Rapid Application Development for KNX/EIB BCUs

KNX Scientific Conference 2005

Martin Kögler, Wolfgang Kastner, Georg Neugschwandtner

Institute of Automation
Automation Systems Group
Vienna University of Technology
Vienna, Austria

www.auto.tuwien.ac.at/knx

Outline

- Introduction
  - KNX/EIB Bus Coupling Units
- Programming model
  - Traditional vs. Rapid Application Development
  - Key features of the BCU SDK
- Engineering work flow support
  - Development, customization, installation
  - Meta data interchange formats
- Behind the scenes
  - Tool chain port

AUTOMATION SYSTEMS GROUP

KNX Scientific Conference 2005
Rapid Application Development for KNX/EIB BCUs - 2

BCUs (Bus Coupling Units) are standardized, generic platforms for embedded KNX/EIB devices. The BCU SDK provides an open-source framework for developing BCU applications using high-level language.

Its programming model is inspired by modern RAD (Rapid Application Development) environments, where objects are instantiated from a range of available classes and customized by changing properties and assigning event handlers. The developer uses a simple specification language to define which BCU system software entities (group objects, interface objects/properties, timers, ...) he would like to use, their desired configuration, and how he would like to refer to them in his C code.

The presentation discusses the work flow when building an KNX/EIB system and the resulting requirements on the tool chain. 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 meta data description is XML based. It is automatically generated from the RAD specification, including a description of the application's behaviour by means of functional blocks and its possibilities for parameterization.

The solution is based on the GNU tool chain. Specific challenges in porting it to the BCU microcontroller are sketched. The BCU SDK also includes an open KNX/EIB network access and management API for PC-based software (presented in "Open-source foundations for PC based KNX/EIB access and management").

Bus coupling units

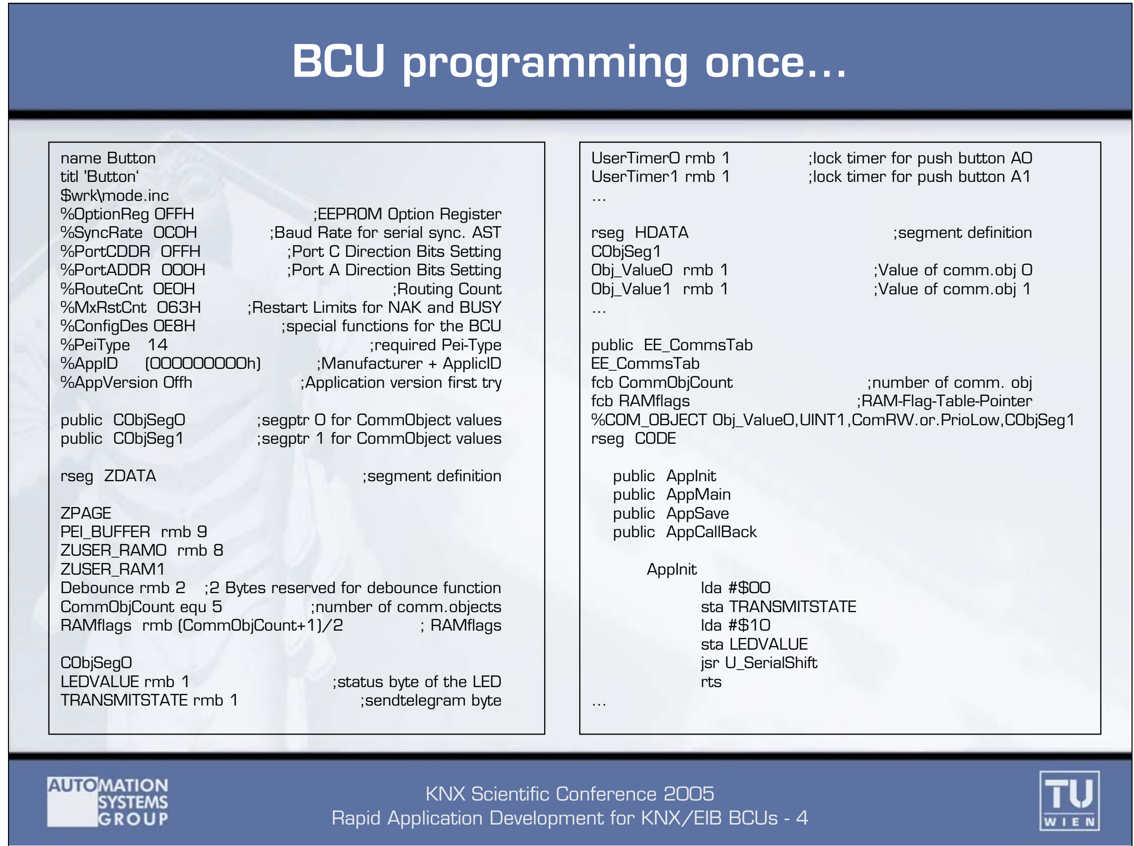
- Standardized, generic platforms for embedded TP1/PL110 devices
  - Microcontroller (Freescale M68HC05 family)
  - Network interface (transceiver, power supply)
  - System software in ROM
- Customization via application modules
  - Standard 10-pin interface (PEI)
  - Matching application programs loaded into EEPROM (via PEI or bus)
- BCU 1 vs. BCU 2
  - Memory size, OS features, speed

KNX Scientific Conference 2005
Rapid Application Development for KNX/EIB BCUs - 3

BCUs (Bus Coupling Units) are standardized, generic platforms for embedded KNX/EIB devices. They include the entire physical layer network interface, the link power supply and a microcontroller holding an implementation of the KNX/EIB protocol stack in ROM. The microcontroller is a member of the Freescale/Motorola M68HC05 family.

BCUs are customized for a specific task (e.g., to act as a wall switch) by combining them with application modules containing the necessary hardware. The appropriate software is stored in the microcontroller EEPROM. Application modules communicate with the BCU via a standardized 10-pin interface (Physical External Interface, PEI). The PEI can also be used for downloading the BCU application. Once it has been assigned its individual address on the network, this can also be done over the network.

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 (still, even the BCU2 is limited to << 1 kb EEPROM and << 100 bytes RAM). While the BCU 1 is a standard M68HC05B6, the BCU 2 is a special variant withTP1-specific on-chip-peripherals. The BCU 2 is also faster than the BCU 1. The M68HC05 variant used in the BCU 2 has two separate RAM sections.



When creating application programs for KNX/EIB nodes, one was till now faced with low-level constructs. For interacting with the BCU system software, configuration data, code hooks and library functions had to be accessed at specific memory addresses, more or less hidden by appropriate symbolic definitions. The code is correspondingly complex (the slide shows part of a BCU 2 example program obtained from www.knx-developer.de).

The BCU SDK offers an increased level of abstraction. Its programming model is inspired by modern RAD (Rapid Application Development) environments, where objects are instantiated from a range of available classes and customized by changing properties and assigning event handlers.

The programmer uses a simple specification language to define which BCU system software entities (group objects, interface objects/properties, timers, ...) he would like to use, their desired configuration, and how he would like to refer to them in his code. The specification file is plain text. Necessary code elements (such as event handlers) are written in C. Inline assembler is also supported.

An example is shown on the next slide (the specification file on the left, the corresponding C file on the right). The example implements a simple timer switch (e.g., for stairway or hallway lighting). Basically, it just closes and opens a relay by switching a PEI output on and off in response to the change of state of a group object. The output is also switched off automatically after some time has passed. A second group object is provided for status output. Whenever the state of the relay changes, it is used to transmit the new state.

... and now

```
GroupObject {
   Title "Switch input";
   Name INPUT;  Type UINT1;     // one bit msg
   StateBased false;   // group-rd not reasonable
   on_update input_changed;    // event handler
};

GroupObject {
   Title "Status output";
   Name OUT;  Type UINT1;       // one bit msg
   Sending true;        // generates OUT_transmit()
   StateBased true;       // answer read requests
};

Timer {
   Name TIMER1;
   Type EnableUserTimer;
   Resolution RES_4266ms;
   on_expire switch_off;              // called when
};                                   // timer expires

on_init init;       // function executed at power on
```

```
void switch_on() {
   _U_ioAST(PEI3_ON);      // BCU API function
   OUT = 1;                  // set value and ...
   OUT_transmit();        // transmit group-write
}

void switch_off() {
   _U_ioAST(PEI3_OFF); // the other way round
   OUT = 0;
   OUT_transmit();
}

void input_changed() {
   if (INPUT == 1)   {
    TIMER1_set(DELAY*10/43); //start tmr
    switch_on();              // ... and switch on
   }
   else   {
    TIMER1_del();                 // stop timer
    switch_off();             //  ... and switch off
   }
}
```

AUTOMATION SYSTEMS GROUP

KNX Scientific Conference 2005
Rapid Application Development for KNX/EIB BCUs - 5

TU WIEN

The `Title` allows to give a short description of the corresponding group object (for later binding). The group objects are assigned the internal names of INPUT and OUT. The length of both messages is one bit (type name `UINT1`).

In the corresponding C code , 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 in C. OUT has the `Sending` attribute set, which makes the function `OUT_transmit()` available. It is used to transmit an `A_GroupValue_write.req` for its associated group object.

OUT is also `StateBased`, which tells the BCU to autonomously answer an incoming `A_GroupValue_Read.req`. This makes the status output readable from the network. Since IN contains no useful data for reading, its `StateBased` attribute is set to false.

When a new value is received for IN (via an `A_GroupValue_write.ind`), the `input_changed()` function will be called. Similar handlers exist for start-up (`on_init`) and save routines.

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 `EnableUserTimer` (which provides additional functionality over the standard BCU user timers). One timer tick corresponds to 4.266s. The timer interval is expected in seconds and scaled to this resolution. When the timer expires, the handler `switch_off()` will be called (once). Appropriate functions are generated for starting the timer with a specified timeout period and canceling it.

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.

Programming model

- Benefits
  - Focus on desired functionality rather than how to achieve it
  - Level learning curve
  - Easier to maintain
- Features
  - GNU tool chain port (GCC/binutils) for M68HC05 family
  - Free and open (GPL)
  - Supports both BCU 1 and BCU 2 (including interface objects/properties and access protection)
  - Transparent EEPROM and optimized Low RAM access
  - Non-standard length integer types (3/5/6/7 byte)
  - OS neutral (Linux tested)
  - Application download over the bus (via eibd)
  - Integration Tool interface

KNX Scientific Conference 2005
Rapid Application Development for KNX/EIB BCUs - 6

The BCU SDK RAD programming model allows defining the functionality of a BCU program in a more structured and transparent way. Besides levelling the learning curve, this can also be expected to reduce the probability of programming errors. It also takes a step towards code portability.

During the course of this project, a set of free development tools for BCU1 and BCU2 (TP1 version) were created. All differences between the BCU1 and BCU2 low-level APIs are hidden by the SDK. The tight memory constraints of the former however limit what can be achieved using this platform. The code generation tools are based on the GNU tool chain and are all under the GPL.

Signed and unsigned integer types are available in any byte width from 1 to 8 to optimize RAM usage. 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;`. A respective storage class and attribute exists for the BCU1 and BCU2 Low RAM section.

Download of BCU applications is handled via a Unix daemon (eibd) that encapsulates the differences of a variety of hardware interfaces for EIB/KNX access. The code generation part is operating system independent (although the code base is currently maintained on Linux).

The separation of development and deployment, as outlined below, 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.

Work flow

- Development
  - Application program
  - Definition of parameters
  - Testing, packaging
- Project Planning
  - Selection of KNX/EIB devices
  - Customize by setting parameters and defining communication relationships
- Installation
  - Physically install devices
  - Define individual address and download configuration

AUTOMATION SYSTEMS GROUP

KNX Scientific Conference 2005
Rapid Application Development for KNX/EIB BCUs - 7

TU WIEN

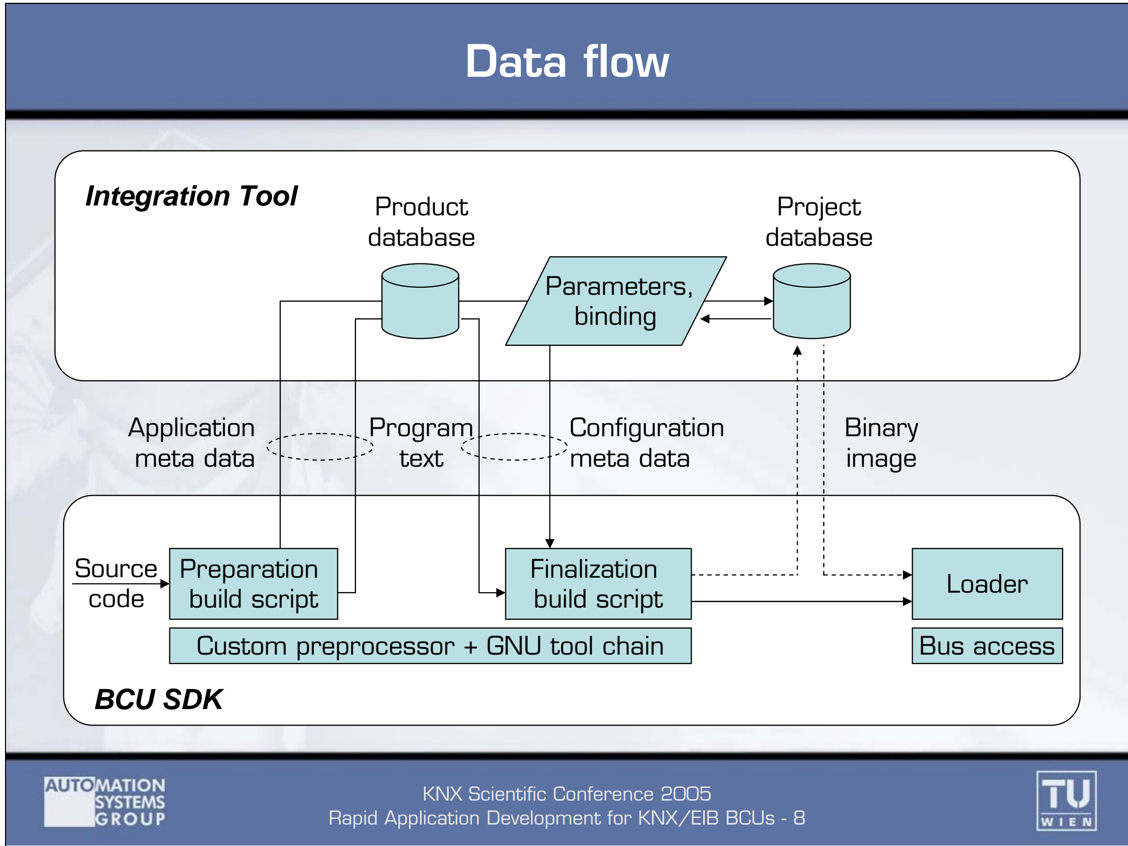
An SDK for KNX/EIB would not be complete without support for the work flow which leads from the device manufacturer to the final application. This work flow is divided into three steps.

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 KNX/EIB 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.

Project planning: A project engineer selects appropriate KNX/EIB devices to fulfill 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.

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.

To allow the roles of software developer and project engineer to be separated, the tool chain has to provide suitable interfaces.

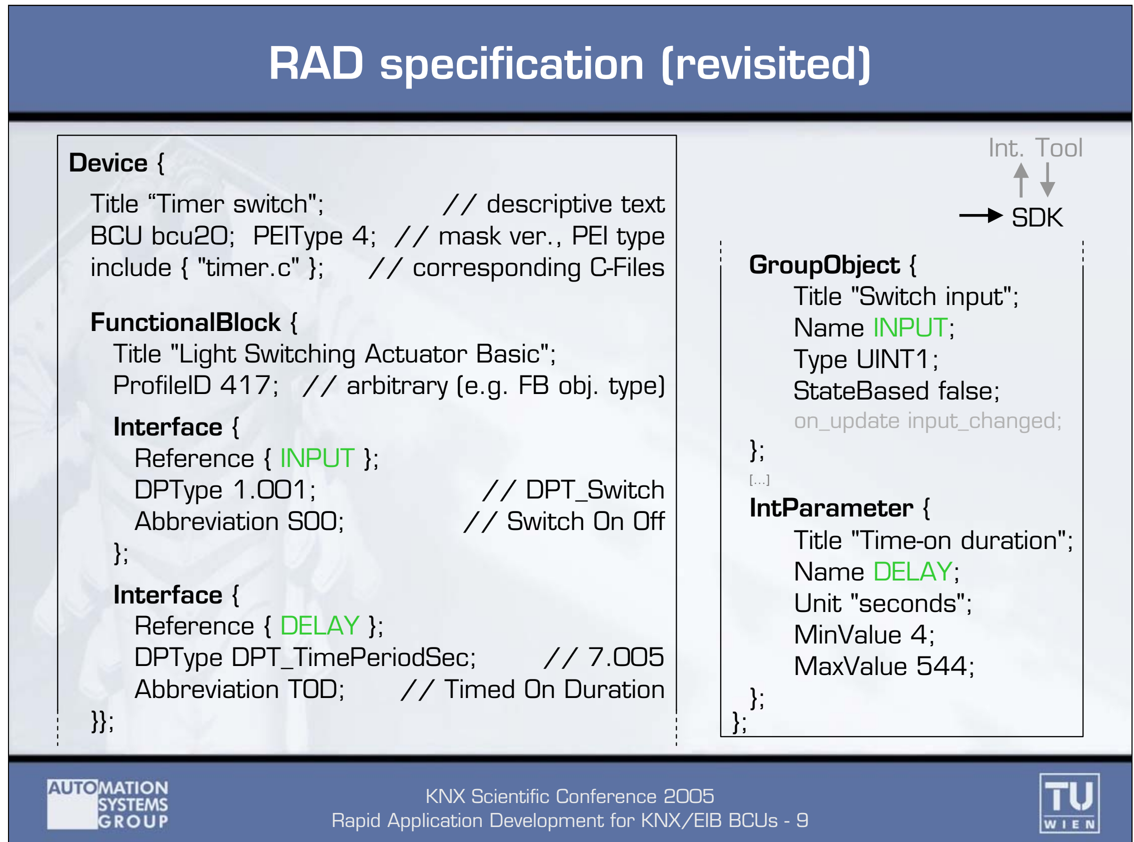


With the BCU SDK, the software developer uses the preparation build script to prepare the source code of the new application for distribution to project engineers. The build script controls the execution of the necessary tools. First, the input files are run through a custom preprocessor which implements the RAD programming model. The code is checked for errors and transformed into a format suitable for distribution (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.

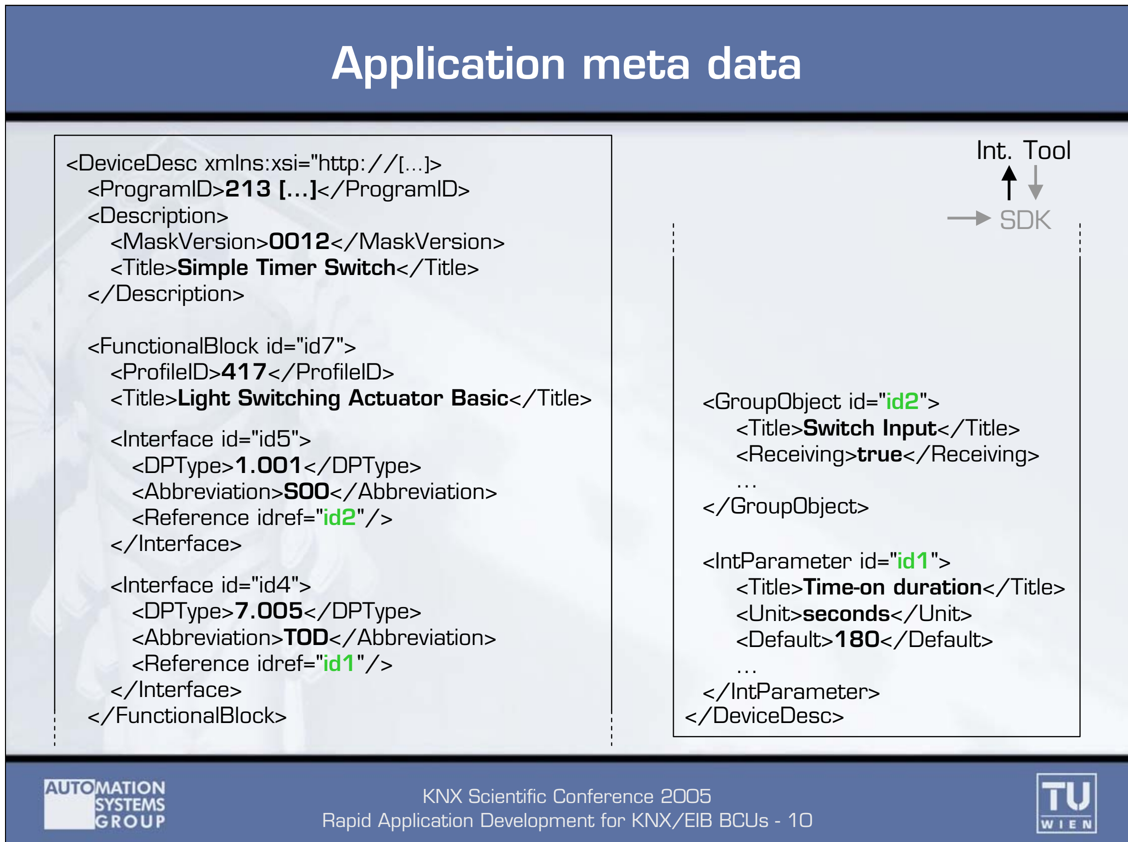


The application meta data are automatically derived from the RAD specification. For this purpose, this file also contains information about other interfaces of the application besides group objects, i.e., application parameters and interface objects and their properties. In our example, the time-on duration is to be customizable via the integration tool. It is therefore specified as an (integer) parameter and described further by its useful range and unit.

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 ``Off''.

To document the behaviour of an application, however, DPTs are not sufficient. Therefore, the developer declares one or more functional blocks. Every such block references one or more interfaces (which in turn are associated with a DPT). It describes a certain processing rule or dependency covering them. Our example actually implements the KNX ''Light Switching Actuator Basic'' (LSAB) functional block. An external, precise definition of the behaviour associated with a functional block is referenced by its ProfileID. In the example, the Interface Object type of the LSAB was chosen. However, this reference could equally be a URL pointing to a custom description.

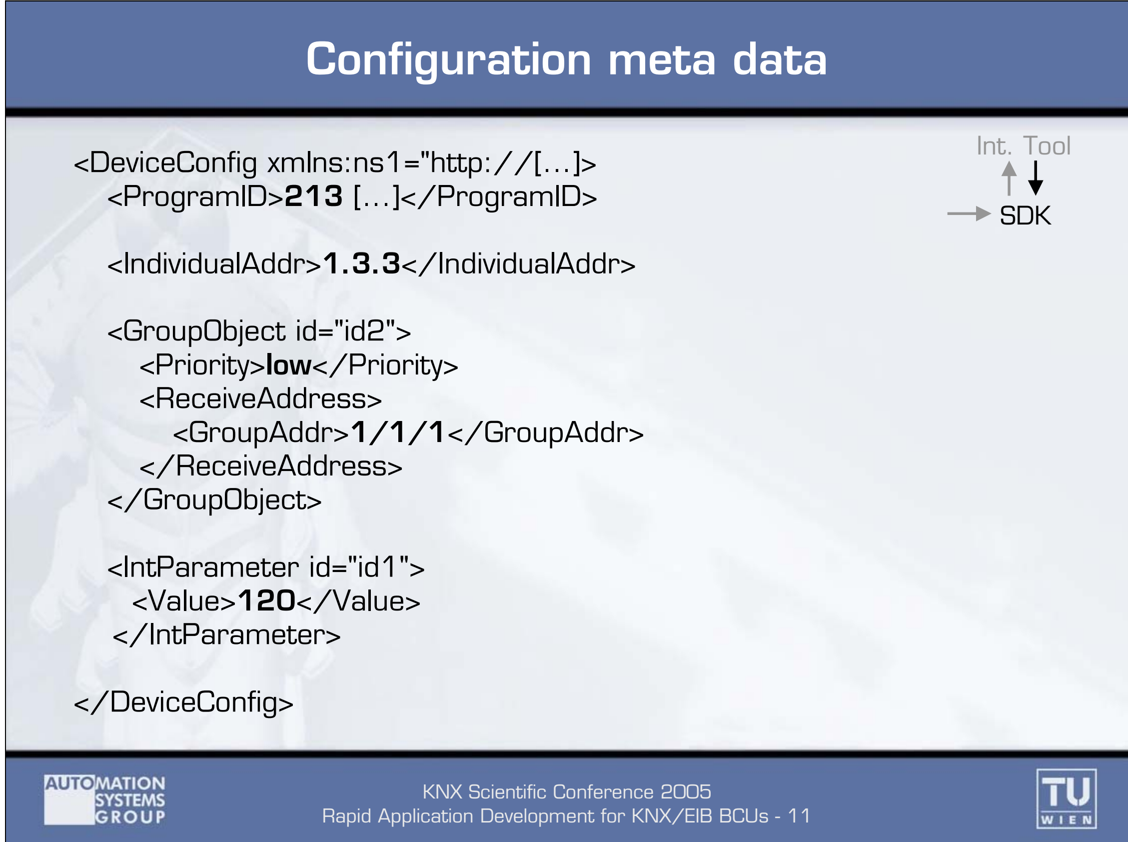
Display rules can be defined for all entities depending on selected parameter values to allow revealing the complexity of a particular application to the project engineer step by step.



The meta data description is XML based. XML Schema definitions are provided. The structure of the application description closely follows the RAD specification. Instead of internal names, references between functional blocks and their interfaces are made via element IDs.

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 meta data interchange formats are designed to be open for node architectures other than BCU1 and BCU2. Rather than being modeled after their specific constraints, they are aligned with the KNX/EIB wire protocol and configuration model (TP1 S-Mode). Also, parameter definitions are generic rather than tailored to BCUs.



The configuration meta data format looks similar. For every configurable entity from the application meta data description, 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 text contained in the application information.

The XML format hides how and by which means this information is applied to generate the final binary image, which is ready for download to the BCU. This is done by the finalization build script. Although the integration tool is responsible for obeying the constraints of a particular node architecture (which it derives from the mask version), the finalization pass performs an additional check. For example, access control cannot be specified for a BCU1.

The binary image may 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.

GNU tool chain

- Ported programs include
  - GCC (GNU C compiler)
  - Binutils (assembler, linker and object file tools)
  - CPU core simulator
  - GDB frontend for the simulator
  - C runtime libraries for the simulator
- GCC expects
  - Many general purpose registers (GPR)
  - A data stack
  - Pointers which can cover the entire address space

AUTOMATION SYSTEMS GROUP

KNX Scientific Conference 2005
Rapid Application Development for KNX/EIB BCUs - 12

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. The GNU Binutils are made up of gas (assembler), ld (linker) ld and tools/libraries for the manipulation of object code in various object file formats (e.g. bfd).

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. 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.

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.

GCC port

- M68HC05 family limitations
  - Two hardware registers (accumulator and index register)
  - Only a call stack
  - 16 bit address space, but only an 8 bit index register for register indirect addressing
- Emulation of missing features
  - RegB–RegN (reserved by BCU OS) used as GPR
  - Data stack
  - Multiplication, division and floating point operations handled by library functions
  - 16 bit pointer emulation uses self modifying code

AUTOMATION SYSTEMS GROUP

KNX Scientific Conference 2005
Rapid Application Development for KNX/EIB BCUs - 13

The constraints of a BCU (<< 1 kb EEPROM, << 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, if enough memory for the stack and virtual registers is available in appropriate memory regions.

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.

GCC expects pointers which can cover the whole address space. Yet, although its address space is 16 bit, the M68HC05 architecture only offers an 8 bit index register for register indirect addressing. To provide the necessary store, load and call operations with 16 bit pointers, they are emulated using self modifying code.

Multiplication, division and floating point operations are handled by library functions. 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 a larger stack than the M68HC05 GCC port supports.

Still, emulation has its limits. For instance, setjmp/longjmp could not be implemented since the call stack pointer is inaccessible and saving the virtual registers would need too much memory.

Conclusion and outlook

- BCU SDK: A free set of tools to develop programs for BCU 1 and BCU 2 in a RAD like way
- GNU tool chain ported to the M68HC05 architecture
- Possible next steps
  - Add graphical user interface
  - Development of integration tool
  - Support PL110 BCU, TP1 polling

Project homepage:

http://www.auto.tuwien.ac.at/~mkoegler/index.php/bcus

AUTOMATION SYSTEMS GROUP

KNX Scientific Conference 2005
Rapid Application Development for KNX/EIB BCUs - 14

A set of software development tools for KNX/EIB BCUs was presented. It supports a RAD like development approach for both BCU variants including the customization of applications.

The SDK is fully functional, although some features still require further testing. 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 integration tool. Moreover, the SDK could be extended to support PL110 BCUs and the TP1 polling mode.

The GCC port needed several tricks to make GCC cope with the limitations of the target architecture. It works reliably. However, the expressiveness of the standard regression test suite is limited, as a number of test cases fail due to insufficient memory and stack overflows. These cases yet have to be investigated and classified correctly as "expected failures" (XFAIL).

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.

All parts of the BCU SDK are placed under the GPL (GNU General Public License) and can be downloaded together with further documentation from the project homepage.