AN OPEN APPROACH TO EIB/KNX SOFTWARE DEVELOPMENT

Wolfgang Kastner, Georg Neugschwandtner, Martin Kögler

TU Wien, Institute of Automation, Automation Systems Group,
Treitlstraße 1, A-1040 Vienna, Austria,
{k,gn,mkoegler}@auto.tuwien.ac.at

Abstract: EIB/KNX is a field bus used in home and building automation. When building application programs for EIB/KNX nodes, one was hitherto faced with low-level constructs. To improve this situation, a RAD (Rapid Application Development) like approach was adopted. This model encapsulates the system software entities in a way which is inspired by the object-oriented paradigm. It also makes use of functional blocks to describe the application behaviour. To allow the roles of software developer and project engineer to be separated, the tool chain is designed for interfacing with an integration tool. The article discusses the work flow when building an EIB/KNX system and the resulting requirements on the tool chain. The GCC-based solution which was developed is presented. Specific challenges in porting the GNU tool chain to the standard microcontroller for EIB/KNX nodes are sketched. The implementation also includes an open PC-based EIB/KNX network access and management server.

Keywords: Embedded Systems, Software Engineering, Fieldbus, Configuration, Programming Approaches

1. INTRODUCTION

The structure of EIB/KNX systems requires that node applications can be customized without low-level programming skills. A common tool software is available for this purpose. Still, the first-time creation of these applications remains a highly technically involved process. The goal of the present project is to apply an RAD approach to this step while respecting these particular work flow requirements.

1.1 EIB/KNX

The European Installation Bus *EIB* (Kastner *et al.*, 2005) is a home and building automation bus system. It is optimized for low-speed control applications like lighting and shading. A European standard since 1998, EIB now forms a subset of the KNX specification (Konnex Association, 2004).

EIB/KNX is specified over various physical media, including powerline, with twisted pair (TP) being the mainstay. Tunnelling EIB/KNX frames over IP networks is known under the term EIBnet/IP. EIBnet/IP provides a point-to-point mode, e. g. for remote configuration, as well as a multicast mode for coupling multiple EIB/KNX subnets via IP. These modes are – in a slightly confusing fashion – labelled Tunnelling and Routing, respectively.

TP and powerline EIB/KNX devices are frequently split into two parts. Generic modules providing bus connectivity (so-called Bus Coupling Units, BCUs) are combined with application specific hardware modules. The standard 10-pin connector between BCUs and these application modules is called the Physical External Interface (PEI). It can be operated in a variety of configurations covering parallel and serial digital I/O and analog input.

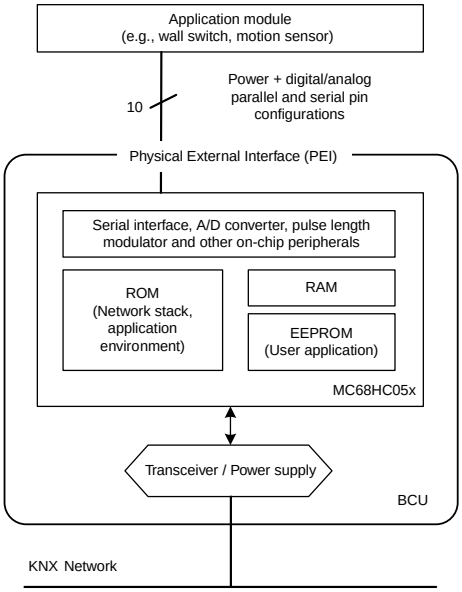

Fig. 1. EIB/KNX Bus Coupling Unit

BCUs are based on Freescale M68HC05 family microcontrollers, which are complemented with a transceiver and link power supply (cf. Fig. 1). They contain an implementation of the entire EIB/KNX network stack and application environment in their ROM. The application software customizing the BCU for a particular application module is stored in the EEPROM.

Currently, two major BCU variants referred to as BCU 1 and BCU 2 exist for the TP medium, plus another one for powerline. Their key differences lie in the system software implementation and the amount of available memory. While the BCU 1 is a standard M68HC05B6, the BCU 2 is a special variant with EIB/KNX-specific on-chip-peripherals.

For more complex user applications which use a separate microcontroller, BCUs offer high-level access to the network stack via the PEI, using it as an asynchronous serial interface. In such cases, however, the TP-UART IC often is a more attractive alternative. It only handles the physical and most of the data link layer, instead of the entire EIB/KNX protocol stack as BCUs do.

For Microsoft Windows based systems, a common high-level API ("Falcon") for accessing functionality throughout the network stack by way of various interfaces is available. For the Linux operating system, which is of ever-increasing importance for embedded platforms, kernel level drivers for BCU access as well as TP-UART based serial interfaces have been implemented (Kastner and Troger, 2003).

1.2 EIB/KNX software development and deployment

Obviously, writing all BCU application programs from scratch is typically not a feasible approach towards building an EIB/KNX system. Therefore, manufacturers provide ready-made applications matching their hardware. The behaviour of these applications can be customized by the project engineer by modifying manufacturer defined parameters. The final configuration is then downloaded to the BCU. The work flow of creating an EIB/KNX system is thus divided into three steps:

(1) *Development:* A *software developer* writes a BCU application program for a particular hardware configuration. He documents its behaviour and defines the parameters available to influence it. The application is brought into a format suitable for distribution to EIB/KNX project engineers. This format also includes the necessary meta information to allow a software tool to display the application parameters. Moreover, it provides this tool with the necessary knowledge on how to apply these changes to the program code.
(2) *Project planning:* A *project engineer* selects appropriate EIB/KNX devices to fulfil the requirements of a particular project. Using a (typically PC-based) integration tool, he makes the necessary adjustments to the application parameters of the chosen devices. He also sets up their communication relationships. While the software developer defines the behaviour (or set of possible behaviours) of one single node, the project engineer thus defines the behaviour of the entire system. This step is entirely off-line, i.e., no target devices are required yet.
(3) *Installation and download:* The BCUs are combined with the appropriate application modules (if not already delivered in a common housing by the manufacturer) and installed to their final location. This step is often carried out by a *site technician*. Before or after installation, the configuration is downloaded to the BCUs. This can be done via the network. Targets are identified via their *Individual address* (a configurable identifier which is unique within the installation) or via a special button on the BCU if they have not yet been assigned such an address.

The second and third step shall together be referred as *system integration*, the software tool which assists the work of the project engineer (and possibly site technician) as *integration tool*. For EIB/KNX systems, only one single integration tool software is necessary. This tool, called *ETS* (EIBA s.c.r.l., n.d.), handles every certified EIB/KNX device, no matter from which manufacturer. This solution significantly lowers the effort involved in setting up a multi-vendor system.

1.3 Project goals

In an effort to ensure that a project built with ETS will always work as desired, ETS will not accept any

device application which has not passed compliance certification. This creates an issue for projects which want to explore new approaches to EIB/KNX node software development and deployment.

The goal of the project presented was to remedy this situation by creating an open-source framework for developing BCU applications. The framework should support the use of high-level languages, in particular C. It should make obtaining the desired behaviour as easy as possible, allowing to focus on the desired functionality rather than how to achieve it. Applications which are easier to understand are also less error prone. Still, the programming model should not be overly restricted in its expressiveness regarding access to the features of the BCU system software. The framework should support both BCU 1 (TP version) and BCU 2.

The separation of development and integration as outlined above was to be supported. For this purpose, a suitable, open interface between the development framework and an integration tool had to be defined. The interface should be generic enough to accommodate hardware platforms other than BCUs. The format should also describe the behaviour of the application, which we believe to be necessary for any advanced approach to configuration. Still, the RAD approach in the context of the BCU SDK was to focus on the development step. Although a suitable interface was desired to support automating the system configuration (project planning) process, the actual automation of this step itself was out of scope.

2. BCU SDK OVERVIEW

Fig. 2 illustrates the approach the BCU SDK (software development kit) developed during the present project takes towards implementing the work flow discussed in Section 1.2. The software developer uses the *preparation build script* to prepare the source code of the new application for distribution to project engineers. The build script controls the execution of the necessary tools. First, the input files are run through a custom preprocessor which implements the RAD programming model. An example program illustrating this model is presented in Section 3. The code is checked for errors and transformed into a format suitable for distribution, which will in the following be referred to as *program text*. Also, a description of the customizable aspects of the application (and its behaviour) is generated (*application meta data*).

The application meta data are collected in some form of *product data base*. From this collection, the project engineer retrieves the descriptions of the devices he has decided to use in a new project. The exact design of this database, which will also hold the matching program texts, and its interfaces are not relevant for the purposes of the BCU SDK. The project engineer may base his choice on external documentation (as it is the case with ETS now) or the integration tool may assist him using the information provided in the application meta data. Either way, he defines the customization options and binding data (communication relationships) for every device. The integration tool generates the appropriate interchange data format for these *configuration meta data*.

It does not, however, touch the contents of the program text. The integration tool is merely responsible for passing it to the finalization build script together with the associated configuration meta data. The finalization build script will then take all necessary actions to make the necessary changes to the program text. How this is done precisely is hidden from the integration tool, which therefore has no need to interpret the program text.

To match up program texts and their associated application meta data, the meta data description contains a reference attribute which uniquely identifies the associated program text. Since it is not limited in length, the current SDK implementation exploits the fact that the most unique identifier is the program text itself. It thus simply stores the (appropriately encoded) program text as the attribute value.

The *finalization build script* generates a final *binary image*, which is ready for download to the BCU. This image can be stored in a *project data base* together with the associated configuration meta data. Finally, the *loader* downloads this image to the device with the matching individual address. Again, the integration tool controls the loader (and the address assignment procedure), but is not required to have any knowledge of the format of the binary image. For these online steps, a *bus daemon* was implemented to provide the necessary EIB/KNX network access. It is presented in detail in Section 4.

The design of the project data base is again not relevant to the BCU SDK. It would equally be possible to have it store the program text and generate the final image on the fly before download. This would, for example, allow to change parameters later without the need to keep the product data base. However, since creating the final image is an offline step, it was included in the middle column of Fig. 2.

Although the data formats for application and configuration meta data have been fully defined, no matching integration tool is available yet. Therefore, the BCU SDK contains a stub implementation. It creates a configuration skeleton which has to be completed manually.

Actually, a developer will always need a special integration tool, since he should not have to manually reapply configuration information for every debug iteration of the program text. This is addressed in the current version by allowing to specify the program text to the finalization build script separately from the configuration meta data.

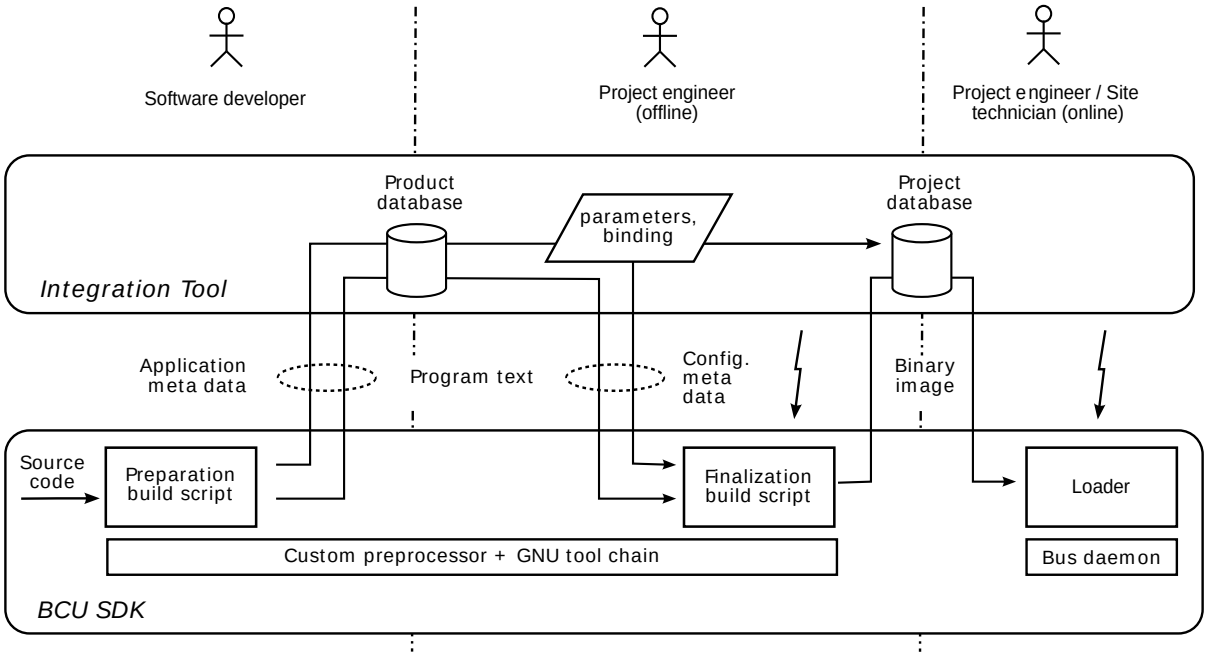

Fig. 2. Work flow and data flow using the BCU SDK

In the BCU SDK, the program text is not a binary image yet. Instead, it contains the preprocessed source code and mapping information between its identifiers and the ones used in the application meta data. The final compiler run is made only when all configuration settings are known to reduce image size.

Still, the preparation build script already builds a binary image for test purposes. This is done to check the source code for syntax errors. It also gives an upper limit on the code size, as no optimizations which could be made in response to certain parameter settings are applied.

This is in contrast to ETS, which modifies parameters and binding information in the executable binary. This approach supports only relatively minor modifications in response to a parameter change (e.g., changing a single byte). Nevertheless, such an approach would be possible using the BCU SDK as well. Experimental patching programs for the test image generated by the preparation build script were implemented.

Code generation relies on the GNU tool chain, which was ported to the Freescale/Motorola M68HC05 family microcontroller architecture. Key features of this port are presented in Section 5.

2.1 Meta data interchange formats

The application and configuration meta data formats are XML based. XML Schema definitions are provided. Fig. 3 shows an example application description (only selected nodes are shown). At the top level, it contains the program text identifier (which, as discussed, currently stores the program text itself). A global description block describes aspects such as the type of BCU the application is designed for. This is necessary since BCU 1 and BCU 2 support different system entities. Parameters can be numeric values (both integers and floating point values are supported), strings, or an enumeration of discrete choices. They carry descriptive text (including the engineering unit for numeric values) as well as tags constraining the range of valid choices.

The communication endpoints of EIB/KNX nodes relevant for process data exchange are referred to as *group objects*. A group object can be regarded as an application related variable exposing a particular aspect of the functionality of an EIB/KNX node. Group objects can be data sources, which provide information to other devices, or data sinks, which carry out certain actions according to the information received. XML nodes describing a group object include information about the supported data flow direction.

The project engineer forms network-wide shared variables by assigning group objects of multiple nodes to a common logical group. The values of all group objects belonging to such a group will be held consistent with the help of the BCU system software. As an example, to have a wall switch control a relay, the project engineer assigns the the group object representing the switch position to the same group as the one the relay node exposes as its control input. These groups are the *data points* (cf. (Kastner *et al.*, 2005)) of an EIB/KNX installation.

Obviously, group objects forming a group need to use a common encoding to ensure that the shared value will be interpreted in a consistent way. The KNX specification refers to these encodings as *data point types* (DPT). For example, DPT 1.001 stands for a Boolean value with the state labels “On” and

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DeviceDesc SYSTEM [...]>
<DeviceDesc xmlns:xsi="http://[...]>
  <ProgramID>213C6172[...]</ProgramID>
  <Description>
    <MaskVersion>0012</MaskVersion>
    <Title>Simple Timer Switch</Title>
  </Description>
  <FunctionalBlock id="id7">
    <ProfileID>417</ProfileID>
    <Title>Light Switching
       Actuator Basic</Title>
    <Interface id="id5">
      <DPType>1.001</DPType>
      <Abbreviation>SOO</Abbreviation>
      <Reference idref="id2"/>
    </Interface>
    [...]
    <Interface id="id4">
      <DPType>7.005</DPType>
      <Abbreviation>TOD</Abbreviation>
      <Reference idref="id1"/>
    </Interface>
  </FunctionalBlock>
  <GroupObject id="id2">
    <Title>Switch Input</Title>
    <Receiving>true</Receiving>
    [...]
  </GroupObject>
  [...]
  <Parameter>
    <IntParameter id="id1">
      <Title>Time-on duration</Title>
      <Unit>seconds</Unit>
      <Default>180</Default>
      [...]
    </IntParameter>
  </Parameter>
</DeviceDesc>
```

Fig. 3. Application meta data format example

"Off". Yet, the expressiveness of DPTs with regard to describing processing rules (i.e., how the change of one data point affects others) is limited. For this reason, the functionality of the application is described by declaring one or more functional blocks. The KNX specification already defines several functional blocks, but they are not yet being applied to EIB/KNX. Every XML node describing a functional block contains an external reference to the precise definition of its behaviour. In the example, the Interface Object type of a functional block from the KNX specification was chosen as reference. However, this reference could as well be a URL pointing to a custom description.

Display rules can be defined for all entities depending on selected parameter values to allow revealing the complexity of a particular application to the project engineer step by step. The configuration meta data format looks similar. For every configurable entity from the application information, it contains the value selected by the project engineer. The relationship is established via the *id* attributes. This allows the easy translation of the descriptive texts contained in the application information. An example configuration description is shown in Fig. 4.

```
<?xml version="1.0"?>
<DeviceConfig xmlns:ns1="http://[...]>
  <ProgramID>213C6172[...]</ProgramID>
  <IndividualAddr>1.3.3</IndividualAddr>
  <GroupObject id="id3">
    <Priority>low</Priority>
    <SendAddress>1/1/2</SendAddress>
    <ReadAddress>
      <GroupAddr>1/1/2</GroupAddr>
    </ReadAddress>
  </GroupObject>
  <GroupObject id="id2">
    <Priority>low</Priority>
    <ReceiveAddress>
      <GroupAddr>1/1/1</GroupAddr>
    </ReceiveAddress>
  </GroupObject>
  <Parameter>
    <IntParameter id="id1">
      <Value>120</Value>
    </IntParameter>
  </Parameter>
</DeviceConfig>
```

Fig. 4. Configuration meta data format example

In EIB/KNX, group membership is managed using what is described in (Thomesse, 1999) as the *producer-consumer* mechanism. Thus, every node is provided with the information which logical groups each of its group objects belongs to. Groups are identified using *group addresses*. In the configuration information, this assignment is made separately for the different service types of transmitting and receiving a shared value update as well as requesting and receiving its current value. Group objects can also belong to multiple groups. In addition, a transmission priority is assigned to every data source group object.

The meta data interchange formats are designed to be open for EIB/KNX node architectures other than BCU 1 and BCU 2 as they are aligned with the wire protocol. They also support the EIB/KNX client-server communication scheme for node management (referred to as *interface objects* and *properties*) and TP1 polling groups.

3. PROGRAMMING MODEL

For interacting with the BCU system software, the BCU SDK offers an increased level of abstraction. Instead of accessing configuration data and library functions at specific memory addresses, the developer uses a simple specification language to define which BCU system entities he would like to use and how he would like to refer to them in his C code. The specification file is plain text. Its syntax is inspired by modern RAD environments, where objects are instantiated from a range of available classes and customized by changing properties and assigning event handlers.

An example is shown in Fig. 5. Here, a standard functional block from the KNX specification is implemented. The functional block describes a light switching actuator. Basically, it just closes and opens a relay

timer.config

```
Device {
  BCU bcu12;
  PEIType 4;
  Title "Simple timer switch";
  include { "timer.c" };

  FunctionalBlock {
    Title "Light Switching
           Actuator Basic";
    ProfileID 417;

    Interface {
      Reference { IN };
      Abbreviation SOO;
        // Switch On Off
      DPType 1.001;  // DPT_Switch
    };

    Interface {
      Reference { OUT };
      Abbreviation IOO;
        // Info On Off
      DPType DPT_Switch; // 1.001
    };

    Interface {
      Reference { DELAY };
      Abbreviation TOD;
        // Timed On Duration
      DPType TimePeriodSec; // 7.005
    };
  };

  GroupObject {
    Title "Switch input";
    Name IN;  Type UINT1;
    StateBased false;
    on_update input_changed;
  };

  GroupObject {
    Title "Status output";
    Name OUT;  Type UINT1;
    Sending true;
    StateBased true;
  };

  Timer {
    Name TIMER1;
    Type EnhUserTimer;
    Resolution RES_4266ms;
    on_expire switch_off;
  };

  IntParameter {
    Title "Time-on duration";
    Name DELAY;
    Unit "seconds";
    Default 180;  Precision 4;
    MinValue 4;  MaxValue 544;
  };
};
```

timer.c

```
#define PEI3_ON 0x11
#define PEI3_OFF 0x10

void switch_on() {
  _U_ioAST(PEI3_ON);
  OUT = 1;
  OUT_transmit();
}

void switch_off() {
  _U_ioAST(PEI3_OFF);
  OUT = 0;
  OUT_transmit();
}

void input_changed() {
  if (IN == 1)
  {
    TIMER1_set
      (DELAY*10/43);
    switch_on();
  }
  else
  {
    TIMER1_cancel();
    switch_off();
  }
}
```

Fig. 5. Source code example

by switching a PEI output on and off in response to the change of state of a group object. Two optional functions are added. First, the actuator shall autonomously switch off after a certain duration (e.g., for use as stairway or hallway lighting). Second, a status output is added which reflects the current state of the relay.

The desired behaviour is as follows. The relay shall be closed when an "On" message is received on the `Switch input` group object. The length of this message is one bit. This length is assigned the type name UINT1 (Siemens AG, 1996; Siemens AG, 2004). It shall open again when an "Off" message is received on the same group object or after a specified timeout period. Whenever the state of the relay changes, its current state shall be transmitted via a second group object (`Status output`, again of type UINT1). The status output shall be readable from the network. Finally, the time-on duration shall be customizable via the integration tool.

The definition of the group objects in the RAD specification (*timer.conf*) is straightforward. The `Title` is what will appear in the respective meta data description XML tag. The group objects are assigned the internal names of `OUT` and `IN`. `OUT` has the `Sending` attribute set. This makes the function `OUT_transmit` available in the C code, which is used to transmit new values via the `SendAddress` associated with this group object. `OUT` is also `State Based`, which tells the BCU to answer read requests. Since `IN` contains no useful data for reading, its `StateBased` attribute is set to false. When a new value is received for `IN`, the `input_changed()` function will be called.

The time-on duration is specified as an integer parameter together with its useful range and unit. To autonomously switch off after a certain time interval, a timer is used. Specifying a `Timer` block sets up the necessary framework. It is of type `EnhUserTimer` (which provides additional functionality over the standard BCU user timers). One timer tick corresponds to 4.266 s. This resolution is reflected in the precision attribute of the `DELAY` parameter. When the timer expires, the handler `switch_off()` will be called.

The corresponding C code is located in *timer.c*. The group object values are available as global variables which will always contain the last value received for the associated group address. The UINT1 type is mapped to an unsigned 8 bit integer. Note the automatically generated functions for transmitting the current value of `OUT`, starting the timer with a specified timeout period, and canceling a runnning timer. The timer interval is scaled to its resolution (it must not exceed 127). The relay output is controlled using the BCU API function for accessing the PEI `_U_ioAST()`. C wrappers like this one are provided for all BCU API calls.

Back in *timer.config*, the target BCU is defined using its system software version (mask version) 1.2, which corresponds to a BCU 1. The desired PEI configuration type for this application is 4 (parallel I/O with 2 inputs and 3 outputs). The program contains a single functional block. The reference code (as discussed in the previous section) is taken out of the KNX specification, as are the interfaces of the functional block (i.e., their `Abbreviation` code and *DPT*). References are made to the appropriate system entities. The preparation build script will automatically transform this information into the application meta data format as shown in Fig. 3.

Other relevant BCU system functionality such as initialization and power-failure handlers and interface objects and their properties can be used in the same way. As far as the RAD specification is concerned, all differences between the BCU 1 and BCU 2 low-level APIs are hidden by the SDK.

In the C code, signed and unsigned integer types are available in any byte width from 1 to 8 to optimize RAM usage. As a special feature, transparent EEPROM access is conveniently available by declaring a variable with the appropriate storage class and attribute, e.g., `int x EEPROM_SECTION EEPROM_ATTRIB;`. Support for interface object parameters located in EEPROM is provided as well.

4. NETWORK ACCESS AND MANAGEMENT

For access to the EIB/KNX TP1 network, the BCU SDK includes a Unix daemon (called *eibd*, Fig. 6). It allows multiple clients to connect simultaneously via IP or Unix domain sockets. Clients can invoke various services of the EIB/KNX protocol stack. This includes sending and receiving unicast, multicast and broadcast telegrams. Also, eibd handles the protocol state machine for the client endpoint of a reliable connection. Based upon this, eibd can also autonomously execute various device and network management procedures, such as assigning individual addresses. Also, a bus monitor can be opened, which optionally can decode EIB frames. A "Raw" interface is available for clients that wish to implement the server endpoint for unicast communication, including reliable connections. This allows the host eibd resides on to be managed remotely via EIB/KNX management procedures.

The method of access to the EIB/KNX network is entirely hidden by the backends. The BCU 1 supports a protocol with RTS/CTS handshake ("PEI 16") which is best handled via a kernel driver. The FT1.2 based protocol of the BCU 2 ("PEI 10") can be implemented using the plain serial driver. Moreover, the respective backends have to adapt the message formats (EMI). For lab use, the TP-UART serial protocol is supported as well. EIBnet/IP Tunnelling and Routing can also be used for network access. Implementation of USB interfacing is currently being added.

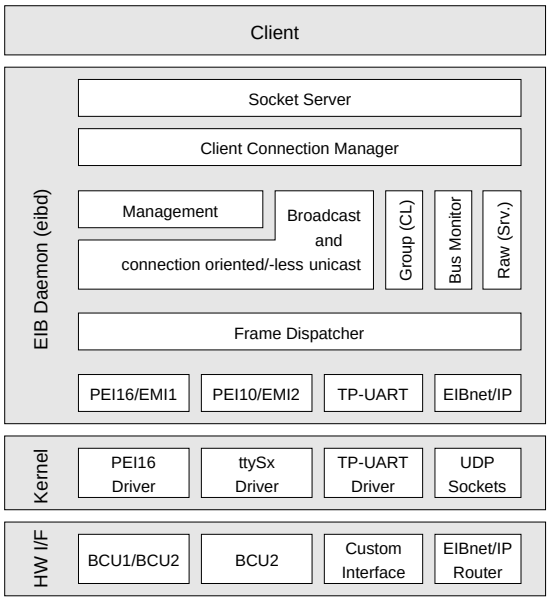

Fig. 6. Network access/management daemon

Higher-level tasks within eibd register with the frame dispatcher and state which frames (based upon addressing mode and destination address) they are prepared to process. To be able to serve multiple clients simultaneously, backends should deliver as many incoming frames as possible and leave filtering to the frame dispatcher.

In principle, this allows one client to maintain a point-to-point connection – where only frames from one single source are relevant – and another to operate in bus monitor mode. Yet, since bus monitor operation entails switching the hardware into a read-only mode, a special "best-effort" monitor mode was introduced which builds upon the fact that telegram filtering is not performed in the access hardware, but in the frame dispatcher. It forwards all frames the backend will provide in normal operation mode. For the BCU SDK, it will also act as a loader for writing applications into the BCU.

5. IMPLEMENTATION

To avoid duplicating work already done in various other places, it was decided to port an existing C compiler to the M68HC05 architecture. GCC (GNU Compiler Collection) was selected for its proven front end and optimizer. GCC is in wide-spread use as it is the standard compiler on most free operating systems. Its core parts are maintained by a large community. The GNU tool chain (Free Software Foundation, n.d.*b*) is supplemented by the GNU binutils (assembler and linker).

Several creative measures had to be taken to make GCC cope with the resource limitations of BCUs ($\ll$ 1 kb EEPROM, $\ll$ 100 bytes RAM). This constituted a key point of the porting activities. Full details are available in (Kögler, 2005). SDCC (SDCC

Project, n.d.) was considered as an alternative candidate. It supports different architectures and implements some optimizations. Compared to GCC, SDCC is much simpler and has less features. If SDCC had been the first choice, the structure would have been much simpler. Yet, some parts, like the automatic removal of unused static variables, would have had to be implemented from scratch.

The port can be used for any member of the M68HC05 microcontroller family. For automated regression tests, a CPU core simulator for the M68HC05 architecture was developed. It is based on its counterpart in the M68HC11 port of the GNU tool chain (Free Software Foundation, n.d.*a*). It is also accessible via a limited interactive GNU debugger frontend.

Since GCC is designed to work with a considerably different hardware architecture, certain missing features need to be emulated. This includes providing a larger number of general purpose registers (using RAM), multiplication and division operations, a data stack, and pointers which can cover the entire address space. The GCC floating point simulator for single and double precision values is available. However, many of its functions need too much memory, which causes them to fail on a BCU. Some functions even need a larger stack than the M68HC05 GCC port supports. The tight memory constraints also affect the expressiveness of the standard regression test suite. Approximately five per cent of the test cases yield unexpected outcomes, but we believe these cases to be unsuitable for the target. Due to the huge overall number of test cases, a detailed assessment was not possible yet.

The port had to make special provisions to allow GCC to automatically distribute variables over the non contiguous RAM segments of a BCU 2. Additionally, transparent access to the EEPROM was added. Special effort was directed towards obtaining small sized output. Still, the compiler output will in most cases be larger than well optimized, hand written assembler code. For the exceptionally resource-constrained BCU 1 environment, this considerably limits the amount of functionality which can be realized.

The interplay of an assembler-level operating system API and an optimizing compiler can bring about interesting effects which prove to be a challenge when implementing the compiler. For example, the BCU API provides optimized functions for multiplication, division, shift and bit operations. Yet, their use may in certain cases prevent M68HC05-GCC from applying optimizations and thus actually result in larger code.

6. CONCLUSION

A set of software development tools for EIB/KNX nodes was presented. It adopts a RAD like programming model, which we believe to be a novel approach for field devices. The separation of development and deployment is supported via an integration tool interface. Its implementation involved porting GCC to the MC68HC05 microcontroller family and the development of an API for EIB/KNX access from PC-based programs. All relevant parts of the SDK are fully implemented, albeit partially still in the testing phase. The entire SDK is placed under the GPL (GNU General Public License) and can be downloaded together with further documentation from `http://www.auto.tuwien.ac.at/projects/hba/`.

Numerous improvements are still possible. Specifically, target specific optimizations within GCC provide ample possibilities for further work. Also, the handling of the build process should be improved. This includes adding a graphical interface (e.g., using Eclipse or KDevelop) for the developer. For system integration, we intend to enhance the open BAsys tool (BASys Project, n.d.) to enable interaction with the BCU SDK.

REFERENCES

BASys Project (n.d.). BASys 2003. `http://www.basys2003.org`.

EIBA s.c.r.l. (n.d.). The ETS 3 Software Family. `http://www.eiba.com/ets`.

Free EibIDE Project (n.d.). Free EibIDE for Linux. `http://sourceforge.net/projects/freeeibide/`.

Free Software Foundation (n.d.*a*). GNU development chain for 68HC11 & 68HC12. `http://www.gnu.org/software/m68hc11/`.

Free Software Foundation (n.d.*b*). GNU tool chain documentation. Distributed with the respective program sources, also available from `http://ftp.gnu.org/gnu/`.

Kastner, Wolfgang and Christian Troger (2003). Interfacing with the EIB/KNX: A RTLinux device driver for the TPUART. In: *5th IFAC Intl. Conf. on Fieldbus Syst. and Appl. (FeT'2003)*.

Kastner, Wolfgang, Georg Neugschwandtner, Stefan Soucek and H. Michael Newman (2005). Communication systems for building automation and control. *Proceedings of the IEEE* **93**(6), 1178–1203.

Kögler, Martin (2005). Free development environment for Bus Coupling Units of the European Installation Bus. Master's thesis. TU Wien.

Konnex Association (2004). *KNX Specifications, Ver. 1.1*.

SDCC Project (n.d.). SDCC – Small device C compiler. `http://sdcc.sourceforge.net/`.

Siemens AG (1996). *BCU 1 Online Help*.

Siemens AG (2004). *BCU 2 Online Help, Version 1.1*.

Thomesse, Jean Pierre (1999). Fieldbuses and interoperability. *Control Engineering Practice* **7**(1), 81–94.