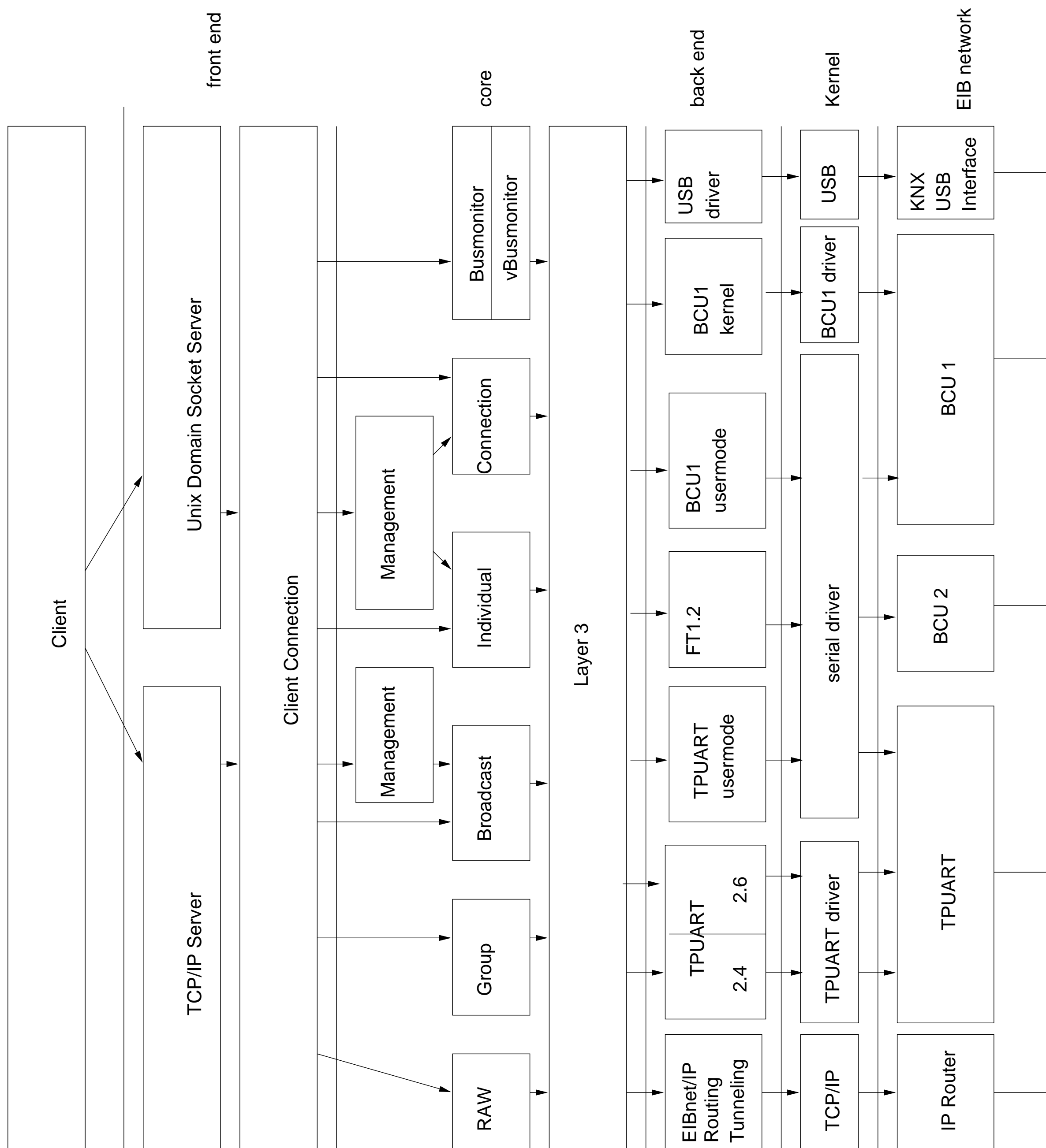

Figure 7.1.: Structure of *eibd*

7.3. Back ends

The back ends provide an interface to the EIB bus. A back end is an implementation of the *Layer2Interface* class.

The EMI1 or EMI2 based back ends only provide generic frame encapsulation and unpacking of EIB frames. The EMI frames are passed to an instance of the *LowLevelDriver-Interface* class, which provides the means to send and receive frames. Because of this, different interfaces, like the BCU1 user mode driver and the BCU1 kernel driver, can be supported by the same class.

7.3.1. EMI2

The EMI2 interface is implemented in the *EMI2Layer2Interface* class. It supports the bus monitor mode. In vBusmonitor mode, all outgoing frames are delivered. Incoming frames are filtered as described below. For normal communication, the same restrictions apply.

At the moment, this back end accepts listen requests for any group address. To actually receive telegrams on a group address, it must be in the BCU address table, or the address table length needs to be set to 0 before eibd is started (using the *bcuaddrtab* command). Telegrams with an individual address as destination are delivered only when this address matches the individual address of the BCU 2.

Changing the address table length involves changing a byte in the BCU EEPROM. The problem with this solution is, that an EEPROM has a limited number of write cycles and that in the case of a crash of eibd the original value would not be restored. Therefore, this change is not made automatically within eibd.

FT1.2

The interface to a BCU 2 over FT1.2 is implemented in the class *FT12LowLevelDriver*. It only requires access to a serial port. The URL for this back end is *ft12:/dev/ttySx*, where /dev/ttySx has to be replaced with the correct serial interface. This back end works reliably.

7.3.2. EMI1

The EMI1 interface is implemented in the class *EMI1Layer2Interface*. Regarding bus monitor mode and frame filtering, it has the same features and limitations as the EMI2 back end described above.

The PEI16 protocol used to transmit EMI1 messages from and to the BCU 1 is highly timing sensitive. RTS/CTS handshaking is done for every character and a message must be transmitted within 130 ms. This complicates implementation on the PC side.

BCU1 kernel driver

An interface for the BCU1 kernel driver is implemented in the class *BCU1DriverLowLevelDriver*. The URL for this back end is *bcu1:/dev/eib*, where /dev/eib has to be replaced with the correct device node of the kernel driver.

As of version 0.2.6.2, the kernel driver is working. Under certain circumstances however (which could not yet be identified precisely), communication with the BCU 1 is lost. Sometimes even incorrect information is delivered by the driver. These problems appear to be due to PEI16 timing issues. Thus, this back end should only be used when no other way of bus access than a BCU 1 is available.

BCU1 user mode driver

This interface is implemented in the class *BCU1SerialLowLevelDriver*. It suffers even more from timing problems than the kernel driver. This back end is more a technical test. It should only be used for testing or demonstration purposes, although longer tasks like a property scan successfully worked.

The critical point is that an EMI frame must be transmitted within 130 ms. Each exchange needs to change the RTS line, do busy waiting until a CTS line changes, send a character, wait for a character, change the RTS line, and do busy waiting until a CTS line changes again. The danger is that the daemon loses the processor too long while a transfer is in progress.

During development, a first version with debugging output was created. Removing the debugging output and replacing it with sleep or a nop loop increased the timing problems. So the delay was left over to the debugging output. Therefore the back end must be run with a trace level of 1023 in a terminal in the foreground. Not every terminal emulation works equally well, e.g. on an old Linux 2.4 systems, a GNOME terminal supports transmitting longer frames than a KDE terminal.

Additionally, it requires the low latency mode. On Linux 2.6, be sure to use Linux 2.6.11 or later. Previous versions contain a deadlock which will freeze the computer after the first byte is transferred over the serial line.

The URL for this back end is *bcu1s:/dev/ttySx*, where /dev/ttySx has to be replaced with the appropriate serial interface.

7.3.3. KNX USB interface

The USB backend consists of several layers. The low level interface is implemented in the class *USBLowLevelDriver*, which is responsible for sending and receiving USB messages. It uses a customized version of a development snapshot of *libusb*, modified to fit in the eibd framework. It accesses the USB interface directly without any special kernel level driver (like the Linux HID driver).

The class *USBConverterInterface* is used to translate between EMI frames and USB messages. The high level interface is implemented in the class *USBLayer2Interface*. It determines the EMI version and creates a Layer 2 interface of the corresponding EMI

version. Therefore all limitations of the EMI1 and EMI2 backends hold depending on the EMI version used.

The URL of this backend is *usb:[bus[:device[:config[:interface]]]]*. The values of bus, device, config and interface can be determined using *findknxusb*.

Currently only EMI1 and EMI2 are supported. cEMI is not implemented, as no device supporting this feature is available for testing.

7.3.4. EIBnet/IP Routing

EIBnet/IP Routing was implemented in the class *EIBNetIPRouter*. It has the disadvantage that there is no bus monitor support. So the vBusmonitor mode is used instead, which delivers every packet that is transmitted over the IP multicast port.

This back end can use arbitrary EIB addresses. The individual address, from which all management connections originate, must be set with a parameter. The main factor to make a connection work is the setup of the IP router. If some necessary packets are filtered by the router, the communication will not work.

If the routing table is correct, the back end works reliably. The URL of the back end is ip:*multicast_address*:*port*. If the default settings are used only ip: is sufficient. If only another multicast address is needed, ip:*multicast address* can be used. It is possible to connect many instances of eibd to an IP router.

7.3.5. EIBnet/IP Tunneling

In a first test with EIBnet/IP Tunneling mode, the embedded EIBnet/IP server (Siemens IP Router) used stopped serving the connection after one second. This problem disappeared after some months (without any change in the program, but some resets of the IP router) and so support for Tunneling mode was added in the class *EIBNetIPTunnel*.

As the embedded server does not support bus monitor mode, bus monitor mode is implemented as a vBusmonitor mode, which can deliver all group communication as well as as well as telegrams directed to the individual address which was assigned to this tunneling endpoint by the IP router. For normal communication, the same applies. Incoming telegrams with any other destination individual address than the one of this tunneling endpoint will be filtered. In addition, telegrams to certain destination group addresses may be filtered by the IP router.

The back end is working. Eibd tries to connect periodically, until a positive response is received. After a disconnect message is received, it periodically attempts to reconnect. It does not detect broken connections (e.g. different views of sequence numbers on both sides or a server which died without sending a disconnect request). In this case, *eibd* does not abort, but no further communication with the IP router will occur. The URL of the back end is ipt:*router-ip-name*:*dst-dport*:*src-port*. Source and/or destination port can be omitted (including the colon), if the default values are sufficient.

7.3.6. TPUART kernel driver

The driver is implemented in the class *TPUARTLayer2Driver*. Two different versions of the TPUART kernel driver exist: for the 2.4 kernel and the 2.6 kernel. Because the API changed (the checksum byte may not be added when a frame is sent), different URLs are needed.

For the 2.4 driver, use tpuart24:/dev/tpuartX as URL, where /dev/tpuartX has to be replaced by the device node. For the 2.6 version, tpuart:/dev/tpuartX has to be used. The primary EIB management address must be set as a command line parameter.

The back end supports the use of arbitrary addresses. In vBusmonitor mode, only frames for addresses to which eibd has subscribed are delivered. A "deliver all" option would be possible, but is not implemented in the kernel driver. This back end works reliably.

7.3.7. TPUART user mode driver

In the class *TPUARTSerialLayer2Driver*, a complete user mode driver for the TPUART is implemented. It has the same addressing capabilities as the kernel driver, but the vBusmonitor mode delivers all frames. Although a kernel module is naturally in a better position to handle the timing requirements of the TPUART communication protocol properly, the user-mode solution is attractive due to its higher flexibility and reduced installation hassle. In tests, the user mode driver performed satisfactorily on a rather heavily loaded Pentium-4/1.8 GHz.

When a frame is received from the TPUART, an acknowledgement request has to be returned within a short time after the destination address has been received. When it is not returned in time, the remote station will repeat its transmission on the EIB network up to three times. On a reasonably recent workstation PC, it is possible to acknowledge at least one of these transmit attempts by using low latency mode. Due to a bug in the low latency mode implementation, running this back end on a Linux 2.6 kernel with a version lower than 2.6.11 will crash the computer (see section 7.3.2).

Because no history of recently received frames is kept, all repeated frames are discarded. With this strategy, it is possible to lose a frame if the first send attempt is corrupted.

For recognizing frame starts, a dual strategy is implemented. It assumes that a frame starts at the first byte received. If the byte sequence starting at this byte is not a correct frame, or if an expected byte is not received after an expected timeout, the head of the receive buffer is discarded and a new receive attempt is made with the new head.

There is no problem regarding the transmission of frames. The URL for this back end is *tpuarts:/dev/ttySx*, where /dev/ttySx has to be replaced with the correct serial interface.

7.4. Core

The core of the driver is organized in layers. The definition of the layers is inspired by their definitions in the KNX specification ([KNX]), but adapted to fulfill all requirements.

7.4.1. Layer 3

The class *Layer3* is the main dispatcher and the interface to the back end. It decides when to enter bus monitor mode.

Each higher layer task registers at the Layer 3 and states what kind of packets and addresses it is interested in. The group address 0/0/0 and the individual address 0.0.0 have a special meaning. Listening on group address 0/0/0 means that all group communication packets from the back end should be delivered. For getting the broadcast packets, which use this address on the bus, a special call back is implemented.

Listening for frames with the source address 0.0.0 means, that all packets to a specific individual address of the back end should be delivered. Listening for frames with the destination address 0.0.0 means, that all packets to the default individual address of the back end should be delivered. The mapping of the address 0.0.0 to the default address is done transparently in both directions. In fact, higher layers cannot even ever determine the real EIB address of the back end.

7.4.2. Layer 4

This layer provides a communication endpoint to applications for one specific Layer 4 service. The services are:

- The class *T_Broadcast* implements an endpoint for broadcast communication. For sending, an APDU is passed (as a character array). The APDU as well as the source of a received broadcast will be returned.

- The class *T_Group* implements an endpoint for group communication with a specific group address. For sending, an APDU is passed (as a character array). Of a received group telegram, the APDU as well as the source will be returned.

- The class *GroupSocket* implements an endpoint for group communication which is not bound to a particular address. Unlike a *T_Group* instance, a single *GroupSocket* can be used to communicate using the whole range of group addresses.

  T_Group registers the group address it is bound to with the backend, so that Layer 2 ACKs can be generated (if supported by the backend). *GroupSocket* does not register any addresses, so that the generation of ACKs must be controlled by other means.[1] Currently, only the TPUART backends support generating Layer 2 ACKs.

[1] E.g., ensure that another device or coupler on the same line will generate these ACKs if required.

- The class *T_Individual* implements a T_Data_Individual communication relation between two devices. Sent APDUs will be automatically transmitted to the right communication partner and only the APDUs of T_Data_Individual frames of the communication partner will be delivered to the higher layers.
- The class *T_Connection* implements the client of a connection between two devices. It implements the necessary state engine and opens the connection when instantiated.

  The connection will be closed when the class gets destroyed. If the connection gets closed by the remote device, an empty APDU will be transmitted to the higher layers. The higher layers must take care, that the connection is not idle too long. This will cause the remote device to close the connection.

  It is possible to use multiple connections if they have different remote targets. This fact is not checked by eibd. If multiple connections are made to one device, they will interfere and in most cases all these connections will get closed. Such a race condition can also happen if a connection is closed and immediately reopened, because the T_Disconnect may not have been sent on the EIB bus yet when the new connection is opened.

  In normal operation, APDUs are exchanged with the higher layers.
- The class *T_TPDU* provides raw access to TPDUs to unicast TPDUs[2]. It is intended for the implementation of a server endpoint of a T_Connection in an application which is not supported in eibd at the moment.

7.5. Layer 7

The call of management relevant Layer 7 functions is implemented in the classes *Layer7_Connection* and *Layer7_Broadcast*.

Layer7_Broadcast can send a A_IndividualAddress_Write or can collect all corresponding responses after sending a A_IndividualAddress_Read.

Layer7_Connection provides functions to send a connection oriented request and parse the result. Functions are, for example, A_Memory_Read, A_Memory_Write, A_ADC_Read .... Additionally there are some functions prefixed with *X_*, which do more complicated tasks:

X_Property_Write writes a property and verifies it.

X_Memory_Write writes to memory and verifies it.

X_Memory_Write_Block writes a block of memory (no size limit) and verifies it.

X_Memory_Read_Block reads a block of memory (no size limit).

[2]No group/broadcast telegrams will be delivered.

High level management procedures are implemented in the class *Management_Connection*:

X_Progmode_On switches the device to programming mode.

X_Progmode_Off turns the programming mode off.

X_Progmode_Toggle toggles the programming mode flag.

X_Progmode_Status gets the state of the programming mode.

X_Get_PEIType gets the PEI type of the application module connected to a device.

X_PropertyScan returns a list of all properties of a device.

7.6. EIBnet/IP server front end

Eibd provides access to the back end device over the EIBnet/IP Routing and Tunneling protocols. This is implemented in the class *EIBnetServer*. Routing, Tunneling and the discovery functions (SEARCH, DESCRIPTION) can be enabled separately.

Enabling this server prohibits the normal bus monitor mode (vBusmonitor is still working). All other *eibd* functions are not affected. Eibd does not provide the ability to use filter tables for routing. This means that telegram loops can easily occur when it is used in parallel with another EIBnet/IP router on the same line.

The EIBnet/IP server front end of eibd performs a kind of network address translation on individual addresses. In outgoing frames, *0.0.0* is replaced with the individual address of the bus access device. Likewise, incoming frames with the individual address of the bus access device as destination have this destination address replaced with *0.0.0*.[3] Also, *0.0.0* is returned as the KNX individual address assigned to the Tunnelling connection in the connection response data block.

Source addresses other than *0.0.0* can be used for outgoing frames, but incoming frames addressed to individual addresses other than that of the bus access device are suppressed.[4] Therefore, *0.0.0* should be used as the local individual address by eibd-EIBnet/IP client applications. Tunneling clients which use the address returned in the connection response will do so automatically.

7.7. EIBD front end

The EIBD front end[5] accepts connections on a TCP/IP port or on a Unix domain socket, receives requests, unmarshals requests, processes them and marshals and sends

[3]For outgoing frames, this translation is consistent with [KNX] AN033, 2.5.3.3 (cEMI L_Data.req). Applying this concept to incoming frames is an extension to the KNX specification.

[4]Note that this restriction, which is due to the limitations of the bus access devices and/or drivers used, significantly limits the use of the Routing protocol with eibd.

[5]This frontend speaks a simple protocol, which is different to the EIBnet/IP protol suite, also supported by EIBD.

the results. Its implementation is distributed over various classes.

As a counter part, a simple client library was written, which does all marshaling and unmarshaling at the client side. The protocol and the corresponding stub functions will be described in the following sections. There are also small example programs for nearly all management functions.

7.7.1. Protocol

The protocol is quite simple. The client connects to the eibd daemon over TCP/IP or Unix domain sockets. Then it sends its requests and receives all responses. To free the connection, the connection is simply closed using the means provided by the operating system.

The relevant functions in the library are:

EIBSocketURL opens a connection to eibd. The connection target is passed as a string. The format is:

- local:*path to Unix socket* connects to a Unix socket.
- ip:*hostname*[:*port*] connects to eibd listening on a TCP port.

EIBSocketLocal connects to eibd over the Unix socket passed in the parameter.

EIBSocketRemote connects to eibd over a TCP port.

EIBClose closes the connection and frees all resources.

Normally a connection is switched to a certain mode and after that only certain types of requests can be processed. Multi byte values are passed in the big-endian format.

Every packet starts with a two byte length field of the data, counting from after the second byte of the packet. Then two bytes determine the purpose of the packet (this will be called the *type* in the following). The possible values of this field are defined in *eibtypes.h*.

If a request is unsuccessful, a packet with the type EIB_INVALID_REQUEST or a more specific response is returned by eibd. Therefore, if the result type is not in the range of the expected types, an error should be returned.

Bus monitor mode

To use the bus monitor services, a packet with a type of EIB_OPEN_BUSMONITOR, EIB_OPEN_BUSMONITOR_TEXT, EIB_OPEN_VBUSMONITOR or EIB_OPEN_VBUSMONITOR_TEXT has to be sent to the server. If the request is successful, a packet with the same type is sent back. Then, each received packet is transmitted with the type EIB_BUSMONITOR_PACKET.

A bus monitor mode with the addition of TEXT means, that a human readable decoded version as string is passed rather than the raw content of the EIB frame. This service can be used for simple bus monitors. If filtering and displaying of parts of the

content is needed, an application can use the normal services and decode the packets itself.

The relevant functions are:

EIBOpenBusmonitor opens a normal binary bus monitor.

EIBOpenBusmonitorText opens a decoded, normal bus monitor.

EIBOpenVBusmonitor opens a binary vBusmonitor.

EIBOpenVBusmonitorText opens a decoded vBusmonitor.

EIBGetBusmonitorPacket receives a bus monitor packet.

Layer 4 connections

A layer 4 connection is opened by a packet of 5 bytes with one of the following types:

- EIB_OPEN_T_CONNECTION opens a T_Connection, the destination address is transmitted in bytes 2–3.
- EIB_OPEN_T_INDIVIDUAL opens a T_Data_Individual connection, the destination address is transmitted in bytes 2–3. Byte 4 is 0, if the connection is only used to send data, else it is 0xff.
- EIB_OPEN_T_GROUP opens a T_Data_Group connection, the group address is transmitted in bytes 2–3. Byte 4 is 0, if the connection is only used to send data, else it is 0xff.
- EIB_OPEN_T_BROADCAST opens a T_Data_Broadcast connection. Byte 4 is 0, if the connection is only used to send data, else it is 0xff.
- EIB_OPEN_T_TPDU opens a raw Layer 4 connection. The local address is transmitted in bytes 2–3 (0 means the default address).
- EIB_OPEN_GROUPCON opens a group socket, which can be used to send and receive group telegrams for any group address. Receiving telegrams over this mechanism does not generate Layer 2 ACKs[6]. Byte 4 is 0, if the connection is only used to send data, else it is 0xff.

The data are exchanged in packets of the type EIB_APDU_PACKET. For some types, two bytes with the EIB address are inserted before the data. A raw connection transmits an EIB address in both directions, group and broadcast connections transmit addresses only from the EIB daemon.

A group socket transmits packets of the type EIB_GROUP_PACKET. For sending group telegrams, the destination address is put in bytes 2–3 followed by the APDU. For

[6]see Section 7.4.2, class *GroupSocket* for details

received telegrams, bytes 2–3 contain the source address, followed by the destination address in bytes 4–5 and the APDU.

Note that a T_Connection is automatically closed if there is no traffic for some seconds. The close event is indicated by receiving an empty APDU. The relevant functions are:

EIBOpenT_Connection opens a T_Connection.

EIBOpenT_Individual opens a T_Data_Individual connection.

EIBOpenT_Group opens a T_Data_Group connection.

EIBOpenT_Broadcast opens a T_Data_Broadcast connection.

EIBOpenT_TPDU opens a raw connection.

EIBOpen_GroupSocket opens a group socket.

EIBSendAPDU sends an APDU over a T_Connection, T_Data_Broadcast, T_Data_Group or T_Data_Individual connection.

EIBGetAPDU receives an APDU over a T_Connection or T_Data_Individual connection.

EIBGetAPDU_Src receives an APDU with source address over a T_Data_Broadcast or T_Data_Group connection.

EIBSendTPDU sends a TPDU to destination address over a raw connection.

EIBGetTPDU receives a TPDU and a source address over a raw connection.

EIBSendGroup sends an APDU and a destination address over a group socket.

EIBGetGroup_Src receives a TPDU and a source and destination address over a group socket.

Connectionless functions

There are some management functions which can only be executed on connections where no mode switch has been executed. After that, the connection remains in the same state. The functions are:

- Switch programming mode: a 5 byte packet of the type EIB_PROG_MODE is sent to the daemon. Bytes 2–3 contain the address of the EIB device, byte 5 the function code:

  0 turns the programming mode on (implemented in EIB_M_Progmode_Off).

  1 turns the programming mode on (implemented in EIB_M_Progmode_On).

  2 toggles the programming mode (implemented in EIB_M_Progmode_Toggle).

3 gets the status of the programming mode (implemented in EIB_M_Progmode_Status).

If the request is successful, a packet of the same type is returned. If the state of the programming mode flag is requested, it is returned in the third byte.

- To list devices in programming mode, a packet of the type EIB_M_INDIVIDUAL_ADDRESS_READ is sent to the daemon. If the request is successful, the result has the same type and at every even address starting with 2 an EIB address is returned. It is implemented in EIB_M_ReadIndividualAddresses.

- To write the address in a device with activated programming mode, a packet of the type EIB_M_INDIVIDUAL_ADDRESS_WRITE with the new address in bytes 2–3 is sent. If the write is successful, a packet of the same type is returned. Other error codes are:

  EIB_ERROR_ADDR_EXISTS Address already in use.

  EIB_ERROR_MORE_DEVICE More than one device is in programming mode.

  EIB_ERROR_TIMEOUT No device is in programming mode.

  EIB_PROCESSING_ERROR An unspecified processing error occurred.

  This function is implemented in EIB_M_WriteIndividualAddress.

- To read the mask version, a packet of the type EIB_MASK_VERSION, with the address in bytes 2–3, is sent. If successful, a packet of the same type with the mask version in bytes 2–3 is returned. This function is implemented in EIB_M_GetMaskVersion.

- EIB_LOAD_IMAGE loads an image into a BCU. The request consists of an image in the BCU SDK image format stored in a packet of the type EIB_LOAD_IMAGE. As a result a packet of the same type will be returned. In the bytes 2–3, the load result is returned. The list of possible values is defined in the type BCU_LOAD_RESULT. The function is implemented in EIB_LoadImage.

Management connection

A management connection is opened by sending a packet of the type EIB_MC_CONNECTION with the address of the EIB device in bytes 2–3 to the daemon. If the request is successful, a packet of the same type is returned. This function is implemented in EIB_MC_Connect.

After opening, various management functions can be called. If a management connection is idle for some seconds, it is automatically closed. This has the result that all future calls will fail. The different functions are:

EIB_MC_PROG_MODE This basically does the same as EIB_PROG_MODE, only the request packet is different. Here the address of the device is not transmitted. Instead, the function code is transmitted at byte 3 (instead of byte 5). The return packet has the same structure, but the type is EIB_MC_PROG_MODE.

The functions are implemented in EIB_MC_Progmode_Toggle, EIB_MC_Progmode_On, EIB_MC_Progmode_Off and EIB_MC_Progmode_Status.

EIB_MC_MASK_VERSION An empty packet of this type is sent to the daemon and a packet of the same type with the mask version in bytes 2–3 is returned. It is implemented in EIB_MC_GetMaskVersion.

EIB_MC_PEI_TYPE to read the PEI type, a packet with this type is sent. If the request is successful, a packet of the same type with the PEI type in bytes 2–3 is returned. It is implemented in EIB_MC_GetPEIType.

EIB_MC_ADC_READ A packet with the ADC channel in byte 2 and the count in byte 3 is sent. A successful result has the same type and the ADC value in bytes 2–3. It is implemented in EIB_MC_ReadADC.

EIB_MC_PROP_READ Byte 2 of the request contains the object index, byte 3 the property ID, bytes 4–5 the start offset and byte 6 the element count. A successful result has the same type and contains the data read. It is implemented in EIB_MC_PropertyRead.

EIB_MC_READ A packet of this type with the address in bytes 2–3 and the length in bytes 4–5 is sent. The result is contained in a packet of the same type. It is implemented in EIB_MC_Read.

EIB_MC_PROP_WRITE Byte 2 of the request contains the object index, byte 3 the property ID, bytes 4–5 the start offset and byte 6 the element count. Then the data to be written follows. A successful write is acknowledged by a packet of the same type with the content of the response packet. It is implemented in EIB_MC_PropertyWrite.

EIB_MC_WRITE Bytes 2–3 contain the address, bytes 4–5 the length. The data to be written follows. A successful write is acknowledged by a packet of the same type. It is implemented in EIB_MC_Write.

EIB_MC_PROP_DESC A packet of this type with the object index in byte 2 and the property ID in byte 3 is sent to the server. The property type is returned in byte 2, the element count in bytes 3–4 and the access level in byte 5. It is implemented in EIB_MC_PropertyDesc.

EIB_MC_AUTHORIZE A packet of this type with the key in bytes 2–5 is sent to the server. The level returned by the device is delivered in byte 3 of a packet of the same type. It is implemented in EIB_MC_Authorize.

EIB_MC_KEY_WRITE Bytes 2–5 contain the key and byte 6 the level of the key write request. A successful request is acknowledged by a request of the same type. It is implemented in EIB_MC_SetKey.

EIB_MC_PROP_SCAN A request of this type is sent to eibd to get a list of all properties. If the request is successful, a packet of the same type is returned. For each property, 6 bytes are returned. Byte 0 contains the object index, byte 1 the property ID, byte 2 the property type. Bytes 3–4 contain the object type if the property ID is 1 and the property type is 4. Otherwise, they contain the element count. Byte 5 contains the access level. It is implemented in EIB_MC_PropertyScan.

Part III.

Using the BCU SDK

8. Input format

8.1. BCU configuration

The *BCU configuration* is defined in a text file:

- As comments, C style comments as well as line comments starting with # are supported.
- Different tokens are separated by white space (space, newline and tabulator). In some situations, white space is not necessary (e.g. between an identifier and a semicolon).
- Strings are used as in C. They can even consist of different parts, which are automatically concatenated. C escape sequences are supported.
- Identifiers start with a letter or an underscore. Any number of letters, numbers or underscores can follow. All identifiers are case sensitive.
- Numbers can be floating point numbers (in C format) or integer numbers, either decimal or hexadecimal. A hexadecimal number is prefixed with *0x*.
- Instead of integer or floating point constants, a subset of C expressions can be used. Such an expression may only contain constant expressions. All arithmetic and logic operands can be used for integer values. For floating point values, only arithmetic operations are supported.
- Boolean values can be set to the values *true* or *false*.
- Each entry consists of a keyword and its value. The entry is ended with a semicolon.
- If an entry is a block, it contains a list of entries enclosed in { and } as its value.
- If an entry contains a mapping, it contains a list of *IDENT = STRING ;* enclosed in { and } as its value.
- If an entry contains an array, it contains a list of array elements separated by commas and enclosed in { and } as its value.

The root element is a block called *Device*.

8.1.1. Device block

This block can have the following attributes:[1]

BCU Mandatory, selects the used BCU. Supported values are *bcu12*, *bcu20* and *bcu21*.

Model Optional identifier to select a different feature set. Available choices are:

- **histack** Only for BCU2; allocates the stack in high memory.

PEIType Mandatory integer, the PEI type used.

ManufacturerCode Optional integer.

InternalManufacturerCode Optional integer.

DeviceType Optional integer.

Version Optional integer.

SyncRate Optional integer, contains the raw value as specified in [BCU1] and [BCU2].

PortADDR Optional integer, DDR setting of Port A.

PortCDDR Optional integer, DDR setting of Port C.

on_run Optional function name, is executed every cycle.

on_init Optional function name, is executed at power on.

on_save Optional function name, is executed in the case of a power failure.

Title Mandatory string, contains the short description of the application program.

AddInfo Optional string, contains additional information text about the application.

OrderNo Optional string, contains the order number of the product.

Manufacturer Optional string, contains the manufacturer of the product.

Category Optional string, contains the hierarchical function class, e.g. *Application Modules / Push Button Sensor / Two Fold.*

Author Optional string, contains the author.

Copyright Optional string, contains copyright information.

Test_Addr_Count Optional integer, number of group addresses used in the test compile (for size estimation in *build.ai*).

[1] Some of these attributes provide access to a specific system setting. Under normal conditions, usable defaults are selected. For details about such settings, refer to [BCU1, BCU2].

Test_Assoc_Count Optional integer, number of associations used in the test compile (for size estimation in *build.ai*).

include An array of strings which contains the name of all used C files.

RouteCount Optional integer, start value for the routing counter.

BusyLimit Optional integer, BUSY retransmission limit.

INAKLimit Optional integer, INAK retransmission limit.

RateLimit Optional integer, set telegram rate limit (default: none).

CPOL Optional boolean, set CPOL (clock phase for serial synchronous interface).

CPHA Optional boolean, set CPHA (clock phase for serial synchronous interface).

AutoPLMA Optional boolean, enable automatic PLMA clear.

A_Event Optional boolean, enable A_Event generation.

BCU1_SEC Optional boolean, set M68HC05 SEC flag.

BCU1_PROTECT Optional boolean, enable BCU 1 EEPROM protection.

BCU2_PROTECT Optional boolean, enable BCU 2 EEPROM protection.

BCU2_WATCHDOG Optional boolean, enable BCU 2 watchdog.

PLM_FAST Optional boolean, select PLM frequency for BCU 2.

U_DELMSG Optional boolean, enables/disables the automatic call to *U_DELMSG*.

Additionally, blocks of the following types can be present:

- *FunctionalBlock*
- *IntParameter*
- *FloatParameter*
- *ListParameter*
- *StringParameter*
- *GroupObject*
- *Object*
- *Debounce*
- *Timer*
- *PollingMaster*
- *PollingSlave*

8.1.2. FunctionalBlock block

A functional block contains *Interface* blocks as well as the following attributes:

ProfileID Mandatory, contains the unsigned integer number describing the object type of the functional block as specified in [KNX] 3/7/3-2.2.

Title Mandatory, short description of the functional block as a string.

AddInfo Optional, additional textual information about the functional block as a string.

A functional block without any interfaces or only containing interfaces which do not reference anything is left out of the *application information*.

8.1.3. Interface block

This block has the following attributes:

DPType Mandatory, contains the DP Type ([KNX] 3/7/3-5) of the interface as floating point value. The DP Types, which can be accessed by name, are listed in Section B.1.

Title Optional, short description of the interface as a string.

AddInfo Optional, additional textual information about the interface as a string.

GroupTitle Optional, specifies the title of the group to which the interface belongs as string. This attribute is intended to provide the name of a property page, on which an integration tool should display the interface if the functional block is unknown to the integration tool.

Abbreviation Mandatory, the abbreviation of the interface as an identifier.

InvisibleIf Optional, contains an expression which indicates if it is appropriate to display this interface (in the context of the settings of other parameters). If the expression is not null, the interface should not be displayed in the integration tool.

The root expression must be a boolean expression (*exprb*). A valid expression conforms to the following grammar:

- exprb := expri
  is true, if the integer value is not 0.
- exprb := '(' exprb ')'
- exprb := '!' exprb
  NOT
- exprb := exprb '&&' exprb
  AND

- exprb := exprb '||' exprb
  OR
- exprb := ident 'IN' '(' ident [',' ident] * ')'
  The first ident must be the name of a ListParameter. All following idents must be the name of an element of the ListParameter. The expression is true, if the name of the current selected value of the ListParameter is in the ident list.
- exprb := exprs '==' exprs | exprs '! =' exprs | exprs '<' exprs | exprs '<=' exprs | exprs '>=' exprs | exprs '>' exprs
  returns true, if the a bytewise compare of the two strings fulfils the conditions.
- exprb := expri '==' expri | expri '! =' expri | expri '<' expri | expri '<=' expri | expri '>=' expri | expri '>' expri
  returns true, if the a byte wise compare of the two integer values fulfils the conditions.
- exprb := exprf '==' exprf | exprf '! =' exprf | exprf '<' exprf | exprf '<=' exprf | exprf '>=' exprf | exprf '>' exprf
  returns true, if the a bytewise compare of the two floating point values fulfils the conditions.
- exprs := '(' exprs ')'
- exprs := C-style string
- exprs := ident
  returns the current value of the StringParameter with the name ident.
- expri := any valid integer constant or constant integer expression
- expri := ident
  returns the current value of the IntParameter with the name ident.
- expri := expri '+' expri | expri '-' expri | expri '%' expri | expri '*' expri | expri '/' expri | '-' expri
- exprf := '(' exprf ')'
- exprf := any valid constant floating point expression
- exprf := ident
  returns the current value of the FloatParameter with the name ident.
- exprf := exprf '+' exprf | exprf '-' exprf | exprf '*' exprf | exprf '/' exprf | '-' exprf
- exprf := expri
- exprf := '(' exprf ')'

The precedence and meaning of operators are the same as in C.

Reference An array of identifiers, which are the names of a *GroupObject*, *PollingMaster*, *PollingSlave*, *Property* or a kind of parameter. If one of these objects is not referenced by an interface, it is left out of the *application information*. Additionally, most functions of these objects are removed, so that they consume less space. Their interface is not changed.

8.1.4. IntParameter block

This block defines a parameter of an integer type. The mandatory attribute *Name* contains the name of the constant which can be used to access the selected value in the program. The size and signedness are selected automatically, so that minimal space is used.

The other attributes are:

Title Mandatory, short description of the parameter as a string.

AddInfo Optional, additional textual information about the parameter as a string.

MinValue Mandatory, contains the lower bound as integer number.

MaxValue Mandatory, contains the upper bound as integer number.

Default Mandatory, contains the integer number which the integration tool should preselect.

Unit Optional, contains a string which represents the unit in which the value is measured.

Precision Optional, contains an integer value which describes the size of the smallest interval whose bounds the final BCU application will consider as separate values.

Increment Optional, contains an integer value which is the default increment value which the integration tool should offer if up/down buttons are shown.

8.1.5. FloatParameter block

This block defines a parameter of a floating point type. The mandatory attribute *Name* contains the name of the constant which can be used to access the selected value in the program.

The other attributes are:

Title Mandatory, short description of the parameter as a string.

AddInfo Optional, additional textual information about the parameter as a string.

MinValue Mandatory, contains the lower bound as a floating point number.

MaxValue Mandatory, contains the upper bound as a floating point number.

Default Mandatory, contains the floating point number which the integration tool should preselect.

Unit Optional, contains a string which represents the unit in which the value is measured.

Precision Optional, contains a floating point value which describes the size of the smallest interval whose bounds the final BCU application will consider as separate values.

Increment Optional, contains a floating point value which is the default increment value which the integration should offer if up/down buttons are shown.

8.1.6. ListParameter block

This block defines a parameter which provides a selection of one element out of a list. The mandatory attribute *Name* contains the name of the constant which can be used to access the selected value in the program. Internally, this parameter creates an enumeration. The enumeration type is called Name_*t*. The names of the enumeration entries are the identifiers of a mapping called *Elements*. As its values, this mapping contains the strings which should be shown in an integration tool.

The other attributes are:

Title Mandatory, short description of the parameter as a string.

AddInfo Optional, additional textual information about the parameter as a string.

Default Mandatory, contains the name of the preselected entry.

Unit Optional, contains a string which represents the unit in which the value is measured.

8.1.7. StringParameter block

This block defines a parameter of string type. The mandatory attribute *Name* contains the name of the constant which can be used to access the selected value in the program. The space allocated for the constant depends on the stored content.

The other attributes are:

Title Mandatory, short description of the parameter as a string.

AddInfo Optional, additional textual information about the parameter as a string.

MaxLength Mandatory integer, contains the maximum string length the application supports.

BCU SDK type name	C type
UINT1	uint1
UINT2	uint1
UINT3	uint1
UINT4	uint1
UINT5	uint1
UINT6	uint1
UINT7	uint1
UINT8	uint1
UINT16	uint2
TIME_DATE	uint3
FLOAT	float
DATA6	uint6
DOUBLE	double
DATA10	uint1[10]
MAXDATA	uint1[14]

Table 8.1.: Group object types

RegExp Optional string, regular expression which the string must match. The format of the regular expression is the same as used in XML Schema ([XML2], Appendix F).

Default Mandatory, contains the value as a string which the integration tool should preselect.

Unit Optional, contains a string which represents the unit in which the value is measured.

8.1.8. GroupObject block

The attribute *Name* contains the name of the variable under which the group object value will be available. The number of the group object will be available under the name Name_*no*.

Other attributes are:

Title Mandatory, short description of the group object as string.

AddInfo Optional, additional textual information about the group object as a string.

Type Mandatory, contains the type. The supported types are listed in Table 8.1.

Sending Optional boolean value, which is true if the application can transmit values via this group object. Generates the function Name_*transmit()*, which initiates the transmission of the group object value.

Receiving Optional boolean value, which is true if the application can receive values via this group object.

Reading Optional boolean value, which is true, if the application can send A_GroupValue_Read telegrams via this group object. Generates the function Name_*readrequest()*, which sends a read request. If the answer is received, the *on_update* handler is called, if present.

If both *Reading* and *Sending* are enabled, the function Name_*clear()* is generated. It is not possible to call Name_*transmit()* before the answer of an outstanding Name_*readrequest()* is received. Therefore Name_*clear()* is available to cancel an open read request, so that a normal transmit is possible again.

StateBased Mandatory boolean value, which is true if the group object contains state information (which means that a A_GroupValue_Read operation will return a meaningful value).

on_update Optional, contains the name of the functions which will be called when the value of the group object changes. It automatically activates *Receiving*.

eeprom Optional boolean value, which is true if the value of the group object should be allocated in the EEPROM. Transparent EEPROM access is automatically activated. The default is false.

8.1.9. Object block

This block describes an EIB user object (interface object). Apart from *Property* blocks, which describe the properties of the object, it has the following attributes:

Name Mandatory identifier, contains the internal name of the object.

ObjectType Object type as integer, as listed in [KNX] 3/7/3-2.

Title Optional string, unused.

AddInfo Optional string, unused.

8.1.10. Property block

The Property block contains the following attributes:

Name The name the property variable is accessible under.

Title Mandatory, short description of the property as a string.

AddInfo Optional, additional textual information about the property as a string.

Type Mandatory, contains the type as listed in Table 8.2.

BCU SDK type name	C type
PDT_CHAR	sint1
PDT_UNSIGNED_CHAR	uint1
PDT_INT	sint2
PDT_UNSIGNED_INT	uint2
PDT_KNX_FLOAT	uint2
PDT_DATE	uint3
PDT_TIME	uint3
PDT_LONG	sint4
PDT_UNSIGNED_LONG	uint4
PDT_FLOAT	float
PDT_DOUBLE	double
PDT_CHAR_BLOCK	uint1[10]
PDT_POLL_GROUP_SETTINGS	uint3
PDT_SHORT_CHAR_BLOCK	uint5
PDT_GENERIC_01	uint1
PDT_GENERIC_02	uint2
PDT_GENERIC_03	uint3
PDT_GENERIC_04	uint4
PDT_GENERIC_05	uint5
PDT_GENERIC_06	uint6
PDT_GENERIC_07	uint7
PDT_GENERIC_08	uint8
PDT_GENERIC_09	uint1[9]
PDT_GENERIC_10	uint1[10]

Table 8.2.: Property types

PropertyID Mandatory, contains the ID of the property as specified in [KNX] 3/7/3-3. Some property IDs can also be referred to using textual names. They are listed in Section B.2.

Writeable Mandatory boolean value, which is true if the property is writable.

MaxArrayLength Optional, contains the maximum number of elements of the property as integer.

eeprom Operation boolean, which is true if the variable should be located in the EEPROM. Transparent EEPROM access is activated.

handler The name of a function which is used to handle a custom property type.

If the *handler* is set, no variable will be allocated.

```
typedef struct
{
  bool write;
  uint1 ptr;
} PropertyRequest;

typedef struct
{
  bool error;
  uint1 ptr;
  uint1 length;
} PropertyResult;
```

The *handler* function takes a variable of the type *PropertyRequest* as its only parameter and returns a variable of the type *PropertyResult*.

If no *handler* is set, but *MaxArrayLength* is greater than one, a structure is allocated under the name given in *Name*. It contains the element *count*, which is the number of used elements. The other element is named *elements*, which is an array of *MaxArrayLength* elements of the type selected by *Type*.

In any other case, a normal variable with the type selected by *Type* is allocated under the name given in *Name*.

8.1.11. Debounce block

This block is used to allocate a debouncer. The mandatory entry *Name* contains the name of the debounce function. As it only implements an interface to the BCU functions (with correct reservation of all resources used), it only works for 8 bit values.

The second optional entry has the name *Time* and can contain the debouncing time in ms. If it is present, the defined function takes only an 8 bit value as parameter and returns the debounced value.

If no time is given, the debouncer function takes a second argument, which is the debouncing time in 0.5 ms steps.

For BCU 1 and BCU 2, only one debouncer is supported.

8.1.12. Timer block

Each timer has the mandatory identifier *Name*. It is used as the base for the names of the interface to the timer. Each timer stores its number in the constant Name*_no*.

The attribute *Type* can be one of the following:

UserTimer allocates a user timer. *Resolution* must contain a value for the user timers from Table 8.3. *on_expire* can contain the name of a function which is called periodically while the timer is expired.

Under the value of *Name*, an 8 bit variable is available which contains the current value. Name*_get()* returns true if the timer is expired. Name*_set(uint1 value)* loads the timer with the new value (between 0 and 127).

EnableUserTimer provides the same interface as a *UserTimer*. The difference is, that the *on_expire* function is only called once, if the time expires (and needs more memory). Name*_del()* cancels a running timer. If the timer is running, can be check by Name*_enable*, which is true as long as the timer is running.

SystemTimer only allocates a system timer. This timer can only be accessed by the BCU API.

CountDownTimer allocates a system timer in count down mode. *Resolution* must contain a value for the BCU 1 timers from Table 8.3. *on_expire* can contain the name of a function which is called periodically while the timer is expired.

Name*_get()* returns true if the timer is expired. Name*_set(uint1 value)* loads the timer with the new value.

DifferenceCounter allocates a system timer in difference counter mode. *Resolution* must contain a value for the BCU 1 timers from Table 8.3.

Name*_get()* returns the time difference to the last call. Name*_set(uint1 value)* loads the timer with the new value.

MessageTimer only works on a BCU 2. *Resolution* must contain a value for the BCU 2 message timer out of Table 8.3.

Name*_del()* stops the timer. Name*_set(uint1 value)* loads the timer with the new value.

MessageCyclicTimer only works on a BCU 2. *Resolution* must contain a value for the BCU 2 cyclic message timer out of Table 8.3.

Name*_del()* stops the timer. Name*_set(uint1 param)* starts the timer (*param* is put into the periodic message).

timer type	name	resolution
user timer	RES_133ms	133 ms
user timer	RES_266ms	266 ms
user timer	RES_533ms	533 ms
user timer	RES_1066ms	1066 ms
user timer	RES_2133ms	2133 ms
user timer	RES_4266ms	4266 ms
user timer	RES_8533ms	8533 ms
user timer	RES_17067ms	17.067 s
user timer	RES_34133ms	34.133 s
user timer	RES_68267ms	68.267 s
user timer	RES_2min16s	2 min 33 s
user timer	RES_4min33s	4 min 6 s
user timer	RES_9min6s	9 min 6 s
user timer	RES_18min12s	18 min 12 s
user timer	RES_36min24s	36 min 24 s
user timer	RES_72min48s	72 min 48 s
BCU 1 timer	RES_0_5ms	0,5 ms
BCU 1 timer	RES_8ms	8 ms
BCU 1 timer	RES_130ms	130 ms
BCU 1 timer	RES_2100ms	2.1 s
BCU 1 timer	RES_33s	33 s
BCU 2 message timer	RES_0_4ms	0.4 ms
BCU 2 message timer	RES_7ms	7 ms
BCU 2 message timer	RES_107ms	107 ms
BCU 2 message timer	RES_27s	27 s
BCU 2 cyclic message timer	RES_100ms	100 ms
BCU 2 cyclic message timer	RES_1s	1 s
BCU 2 cyclic message timer	RES_1min	1 min

Table 8.3.: Timer resolutions

The count of user timers is only limited by the memory available. For the user, two system timers are available. If the debouncer is used, only one of them is available.

8.1.13. PollingMaster block

It has the attributes:

Name The name which is used as the base name for all functions which are related to the polling master interface.

Title Mandatory, short description of the polling master interface as string.

AddInfo Optional, additional textual information about the polling master interface as a string.

This block needs a BCU 2. It is not implemented at the moment.

8.1.14. PollingSlave block

It has the attributes:

Name The name which is used as the base name for all functions which are related to the polling slave interface.

Title Mandatory, short description of the polling slave interface as string.

AddInfo Optional, additional textual information about the polling slave interface as a string.

This block needs a BCU 2. It is not implemented at the moment.

8.2. C files

The C files can contain any C code which is allowed by GCC. To avoid problems and create the smallest code, follow these guidelines:

- The preprocessor is run over all C sources in an extra pass to combine them into one file. At this time, the BCU header files are not included. Therefore you can only refer to your own defines.

  Some functions of the BCU SDK may be implemented as defines. Since they are handled specially and may be implemented differently in future versions, they only should be used in the specified way.

- Make all your functions and global variables static. This allows GCC to optimize the code better or even remove variables and functions.

  For event handlers, do not specify *static*, as this is already done in the header files if necessary. Adding the *static* attribute to an event handler may cause the program to break.

- The GCC floating point emulator is linked if necessary. However, try to avoid the use of floating point values, as it needs lots of memory. The floating point types have the standard names *float* and *double*.

- The BCU SDK includes *newlib* as runtime library by default. Operating system independent functions like *strcpy* should work, but need a lot of space. They are used by including the header file, as in any C program.

  Functions which perform I/O, need an operating system call, or do dynamic memory management (except *alloca*), are not supported. Newlib functions are documented in [NEW2, NEW3].

- Integer types from 1 to 8 bytes are available. A bigger type needs more code to handle and uses more RAM. The unsigned types are called *uint1* to *uint8*, the signed types *sint1* to *sint8*.

- Normal variables are placed in a section where enough free space is available. If the variable must be in a certain memory region (e.g. because the BCU operating system expects it there), the following modifiers can be added to the variable declaration:

  RAM_SECTION place it in low RAM at an address lower than 0x100. Such a variable is initialized with zero, even if an initializer is added.

  LOW_CONST_SECTION place a constant in the EEPROM between 0x100 and 0x1ff. You may not change the value of such a variable, as this will cause a checksum mismatch which will stop the application program. If possible, add the *const* qualifier.

  EEPROM_SECTION place it in the EEPROM between 0x100 and 0x1ff. The value of such a variable may be changed.

  An example:

  ```
  int x EEPROM_SECTION;
  ```

- To guarantee that variables are allocated in a certain order, place them in a structure.

- Avoid recursion, as the call stack is very limited.

- The present version of the GCC port only accesses byte blocks, as it does not uses bit operations. If you need to access only a single bit, this must be done with inline assembler statements.

- In some cases, the use of local variables results in larger code, because stack handling code is needed. Otherwise the memory locations of global variables cannot be reused for different functions.

- The BCU SDK supports transparent access to the EEPROM, even using pointers. EEPROM write access is only supported for variables which are allocated with the *EEPROM_SECTION* modifier. If the address of any other memory location is loaded into a transparent EEPROM access pointer and the pointer is dereferenced, anything can happen (even the EEPROM content may be changed in a way that the program will be destroyed).

  To use this feature, add *EEPROM_ATTRIB* to the variable declaration (e.g. *int x EEPROM_SECTION EEPROM_ATTRIB;*). If you use a pointer to an EEPROM variable, add *EEPROM_PTR_ATTRIB* to the pointer definition (*int EEPROM_PTR_ATTRIB* ptr;*). If the pointer variable itself is located in the EEPROM, *EEPROM_ATTRIB* is necessary.

  Be sure to use the right attribute. If an attribute cannot be added, GCC issues a warning. It cannot give any feedback for a wrong attribute of a pointer variable.

  Mixing the use of pointers to the EEPROM and normal memory can produce errors and warnings.

- Similar to the transparent EEPROM access attributes, *LORAM_ATTRIB* and *LORAM_PTR_ATTRIB* are available for memory locations between 0x000 and 0x0FF. The use of these attributes allows the generation of smaller code for memory access operations using pointers. If any other pointer is assigned to such a pointer, the user must take care that the other pointer really points into this low memory region.

- If you are doing multiplications or divisions, cast the operands to unsigned, as the signed functions are larger.

  On the BCU 1, the use of the API functions for multiplication and division can be beneficial in some situations, as it avoids the additional code for the GCC multiplication and division functions.

8.3. API functions

The following functions, which are not covered elsewhere, are provided by the BCU SDK:

- `void reset_watchdog();`

  Resets the watchdog counter in longer calculations.

- `void __U_transRequest (uchar no);`

  Instructs the BCU to transmit the content of group object *no*. It results in smaller code than the BCU API function *_U_transRequest()*.

- `void __U_flag_Set(uchar no, uchar bit);`

  Sets bit *bit* of the RAM flags of group object *no*.

- `void __U_flag_Clear(uchar no, uchar bit);`
  Clears bit *bit* of the RAM flags of group object *no*.

9. File format for data exchange with integration tools

Configuring an EIB system can be a complex task. Different, more or less intelligent, tools for solving this problem have been implemented. To aid the interworking between the BCU SDK and these tools, a clear interface is needed.

The most important existing solution is the ETS format, which is not publicly documented (and uses ZIP files encrypted with a password unknown to the public). Projects like [BASYS] have found workarounds to read this file and have partially decoded the file format.

As the password of the ETS format is unknown, writing such files is impossible. Using a similar file format (e.g. the same as the ETS format, but not encrypted) was not considered useful, because defining a new format gives the possibility to include new features. The primary goals of the new format are:

- Fully specified and freely available, so that it is easy to integrate into applications
- Easy to read and write
- Support for the current and upcoming features of EIB
- No direct image patching, so that it is suited for compile time parameterization, too
- Provide high level information to aid automatic configuration.

The format is based on XML and provides means to create custom formats based on it. The format provides support for nearly all constructs which can be implemented on a BCU, even in finer detail than supported by the current BCU system software.

An important point is that the format hides how an image is finalized and by which means this is done. The ETS uses image patching, which supports only small modifications (e.g. change of a byte). Because modern compilers create better code when they have more information, compiling the image at download time with all settings from the integration tool leads to smaller images in some cases.[1]

The format groups everything into functional blocks, which are a kind of high level description of the application. The organizational process of how functional blocks and their interfaces are defined is beyond the scope of this document. Some are defined in

[1]This is especially important because the GCC port produces larger code than a good assembler programmer.

the KNX standard [KNX]. If no appropriate functional block exists, the application programmer may (and will have to) create his own.

The interfaces of the functional blocks contain all necessary information concerning the format of the data types and how to interpret them. The objects which actually implement the interface only contain a basic type, which in most cases does not represent more than the size of the data type.

9.1. Configuration process

The process from the creation of a BCU application program to its download is as follows:

1. The developer writes an application program.

2. It is turned into an *application description*.

3. Some *application descriptions* are collected, their meta data are unified and turned into a *product catalog* for an integration tool. Additionally, language translations are included in this step.

4. The *product catalog* is imported into an integration tool.

5. The configuration of an EIB installation is planned based on the imported data.

6. The integration tool sends the necessary configuration for each device to the download part as a *configuration description*.

7. The download part of the BCU SDK creates a downloadable image and writes it into the BCU memory.

The format and creation of the *product catalog*, as well as how the integration tool works, are out of scope of this document.

For the translations, two alternatives exist:

- After an *application description* has been created by the BCU SDK, translations into other languages are put in XML tags using the *lang* attribute.

  The big disadvantage is that, for new translations, the whole file has to be changed.

- Use the GNU gettext system. After creating the *application information*, all texts are extracted. The resulting file is passed to a translator, who creates the translations. Then the translations for one language are bundled in catalogs, which are used by integration tools to replace the text.

  This system is used by nearly all Linux applications and has proven to work. Additionally, the automatic reuse of translations is possible and there is no need for releasing translations and applications at the same time.

How the code is passed between the BCU SDK and the download tool is not important for the integration tool. Two alternatives are possible:

- The code is stored in another file (using an internal file format) and imported in the download tool.
- The code is embedded as a large text block in the *application description* (again, using an internal format).

9.2. Basic definitions

The exchange format is based on XML. For the two exchange directions, different formats are described using XML Schema as well as a DTD. As the DTD is not as powerful, the XML Schema should be used for validation, if possible.

To reference nodes in the XML tree, unique IDs are used. The structure of them may be freely chosen as long as they are compatible with XML.

Each exchange format has a version attribute, which allows the application to recognize when it is receiving data in an older or newer exchange format. If an older format is received, it should be readable without any problems because the structure is known to the programmer. If a newer format is received, a warning should be issued. All known elements should be read, the rest ignored. In this case, the document structure should not be checked, because the check will fail.

The current version is *0.0.0*. All text fields should contain English text in *UTF-8*. All group addresses and polling addresses are stored as a hexadecimal number. For group addresses, additionally the formats *x/y/z* (5/3/8 bit) and *x/y* (5/11 bit) must be understood.

Each document should contain a reference to the corresponding schema or DTD. Representation details which are covered by the Schema definition (e.g. how to represent a boolean, ordering implied by the DTD, ...) are left out in the following description.

9.3. Application information

The root node is called *DeviceDesc* and stores the version in the attribute *version*.

The first mandatory element is called *ProgramID*. This node contains some text, which may not be interpreted by the integration tool. A node with the same content has to be written to the *configuration description* to indicate on which application description it is based. There are no limitations concerning the size or the structure of the text. This attribute can be used to pass a globally unique identifier or a text encoded intermediate format of the code.

The next mandatory node is called *Description*, which contains meta information about the program. It can have the following nodes:

MaskVersion Mandatory, contains the mask version of the device the application is designed for as a hexadecimal number string. Additionally, this can be used by the integration tool to determine if certain features are available in the target BCU.

Title Mandatory, contains the short description of the application program.

AddInfo Optional, contains additional information text about the application.

OrderNo Optional, contains the order number of the product as a string.

Manufacturer Optional, contains the manufacturer of the product as a string.

Category Optional, contains the hierarchical function class, e.g.: *Application Modules / Push Button Sensor / Two Fold* as a string.

Author Optional, contains the author as a string.

Copyright Optional, contains copyright information as a string.

9.3.1. Functional block

One or more functional blocks follow the *Description* node. For each functional block, a node with the name *FunctionalBlock* is present. It has the attribute *id*, which contains a unique ID. It contains the following nodes:

ProfileID Mandatory, contains the unsigned integer number describing the object type of the functional block as specified in [KNX] 3/7/3-2.2.

Title Mandatory, short description of the functional block as a string.

AddInfo Optional, additional textual information about the functional block.

9.3.2. Interface

For each interface, the functional block contains a node with the name *Interface*. This node also has a unique ID in the attribute *id*. It contains the following nodes:

DPType Mandatory, contains the DP-Type ([KNX] 3/7/3-5) of the interface.

Title Optional, short description of the interface as a string.

AddInfo Optional, additional textual information about the interface.

Abbreviation Mandatory, the abbreviation of the interface as a string.

GroupTitle Optional, specifies the title of the group to which the interface belongs. This attribute is intended to provide the name of a property page on which an integration tool should display the interface, if the functional block is unknown to the integration tool.

InvisibleIf Optional, contains an expression which indicates if it is appropriate to display this interface (in the context of the settings of other parameters). If the expression is not empty, the interface should not be displayed in the integration tool.

The root expression must be a boolean expression (*exprb*). A valid expression conforms to the following grammar:

- exprb := ’(’ exprb ’)’
- exprb := ’!’ exprb
  NOT
- exprb := exprb ’&&’ exprb
  AND
- exprb := exprb ’||’ exprb
  OR
- exprb := ident ’IN’ ’(’ ident [’,’ ident] * ’)’
  The first ident must be the ID of a *ListParameter*. All following idents must be the IDs of an element of the *ListParameter*. The expression is true, if the ID of the current selected value of the *ListParameter* is in the ident list.
- exprb := exprs ’==’ exprs | exprs ’! =’ exprs | exprs ’<’ exprs | exprs ’<=’ exprs | exprs ’>=’ exprs | exprs ’>’ exprs
  returns true, if the a byte wise compare of the two strings fulfils the conditions.
- exprb := expri ’==’ expri | expri ’! =’ expri | expri ’<’ expri | expri ’<=’ expri | expri ’>=’ expri | expri ’>’ expri
  returns true, if the a byte wise compare of the two integer values fulfils the conditions.
- exprb := exprf ’==’ exprf | exprf ’! =’ exprf | exprf ’<’ exprf | exprf ’<=’ exprf | exprf ’>=’ exprf | exprf ’>’ exprf
  returns true, if the a byte wise compare of the two floating point values fulfils the conditions.
- exprs := ’(’ exprs ’)’
- exprs := C-style string
- exprs := ident
  returns the current value of the *StringParameter* with the ID ident.
- expri := $[0-9]+$ decimal integer expression
- expri := ident
  returns the current value of the *IntParameter* with the ID ident.
- expri := expri ’+’ expri | expri ’-’ expri | expri ’%’ expri | expri ’*’ expri | expri ’/’ expri | ’-’ expri
- exprf := ’(’ exprf ’)’
- exprf := C-style floating point number

- exprf := ident
  returns the current value of the *FloatParameter* with the ID ident.
- exprf := exprf '+' exprf | exprf '-' exprf | exprf '*' exprf | exprf '/' exprf | '-' exprf
- exprf := expri
- exprf := '(' exprf ')'

The precedence and meaning of operators is the same as in C.

The link to the actual objects is made by one or more references with the element *Reference*. It has no content, only the attribute *idref*. *idref* contains the ID of a group object, property, polling object or parameter, which the interface represents.

Then the definitions of the properties, parameters, polling objects and group objects follow.

9.3.3. Group objects

A group object is defined by the node *GroupObject* and contains a unique ID in the attribute *id*. It contains the following elements:

Title Mandatory, short description of the group object as a string.

AddInfo Optional, additional textual information about the group object.

GroupType Mandatory, contains the basic type number of the group object as specified in [KNX] 3/5/1-4.3.2.1.2.1.

Sending Mandatory boolean value, which is true, if the application can send values (i.e., transmit A_GroupValue_Write requests) via this group object.

Receiving Mandatory boolean value, which is true, if the application can receive values via this group object. This includes both A_GroupValue_Write indications and A_GroupValue_Read responses.

Reading Mandatory boolean value, which is true, if the application can send A_GroupValue_Read requests via this group object.[2]

StateBased Mandatory boolean value, which is true if the group object contains state information, i.e., if an A_GroupValue_Read request will return a useful answer (cf. the *StateBased* keyword in the *BCU configuration*, Section 8.1).

[2]Although it will usually be reasonable that *Receiving* is enabled together with this option, it is not mandatory.

9.3.4. Properties

A group object is defined by the node *Property* and contains a unique ID in the attribute *id*. It contains the following elements:

Title Mandatory, short description of the property as a string.

AddInfo Optional, additional textual information about the property.

PropertyType Mandatory, contains the property type as specified in [KNX] 3/7/3-4.

ObjectIndex Mandatory, contains the index at which the property is available.

PropertyId Mandatory, contains the ID of the property as specified in [KNX] 3/7/3-3.

Writeable Mandatory boolean value, which is true if the property is writable.

MaxArrayLength Mandatory, contains the maximum number of elements of the property.

9.3.5. Polling master

A group object is defined by the node *PollingMaster* and contains a unique ID in the attribute *id*. It has the following elements:

Title Mandatory, short description of the polling master interface as a string.

AddInfo Optional, additional textual information about the polling master interface.

Here, no type information is given because the basic type is only one byte. The actual interpretation is defined by the DPType in the interface.[3]

9.3.6. Polling slave

A group object is defined by the node *PollingSlave* and contains a unique ID in the attribute *id*. It contains the following elements:

Title Mandatory, short description of the polling slave interface as a string.

AddInfo Optional, additional textual information about the polling slave interface.

[3] At the moment, there is no DPType for this. When functional blocks with polling will be introduced, a DPT needs to be allocated.

9.3.7. Parameter

All parameters are grouped together in the node *Parameter*. For each parameter, there is a child node which carries its unique ID in the attribute *id*.

Possible parameters are:

ListParameter Represents a selection of one element out of a list. It has the following elements:

- **Title** Mandatory, short description of the parameter as a string.
- **AddInfo** Optional, additional textual information about the parameter.
- **ListDefault** Mandatory, an empty node which contains the ID of the preselected list element in the attribute *idref*.
- **ListElement** For each list element, there is one node. It contains the element text as a string and has a unique ID stored in the attribute *id*.
- **Unit** Optional, contains some text which represents the unit the value is measured in.

IntParameter This parameter represents an integer value. It has the following elements:

- **Title** Mandatory, short description of the parameter as a string.
- **AddInfo** Optional, additional textual information about the parameter.
- **MinValue** Mandatory, contains the lower bound as integer number.
- **MaxValue** Mandatory, contains the upper bound as integer number.
- **Default** Mandatory, contains the value which the integration tool should preselect.
- **Unit** Optional, contains some text which represents the unit for measuring the value.
- **Precision** Optional, contains an integer value which describes the size of the smallest interval whose bounds the final BCU application will consider as separate values.
- **Increment** Optional, contains an integer value which is the default increment value the integration tool should offer if up/down buttons are shown.

FloatParamter This parameter represents a floating point value. It has the following elements:

- **Title** Mandatory, short description of the parameter as a string.
- **AddInfo** Optional, additional textual information about the parameter.
- **MinValue** Mandatory, contains the lower bound as a floating point number.
- **MaxValue** Mandatory, contains the upper bound as a floating point number.
- **Default** Mandatory, contains the value which the integration tool should preselect.

Unit Optional, contains some text which represents the unit for measuring the value.

Precision Optional, contains a floating point value which describes the size of the smallest interval whose bounds the final BCU application will consider as separate values.

Increment Optional, contains a floating point value which is the default increment value the integration tool should offer if up/down buttons are shown.

StringParameter Asks for a string. It has the following elements:

Title Mandatory, short description of the parameter as a string.

AddInfo Optional, additional textual information about the parameter.

MaxLength Mandatory, contains the maximum string length which the application supports.

RegExp Optional, regular expression the string must match. The format of the regular expression is the same as used in XML Schema ([XML2], Appendix F).

Default Mandatory, contains the value which the integration tool should preselect.

Unit Optional, contains some text which represents the unit for measuring the value.

9.4. Configuration description

The root node is called *DeviceConfig* and stores the version in the attribute *version*. The first mandatory attribute is called *ProgramID* and contains the same value as in the associated *application information*.

All IDs have the same values as in the *application information*, if they refer to the same element. Also, every parameter of the *application information* must be represented by the corresponding node in this exchange format.

The next mandatory node is *PhysicalAddress*, which contains the individual address of the device to which the program is to be downloaded.[4] The format is "x.y.z". Additionally a 16 bit hex value must be also understood.

If the device is already locked by a key at download time, this key is stored as a hexadecimal number in the optional node *InstallKey*. For each key, which should be stored in the device after download, a *Key* node is present. It contains the key as hexadecimal number and the level in the attribute *id*.

Then the settings of the properties, parameters, polling objects and group objects follow.

[4]The assignment of individual addresses has to be done separately from the download process.

9.4.1. Group objects

A group object is configured with the node *GroupObject* and contains the unique ID this group object has in the *application information* in the attribute *id*. It contains the following elements:

Priority Optional, contains one of the values *low*, *normal*, *urgent* or *system* specifying the priority of the messages transmitted via this group object.

SendAddress Optional, contains the group address to send A_GroupValue_Write telegrams to. May only be present if *Sending* is true. If not present, sending is disabled.

ReadRequestAddress Optional, contains the group address to send A_GroupValue_Read telegrams to. May only be present if *Reading* is true. Note that processing the answer must be enabled separately (via *ReceiveAddress* for a BCU 1 or *UpdateAddress* for a BCU 2).

ReceiveAddress Optional, contains a list of group addresses (each in a *GroupAddr* node). A_GroupValue_Write telegrams with these destination addresses will be processed by the group object. May only be present if *Receiving* is true. If not present, receiving is disabled.

ReadAddress Optional, contains a list of group addresses (each in a *GroupAddr* node). A_GroupValue_Read request with these destination addresses will be answered with the value of the group object. If not present, no A_GroupValue_Read will be processed.

UpdateAddress Optional, contains a list of group addresses (each in a *GroupAddr* node). A_GroupValue_Response frames with these destination addresses will update the value of the group object. If not present, no A_GroupValue_Response frame will be processed.

9.4.2. Property

A property is configured with the node *Property* and contains the unique ID in the attribute *id*. It contains the following elements:

Disable Optional boolean value; if it is true, the property should be deactivated. The default value is false.

ReadOnly Optional boolean value; if it is true, the property should be set read only. The default value is false. This option only makes sense if *Writeable* is true.

ReadAccess Optional; contains the read access level of the property. The default value is set to no access restriction.

WriteAccess Optional; contains the write access level of the property. The default value is set to no access restriction.

9.4.3. Polling master

A polling master is configured with the node *PollingMaster* and contains the unique ID in the attribute *id*. If present, it contains the following elements:

PollingAddress Mandatory, contains the polling address which the master uses.

PollingCount Mandatory, contains the number of the highest polling slot used for this address (numbering starts with 1).

9.4.4. Polling slave

A polling slave is configured with the node *PollingSlave* and contains the unique ID in the attribute *id*. If present, it contains the following elements:

PollingAddress Mandatory, contains the polling address which the slave listens to.

PollingSlot Contains the number of the polling slot to send the polling result (numbering starts with 0).

9.4.5. Parameter

All parameters are grouped together in the node *Parameter*. For each parameter, there is a child node, which has its unique ID stored in the attribute *id* (with the same *id* as in the *application information*).

Possible parameters are:

ListParameter Contains the ID of the selected list element in the node *Value*.

IntParameter contains the selected integer value in the node *Value*. It must meet all constraints defined in the *application information*.

FloatParameter contains the selected floating point value in the node *Value*. It must meet all constraints defined in the *application information*.

StringParameter contains the selected string value in the node *Value*. It must meet all constraints defined in the *application information*.

9.5. Limitations

As a BCU has only limited capabilities, an integration tool must know that the following limitations exist:

- All access control features will not work for the BCU 1.
- If different access levels are specified for properties with the same object index, the highest one will be used for the BCU 2.

- The sending address for group objects is implicitly added to the other address lists used, if it is not present there (for BCU 1 and BCU 2).
- The union of the different listening addresses for a group object will be used, if they are not the same (for BCU 1 and BCU 2).
- The *UpdateAddress* is not supported for the BCU 1.
- For BCU 1 and BCU 2, *SendAddress* and *ReadRequestAddress* must have the same value, if both are set.

If such a situation is configured, a warning will be issued by the downloader.

BCU 1 have the mask versions (hexadecimal) 0010, 0011 and 0012. BCU 2 have the mask versions 0020 and 0021.

10. Usage/Examples

10.1. Installation

The BCU SDK can either be installed as *root* in a global location, or in the home directory of a user. In the latter case, no root access is needed. Loading the EIB kernel drivers requires root access. The rights to use all eibd back ends can be granted to a normal user. Up to 2 GB of disk space will be used during compilation.

10.1.1. Installation in a home directory

In the following, it will be assumed that the home directory is */home/user*.

First, add */home/user/install/bin* to the *PATH* and */home/user/install/lib* to *LD_LIBRARY_PATH*. The simplest way to achieve this is to add

```
export PATH=/home/user/install/bin:$PATH
export LD_LIBRARY_PATH=/home/user/install/lib:$LD_LIBRARY_PATH
```

to the user's *.bashrc* and *.bash_profile* files. After logging out and logging in again, the environment should contain the changes.

For each run of *configure*, add *–prefix=/home/user/install* to the command line.

10.1.2. Prerequisites

The BCU SDK needs the following software to be installed:

- A reasonably recent Linux system (e.g. Fedora Core 2 or Debian Sarge)
- GCC and G++ in version 3.3 or higher
- libxml2 (plus development files) 2.6.16 or higher
- flex (Debian users must install flex-old, if it is present, instead of flex)
- bison
- GNU make
- xsltproc (part of libxslt)
- xmllint (part of libxml2 sources)

If a tool is too old, it is possible to install a newer version in the home directory of the user.

10.1.3. Getting the source

Download the following from *http://www.auto.tuwien.ac.at/~mkoegler/index.php/bcus*:

- BCU SDK sources
- pthsem sources
- all parts of a snapshot of the GNU utilities[1]

10.1.4. Installing GCC

All parts of the snapshot have to be extracted in the same, empty directory. Then enter this directory and run

```
./configure --target=m68hc05 --enable-gdbtk=no --disable-newlib-io-float
make
make install
```

After the process has finished, m68hc05-gcc should be available for execution.

10.1.5. Installing pthsem

For installing pthsem, either a RPM or deb file can be built and installed as root. As another alternative, extract the pthsem sources into a directory, enter this directory and run

```
./configure
make
make install
```

10.1.6. Installing the BCU SDK

Extract the BCU SDK sources, run *./configure –help* and decide which back ends are necessary. Then run *./configure* with the appropriate parameters. Finally, run *make install*. By default, all back ends are disabled, but it is no problem to enable them all at the same time.

10.1.7. Granting EIB access to normal users

To allow all local users to use eibd back ends, which use the serial driver, run as root:

```
chmod 666 /dev/ttyS0
chmod 666 /dev/ttyS1
```

[1]Starting with version 0.0.1, only one big tarball exists.

If the TPUART or bcu1 driver is used, load the driver as root and run

```
#for TPUART driver
chmod 666 /dev/tpuart0
chmod 666 /dev/tpuart1
#for BCU1 driver
chmod 666 /dev/eib
```

to grant access to all users.

If you are loading a driver, unload the serial driver or detach the serial driver from the port (as root):

```
setserial /dev/ttySx uart none
```

The interface file used by the USB backend is generated dynamically. It changes its name with every reconnect of the USB cable or reboot. To determine the name for the current session, use *findknxusb* (as discussed in Section 10.2.2) to determine the USB bus number and device number of the KNX interface. The interface file has the name */proc/bus/usb/*bus-number/device-number. Both numbers have to be expanded to 3 digits with leading zeros. Issue *chmod 666* filename to grant access to all users.

10.1.8. Development version

The development versions are only available as GIT repositories. All repositories are located at `http://www.auto.tuwien.ac.at/~mkoegler/git`. To access these versions, install Cogito and GIT[2], which are available at `http://www.kernel.org/pub/scm`.

To get a copy of a repository, issue

```
cg-clone http://www.auto.tuwien.ac.at/~mkoegler/git/REPOSITORY-NAME.git/
```

To update a copy, issue

```
cg-update
```

A tar file can be created by

```
cg-export TARFILE
```

Before the bcusdk repository can be built, the following commands must be issued:[3]

```
aclocal
autoheader
automake -a --foreign
autoconf
```

To build a tar file of the bcusdk, you must issue a *configure* command followed by

```
make dist
```

[2]At the time of writing, the most recent versions are Cogito 0.15.1 and GIT 0.99.9d

[3]If multiple automake or autoconf versions are installed, use the coresponding command of the lastest version.

10.1.9. Building install packages

Starting with the release of BCU SDK 0.0.1, all files to build a native package for rpm and deb based distributions are included. To build a rpm package, issue (as root)

```
rpmbuild -ta TARFILE
```

For the build process, the same order must be followed as for a normal installation (first pthsem build and install, then m68hc05-gnu and finally the bcusdk).

To build a Debian package, first extract the tar file and then issue:

```
chmod a+x debian/rules
dpkg-buildpackage -rfakeroot
```

10.2. Using eibd

It is possible to run eibd as daemon. However, this is not recommended for security reasons. Do not use the *bcu1s* and *tpuarts* back ends on Linux 2.6.X systems where X is lower than 11. In general, Linux 2.4 works more reliably, as it was noticed that a 2.6 kernel will in some situations delay a program for several seconds, so that deadlines in the bus communication will be missed.

10.2.1. Command line interface

The usage of eibd is:

```
eibd [OPTION...] URL
```

Possible URLs are:

ft12:/dev/ttySx connects to a BCU 2 over a serial line

bcu1:/dev/eib connects to a BCU 1 via the BCU1 kernel driver

tpuart24:/dev/tpuartX connects to a TPUART using the kernel driver on a Linux 2.4 kernel

tpuart:/dev/tpuartX connects to a TPUART using the kernel driver on a Linux 2.6 kernel

ip:[multicast_addr[:port]] connects via an EIBnet/IP router using the EIBnet/IP Routing protocol (its routing tables must be set up correctly)

ipt:router-ip[:dest-port[:src-port]] connects via an EIBnet/IP router using the EIBnet/IP Tunneling protocol (the router must be set up correctly)

bcu1s:/dev/ttySx connects to a BCU 1 over the experimental user mode driver. For more details, see 7.3.2.

tpuarts:/dev/ttySx connects to a TPUART using the user mode driver.

usb:[bus[:device[:config[:interface]]]] connects over a KNX USB interface

The options are:

-d, –daemon[=FILE] Start the program as daemon. The output will be written to FILE, if the argument is present.

-e, –eibaddr=EIBADDR Set our own EIB-address to EIBADDR (default 0.0.1) for drivers which need such an address.

-i,–listen-tcp[=PORT] Listen at TCP port PORT (default 6720).

-p, –pid-file=FILE Write the PID of the process to FILE.

-t, –trace=LEVEL Set the trace level.

-u, –listen-local[=FILE] Listen at Unix domain socket FILE (default /tmp/eib).

The TPUART and EIBnet/IP Routing back ends need an EIB address, which must be specified by the *-e* option. For better security, only pass the *-u* option and avoid using *-i*.

If you are using eibd with the ft12 back end using */dev/ttyS0*, run the following in a terminal:

```
eibd -u ft12:/dev/ttyS0
```

If you experience any problems, pass *-t1023* as option, which will print all debugging information. The program can be ended by pressing Ctrl+C.

10.2.2. USB backend

USB devices have no fixed name. The command *findknxusb* lists the addresses and endpoints of all USB devices connected which match the interface properties of a KNX USB interface (HID device class with 64 byte interrupt endpoints for input and output). If the access rights for the current user are correct, the manufacturer and product name are printed, else *Unreadable*. If the name can not be displayed, it does not make sense to run eibd on this interface, as communication with the device is not possible for the current user.

Its output is one or more lines like the following:

```
device 4:17:1:0 (<Unreadable>:<Unreadable>)
```

The first part is the bus number, the second the device number. Beware that this list may contain devices which are not KNX interfaces. Please use the product name to verify that the correct device is used by eibd. If eibd interacts with another USB device, it could harm the device.

The address of an USB device consists of four parts separated by colons. This address is passed with the prefix *usb:* as URL to eibd. The last parts can be omitted if only one device matches the specified first parts. Otherwise, eibd will randomly select one of the matching devices. If only one device is found at all, the address can be left empty.

10.2.3. EIBnet/IP server

Eibd can also act as a simple EIBnet/IP server, which provides access to the specified back end. This function is enabled with the *-S* switch. If no multicast address is given, the default value is used. Beware, that any character following the *-S* is interpreted as IP address (or DNS name). The supported service protocols are enabled using other options (e.g. pass the parameter *-TRDS* to eibd to enable all functions).

The options are:

-D, –Discovery enable the EIBnet/IP server to answer discovery and description requests(SEARCH, DESCRIPTION).

-R, –Routing enable EIBnet/IP Routing in the EIBnet/IP server

-S, –Server[=ip[:port]] starts the EIBnet/IP server part

-T, –Tunnelling enable EIBnet/IP Tunneling in the EIBnet/IP server

Enabling this server prohibits the normal bus monitor mode (vBusmonitor is still working). All other *eibd* functions are not affected. Eibd does not provide the ability to use filter tables for routing. This means that telegram loops can easily occur when it is used in parallel with another EIBnet/IP router on the same line.

The EIBnet/IP server front end of eibd performs a kind of network address translation on individual addresses. In outgoing frames, *0.0.0* is replaced with the individual address of the bus access device. Likewise, incoming frames with the individual address of the bus access device as destination have this destination address replaced with *0.0.0*.[4] Also, *0.0.0* is returned as the KNX individual address assigned to the Tunnelling connection in the connection response data block.

Source addresses other than *0.0.0* can be used for outgoing frames, but incoming frames addressed to individual addresses other than that of the bus access device are suppressed.[5] Therefore, *0.0.0* should be used as the local individual address by eibd-EIBnet/IP client applications. Tunneling clients which use the address returned in the connection response will do so automatically.

[4]For outgoing frames, this translation is consistent with [KNX] AN033, 2.5.3.3 (cEMI L_Data.req). Applying this concept to incoming frames is an extension to the KNX specification.

[5]Note that this restriction, which is due to the limitations of the bus access devices and/or drivers used, significantly limits the use of the Routing protocol with eibd.

10.2.4. Example programs

If eibd is started as described above, use *local:/tmp/eib* as URL.
The following useful example programs are installed:

busmonitor1 Decoding bus monitor

busmonitor2 Raw bus monitor

vbusmonitor1 Decoding vBusmonitor

vbusmonitor2 Raw vBusmonitor

progmodeon Turn programming mode of a device on

progmodeoff Turn programming mode of a device off

progmodestatus Return the state of the programming mode flag

progmodetoggle Toggle programming mode of a device

readindividual List all devices in programming mode

writeaddress Write an individual address to a device in programming mode

groupwrite Send a group write telegram to a group address (for values with more than 6 bit width)

groupswrite Send a group write telegram to a group address (for values with less than 6 bit width)

groupresponse Send a GroupValue_Response response message to a group address (for values with more than 6 bit width)

groupsresponse Send a GroupValue_Response response message to a group address (for values with less than 6 bit width)

groupread Send a group read telegram to a group address. The result is not captured by this tool. It has to be monitored by a bus monitor.

grouplisten Displays all received frames with a particular (destination) group address.

mmaskver Read the mask version of a device

mpropscan Return a list of all properties of a device

mpropdesc Return information about a property

mpeitype Return the current PEI type of a device

madcread Read an ADC channel

mread Read program memory of a device

mwrite Write program memory of a device

mpropwrite Write a property of a device

mpropread Read a property of a device

msetkey Set key for an access level

mprogmodeon Turn programming mode of a device on

mprogmodeoff Turn programming mode of a device off

mprogmodestatus Return the state of the programming mode flag

mprogmodetoggle Toggle programming mode of a device

All connection oriented management programs (i.e. programs with a *m* as prefix in the name) accept a optional key which will be used for authentication before the respective management procedure is executed. This key is passed as -*k* KEY before the URL of the *eibd* connection.

10.2.5. Usage examples

```
#turn the programming mode on on 1.3.1
progmodeon local:/tmp/eib 1.3.1
#get the mask version of of 1.3.1
mmaskver local:/tmp/eib 1.3.1
#write the address 1.3.1 to the device in programming mode
writeaddress local:/tmp/eib 1.3.1
#run a bus monitor
busmonitor1 local:/tmp/eib
#run a vBusmonitor
vbusmonitor1 local:/tmp/eib
#send a group write telegram with value 1 to 0/1/1
groupswrite local:/tmp/eib 0/1/1 1
#send group write for values > 6 bits, here 0x01 0x02
groupwrite local:/tmp/eib 0/1/1 1 2
#read memory 0x100 (16 bytes) on 1.3.1
mread local:/tmp/eib 1.3.1 100 16
#write 0x11 0x12 to 0x105 on 1.3.1
mwrite local:/tmp/eib 1.3.1 105 11 12
```

10.2.6. *eibd* utilities

The following helper programs are distributed with eibd:

eibnetsearch discovers all EIBnet/IP server listening on a particular multicast address. For the default address, pass only – as parameter. A list of all EIBnet/IP servers that answered the search request, together with their IP addresses and ports will be returned.

If an EIBnet/IP server supports service type 4, it supports the EIBnet/IP Tunneling mode of eibd. If an EIBnet/IP server supports service type 5, it supports the EIBnet/IP Routing mode of eibd. The returned multicast address can be used to construct a EIBnet/IP Routing URL, the returned individual IP address to construct a EIBnet/IP Tunneling URL.

eibnetdescribe expects the IP address of a IP router and returns its capabilities (similar to eibnetsearch, but only for a specific device).

findknxusb lists the addresses of the available KNX USB interfaces.

bcuaddrtab expects a URL of a BCU back end (FT1.2, USB or BCU 1) and displays the address table size of the interface BCU. Using the *–w* parameter, it can be changed to an other value.

If a BCU is used as EIB bus interface, this value should be changed to 0 to allow all incoming group telegrams to be received by eibd. While this setting is changed, the application program will not work. The original value must be restored to make it work again.

bcuread expects the URL of a BCU back end (similar to *bcuaddrtab*, FT1.2, USB or BCU 1), a starting memory address (in hexadecimal notation) and a length value between 1 and 8 (bytes). It displays the content of the specified memory region of the local BCU.

10.3. Recovering from errors

As the BCU SDK provides a wide range of possibilities to cause damage, methods for how to recover from errors will be shown in the following. As access to the PEI is needed for some error recovery techniques, it is not recommended to use the BCU SDK when the PEI connector is not accessible.

- If a BCU program contains a severe error, the BCU 1 can stop responding to all telegrams. In such a case, directly connect pins 5 and 6 of the PEI. These are the pins at the short side of the PEI. If the pins on the other side are connected, no harm will be done because these are both ground pins.

  While they are connected, the BCU should start to respond again. In this situation, it may be necessary to reprogram the individual address. Then a new application program can be downloaded. After this, the original PEI type can be restored.

- If a BCU 2 shows a malfunction which does not disappear after a reset (power cycle), you need to perform a master reset. Connect pins 5 and 6 of the PEI. Then turn off the bus power (or disconnect the BCU from the bus). Keep the programming button pressed and restore the power.

  Then the relevant content of the BCU memory should have been deleted. The PEI type can be restored and after reprogramming the individual address, a new application program can be loaded.

- ETS says that a BCU has the wrong manufacturer (or is an other type): Experience has shown that in this case typically the content of the manufacturer data is destroyed. To solve the problem, rewrite the memory locations between 0x101 and 0x106. The correct values can either be read out of another device of the same type or can be read before starting programming attempts.

- In any other cases, reprogramming the BCU should be sufficient.

10.4. Developing BCU applications

To develop a BCU application, first of all a *BCU configuration* (in the following *bcu.config*) and all necessary source files have to be created.

In the source directory, run

```
build.ai -cbcu bcu.config
```

This will generate the *application information* in the file *bcu.config.ai* and the program text (value of the program ID tag) will be stored in a file with the name *bcu*.

If you want to see what is going on internally, add the option *-Dtmp*. With this option, a directory named *tmp* will be created where all temporary files are kept.

Next, generate a *configuration description*. It is possible to write one from scratch according to the specification, but you can generate a template. Run

```
gencitemplate bcu.config.ai bcu.config.ci
```

This template has many possible structures created with the right IDs. In this template, all values must be adjusted (and some entries must even be removed in some cases). The syntax is described in section 9.4.

Build the image with

```
build.img -cbcu -ibcu.load bcu.config.ci
```

Again, if you want to see what happens internally, add the option *-Dtmp*. A directory named *tmp* will be created, where all temporary files are kept.

If you want to check if the image is loadable (e.g. if it is small enough), run *viewimage bcu.load*. To download it, run *loadimage local:/tmp/eib bcu.load*.

If an error is found, it is sufficient to rerun *build.ai* and then *build.img*, if the structure has not changed. Otherwise the *configuration description* needs to be updated or recreated.

10.5. Generating BCU applications

When development is finished, the extra program text file is no longer necessary. In this case, only

`build.ai bcu.config`

for building the *application information* is used. As the *configuration description* contains the program text (program ID) from the application description, the image is built with

`build.img -ibcu.load bcu.config.ci`

There are two useful utilities, *embedprogid* and *extractprogid* to extract and embed program IDs in *application informations* and *configuration descriptions*.

10.6. Example program

10.6.1. A negation which can be disabled

The following program passes changes of the group object *recv* to the group object *send* as negated values, while the *cond* group object is enabled. All group objects are of type DPT_Bool (1.002).

cond.config

```
Device {
  PEIType 0;
  BCU bcu12; // use bcu20 for a BCU 2.0
  Title "Conditional negation";

  FunctionalBlock {
    Title "Conditional negation";
    ProfileID 10000;
    Interface {
      Reference { send };
      Abbreviation send;
      DPType DPT_Bool; // same as 1.002
    };
    Interface {
      Reference { recv };
      Abbreviation recv;
      DPType DPT_Bool;
    };
    Interface {
```

```
      Reference { cond };
      Abbreviation cond;
      DPType DPT_Bool;
    };
  };

  GroupObject {
    Name send;
    Type UINT1;
    Sending true;
    Title "Output";
    StateBased true;
  };

  GroupObject {
    Name recv;
    Type UINT1;
    on_update send_update;
    Title "Input";
    StateBased true;
  };

  GroupObject {
    Name cond;
    Type UINT1;
    Receiving true;
    Title "Condition";
    StateBased true;
  };

  include { "cond.c" };
};
```

cond.c

```
void send_update()
{
  if(cond){
  send=recv+1;
  send_transmit();
  }
}
```

cond.ci

```
<?xml version="1.0"?>
<DeviceConfig >
  <ProgramID>xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx</ProgramID>
  <PhysicalAddress >1.3.1</PhysicalAddress>
  <GroupObject id="id0">
    <Priority>low</Priority>
    <SendAddress>0/0/1</SendAddress>
  </GroupObject>
  <GroupObject id="id2">
    <Priority>low</Priority>
    <ReceiveAddress>
      <GroupAddr>0/0/5</GroupAddr>
    </ReceiveAddress>
  </GroupObject>
  <GroupObject id="id4">
    <Priority>low</Priority>
    <ReceiveAddress>
      <GroupAddr>0/0/7</GroupAddr>
    </ReceiveAddress>
  </GroupObject>
</DeviceConfig>
```

Description

The program will be loaded on the device *1.3.1.* It uses *0/0/1* as group address for the output, *0/0/5* for the input and *0/0/7* for the condition.

Each *GroupObject* command creates a group object, as well a variable to receive values. The *send* object has set *Sending* to true, so that a transmit function named object_name_*transmit* is generated. If a value is to be sent, the variable is changed and the transmit function is called.

The *recv* object contains the *on_update* statement, which causes the given function to be called when a value for this group object is received.

Setting *Receiving* to true for *cond* causes a group object to receive new values in its associated variable, but does not cause any action in that case.

As a special feature, group objects (and other objects) can be declared in any number without necessarily having impact on the code size. Objects which are to be visible in the *application information* must be referenced in an *Interface* within a *FunctionalBlock*. Otherwise, they will not consume space in the binary image. Also, objects which are disabled or not referenced in the *configuration description* will not increase the image size. However, this automatic removal is not possible when some parts of the object are used in the C code which cannot be removed by GCC.

10.6.2. Cyclic switching

The following program periodically changes the state of the group object *out* and transmits its value. The initial period can be selected by the parameter *time*. At runtime, the period can be changed via a property.

timer.config

```
Device {
  BCU bcu20;
  PEIType 0;
  Title "Cyclic switching";

  FunctionalBlock{
    Title "Timer";
    ProfileID 10001;

    Interface {
      DPType 1.000;
      Reference { out };
      Abbreviation out;
    };
    Interface {
      DPType 5.000;
      Reference { time, ctime };
      Abbreviation period;
    };
  };

  GroupObject {
    Name out;
    Type UINT1;
    Title "Output";
    StateBased true;
    Sending true;
  };

  Timer {
    Name timeout;
    Type UserTimer;
    Resolution RES_1066ms;
    on_expire send;
  };
```

```
  IntParameter {
    Title "Period";
    Name time;
    MinValue 1;
    MaxValue 127;
    Default 10;
  };
  include { "timer.c"};
  on_init init;

  Object {
    Name prop;
    ObjectType 100;

    Property {
      Name ctime;
      PropertyID 100;
      Type PDT_UNSIGNED_CHAR;
      Title "Transmit period";
    };
  };
};
```

timer.c

```
void init()
{
  ctime=time;
  timeout_set(ctime);
}
void send()
{
  out=!out;
  out_transmit();
  timeout_set(ctime);
}
```

timer.ci

```
<?xml version="1.0"?>
<DeviceConfig>
  <ProgramID>YYYYYYYYYYYYYYYYYYYYY</ProgramID>
  <PhysicalAddress>1.3.2</PhysicalAddress>
  <GroupObject id="id4">
    <Priority>low</Priority>
```

```
    <SendAddress>0/0/9</SendAddress>
  </GroupObject>
  <Parameter>
    <IntParameter id="id1">
      <Value>20</Value>
    </IntParameter>
  </Parameter>
</DeviceConfig>
```

Description

The program will be loaded into the device *1.3.2*. It uses group address *0/0/9* as destination.

An integer parameter named *ctime* is defined, which accepts values between 1 and 127. Additionally, a property named *ctimer* is defined.

The *on_init* clause causes *init* to be executed. This function copies the initial value from *time* to *ctime* and starts the timer *timeout*. This timer is defined as a user timer.

When it expires, *on_expire* causes that *send* is executed. This function inverts the current state of the group object, transmits its state and starts the timer again.

Part IV.

Appendix

A. Image format

The image format is a binary format. All multi byte values are stored in big endian format. The image is divided into streams.

A valid image starts with a 10 byte header. After this header, it may only include well formed streams as described in this chapter, without any space between them. It is possible to include new stream types, if a new stream identifier number is chosen for them.

The header has the following content:

- Bytes 0–3: 0xBC 68 0C 05
- Bytes 4–7: 0xBC 68 0C 05
- Bytes 8–9: Size of the whole file

The header of each stream has the following structure:

- Bytes 0–1: Size of the stream (excluding this field)
- Bytes 2–3: Type code of this stream (see *common/loadctl.h*)
- Bytes 4–end: Any stream specific data.

A.1. Streams

In general, a stream which is specified in the following section must have the described format and may not be extended.

A.1.1. L_BCU_TYPE

This stream specifies for which BCU the image is intended. It contains the mask version in bytes 4–5.

A.1.2. L_CODE

This stream contains the memory image of the BCU program. For BCU 1.2 and 2.x, this is the content of the EEPROM starting at address 0x100.

A.1.3. L_STRING_PAR

This stream describes the location of a string parameter. Bytes 4–5 contain the address of the parameter, bytes 6–7 the length of the reserved space. Then, the ID value of the parameter of the *application information* follows as a null terminated string.

A.1.4. L_INT_PAR

This stream describes the location of an integer parameter. Bytes 4–5 contain the address of the parameter, byte 6 the type of the integer. Then, the ID value of the parameter from the *application information* follows as a null terminated string.

If the type value is negative, the type is signed, else unsigned. The allocated size is $2^{abs(type)-1}$.

A.1.5. L_FLOAT_PAR

This stream describes the location of a float parameter. Bytes 4–5 contain the address of the parameter. Then the ID value of the parameter of the *application information* follows as null terminated string.

A.1.6. L_LIST_PAR

This stream describes the location of a list parameter. Bytes 4–5 contain the address of the parameter, bytes 6–7 the count of list elements. Then, the ID value of the parameter, which is used in the *application information*, follows as a null terminated string. After that, the ID of the list elements follows (as null terminated strings) in the same order they have in the C enumeration.

A.1.7. L_GROUP_OBJECT

Byte 4 contains the number of the group object, then the ID value of the group object which is used in the *application information* follows as a null terminated string.

A.1.8. L_BCU1_SIZE

- Bytes 4–5: size of the program in the EEPROM
- Bytes 6–7: size of the available stack
- Bytes 8–9: size of the data segment
- Bytes 10–11: size of the bss segment

A.1.9. L_BCU2_SIZE

- Bytes 4–5: size of the program in the EEPROM
- Bytes 6–7: size of the available stack
- Bytes 8–9: size of the data segment in lo-RAM
- Bytes 10–11: size of the bss segment in lo-RAM
- Bytes 12–13: size of the data segment in hi-RAM
- Bytes 14–15: size of the bss segment in hi-RAM

A.1.10. L_BCU2_INIT

- Bytes 4–5: start of the address table
- Bytes 6–7: size of the address table
- Bytes 8–9: start of the association table
- Bytes 10–11: size of the association table
- Bytes 12–13: start of the readonly text segment
- Bytes 14–15: end of the readonly text segment
- Bytes 16–17: start of the parameter segment
- Bytes 18–19: end of the parameter segment
- Bytes 20–21: pointer to the EIB User Object List
- Bytes 22–23: count of the EIB User Objects
- Bytes 24–25: address of the application call back
- Bytes 26–27: pointer to the group object table
- Bytes 28–29: pointer to segment 0 (for group objects)
- Bytes 30–31: pointer to segment 1 (for group objects)
- Bytes 32–33: address of handler for PEI handler
- Bytes 34–35: address of the init function
- Bytes 36–37: address of the run function
- Bytes 38–39: address of the save function

- Bytes 40–41: start of the *.eeprom* segment
- Bytes 42–43: end of the *.eeprom* segment
- Bytes 44–45: polling address use by the polling slave
- Byte 46: polling slot used by the polling slave

A.1.11. L_BCU2_KEY

The first four bytes contain the installation key. Then the keys to be set follow (as four byte values). The first key becomes number 0, the second number 1, and so on.

A.2. Valid images

A valid image for a BCU 1.2 contains a L_BCU_TYPE, a L_CODE and a L_BCU1_SIZE stream. The L_BCU_TYPE contains the mask version 0x0012. Additionally, all memory requirements of the program must be within the limitation of the BCU. Other valid streams are ignored.

A valid image for a BCU 2.0 (or 2.1) contains a L_BCU_TYPE, a L_CODE, L_BCU2_SIZE and a L_BCU2_INIT stream. The L_BCU_TYPE contains the mask version 0x0020 (or 0x0021 for a BCU 2.1). All memory requirements of the program must be within the limitation of the BCU. Other valid streams are ignored.

If a L_BCU2_KEY stream is present, its keys are used. It must contain one installation key and three keys for the three privileged access levels. If it is missing, all keys are assumed to be 0xFFFFFFFF.

During loading, the following segments in the EEPROM are allocated (and loaded):

header The memory between 0x100 and 0x115.

addrtab Address table, must start at 0x116 and is checksum protected; needs access level 1 for write access.

assoctab Association table, is checksum protected; needs access level 1 for write access.

readonly Read only text segment, is checksum protected; needs access level 0 for write access.

eeprom *.eeprom* segment, is not checksum protected, e.g. for group objects in the EEPROM; needs access level 1 for write access.

param Parameter segment, is checksum protected, intended for run time parameters which are changed by A_Memory_Write; needs access level 2 for write access.

The order and location of these segments is determined by the L_BCU2_INIT stream. The segments may not overlap. If a checksum is used, the last byte of the segment is used for this purpose. If any space is left between segments, the values of these EEPROM locations are undefined.

B. Tables

B.1. Available DP Types

BCU SDK knows the following DPT types by name. This list is based on the proposed list of 2004-02-12.

DPT Number	Name
1.001	DPT_Switch
1.002	DPT_Bool
1.003	DPT_Enable
1.004	DPT_Ramp
1.005	DPT_Alarm
1.006	DPT_BinaryValue
1.007	DPT_Step
1.008	DPT_UpDown
1.009	DPT_OpenClose
1.010	DPT_Start
1.011	DPT_State
1.012	DPT_Invert
1.013	DPT_DimSendStyle
1.014	DPT_InputSource
1.015	DPT_Reset
1.016	DPT_Ack
1.017	DPT_Trigger
1.018	DPT_Occupancy
1.019	DPT_Window_Door
1.020	DPT_Toggle_Mode
1.021	DPT_LogicalFunction
1.022	DPT_Scene_AB
1.023	DPT_ShutterBlinds_Mode
1.100	DPT_Heat_Cool
2.001	DPT_Switch_Control
2.002	DPT_Bool_Control
2.003	DPT_Enable_Control
2.004	DPT_Ramp_Control
2.005	DPT_Alarm_Control

DPT Number	Name
2.006	DPT_BinaryValue_Control
2.007	DPT_Step_Control
2.008	DPT_Direction1_Control
2.009	DPT_Direction2_Control
2.010	DPT_Start_Control
2.011	DPT_State_Control
2.012	DPT_Invert_Control
3.007	DPT_Control_Dimming
3.008	DPT_Control_Blinds
3.009	DPT_Mode_Boiler
4.001	DPT_Char_ASCII
4.002	DPT_Char_8859_1
5.001	DPT_Scaling
5.003	DPT_Angle
5.004	DPT_Percent_U8
5.005	DPT_DecimalFactort
5.010	DPT_Value_1_Ucount2
6.001	DPT_Percent_V8
6.010	DPT_Value_1_Count
6.020	DPT_Status_Mode3
7.001	DPT_Value_2_Ucount
7.002	DPT_TimePeriodMsec
7.003	DPT_TimePeriod10MSec
7.004	DPT_TimePeriod100MSec
7.005	DPT_TimePeriodSec
7.006	DPT_TimePeriodMin
7.007	DPT_TimePeriodHrs
7.010	DPT_PropDataType
7.011	DPT_Length_mm
7.012	DPT_UElCurrentmA
8.001	DPT_Value_2_Count
8.002	DPT_DeltaTimeMsec
8.003	DPT_DeltaTime10MSec
8.004	DPT_DeltaTime100MSec
8.005	DPT_DeltaTimeSec
8.006	DPT_DeltaTimeMin
8.007	DPT_DeltaTimeHrs
8.010	DPT_Percent_V16
8.011	DPT_Rotation_Angle
9.001	DPT_Value_Temp
9.002	DPT_Value_Tempd

DPT Number	Name
9.003	DPT_Value_Tempa
9.004	DPT_Value_Lux
9.005	DPT_Value_Wsp
9.006	DPT_Value_Pres
9.007	DPT_Value_Humidity
9.008	DPT_Value_AirQuality
9.009	DPT_Value_AirFlow
9.010	DPT_Value_Time1
9.011	DPT_Value_Time2
9.020	DPT_Value_Volt
9.021	DPT_Value_Curr
9.022	DPT_PowerDensity
9.023	DPT_KelvinPerPercent
9.024	DPT_Power
10.001	DPT_TimeOfDay
11.001	DPT_Date
12.001	DPT_Value_4_Ucount
13.001	DPT_Value_4_Count
13.002	DPT_ElectricEnergy_10Wh
13.003	DPT_ElectricReactiveEnergy_10VAh
13.100	DPT_LongDeltaTimeSec
14.000	DPT_Value_Acceleration
14.001	DPT_Value_Acceleration_Angular
14.002	DPT_Value_Activation_Energy
14.003	DPT_Value_Activity
14.004	DPT_Value_Mol
14.005	DPT_Value_Amplitude
14.006	DPT_Value_AngleRad
14.007	DPT_Value_AngleDeg
14.008	DPT_Value_Angular_Momentum
14.009	DPT_Value_Angular_Velocity
14.010	DPT_Value_Area
14.011	DPT_Value_Capacitance
14.012	DPT_Value_Charge_DenstySurface
14.013	DPT_Value_Charge_DenstyVolume
14.014	DPT_Value_Compressibility
14.015	DPT_Value_Conductance
14.016	DPT_Value_Electrical_Conductivity
14.017	DPT_Value_Density
14.018	DPT_Value_Electric_Charge
14.019	DPT_Value_Electric_Current

DPT Number	Name
14.020	DPT_Value_Electric_CurrentDensity
14.021	DPT_Value_Electric_DipoleMoment
14.022	DPT_Value_Electric_Displacement
14.023	DPT_Value_Electric_FieldStrength
14.024	DPT_Value_Electric_Flux
14.025	DPT_Value_Electric_FluxDensity
14.026	DPT_Value_Electric_Polarization
14.027	DPT_Value_Electric_Potential
14.028	DPT_Value_Electric_PotentialDifference
14.029	DPT_Value_ElectromagneticMMoment
14.030	DPT_Value_Electromotive_Force
14.031	DPT_Value_Energy
14.032	DPT_Value_Force
14.033	DPT_Value_Frequency
14.034	DPT_Value_Angular_Frequency
14.035	DPT_Value_Heat_Capacity
14.036	DPT_Value_Heat_FlowRate
14.037	DPT_Value_Heat_Quantity
14.038	DPT_Value_Impedance
14.039	DPT_Value_Length
14.040	DPT_Value_Light_Quantity
14.041	DPT_Value_Luminance
14.042	DPT_Value_Luminous_Flux
14.043	DPT_Value_Luminous_Intensity
14.044	DPT_Value_Magnetic_FieldStrength
14.045	DPT_Value_Magnetic_Flux
14.046	DPT_Value_Magnetic_FluxDensity
14.047	DPT_Value_Magnetic_Moment
14.048	DPT_Value_Magnetic_Polarization
14.049	DPT_Value_Magnetization
14.050	DPT_Value_MagnetomotiveForce
14.051	DPT_Value_Mass
14.052	DPT_Value_MassFlux
14.053	DPT_Value_Momentum
14.054	DPT_Value_Phase_AngleRad
14.055	DPT_Value_Phase_AngleDeg
14.056	DPT_Value_Power
14.057	DPT_Value_Power_Factor
14.058	DPT_Value_Pressure
14.059	DPT_Value_Reactance
14.060	DPT_Value_Resistance

DPT Number	Name
14.061	DPT_Value_Resistivity
14.062	DPT_Value_SelfInductance
14.063	DPT_Value_SolidAngle
14.064	DPT_Value_Sound_Intensty
14.065	DPT_Value_Speed
14.066	DPT_Value_Stress
14.067	DPT_Value_Surface_Tension
14.068	DPT_Value_Common_Temperature
14.069	DPT_Value_Absolute_Temperature
14.070	DPT_Value_TemperatureDifference
14.071	DPT_Value_Thermal_Capacity
14.072	DPT_Value_Thermal_Conductivity
14.073	DPT_Value_ThermoelectricPower
14.074	DPT_Value_Time
14.075	DPT_Value_Torque
14.076	DPT_Value_Volume
14.077	DPT_Value_Volume_Flux
14.078	DPT_Value_Weight
14.079	DPT_Value_Work
15.000	DPT_Access_Data
16.000	DPT_String_ASCII
16.001	DPT_String_8859_1
17.001	DPT_SceneNumber
18.001	DPT_SceneControl
19.001	DPT_DateTime
20.001	DPT_SCLOMode
20.002	DPT_BuildingMode
20.003	DPT_OccMode
20.004	DPT_Priority
20.005	DPT_LightApplicationMode
20.006	DPT_ApplicationArea
20.007	DPT_AlarmClassType
20.008	DPT_PSUMode
20.011	DPT_ErrorClass_System
20.012	DPT_ErrorClass_HVAC
20.100	DPT_FuelType
20.101	DPT_BurnerType
20.102	DPT_HVACMode
20.103	DPT_DHWMode
20.104	DPT_LoadPriority
20.105	DPT_HVACContrMode

DPT Number	Name
20.106	DPT_HVACEmergMode
20.107	DPT_ChangeoverMode
20.108	DPT_ValveMode
20.109	DPT_DamperMode
20.110	DPT_HeaterMode
20.111	DPT_FanMode
20.112	DPT_MasterSlaveMode
20.600	DPT_Behaviour_Lock_Unlock
20.601	DPT_Behaviour_Bus_Power_Up_Down
21.001	DPT_StatusGen
21.002	DPT_Device_Control
21.100	DPT_ForceSign
21.101	DPT_ForceSignCool
21.102	DPT_StatusRHC
21.103	DPT_StatusSDHWC
21.104	DPT_FuelTypeSet
21.105	DPT_StatusRCC
21.106	DPT_StatusAHU
22.100	DPT_StatusDHWC
23.001	DPT_OnOff_Action
23.002	DPT_Alarm_Reaction
23.003	DPT_UpDown_Action
23.100	DPT_PresenceSensorType
23.101	DPT_HVACOperationModeType
23.102	DPT_HVAC_PB_Action
24.001	DPT_VarString_8859_1
25.001	DPT_DoubleNibble
200.001	DPT_DelayTimeMinSc
200.100	DPT_Heat_Cool_Z
200.102	DPT_WindowState
201.100	DPT_HVACMode_Z
201.101	DPT_HVACModeUser
201.102	DPT_DHWMode_Z
201.103	DPT_DHWModeUser
201.104	DPT_HVACContrMode_Z
201.105	DPT_EnablH_CStage_Z
201.106	DPT_EnableH_Energy
201.107	DPT_BuildingMode_Z
201.108	DPT_OccMode_Z
201.109	DPT_HVACEmergMode_Z
201.110	DPT_EmergMode_Z

DPT Number	Name
202.001	DPT_RelValue_Z
202.002	DPT_UCountValue8_Z
202.100	DPT_ActPosSetp
203.002	DPT_TimePeriodMsec_Z
203.003	DPT_TimePeriod10Msec_Z
203.004	DPT_TimePeriod100Msec_Z
203.005	DPT_TimePeriodSec_Z
203.006	DPT_TimePeriodMin_Z
203.007	DPT_TimePeriodHrs_Z
203.011	DPT_UFlowRateLiter_h_Z
203.012	DPT_UCountValue16_Z
203.013	DPT_UElCurrentA_Z
203.014	DPT_PowerKW_Z
203.016	DPT_TimePeriodMin_Z
203.100	DPT_HVACAirQual_Z
203.101	DPT_WindSpeed_Z
203.102	DPT_SunIntensity_Z
203.103	DPT_ElPower
203.104	DPT_HVACAirFlowAbs_Z
204.001	DPT_RelSignedValue
205.002	DPT_DeltaTimeMsec_Z
205.003	DPT_DeltaTime10Msec_Z
205.004	DPT_DeltaTime100Msec_Z
205.005	DPT_DeltaTimeSec_Z
205.006	DPT_DeltaTimeMin_Z
205.007	DPT_DeltaTimeHrs_Z
205.100	DPT_TempHVACAbs_Z
205.101	DPT_TempHVACRel_Z
205.102	DPT_HVACAirFlowRel_Z
206.100	DPT_HVACModeNext
206.101	DPT_HVACModeExceptPer
206.102	DPT_DHWModeNext
206.103	DPT_DHWModeExceptPer
206.104	DPT_OccModeNext
206.105	DPT_BuildingModeNext
206.106	DPT_TariffNext
207.100	DPT_StatusBUC
207.101	DPT_LockSign
207.102	DPT_ValueDemBOC
207.103	DPT_LockSignHFDM
207.104	DPT_ActPosDemAbs

DPT Number	Name
207.105	DPT_StatusAct
207.106	DPT_RelValUser
208.100	DPT_HVACAirQualSetpSet
208.101	DPT_ElPowerNext
208.102	DPT_TempNext
209.100	DPT_StatusHPM
209.101	DPT_TempRoomDemAbs
209.102	DPT_StatusCPM
209.103	DPT_StatusWTC
210.100	DPT_TempFlowWaterDemAbs
21.100	DPT_LM_Priority
211.100	DPT_EnergyDemWater
211.100	DPT_RoomTempSetpShift
212.100	DPT_TempRoomSetpSetShift
213.100	DPT_TempRoomSetpSet
213.101	DPT_TempDHWSetpSet
214.100	DPT_PowerFlowWaterDemHPM
214.101	DPT_PowerFlowWaterDemCPM
215.100	DPT_StatusBOC
215.101	DPT_StatusCC
216.100	DPT_SpecHeatProd
217.001	DPT_Version
218.001	DPT_VolumeLiter_Z
219.001	DPT_AlarmInfo
220.100	DPT_TempHVACAbsNext
221.001	DPT_SerNum
222.001	DPT_Current_Set_F16
222.100	DPT_TempRoomSetpSetF16
222.101	DPT_TempRoomSetpSetShiftF16
223.100	DPT_EnergyDemAir
224.100	DPT_TempSupplyAirSetpSet
225.001	DPT_ScalingSpeed
225.002	DPT_Scaling_Step_Time
226.500	DPT_PowerPriority
227.001	DPT_PowerPeriod

B.2. Available property IDs

BCU SDK knows the following property IDs by name. This list is based on the proposed list of 2004-02-12.

Property ID	Name
1	PID_OBJECT_TYPE
2	PID_OBJECT_NAME
3	PID_SEMAPHOR
4	PID_GROUP_OBJECT_REFERENCE
5	PID_LOAD_STATE_CONTROL
6	PID_RUN_STATE_CONTROL
7	PID_TABLE_REFERENCE
8	PID_SERVICE_CONTROL
9	PID_FIRMWARE_REVISION
10	PID_SERVICES_SUPPORTED
11	PID_SERIAL_NUMBER
12	PID_MANUFACTURER_ID
13	PID_PROGRAM_VERSION
14	PID_DEVICE_CONTROL
15	PID_ORDER_INFO
16	PID_PEI_TYPE
17	PID_PORT_CONFIGURATION
18	PID_POLL_GROUP_SETTINGS
19	PID_MANUFACTURER_DATA
20	PID_ENABLE
21	PID_DESCRIPTION
22	PID_FILE
23	PID_TABLE
24	PID_ENROL
25	PID_VERSION
26	PID_GROUP_OBJECT_LINK
27	PID_MCB_TABLE
28	PID_ERROR_CODE
51	PID_ROUTING_COUNT
52	PID_MAX_RETRY_COUNT
53	PID_ERROR_FLAGS
54	PID_PROGMODE
55	PID_PRODUCT_ID
56	PID_MAX_APDULENGTH
57	PID_SUBNET_ADDR
58	PID_DEVICE_ADDR

Property ID	Name
59	PID_PB_CONFIG
60	PID_ADDR_REPORT
61	PID_ADDR_CHECK
62	PID_OBJECT_VALUE
63	PID_OBJECTLINK
64	PID_APPLICATION
65	PID_PARAMETER
66	PID_OBJECTADDRESS
67	PID_PSU_TYPE
68	PID_PSU_STATUS
69	PID_PSU_ENABLE
70	PID_DOMAIN_ADDRESS
71	PID_IO_LIST
72	PID_MGT_DESCRIPTOR_01
73	PID_PL110_PARAM
74	PID_RF_REPEAT_COUNTER
75	PID_RECEIVE_BLOCK_TABLE
76	PID_RANDOM_PAUSE_TABLE
77	PID_RECEIVE_BLOCK_NR
78	PID_HARDWARE_TYPE
79	PID_RETRANSMITTER_NUMBER
80	PID_SERIAL_NR_TABLE
81	PID_BIBATMASTER_ADDRESS
101	PID_CHANNEL_01_PARAM
102	PID_CHANNEL_02_PARAM
103	PID_CHANNEL_03_PARAM
104	PID_CHANNEL_04_PARAM
105	PID_CHANNEL_05_PARAM
106	PID_CHANNEL_06_PARAM
107	PID_CHANNEL_07_PARAM
108	PID_CHANNEL_08_PARAM
109	PID_CHANNEL_09_PARAM
110	PID_CHANNEL_10_PARAM
111	PID_CHANNEL_11_PARAM
112	PID_CHANNEL_12_PARAM
113	PID_CHANNEL_13_PARAM
114	PID_CHANNEL_14_PARAM
115	PID_CHANNEL_15_PARAM
116	PID_CHANNEL_16_PARAM
117	PID_CHANNEL_17_PARAM
118	PID_CHANNEL_18_PARAM

Property ID	Name
119	PID_CHANNEL_19_PARAM
120	PID_CHANNEL_20_PARAM
121	PID_CHANNEL_21_PARAM
122	PID_CHANNEL_22_PARAM
123	PID_CHANNEL_23_PARAM
124	PID_CHANNEL_24_PARAM
125	PID_CHANNEL_25_PARAM
126	PID_CHANNEL_26_PARAM
127	PID_CHANNEL_27_PARAM
128	PID_CHANNEL_28_PARAM
129	PID_CHANNEL_29_PARAM
130	PID_CHANNEL_30_PARAM
131	PID_CHANNEL_31_PARAM
132	PID_CHANNEL_32_PARAM
51	PID_EXT_FRAMEFORMAT
52	PID_ADDRTAB1
53	PID_GROUP_RESPONSER_TABLE
51	PID_PARAM_REFERENCE
51	PID_MEDIUM_TYPE
52	PID_COMM_MODE
53	PID_MEDIUM_AVAILABILITY
54	PID_ADD_INFO_TYPES
55	PID_TIME_BASE
56	PID_TRANSP_ENABLE
51	PID_GRPOBJTABLE
52	PID_EXT_GRPOBJREFERENCE
51	PID_POLLING_STATE
52	PID_POLLING_SLAVE_ADDR
53	PID_POLL_CYCLE
51	PID_AR_TYPE_REPORT
51	PID_PROJECT_INSTALLATION_ID
52	PID_KNX_INDIVIDUAL_ADDRESS
53	PID_ADDITIONAL_INDIVIDUAL_ADDRESSES
54	PID_CURRENT_IP_ASSIGNMENT_METHOD
55	PID_IP_ASSIGNMENT_METHOD
56	PID_IP_CAPABILITIES
57	PID_CURRENT_IP_ADDRESS
58	PID_CURRENT_SUBNET_MASK
59	PID_CURRENT_DEFAULT_GATEWAY
60	PID_IP_ADDRESS
61	PID_SUBNET_MASK

Property ID	Name
62	PID_DEFAULT_GATEWAY
63	PID_DHCP_BOOTP_SERVER
64	PID_MAC_ADDRESS
65	PID_SYSTEM_SETUP_MULTICAST_ADDRESS
66	PID_ROUTING_MULTICAST_ADDRESS
67	PID_TTL
68	PID_EIBNETIP_DEVICE_CAPABILITIES
69	PID_EIBNETIP_DEVICE_STATE
70	PID_EIBNETIP_ROUTING_CAPABILITIES
71	PID_PRIORITY_FIFO_ENABLED
72	PID_QUEUE_OVERFLOW_TO_IP
73	PID_QUEUE_OVERFLOW_TO_KNX
74	PID_MSG_TRANSMIT_TO_IP
75	PID_MSG_TRANSMIT_TO_KNX
76	PID_FRIENDLY_NAME

Bibliography

The entire GNU tool chain documentation is distributed with the respective program sources. Different versions are available at http://ftp.gnu.org/gnu/. However, the best information source is the version distributed with the sources you are using. Online references are valid as of 2005-05-05.

[ANYC] AnyC – GPL C compiler for 8-bit microcontrollers. http://anyc.sf.net/

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman, Compilers – Principles, Techniques and Tools. Addison-Wesley, 1986

[BASYS] BASys 2003 home. http://www.basys2003.org

[BCU1] Siemens AG, BCU1 Helpfile. 1996

[BCU2] Siemens AG, BCU2 Helpfile. Version 1.1, 2004

[BFD] Steve Chamberlain and others, libbfd – The Binary File Descriptor Library. Available as part of the binutils sources

[BFDINT] Ian Lance Taylor and others, BFD Internals. Available as part of the binutils sources

[BIN01] Unsupported targets slated for removal. http://sourceware.org/ml/binutils/2005-03/msg00618.html

[BSD] A sample of a BSD style licence. http://www.debian.org/misc/bsd.license

[CEXT] ISO/IEC TR 18037:2004, Programming languages – C – Extensions to support embedded processors.[1]

[DFSG] The Debian Free Software Guidelines (DFSG). http://www.debian.org/social_contract#guidelines

[EIBIDE] Free EibIDE for Linux. http://sourceforge.net/projects/freeeibide

[1] The last draft version is available at http://www.open-std.org/jtc1/sc22/wg14.

[GAS] Dean Elsner, Jay Fenlason and others, Using as – The GNU assembler. Available as part of the binutils sources

[GASINT] Free Software Foundation, Assembler Internals. Available as part of the binutils sources

[GCC] Free Software Foundation, Using the GNU Compiler Collection (GCC). Available as part of the GCC sources

[GCCINT] Free Software Foundation, GNU Compiler Collection (GCC). Available as part of the GCC sources

[GDB] Richard Stallman, Roland Pesch and others, Debugging with GDB. Available as part of the GDB sources

[GDBINT] John Gilmore and others, Using GDB – A guide to the internals of the GNU debugger. Available as part of the GDB sources

[GNU11] GNU Development Chain for 68HC11 & 68HC12. `http://www.gnu.org/software/m68hc11/`

[GPL] GNU General Public License. `http://www.gnu.org/copyleft/gpl.html`

[KNX] Konnex Association, KNX Specification. Version 1.1, 2004

[LD] Steve Chamberlain, Ian Lance Taylor and others, Using ld – The GNU linker. Available as part of the binutils sources

[LDINT] Per Bothner, Steve Chamberlain and others, A guide to the internals of the GNU linker. Available as part of the binutils sources

[M68HC05] Motorola Inc., M68HC05 Family, Understanding Small Microcontrollers. Revision 2, undated

[NEW1] Rob Savoye and others, Porting The GNU Tools To Embedded Systems. Available as part of the newlib sources

[NEW2] Steve Chamberlain and others, The Red Hat newlib C Library. Available as part of the newlib sources

[NEW3] Steve Chamberlain and others, The Red Hat newlib Math Library. Available as part of the newlib sources

[PTH] GNU PTH – The GNU Portable Threads. `http://www.gnu.org/software/pth/`

[PTHSEM] Semaphore support for GNU PTH. `http://www.auto.tuwien.ac.at/~mkoegler/index.php/pth`

[SDCC] SDCC – Small Device C Compiler. `http://sdcc.sf.net/`

[XML2] XML Schema Part 2: Datatypes Second Edition. W3C Recommendation, 28 October 2004, `http://www.w3.org/TR/xmlschema-2/`