Diplomarbeitspräsentation der Fakultät Informatik

FAKULTÄT FÜR !NFORMATIK

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

Diplom-Studium:
Informatik

Martin Kögler <mkoegler@auto.tuwien.ac.at>

Technische Universität Wien
Institut für Rechnergestützte Automation
Arbeitsgruppe Automatisierungssysteme
Betreuer: ao. Univ.-Prof. Dr. Wolfgang Kastner
Betreuer: Dipl.-Ing. Georg Neugschwandtner

Abstract

The European Installation Bus (EIB) is a field bus for home and building automation. Bus Coupling Units (BCUs) provide a standardized 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. Its RAD-like (Rapid Application Development) approach is introduced.
The tool set supports the separation of application development and deployment and includes a multi-user and network-capable Linux daemon for EIB access and network management.
The GCC port needed several creative measures to make GCC cope with the limitations of the architecture.

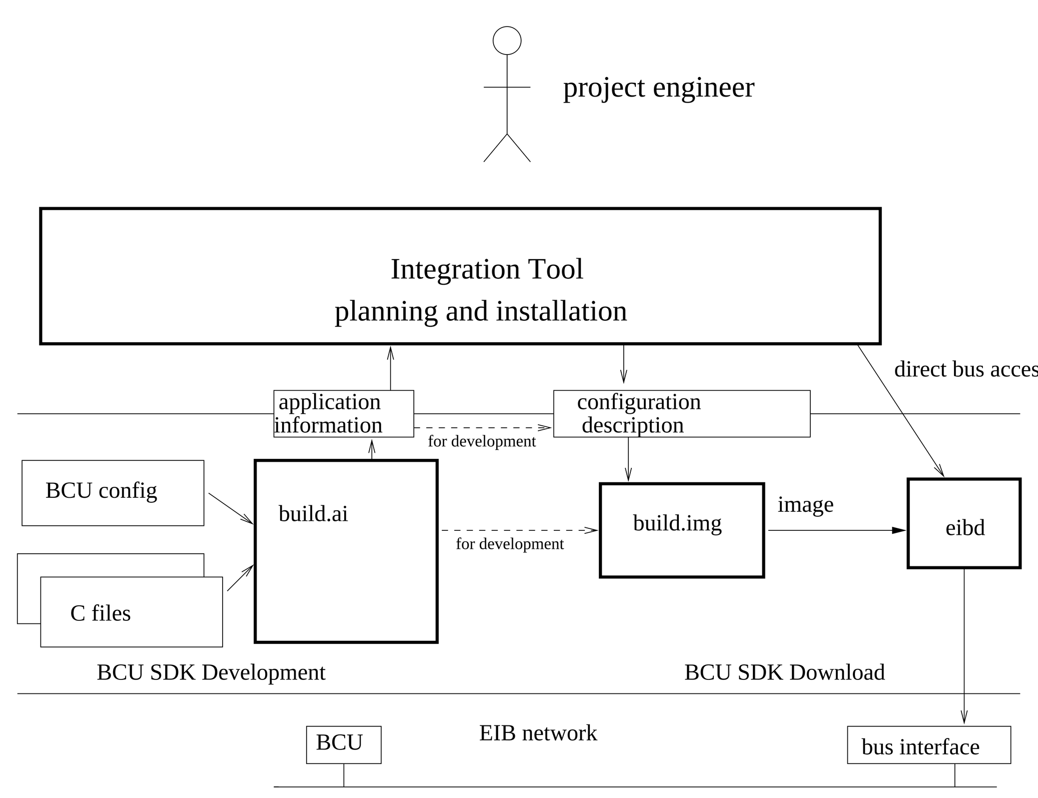
Features of the BCU SDK

- based on the GNU utilities (GCC, Binutils)
- provides RAD like concept (instead of a plain assembler interface), requiring the programmer to specify properties and event handlers only
- C (with inline assembler) is used for programming event handlers
- includes an interface for integration tools (this will be parts of future projects)
  - no ETS interface
  - provides support for compilation at download time
- provides access to same management functions over different bus access devices
  - provides an API to provide EIB access in other programs. Several utility programs, which also illustrate the use of this API, are included.
  - includes a standard bus monitor, which optionally can decode EIB frames.
  - special monitor mode (called *vBusmonitor*) even allows some traffic to be traced without switching to bus monitor mode.
- no GUI interface

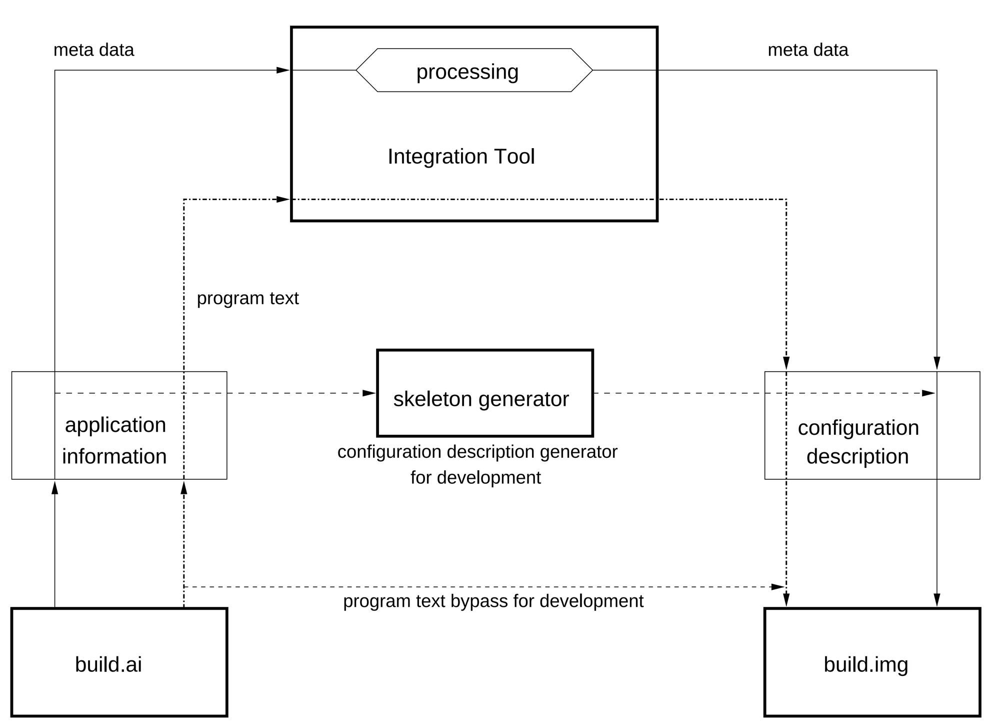
The following limitations are present:

- generates larger code, than optimized, hand written assembler code
- not compatible with the original, commercial BCU SDK
- If the bus is accessed via a BCU 1 or BCU 2, this BCU is inaccessible to the *BCU SDK*.

Workflow



Dataflow



Port of the GNU Utilities

- Binutils (assembler, linker and object file tools)
  - uses a different syntax than Motorola, e.g. *%X* instead of *X* for the X register
- GCC (GNU C compiler)
- CPU core simulator
- GDB frontend for the simulator
- C runtime libraries for the simulator

Use of the simulator

- Simulator and C runtime libraries needed for GCC regression tests
- GDB for analyzing GCC generated code
- ⇒ incomplete

Relaxation

As small code size is needed, relaxation is implemented (shrinking code sections at link time):

- Instruction formats with different length exist.
- The longest one has to be chosen at assembler runtime, if the precise requirements are unknown.
- The linker replaces longer variants if possible.
- Expanded conditional jumps are converted back, if possible.

Section movement

- The BCU 2 has non contiguous RAM sections.
- GCC needs automated distribution of variables.
- GCC prefixes each variable with a special command (*.section* command with name ending in *!!!!!*).
- The assembler creates a unique section (by replacing *!!!!!* with a unique number).
- The linker can be instructed to move sections from a full memory region into another memory region.

GCC

Limitations of the M68HC05 family:

- Two hardware registers (accumulator and index register)
- Only a small call stack
- Only *8 bit index plus address* addressing mode (besides a fixed 8 or 16 bit address).

GCC has different requirements:

- Many GPR (general purpose registers)
- A data stack
- Pointers, which can cover the entire address space

⇒ Emulation of missing features — available memory limits useable functions.

GCC internals

- 13 Bytes of RAM (RegB–RegN, reserved by BCU OS) are used as GPR.
- A byte of RAM is used as data stack pointer. Data stack starts at a 256 byte boundary. Using a different initialization value, a smaller stack area can be used.
- 16 bit pointers are emulated with self modifying code.
- mul, div and floating point operations are handled by library functions.
- Support for 1 to 8 byte integer types
- Support for transparent eeprom access (named address spaces)

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

- The *eeprom* attribute enables transparent access.
- Pointers pointing to such an EEPROM location need the attribute *eepromt* instead.
- A write access to the EEPROM is replaced by a library call.
- Actually placing variables in the EEPROM is done with other attributes.
- Expensive operations like setjmp/longjmp are left out.

Compilation process

- GCC parses a function
- GCC performs target independent optimizations on a tree representation.
- GCC converts it to high level RTL (Register Transfer Language)
  - uses only GPRs and memory locations as operands.
  - uses pseudo instructions for the 8/16/24/32/.. bit operands
- some optimizations are done
- register allocator replaces pseudo registers with GPRs and stack locations.
- Each high level RTL instruction is split into multiple low level RTL instructions
  - each instruction corresponds to an assembler instruction or library call.
  - stack pointer is cached in X register
- some optimizations are redone.
- assembler code is generated

Current GCC status

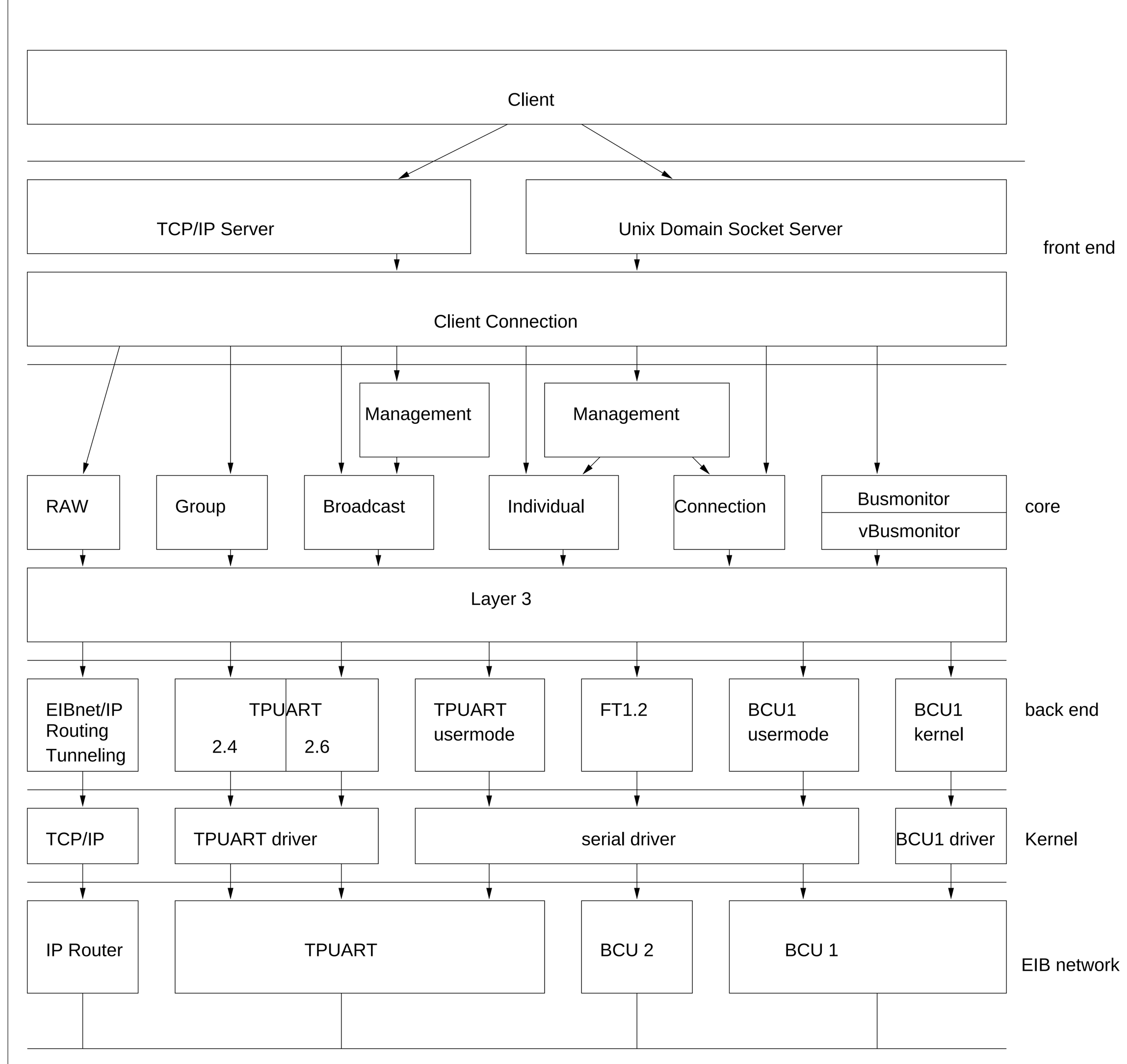
- GCC is working
  - 1335 of 36394 failed regression test cases
  - large parts fail because of insufficient memory and stack overflows.
- No target specific optimizations (e.g. peephole optimizations) implemented.
- G++ frontend is partially working (e.g. no exceptions).
- Some limitations:
  - no overflow detection
  - overflows can occur in compare operations
  - ...

⇒ Lots of improvements are possible

Future work:

- Low level RTL generated at expand time
- Conditon code handled as register

Bus access with eibd



- A network capable, multi user daemon (named *eibd*) was developed.
- Provides access to Layer 4 as well as complex management functions over a simple protocol.
- best effort, cooperative *vBusmonitor* mode, which do not prohibit sending activity
- runs under Linux (some backends even work on Windows using Cygwin)
- The bus access is hidden by the backends:

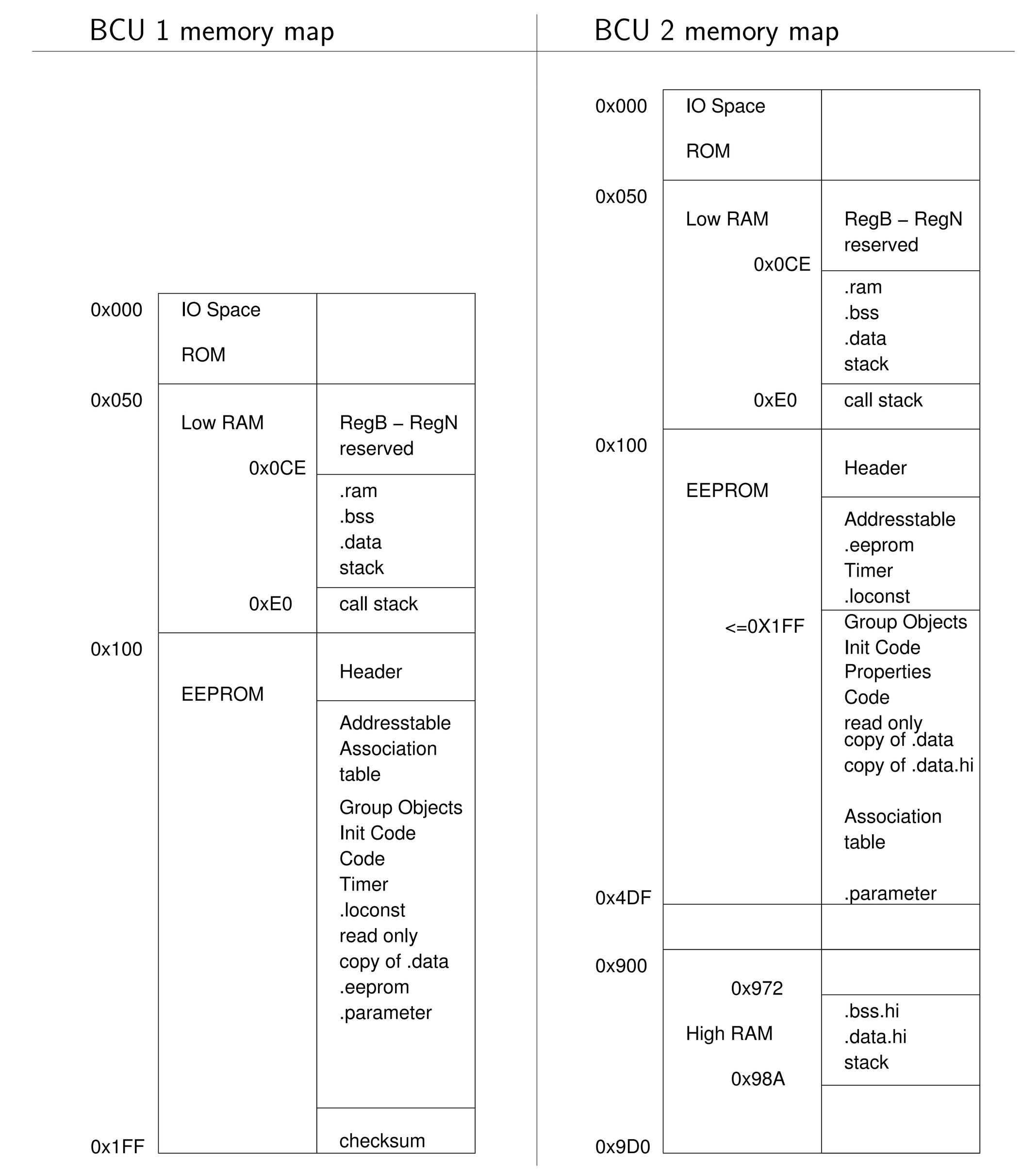
FT1.2 protocol of the serial interface of the BCU 2.
EIBnet/IP EIBnet/IP Routing and EIBnet/IP Tunneling client.
TPUART protocol of the TPUART IC. It uses the plain serial driver or a Linux kernel driver.
PEI16 protocol of the serial interface of a BCU 1 using a kernel driver, which does the time critical data exchange. An experimental version using the plain serial driver exists.

BCU SDK

The build process is done in two steps:

- The *application information* is created by the *build.ai* program. It contains all neccessary information to build a real program including all meta data for use by an integration tool. All errors which could occur in the second step should be detected by this program.
- In the second step, the final binary image is built using the *configuration description*.

Memory maps used by the BCU SDK:



Example program - A negation which can be disabled

The following program passes changes of the group object *recv* to the group object *send*, while the *cond* group object is enabled. The transmitted values are negated. All group objects are of type DPT_Bool (1.002).

BCU configuration - 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;
   DPTType DPT_Bool; // same as 1.002
  };
  Interface {
   Reference { recv }; Abbreviation recv;
   DPTType DPT_Bool;
  };
  Interface {
   Reference { cond }; Abbreviation cond;
   DPTType DPT_Bool;
  };
 };
 GroupObject {
  Name recv; Type UINT1; on_update send_update;
  Title "Input"; StateBased true;
 };
 GroupObject {
  Name send; Type UINT1;
  Sending true; Title "Output";
  StateBased true;
 };
 GroupObject {
  Name cond; Type UINT1;
  Receiving true; Title "Condition";
  StateBased true;
 };
};
```

C code fragment - cond.c

```
void send_update() {
  if(cond)
  { send=recv+1;  send_transmit(); }
}
```

Configuration description - cond.ci

```
<?xml version="1.0"?>
<DeviceConfig>
  <ProgramID>xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx</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>
```

Further details

Project homepage `http://www.auto.tuwien.ac.at/~mkoegler/index.php/bcus`
EIB/KNX projects of the group: `http://www.auto.tuwien.ac.at/knx/`