Programming fieldbus nodes: A RAD approach to customizable applications

G. Neugschwandtner, W. Kastner and M. Kögler
Automation Systems Group, Institute of Automation, TU Wien
{gn,k,mkoegler}@auto.tuwien.ac.at

Abstract

The European Installation Bus (EIB), part of the KNX standard, is a field bus for home and building automation. Bus Coupling Units (BCUs) provide a generic platform for embedded nodes based on the M68HC05 microcontroller family. A set of open source tools for developing and downloading BCU programs based on the GNU tool chain is presented. It uses a RAD-like (Rapid Application Development) approach. The tool set also supports separating application development and deployment and includes a multi-user and network-capable daemon for EIB access and network management. Issues in porting the GNU C compiler to the target platform are highlighted.

1. Introduction

The European Installation Bus *EIB* [4] is a home and building automation bus system. It is optimized for low-speed control applications like lighting and blinds control. EIB forms a part of the KNX specifications [6]. BCUs (Bus Coupling Units) are standardized, generic platforms for embedded EIB devices. They include the entire physical layer network interface, the link power supply and a microcontroller holding an implementation of the EIB/KNX protocol stack in ROM. The microcontroller is a member of the Freescale/Motorola M68HC05 family. Currently, two major variants, referred to as BCU 1 and BCU 2, exist. Their key differences lie in the system software implementation and the amount of available memory. BCUs are customized for a specific task (e.g., a wall switch) by combining them with application modules containing the necessary hardware. The appropriate software is stored in the microcontroller EEPROM.

Having to write all this software from scratch would prohibitively increase the effort involved in building an EIB/KNX system. Therefore, manufacturers provide ready-made applications which are downloaded during the configuration phase. The project engineer can further customize the behaviour of the node by modifying manufacturer defined application parameters. The BCU applications are distributed in a format which includes the necessary meta information to allow an integration tool to display the parameters. Moreover, it provides the tool with the necessary knowledge how to apply these changes to the program code.

For setting up the communication relationships between nodes, the project engineer has to define bindings between communication endpoints. In EIB/KNX systems, the communication endpoints relevant for process data exchange are referred to as *group objects*. The project engineer combines several of them into a network-wide shared variable by assigning them to a common logical group. This binding information is also stored in the BCU EEPROM and downloaded by the integration tool. Therefore, the meta information also has to cover the group objects of a BCU application.

For EIB/KNX systems, only one single integration tool is necessary to handle every certified device (and application) – no matter from which manufacturer. This approach significantly eases the setup of multi vendor systems. The tool, called *ETS*, will not accept any device application which has not passed compliance certification.

The BCU system software provides applications with an appropriate API for handling data exchange over group objects (and other helpful functions). Yet, with the SDKs presently available an application developer still has to use low-level constructs, e.g., manipulate configuration data or call subroutines at specific memory locations. The project presented instead adopts a RAD like programming model, which we believe to be a novel approach for field devices.

This model encapsulates the system software entities in a way which is inspired by the object-oriented paradigm. It allows defining the functionality of a BCU program in a far more structured and transparent way. Besides levelling the learning curve, this can also be expected to reduce the probability of programming errors. In addition, the customization of applications by a project engineer is supported. This concerns both the generation of the necessary meta information as well as applying the selected configuration to the BCU applications.

2. Key Features

A complete set of development tools for both BCU 1 and BCU 2 – in the following referred to as *BCU SDK* – has been developed. Currently, it only supports the versions for the EIB/KNX twisted-pair medium. The code generation tools are based on the GNU tool chain [2]. The RAD programming model is implemented by preprocess-

0-7803-9402-X/05/$20.00 © 2005 IEEE

ing the input files to yield plain C code. The separation of development and deployment, as outlined above, is supported via an integration tool interface. To allow the use of custom integration tools, a new exchange format was defined which also allows better description of the application behaviour. A Unix daemon encapsulates the differences of a variety of hardware interfaces for EIB/KNX access. Besides supporting the download of BCU applications, it also autonomously executes network management procedures. Its protocol is open for use by other programs.

2.1. Programming model

For interacting with the BCU system software, the BCU SDK offers an increased level of abstraction. The programmer uses a simple specification language to define which 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. Setting up a group object for use as a network variable is accomplished as follows:

```
GroupObject {
  Title "Input";      Name Recv;
  Type UINT1;         on_update Do_io;
};
```

This declaration makes a global variable `Recv` available which will always contain the current value of the network variable. It is of the type "EIB/KNX 1 bit" (this is mapped to an unsigned 8 bit integer in the C code). Whenever its value changes, the function `Do_io()` is called. This handler has to be provided by the programmer.

The value of this network variable will be updated whenever another node transmits a value modification request to a group address which is associated with this group object. The selection of these group addresses is defined separately to allow customization, as will be discussed below. Application parameters are handled in a very similar fashion. All relevant BCU system functionality is accessible this way. This also includes timers, initialization and power-failure handlers and the EIB/KNX client-server communication scheme for node management (referred to as *interface objects* and *properties*). All differences between the BCU 1 and BCU 2 low-level APIs are hidden by the SDK. Additionally, C wrappers for the low-level API calls are provided.

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 possible by declaring a variable with the appropriate storage class and attribute, such as
`int x EEPROM_SECTION EEPROM_ATTRIB;`.

Besides describing the system entities to be used, the specification file is also used to declare meta information. As an example, the *Title* of a *GroupObject* will constitute its textual reference in the integration tool. However – in contrast to ETS – the meta information includes behavioural description.

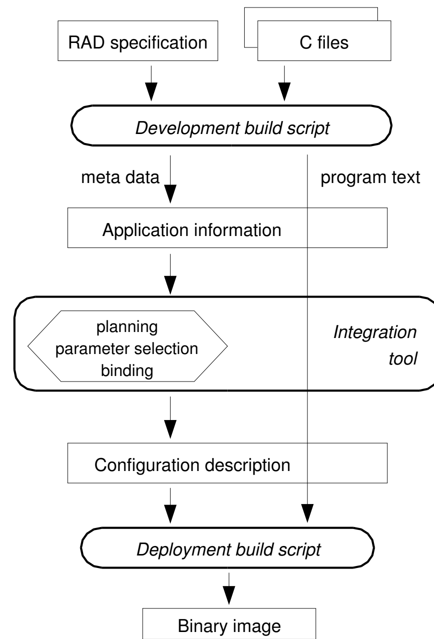

Figure 1. BCU SDK data flow

The functionality of the application is described by declaring one or more functional blocks, which define a processing rule over a set of interfaces. These interfaces can be group objects, parameters or interface object properties. Every block contains an (external) reference to the precise definition of its behaviour.

2.2. Data flow

To allow customization by the project engineer, the input files provided by the programmer are not directly transformed into a BCU memory image which is ready for download. Instead, they take a number of intermediate steps, which are illustrated in Fig. 1. First, they are processed to yield the distribution format. This step is under the control of the software developer. In the current implementation, it is controlled by the development build script. Actually, two separate distribution data formats exist for *meta data* and *program text*.

The meta data format is XML based and referred to as *application information*. An XML Schema definition is provided. The application information describes global aspects like the type of BCU and application module the program is designed for, its functional blocks, group objects, interface objects and properties, and parameters.

Application information data covering all devices available for a project are imported into the integration tool. Based upon these descriptions, the project engineer selects appropriate applications and parameter values, and assigns binding information. For every device used, the integration tool creates a *configuration description* containing these data (e.g., the group addresses to be bound to a `GroupObject`). This format is again XML based. The configuration descriptions are passed back to the deployment part of the SDK, which applies the necessary changes to the program text and generates a downloadable

image. This step happens under the control of the project engineer. In the current implementation, it is controlled by the deployment build script.

An important point is that the XML format hides how an image is finalized and by which means this is done. The format of the program text thus becomes an internal matter of the BCU SDK. It is entirely opaque to the integration tool. The application information contains a reference attribute which uniquely specifies the matching program text. How this text gets from the developer/manufacturer to the project engineer (i.e., deployment build script) is left open. The integration tool may for example store it together with the meta data in a local database, or it may be retrieved from the Web by the SDK deployment part. Since this reference uniquely identifies the program text, it is even possible to store the (appropriately encoded) program text in its place, which is actually the path chosen by the current implementation.

In the BCU SDK, the program text is not a binary image. Instead, it contains the preprocessed C code and mapping information between its identifiers and the ones used in the application information in encoded form. The programs are compiled by the deployment part after all configuration settings are known to reduce image size. This is in contrast to the ETS, which operates on binary images and thus only supports relatively minor modifications in response to a parameter change.

The exchange format is open to other hardware architectures. Although it has been fully defined, no full-fledged integration tool is available yet. Therefore, the BCU SDK contains a minimal implementation which transforms an *application information* into a *configuration description* skeleton using an XSLT transformation.

2.3. Network access and management

To access the EIB bus, the BCU SDK uses a Unix daemon (called *eibd*, Fig. 2). Multiple clients can connect simultaneously via IP or Unix domain sockets. The relevant parts of the EIB protocol stack to send and receive unicast, multicast and broadcast telegrams are provided. 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 setting of node addresses. Also, a bus monitor can be opened, which optionally can decode EIB frames.

The method of access to the EIB/KNX network is entirely hidden by the backends. The backends for the BCU 1, BCU 2 and the TP-UART IC communicate over a serial link, making use of a selection of the low-level drivers described in [5]. The TP-UART is a lean interface IC which only implements medium access control instead of the entire EIB/KNX protocol stack as BCUs do. EIBnet/IP provides tunneling of EIB frames over IP networks.

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.

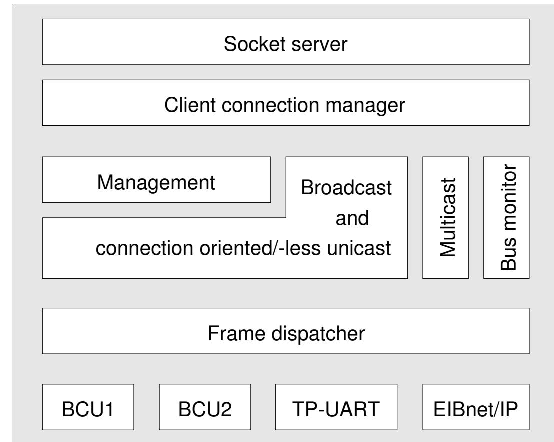

Figure 2. Bus access/management daemon

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 forwards all frames the backend will provide in normal operation mode.

3. GNU tool chain port

For the BCU SDK a C compiler, assembler and linker were needed. As there was no free tool chain available, a new one needed to be created. However, writing a complete C parser with type checking would have meant duplicating work already done in various other places. Therefore, 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. GCC typically uses the GNU binutils as assembler and linker, which were ported too.

Since finding all errors in GCC only by reviewing its output is not a feasible task, 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 [1]. Only aspects necessary for the regression tests are simulated (e.g., neither the interrupt subsystem nor I/O are implemented). The DejaGnu regression testing framework was adapted to be able to run the standard GCC test suites. A limited GNU debugger frontend is also provided.

The M68HC05 family processor core follows a von Neumann architecture with a linear 16 bit address space. The different memory types (RAM, ROM, EEPROM) are mapped at different addresses. The M68HC05 variant used in the BCU 2 has two separate RAM sections. For read accesses, there is no difference for all memory types. Write access to the EEPROM involves a certain control

sequence. The opcodes have a length of 1 byte with 0 to 2 bytes of address information. There are no alignment constraints for instructions or data. The constraints of a BCU ($\ll$ 1 kb EEPROM, $\ll$ 100 bytes RAM) were used as the design driver for the GCC port. It can however be used for any member of the M68HC05 microcontroller family.

The M68HC05 family has only two hardware registers (accumulator and index register). Its stack is a call stack only and thus inaccessible to user programs. Besides, its size is limited as the stack pointer is only 8 bits wide. Although the address space is 16 bit, only an 8 bit index register is available for register indirect addressing. Since GCC is designed to work with a considerably different hardware architecture, certain missing features need to be emulated. 13 bytes of RAM are used to provide additional general-purpose registers expected by GCC. They are located in a region the BCU system routines reserve for temporary use by the user application. Likewise, a data stack is emulated. Multiplication, division and floating point operations are handled by library functions.

GCC expects pointers which can cover the whole address space. Since only an 8 bit index register is available, store, load and call operations with 16 bit pointers are emulated with self modifying code. Four bytes of RAM (plus one to save data for the store operation) are reserved for this purpose. On a pointer operation, the opcode of the corresponding instruction with the appropriate addressing mode is stored at the first byte, followed by the pointer address and finally a return instruction. A call to this RAM region starts the operation.

However, emulation cannot overcome all restrictions. Especially the tight memory constraints of a BCU 1 (256 bytes EEPROM and 18 bytes RAM available to the user) severely limit the available possibilities. Thus, expensive operations like setjmp and longjmp are left out.

Transparent EEPROM access and support for 3, 5, 6, and 7 byte integer types are accessible by using GCC attributes. The semantics for transparent EEPROM access is similar to the named address spaces of [3], but uses two GCC attributes instead of one directive for each named address space. Unlike [3], transparent access when dereferencing a pointer to an EEPROM location involves assigning a different attribute than for a plain variable. Solving this issue would require changes to the GCC frontend, which we wanted yet to avoid for maintainability reasons.

Since the target architecture uses instruction formats with different length, the necessary one is often unknown at assembler runtime and the longest variant has to be chosen. The linker will modify the code to use shorter ones where possible (relaxation). To automatically distribute variables over the non-contiguous RAM sections of a BCU 2, the linker is extended to be able to conditionally move sections between memory regions. GCC and the assembler cooperate to assign each variable a section of its own. For BCU 2 programs, all these sections are initially assigned to one RAM area. If its size is exceeded, the linker will move the necessary amount into the other.

4. Conclusion

A set of software development tools for EIB/KNX nodes was presented. It supports a RAD like development approach for both BCU variants, the customization of applications, and image download and network management via a variety of bus interfaces. The SDK is fully functional, although additional features are still being added and require further testing. All parts are placed under the GPL (GNU General Public License) and can be downloaded together with further documentation from [7].

The GCC port needed several tricks to make GCC cope with the limitations of the target architecture. Also, the expressiveness of the standard regression test suite is limited, as a number of test cases fail due to insufficient memory and stack overflows. 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. Still, it is possible (although tedious) to let the SDK generate the application skeleton only and use inline assembler. The C++ language front end of GCC can be used to compile C programs with the better type checking of C++, as long as features like exceptions and RTTI are disabled. RAD-style BCU programming is only possible in C, however. First evaluation results have shown interesting effects due to the emulation strategy necessary to use GCC with the target architecture. For example, the use of local instead of global variables will result in larger code in some cases due to the pronounced effect of the data stack emulation.

Besides further investigation of these effects, target specific optimizations within GCC provide ample possibilities for further work. Also, the handling of the build process should be improved. The next steps will include adding a graphical interface (e.g., using Eclipse or KDevelop) and the development of a basic, possibly template driven integration tool.

References

[1] GNU development chain for 68HC11 & 68HC12. `http://www.gnu.org/software/m68hc11/`.

[2] GNU tool chain documentation. Distributed with the respective program sources, also available from `http://ftp.gnu.org/gnu/`.

[3] ISO/IEC TR 18037:2004, Programming languages – C – extensions to support embedded processors.

[4] W. Kastner and G. Neugschwandtner. EIB: European Installation Bus. In *The Industrial Communication Technology Handbook*. CRC Press, 2005.

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

[6] Konnex Association. *KNX Specifications, Ver. 1.1*, 2004.

[7] TU Wien, Automation Systems Group. Home and building automation topics/projects. `http://www.auto.tuwien.ac.at/projects/hba/`.