Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Real-Time Field Bus systems with Linux

Semester Thesis

Fabian Dalbert

dalbertf@ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Pengcheng Huang
Georgia Giannopoulou
Tonio Gsell
Gert Brettlecker (Ergon Informatik AG)
Prof. Dr. Lothar Thiele

Thesis ID:

SA-2015-21

January 13, 2016

Acknowledgements

This thesis was written under the supervision of Pengcheng Huang (`pengcheng.huang@tik.ee.ethz.ch`), Georgia Giannopulou (`ggeorgia@ee.ethz.ch`) and Tonio Gsell (`tgsell@ethz.ch`) at the Computer Engineering and Networks Laboratory lead by Professor Dr. Lothar Thiele (`thiele@ethz.ch`) at ETH Zurich and Gert Brettlecker (`gert.brettlecker@ergon.ch`) at the Ergon Informatik AG. I would like to thank my supervisors as well as the industry team at Ergon for their incredible patience, their supervision and support and Professor Thiele for the opportunity to write this thesis in his research group.

Abstract

Can the timing constraints of a soft real-time field bus protocol be met on a microcontroller running a real-time Linux framework without additional hardware? This semester thesis implements and evaluates the MP-Bus protocol, a proprietary field bus protocol developed by the Belimo AG, with a preempt-RT patched Linux operating system. It concludes that software solutions can replace dedicated hardware modules for bus control in soft real-time environments. Different real-time capable Linux frameworks and the BACnet MS/TP protocol were additionally assessed in the theoretical part. A kernel module bit banging the UART port using pin multiplexing at runtime to change pin modes to GPIO was implemented to comply with the MP-Bus' collision avoidance scheme. The requirements were tested with a test setup specifically designed for this semester thesis and a range of measurements on the protocol implementation.

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**

1.1 Motivation 1

1.2 Related Work 2

1.3 Contributions 2

1.4 Outline 3

2 Background **4**

2.1 Linux with Real-Time Capabilities 4

2.2 Real-Time Frameworks 5

2.2.1 Preempt-RT 5

2.2.2 Xenomai 6

2.2.3 L4/Fiasco with DROPS 7

2.2.4 FreeRTOS 7

2.2.5 Xen 8

2.3 The MP-Bus and BACnet Protocols 9

2.3.1 MP-Bus 9

2.3.2 BACnet MS/TP 13

2.4 The Belimo Shared Logic Platform 14

3 Design & Implementation **16**

3.1 Architecture Overview 17

3.1.1 Hardware 17

3.1.2 Software 18

3.2 Linux with the Preempt-RT patch 18

3.3 Bit Banging UART — The Driver 20
3.3.1 User Space — Kernel Space Communication with Ioctl . . 20
3.3.2 Bit Banging Functionality 23
3.3.3 Pin Multiplexing 24
3.3.4 KThread 26
3.4 Real-Time Scheduling 26
3.5 Outlook: BACnet MS/TP 27

4 Evaluation 28
4.1 Test Setup 28
4.2 Measurements 30
4.2.1 Byte and Roundtrip Times 30
4.2.2 System Latency 32

5 Conclusion and Future Work 35

Bibliography 36

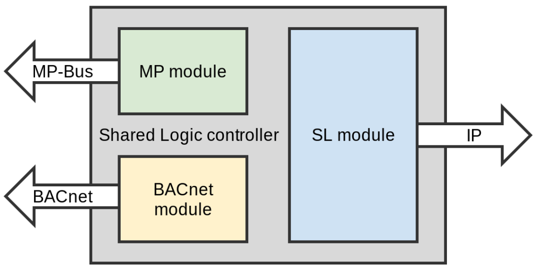
Chapter 1

Introduction

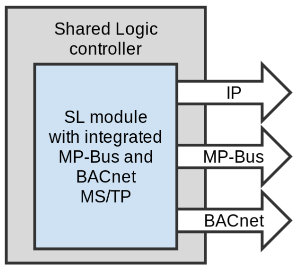
1.1 Motivation

This semester thesis seeks to evaluate the capabilities of Linux in a soft real-time environment, more specifically, when using field bus systems in embedded control. It was conducted in collaboration with Ergon Informatik AG and based on embedded control systems programmed for Belimo AG. Belimo is a major developer and manufacturer of actuators and valves for HVAC (heating, ventilation and air conditioning) control. The control software on the embedded systems runs in a Java virtual machine on top of a light-weight Linux distribution. Communication between actuators, valves and control modules uses the MP-Bus and BACnet MS/TP protocols. Additionally, the control module can be addressed over IP. MP-bus, short for multi point bus, is a proprietary serial communication protocol developed by Belimo, frequently used in the HVAC sector. BACnet, an abbreviation for building automation and control networks, is a widely used standard in building automation. Here the MS/TP (Master-Slave/Token Passing) protocol defined by the BACnet standard will be considered, however the implementation focuses on the MP-Bus protocol.

Can all constraints of the MP-Bus and BACnet protocols be met with a real-time capable Linux framework? The control modules by Ergon and Belimo have, as of today, dedicated microcontrollers in the control module to handle bus communication. These microcontrollers ensure reliability, predictability and compliance with all timing constraints (see Figure 1.1a). Some parts of the MP-Bus protocol, namely the PPX communication mode and dominance arbitration, are however not currently implemented in the control modules. If a framework manages to provide the same reliability and meet all soft real-time constraints imposed by the bus protocols, the physical layout can be reduced to the target configuration (see Figure 1.1b). From a manufacturing standpoint, this allows for significant cost reductions, from the programming standpoint, for reduced complexity of the control development and from a research standpoint, for assessing the time predictability of real-time Linux kernels.



(a) Current configuration



(b) Target configuration

Figure 1.1: Configuration of the Belimo Shared Logic module

1.2 Related Work

Real-time Linux environments have received quite some attention in recent years, mainly focused on scheduling and worst-case latency (see for instance Emde and Gleixner [10] or Bertolotti and Manduchi [13]). Some of this research will be tapped into in the background chapter. Fieldbus systems have also been in researchers' focus (an overview is given by Mahalik [25] for example), BACnet as one of the main protocols used in building automation among them (for instance Merz et al. [26]). No research could however be found on the MP-Bus, being a proprietary bus protocol, or its integration with real-time Linux.

1.3 Contributions

This thesis' contributions are a theoretical overview of selected real-time frameworks and the MP-Bus and BACnet MS/TP communication protocols as well as a practical implementation of the MP-Bus. As the software, called Belimo Shared Logic [7], on the control modules is written in Java, it needs a Linux environment capable of running a Java virtual machine (jvm). A regular Debian Linux with the preempt-RT patch has therefore been chosen for implementation. For the MP-Bus a proof of concept that all timing requirements can be fulfilled without any dedicated hardware is sought. The practical part of the project consists of the setup and configuration of a Beaglebone Black development board with Debian Linux with the preempt-RT patch. Furthermore the current MP-Bus implementation done by Ergon in the Shared Logic framework is extended to meet all real-time constraints. Scheduling in the real-time Linux system as well as tests and measurements round out the practical part.

1.4 Outline

After this introduction the second chapter will present the background of this semester thesis. Its first part gives an overview over five real-time frameworks that can be used with Linux and the implementation choice for this thesis will be explained. In the second part the MP-Bus and BACnet MS/TP protocols will be presented and their real-time requirements explored. The practical configuration of the real-time Linux framework on the Beaglebone Black and the implementation of the MP-Bus protocol are presented in detail in chapter three. The fourth chapter presents an evaluation, the test setup and measurements for the implementation. Finally there will be some concluding remarks and an outlook on possible future work on the topic.

Chapter 2

Background

2.1 Linux with Real-Time Capabilities

In the embedded world, real-time capabilities can be achieved through a variety of frameworks. Five of these have been selected for more detailed analysis here (see Table 2.1).

Name	**Type**	**Open Source**	**Hard Real-Time Capable**	**Full Linux OS**
preempt-RT	Linux kernel patch	Yes	No	Yes
Xenomai	Kernel addition	Yes	Yes	(Yes)[1]
FreeRTOS	Real-time OS	Yes	Yes	No
L4/Fiasco	Microkernel	Yes	Yes	(No)[2]
Xen	Hypervisor	Yes	Depends[3]	(Yes)[1]

Table 2.1: Overview of Frameworks

The criteria for choosing a framework were:

1. Support for hard real-time applications.
2. Ability to run a customised Linux operating system on top.
3. Complexity of adaptation of the existing code base as well as the new MP-Bus implementation.

Compatibility with a full-fledged Linux operating system, the second criterion, is needed to run the Shared Logic control software in a Java virtual machine. This already disqualified FreeRTOS as it has very minimalistic functionality. Xenomai and L4/Fiasco (with the DROPS architecture and L^{4}Linux,

[1]Compatible with any Linux flavour.
[2]The DROPS architecture and L^{4}Linux add full Linux support
[3]See Subsection 2.2.5 for the function of a hypervisor.

see Section 2.2.3) meet requirements 1 and 2 but would need extensive adaptation of code and/or an extensive programming effort to port bus communication to the real-time part of the system. A solution using a hypervisor like Xen is not feasible at this point in time as can be seen in section 2.2.5. Therefore the preempt-RT patch has been chosen for implementation in the scope of this semester thesis.

2.2 Real-Time Frameworks

2.2.1 Preempt-RT

The preempt-RT patch aims at adding real-time capabilities to the mainline Linux kernel. It is based on Ingo Molnàr's work and maintained by Linux Foundation Fellow Robert Gleixner. The first efforts to add real-time capabilities to the Linux kernel got attention in 2006 [10]. After quite some struggle to find funding, the Linux Foundation started a collaborative project to advance real-time Linux in October 2015, with Google as a Platinum member and several other companies, Texas Instruments among them, involved as well [11].

The preempt-RT patch renders the Linux kernel fully preemptible through five changes/additions. First spinlocks are replaced by rtmutexes. Secondly most critical sections are made preemptible, non-preemptible sections are still possible using raw_spinlock_t. Priority inheritance is added to in-kernel mutexes, spinlocks and rw_semaphores. The preempt-RT patch also turns interrupt handlers into preemptible kernel threads and, last but not least, adds high-resolution timers to the Linux kernel [12]. Many of these patches, such as the high-resolution timers, have in the mean time been integrated into the mainline kernel.

The performance of a preempt-RT patched Linux has been improving over the last years, version 3.6.6 (i.e. Linux kernel version 3.6.6) fared 36% better on average over eight tests (thread switch latency, interrupt latency and semaphore release duration among others) compared to version 2.6.33.7 according to Fayyad-Kazan et al. [19]. The worst-case values derived in their benchmarking test were all in a medium two-digit microseconds range on a Intel ATOM based platform comparable to the Beaglebone Black in performance [19].

In theory the preempt-RT patch makes any Linux system hard real-time capable, in practice however the latency depends on proper configuration of the system. As Linux is nowhere near as deterministic as a minimalistic microkernel or real-time operating system, properly tuning the system and finding worst-case execution times is non-trivial [13, p. 409*ff.*]. Guaranteeing response times in milliseconds and execution times in a higher three digit microseconds range should be achievable with the preempt-RT patch [14]. As the time constraints in the case of the field bus systems focused on in this thesis are not as strict, the preempt-RT patch has been selected as the framework of choice for the implementation.

2.2.2 Xenomai

Xenomai uses an *interrupt abstraction* approach to achieve real-time properties with Linux. This means the Xenomai environment controls all real-time applications and runs the operating system as a thread that is only called when no higher priority application needs the CPU. Xenomai is based on Adeos, the Adaptive Domain Environment for Operating Systems. Software solutions like Adeos, providing a Hardware Abstraction Layer (HAL) underneath the operating system, are often called nanokernels [13, p. 412].

Like all Adeos-based projects, Xenomai is patched onto the Linux kernel and uses its HAL. It creates three domains, that are executed simultaneously but with different priorities; the primary domain hosting the real-time nucleus, an interrupt shield that can block interrupts from reaching the secondary domain and the secondary domain, the Linux kernel (see Figure 2.1 for domains and interruption pipeline). Real-time threads started in the primary domain can be switched to the secondary Linux domain to use Linux system calls and be switched back afterwards. This adds some unpredictability but allows real-time tasks to use the full Linux operating system. Xenomai developers are looking into greater integration with the preempt-RT patch to improve secondary domain performance [15, p. 434*f*.].

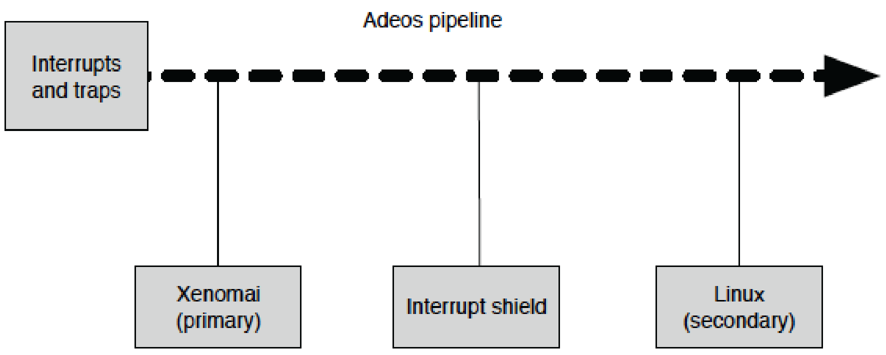


Figure 2.1: The Xenomai domains in the Adeos pipeline [13, p. 414]

Xenomai can reliably be used for time constraints as present in the MP-Bus and BACnet, it does however require additional programming efforts and code adaptations that are beyond the scope of this thesis. It is certainly an interesting framework for real-time applications requiring limited Linux system resources to run in a Linux environment.

2.2.3 L4/Fiasco with DROPS

L4/Fiasco is a second generation microkernel, part of the L4 microkernel family. It was developed at the TU Dresden [4]. In general microkernels are very minimalistic in the functionality they provide, allowing for deterministic behaviour and hard real-time applications. The L4/Fiasco microkernel adds full operating system support through L^4Linux. On top of the microkernel, a framework called DROPS (Dresden Realtime Operating System) manages the real-time and Linux personalities (see Figure 2.2). The approach is similar to Xenomai, does however differ in one important detail; the whole legacy Linux kernel is shifted to user-space, an approach that, according to the developers, does not significantly hamper Linux performance [16].

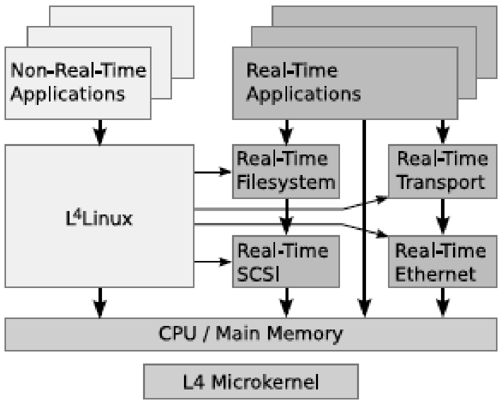


Figure 2.2: The DROPS architecture [16, p. 3]

L4/Fiasco with the DROPS architecture and L^4Linux can provide the real-time capability and full Linux support needed for the implementation, however the extensive adaptation needed and the possible unreliability of running the control module in the user-space Linux system make this approach non-viable for the scope of this thesis.

2.2.4 FreeRTOS

FreeRTOS is a small and highly portable open-source operating system with a focus on real-time capabilities. Its main target are small-scale embedded systems with very limited or no need for high-level and multimedia applications. It only allows for one process with different threads (or tasks) to run simultaneously, highly improving deterministic behaviour while limiting the power of the operating system. [13, p. 191*ff.*]

[4]See `http://os.inf.tu-dresden.de/L4/` for the L4 microkernel family

FreeRTOS provides hard real-time capability with very little overhead. The lack of support for a full Linux operating system makes FreeRTOS infeasible for any system running the Java based control software developed by Ergon.

2.2.5 Xen

Xen is an open-source type-1 or bare-metal hypervisor. This means Xen runs directly on the hardware using a microkernel design. It provides virtualisation for one or several guest operating systems on top. One of these operating systems, the control domain or dom0, provides drivers and has the ability to control the virtual machines [17]. Parallel setups with Linux and a real-time operating system have been shown to work with Xen. Avanzini et al. [18] implemented a single-board dual-OS system with a full-featured Linux and the ERIKA enterprise real-time operating system. They designed their study as a proof of concept that can be ported to other real-time operating systems such as FreeRTOS but did not actually evaluate performance. A dual-core ARM CPU was used and the cores statically dedicated to one operating system each, a setup not possible on the single-core Beaglebone Black. Imperfect isolation of the two operating systems and the use of Linux as the higher privileged control domain system could still pose difficulties for real-time predictability in this framework.

The limited experience with dual operating systems with Linux and a small real-time solution keep Xen from being the setup of choice for an application such as a HVAC control system for the moment. Additionally the framework tested by Avanzini et al. [18] is not feasible for a single-core CPU system like the Beaglebone Black. With additional research hypervisors like Xen can become a competitive solution for dual-use systems with real-time and full-featured parts in the future.

2.3 The MP-Bus and BACnet Protocols

2.3.1 MP-Bus

Overview

MP-Bus (short for MultiPoint bus) is a proprietary communication protocol developed by Belimo AG. It is mostly compliant to the RS-232 standard but uses a higher voltage than defined *ibid* with 24V. The MP-Bus is a master/slave bus and can be used to connect up to 8 actuators to one master. Sensors can be connected through actuators using the Belimo Multi-Function Technology (MFT). MP masters can be MP cooperation nodes (PLC/DDC[5] controllers) or MP Gateways (they link to a different field bus system such as BACnet). The MP-Bus uses bidirectional half-duplex communication via the U5 wire, normally used for analog transmission. The dominant voltage level is low, the idle line high. Table 2.2 gives an overview over the communication parameters [20].

Baudrate	1200 baud
Structure	1 startbit, 8 databits, 1 stopbit
Parity	no
Data order	LSB first
Package structure	1 startbyte, 0–7 databytes, crossparity, lengthparity

Table 2.2: MP-Bus communication parameters [1]

The MP-Bus can operate in two different modes, PP-mode (Point-to-Point) and MP-mode (Multipoint). Additionally there is the PPX-mode, only used for automatic addressing. The operating modes are defined by the address programmed on the slave. The default mode is the PP-mode, programming an address different from its PP-address into a device will automatically switch it to MP-mode. Figure 2.3 shows the different communication and address modes for the MP-Bus protocol with short descriptions for each of them [1].

MP-slaves only answer to commands received, never start communication from their side. In case of error, the master has to resend commands for them to be properly executed. This holds for all possible errors except those occurring on the application layer, where the slave will return an error code in PP-mode or addressed communication. An example of any non-application layer error can be seen in Figure 2.4 [1].

[5]PLC — Programmable Logic Controller, DDC — Direct Digital Control

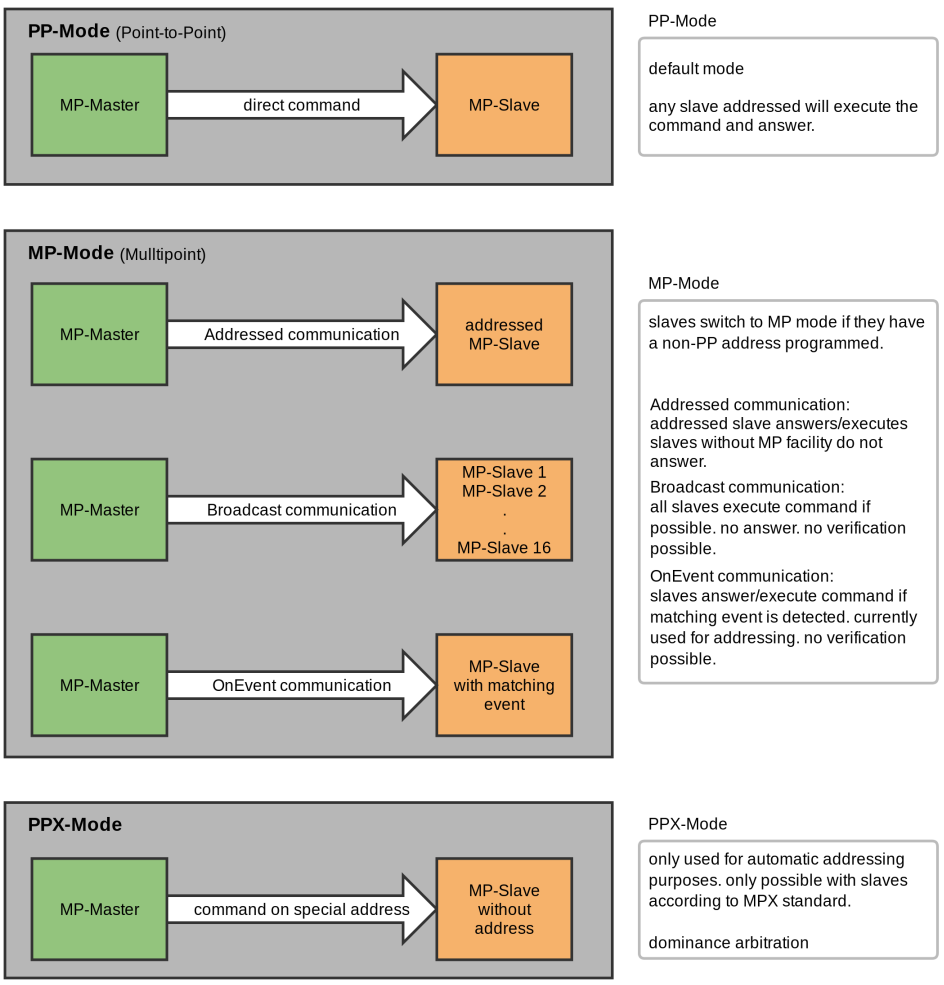

PP-Mode (Point-to-Point)
MP-Master
direct command
MP-Slave
PP-Mode
default mode
any slave addressed will execute the command and answer.
MP-Mode (Mulltipoint)
MP-Master
Addressed communication
addressed MP-Slave
MP-Master
Broadcast communication
MP-Slave 1
MP-Slave 2
.
.
MP-Slave 16
MP-Master
OnEvent communication
MP-Slave with matching event
MP-Mode
slaves switch to MP mode if they have a non-PP address programmed.
Addressed communication: addressed slave answers/executes slaves without MP facility do not answer.
Broadcast communication: all slaves execute command if possible. no answer. no verification possible.
OnEvent communication: slaves answer/execute command if matching event is detected. currently used for addressing. no verification possible.
PPX-Mode
MP-Master
command on special address
MP-Slave without address
PPX-Mode
only used for automatic addressing purposes. only possible with slaves according to MPX standard.
dominance arbitration

Figure 2.3: Communication and address modes for the MP-Bus protocol [1]

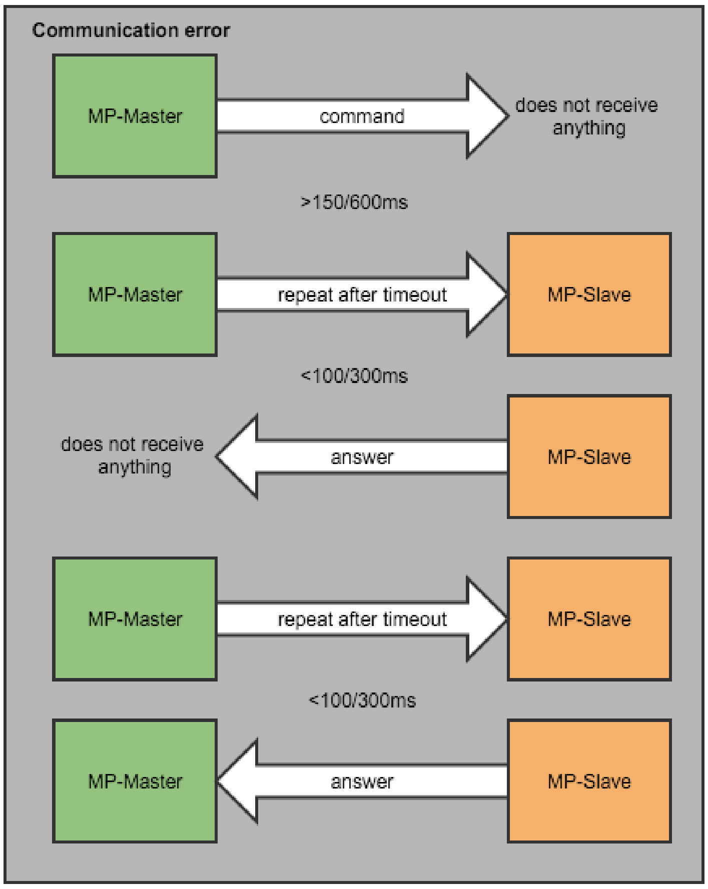


Figure 2.4: Example of a communication error on the MP-Bus [1, p. 13]

Timing Requirements

Most timing requirements of the MP-Bus protocol are in a range achievable with standard Linux operating systems. Table 2.3 gives an overview of the timing requirements in the protocol. The bit time will usually be handled by the UART driver.

Parameter	**Min**	**Typical**	**Max**	**Unit**
bit time	821	833.3	846	μs
frame time[6]	8.21	8.33	8.46	ms
byte gap	0		5	ms
master timeout/repeat interval				
- addressed/OnEvent communication	600			ms
- broadcast communication	150			ms
answer delay				
- config. commands	8.46[7]		300	ms
- other	8.46[7]	50	100	ms
command delay	8.46[7]			ms

Table 2.3: MP-Bus timing requirements [1]

There is however one special case that needs stricter timing. According to the MPX standard, slaves must implement the MP_Get_SeriesNo command. All slaves, which have no MP address must answer with their serial number. To avoid collisions on the bus, slaves must implement the dominance arbitration method. Before sending the start byte, any slave must check whether the bus is idle, i.e. "high". While sending its serial number the slave must continuously observe the bus and immediately stop transmission if the line is not equal to the previously sent bit as this can only be the case when another slave is sending at the same time. Only one slave should be able to transmit its serial number at any given time. After receiving the serial number, the MP-master can change the address of a slave using regular commands, then repeat the procedure until all slaves have received an MP-address [1].

To implement dominance arbitration, a slave must be able to transmit a bit, sense the bus state and, depending on the outcome, break or send the next bit while strictly meeting all timing requirements, i.e. the bit time of maximal 846μs. The roundtrip timing requirements are not as hard to meet, the lowest timing requirement between receiving a command and sending the answer is 100ms as can be seen in Table 2.3 [1].

[6] 10 bits, see table 2.2

[7] i.e. frame time

2.3.2 BACnet MS/TP

Overview

BACnet is a very broad protocol, developed and maintained by the American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE). As can be seen in Figure 2.5, the protocol includes different standardised data links but one common network and application layer. The BACnet object model and services on the application layer are the core of the protocol. An object contains a collection of properties and was designed to be extensible. Only the Device object is required in a ”BACnet device“, all else can be defined by the implementing party. BACnet services are designed around a ”client-server“ model, where a client can send a request which is answered by the server. On the network layer internetwork communication is defined. The details will not be elaborated here except for two points. Each BACnet device has a unique BACnet address consisting of a network number and a medium access control (MAC) address and there can be maximum one active path between any two devices. A maximum of 128 devices can be connected to a network. On the data link layer, there are currently seven technologies defined in the protocol. The protocol based on the EIA-485 serial communication standard, BACnet ”Master-Slave/Token-Passing“ (MS/TP) will be considered in more detail [21, p. 31*ff.*].

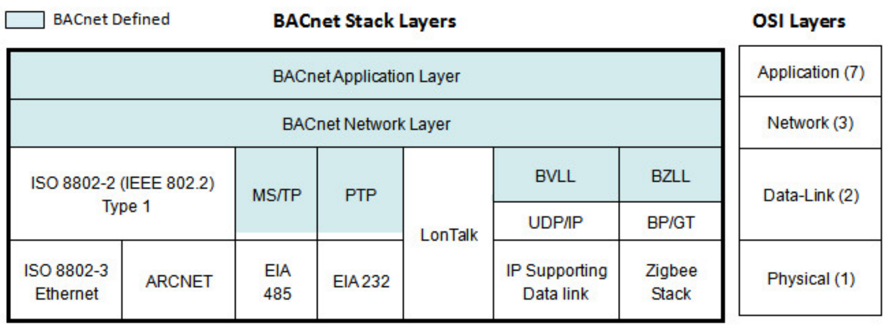


Figure 2.5: BACnet protocol architecture [21, p. 35]

BACnet MS/TP uses the UART pins on a device. A BACnet master uses a peer-to-peer token passing ring to coordinate with other masters and can initiate and respond to service requests. Slaves can only answer a direct request from a master and are not very common. The MS/TP protocol allows for several baud rates to be used, baud rates of 9’600 and 38’400 bps must be implemented though. Eight frame types that can be sent are currently defined by ASHRAE. The operation of a MS/TP master node can be described by a finite state machine (FSM, see Figure 2.6). A master can only actively send after receiving a Frame Type 0, i.e. the token. The token must be passed on to the next master in the token ring after a defined maximum number of frames. To prevent token loss, tokens are not acknowledged. Instead the master that has just passed the token expects at least one frame to be sent by the next master within a certain time frame, otherwise it will resend the token [21, p. 98*ff.*].

Real-Time Constraints

There are no strict real-time constraints in the BACnet MS/TP protocol. It can run relatively stable in a non real-time desktop Linux environment[8]. The main difficulty identified by Ergon with running BACnet MS/TP on a microcontroller is the CPU load generated by the token passing. The timing requirements depend on the baud rate used, as response timeouts, token loss, etc. are defined by octet times (the time it takes to send a byte/octet as well as start- and stopbits) [21, p. 101*ff.*].

2.4 The Belimo Shared Logic Platform

The Belimo Shared Logic platform is the the core software running on Belimo control modules. It is developed by the Ergon Informatik AG and runs hardware independent in a Java virtual machine. The Shared Logic platform implements control for all bus systems supported by the control module as well as ethernet connectivity and a web server. It additionally provides an "Application Designer" for HVAC engineers to program product-specific control models without requiring specific programming knowledge. After such a model is deployed to a node using the "Commissioning Tool", the Shared Logic platform controls connected sensors and actuators, according to the algorithms defined by the HVAC engineer. The state of the system can be checked and reconfigured via web application through a web browser at runtime [7].

[8]according to the BACnet protocol stack sourceforge site `http://sourceforge.net/projects/bacnet/`

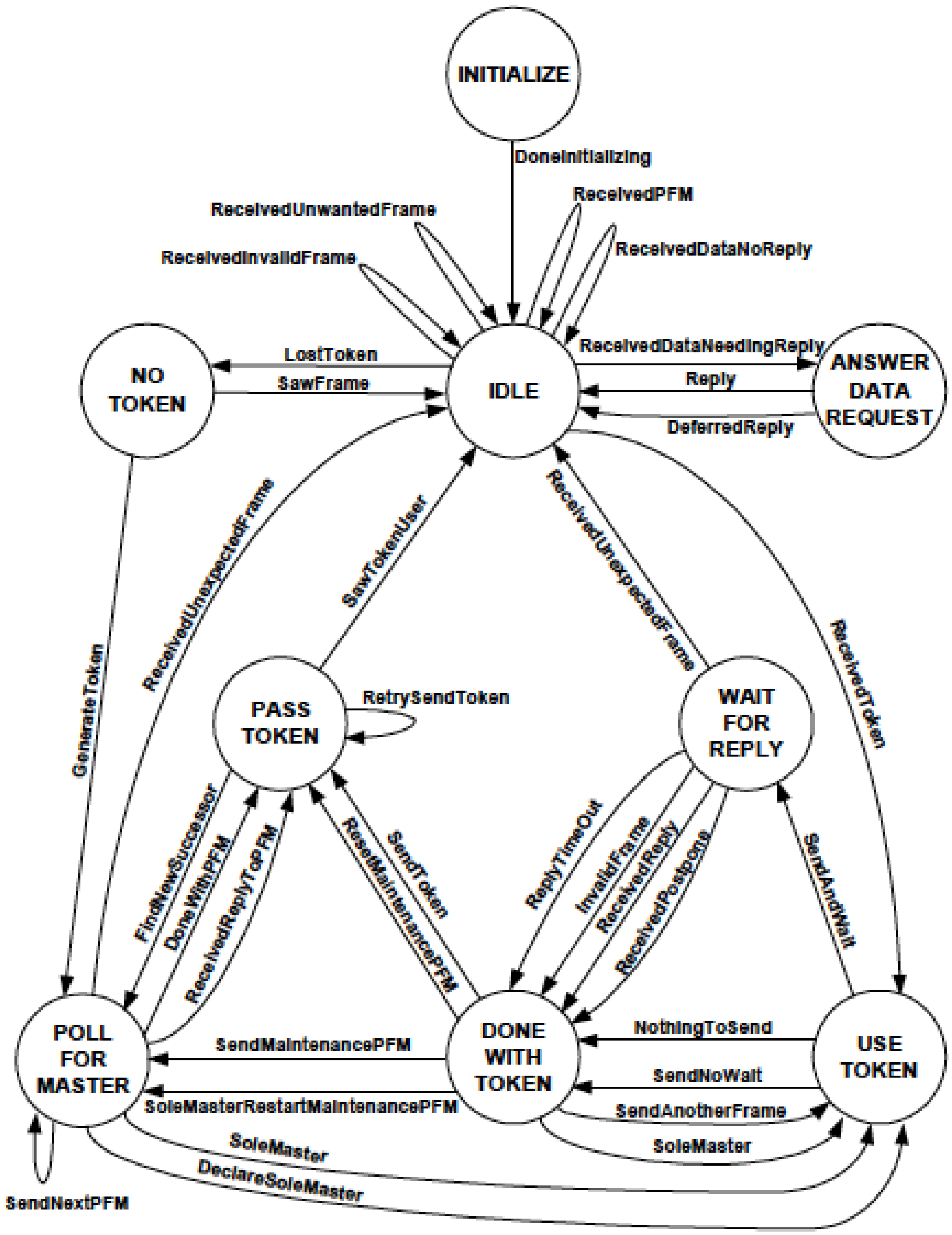


Figure 2.6: Finite State Machine of a BACnet master node [6, p. 86]

Chapter 3

Design & Implementation

The practical part of this semester thesis consisted of developing an implementation for the MP-Bus that can fulfil all timing requirements and integrate it into the Shared Logic setup by Ergon AG. The next Section, Architecture Overview, introduces the hardware and software design of the implementation. To achieve the task specified, first a Beaglebone Black (see Figure 3.1, referred to as beaglebone for the rest of this chapter) had to be setup with a preempt-RT patched Linux system and configured to ensure correct functionality and reach low latency levels. The configuration of the operating system is described in Section 3.2. Next a kernel module to bitbang the UART TX pin while observing the state of the UART RX pin in between bits was written. It is to be used for dominance arbitration when answering MP_Get_SeriesNo commands on the MP-slave side alongside the regular UART driver. As the MP-Bus protocol uses bidirectional half-duplex communication the same signal will lay at both the UART RX and TX pin at any given time. The implementation of the kernel module is described in Section 3.3. Finally, the kernel module had to be integrated into the Linux kernel and adapted for real-time scheduling as described in the last part of this chapter.

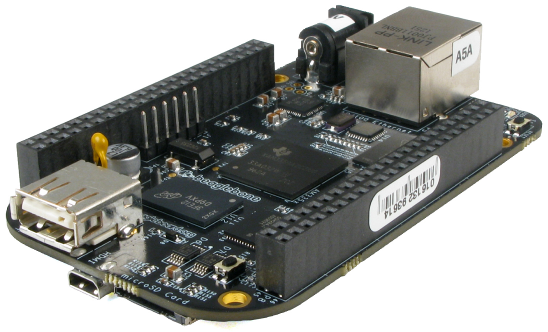


Figure 3.1: Beaglebone Black

3.1 Architecture Overview

3.1.1 Hardware

As previously mentioned, a Beaglebone Black, an open-source microcontroller manufactured by Texas Instruments, was used for the implementation. The beaglebone features the same AM335x family processor used by the newest Belimo control boards, which simplifies the application of any findings to the Belimo system, while its open-source architecture provides better support for the implementation and increases its significance for research. An overview of the most important technical specifications of the Beaglebone Black can be seen in Table 3.1.

Part	**Feature**
Processor	1GHz Sitara AM3358
SDRAM	512MB DDR3L 800MHz
Onboard Flash	4GB
Additional Memory	microSD slot
Ouput pins	69 Pins, up to 8 modes each

Table 3.1: Beaglebone Black technical specification [5]

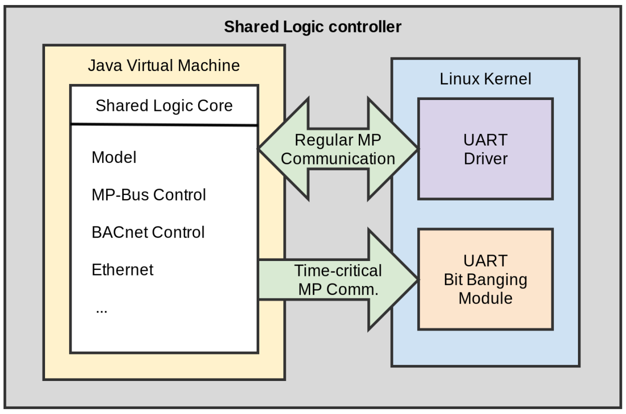


Figure 3.2: Shared Logic software architecture

3.1.2 Software

An overview of the Shared Logic software architecture as described in Chapter 2.4 is depicted in Figure 3.2. The implementation of the Shared Logic core in Java seen on the left is existing code by the Ergon AG. This thesis focused on the right-hand side, namely the Linux Kernel and time-critical MP-Bus communication through the new UART Bit Banging Module. Dominance arbitration as the most time sensitive part of the MP-Bus protocol has not previously been implemented by Ergon.

3.2 Linux with the Preempt-RT patch

Setting up the beaglebone with a working preempt-RT patched Debian Linux (see Table 3.2 for the setup) turned out to be a lot more work than expected. The Debian distribution was chosen, as it is a standard, versatile distribution with a large community and as the BeagleBoard.org Foundation provides pre-configured Debian images. Linux kernel version 4.1.15 is used, prepatched by Robert Nelson for BeagleBoard.org with the beaglebone and preempt-RT patches according to his instructions on eewiki.net[1]. The kernel configuration had to be modified to make the kernel fully preemptible and reduce possible latency (see Table 3.3 for kernel configuration changes), most configuration entries were already adapted to the preempt-RT patch though.

Linux Distribution	Debian 7 Wheezy
BeagleBoard.org image	2015-03-01
Linux kernel	4.1.15-bone-rt-r17

Table 3.2: The setup used on the BeagleBone Black

Configuration	**Effect**
CONFIG_PREEMPT_RT_FULL	make kernel fully preemptible
Disable CPU Frequency scaling	disable processor frequency scaling as frequency changes introduce high latency
Disable CPU Idle	disable processor sleep as waking up from deep sleep takes time

Table 3.3: Changes made to the BeagleBoard default preempt-RT kernel configuration

[1]see `https://eewiki.net/display/linuxonarm/BeagleBone+Black`, accessed 2015–12–01

Instability in the USB-Ethernet connection between the beaglebone and the host computer posed additional difficulty. The connection could not be enabled after installing a new kernel to the beaglebone and/or restarting it. The reason could not be pinned down even with extensive research and help from the BeagleBoard community[2]. A kernel update on the host side and restarting the host everytime after the beaglebone had fully booted had to be used as a workaround.

Two shell scripts had to be written to gain internet access through USB-Ethernet on the beaglebone (see Code 3.1 and Code 3.2). These had to be executed after each respective reboot.

Code 3.1: network.sh script on the beaglebone

```
ifconfig usb0 192.168.7.2
route add default gw 192.168.7.1
echo "nameserver 8.8.8.8" > /etc/resolv.conf
```

Code 3.2: network.sh script on the host

```
#eth0 is my internet facing interface, eth1 is the BeagleBone
    ↪ USB connection
ifconfig eth1 192.168.7.1
iptables --table nat --append POSTROUTING --out-interface eth0 -
    ↪ j MASQUERADE
iptables --append FORWARD --in-interface eth1 -j ACCEPT
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Another implementation challenge related to the kernel was the impossibility to cross-compile kernel headers for the beaglebone. Precompiled headers in the repository do not match a manually compiled kernel whereas cross-compiling the kernel-headers leads to architecture mismatch errors when trying to compile the kernel module on the beaglebone. This is a known problem according to Robert Nelson from Beagleboard.org[3]. A workaround for this problem is to cross-compile the kernel module on the host computer.

Some additional steps were needed to get the Shared Logic framework running on the beaglebone. As Shared Logic is currently implemented on a ptxdist[4] Linux with kernel 2.6.30, the return of “uname -r” had to be tweaked to deceive the application. The options for the Java virtual machine (jvm) had to be adapted, the virtual machine needed more memory on the beaglebone. Finally the Shared Logic software watchdog needed to be disabled.

[2]https://groups.google.com/forum/#!forum/beagleboard

[3]see https://groups.google.com/forum/#!category-topic/beagleboard/support/vEOFtqUJGw0

[4]http://www.ptxdist.org/

3.3 Bit Banging UART — The Driver

The development of a kernel module to bit bang the UART pins was chosen as the solution to be pursued as dominance arbitration has the single most strict timing requirements when implementing MP-Bus support on the beaglebone. Furthermore the ability to check the current bus state is integral to dominance arbitration and can best be provided by a driver addition or a separate kernel module. Bit banging can be described as software controlled serial communication, where a pin, usually a general input/output (GPIO) pin, is toggled with the exact timing required for the serial communication protocol chosen [22]. Bit banging using the UART pins is needed as the UART driver provided by Linux does not support sensing the bus while sending and as regular MP-Bus wiring connects to these UART pins. This module will only be used to answer MP_Get_SeriesNo commands as the regular UART driver provides the functionality needed for all communication without dominance arbitration.

The kernel module was developed in four steps:

1. Implementation of a kernel module with ioctl for user space communication.
2. Bit banging on the beaglebone's GPIO port.
3. Taking control of the UART pins via pin multiplexing through direct memory access to the processor's control module.
4. Converting the function for sending a response[5] to a kernel thread (kthread) for better schedulability.

Code 3.3 gives an overview over all methods used in the kernel module.

3.3.1 User Space — Kernel Space Communication with Ioctl

The ioctl part of the kernel module was implemented following the instructions by Ariane Keller [23]. Ioctl was included as a user space — kernel space interface. Two ioctl commands were implemented, read and write (see Code 3.4). The kernel device registers the character device used for ioctl during initialisation of the kernel mdoule. The memory used for the buffer is also allocated during initialisation using `kmalloc`.

[5]i.e. up to 10 bytes, see Table 2.2

Code 3.3: Overview over kernel module methods

```
// ioctl read method, called when a process has already opened the
    ↪ char device
static ssize_t device_read(struct file *filp, char __user *buffer,
    ↪ size_t length, loff_t *offset)

// ioctl write method, called when a process has already opened the
    ↪  char device
static ssize_t device_write(struct file *filp, const char __user *
    ↪ buff, size_t len, loff_t *off)

// set pin modes to GPIO for bit banging
int mux_pins(void)

// reset pinmodes
int unmux_pins(void)

// method bit banging a char over the GPIO pin, in compliance with
    ↪ UART/MP-Bus specifications
int send_byte(int place)

// this method is called using kthread_run to create a kthread,
    ↪ calls send_byte for each byte in the message
void send_text(void *data)

// ioctl method, called upon an ioctl on our char device
long device_ioctl(struct file *filep, unsigned int cmd, unsigned
    ↪ long arg)

// module entry point, called by insmod or during boot time
static int __init GPIObbModule_init(void)

// module exit point, called by rmmod
static void __exit GPIObbModule_exit(void)
```

Code 3.4: ioctl in kernel module

```
...
// magic number and command numbers for module & ioctl
#define MY_MACIG '2'
#define READ_IOCTL _IOR(MY_MACIG, 5, int)
#define WRITE_IOCTL _IOW(MY_MACIG, 6, int)
...
long device_ioctl(struct file *filep, unsigned int cmd, unsigned
    ↪ long arg) {
        long len = LEN;
        switch(cmd) {
        case READ_IOCTL:
                copy_to_user((char *)arg, buf, LEN);
                break;

        case WRITE_IOCTL:
                copy_from_user(buf, (char *)arg, len);
                break;

        default:
                printk(KERN_ERR "UARTbitbangModule: invalid ioctl
                    ↪ detected!\n");
                return -ENOTTY;
        }
        return len;

}

static struct file_operations fops = {
        .read = device_read,
        .write = device_write,
        .unlocked_ioctl = device_ioctl,
};
...
```

3.3.2 Bit Banging Functionality

Bit banging any GPIO pin while listening to another one in between bits was the next implementation step. Examples on how to do bit banging are widely available, however they needed to be adapted for this particular application. The bit banging was implemented as can be seen in Algorithm 1. The pins are written by direct memory access, i.e. writing out 1 to GPIO_SETDATAOUT or GPIO_CLEARDATAOUT of the respective GPIO pin, and read by directly reading GPIO_DATAIN for the input GPIO pin. All memory addresses can be looked up in the Technical Reference Manual for AM335x ARM Cortex-A8 Microprocessors [3].

Algorithm 1 Pseudo code for sending a byte using bit banging

```
if $RXpin \neq idle$ then
  break
end if
send startbit
sleep for bittime/2
if $RXpin \neq startbit$ then
  break
end if
sleep for bittime/2
while $currentbit < 8$ do
  if $char(currentbit) == 1$ then
    send 1
    sleep for bittime/2
    if $RXpin \neq 1$ then
      break
    end if
    sleep for bittime/2
  else
    send 0
    sleep for bittime/2
    if $RXpin \neq 0$ then
      break
    end if
    sleep for bittime/2
  end if
  current bit ++
end while
send stopbit
sleep for bittime
```

For the kernel module bit banging GPIO pins GPIO1_17 and GPIO3_19, i.e. pin 17 in GPIO group 1 and pin 19 in GPIO group 3, were used. As an example: To clear an output pin, the bit at the GPIO number at the GPIO_CLEARDATAOUT offset of the groups base address has to be set to 1. To clarify, memory addresses for the GPIO groups used are listed in table 3.4.

GPIO group	**Base address**
GPIO1	0x44E07000
GPIO3	0x481AE000
Register name	**Offset**
GPIO_DATAIN	0x138
GPIO_CLEARDATAOUT	0x190
GPIO_SETDATAOUT	0x194

Table 3.4: Memory addresses for GPIO groups on the Beaglebone Black

3.3.3 Pin Multiplexing

The AM335x processor family defines up to 8 modes per pin [5]. Switching between pin modes is done through pin multiplexing, for which the beaglebone provides a tool called cape manager in user space[6]. Changing modes in kernel space is not as straight forward though. Mode 7 puts any pin into GPIO mode if available. In kernel space pin multiplexing can theoretically be controlled through the pinctrl framework and GPIO pins can be enabled through the GPIO or GPIO legacy framework provided by the Linux kernel[7]. However, in practice there were difficulties with both methods. The GPIO framework does not manage to enable GPIO pins when active in another mode. The pinctrl framework has not been used due to its sheer complexity.

The kernel module implemented in this thesis changes pin modes by directly writing to the control module registers of the processor. This method is not foreseen to be used by the Linux kernel developers, does however allow for quick mode changes at runtime. Code 3.5 shows the implementation to change GPIO1_17 and GPIO3_19 pins to mode 7. Table 3.5 explains the possible values for the control module. After bit banging the message to be sent, the pin modes are restored to their previous value using a similar function. After first implementing the bit banging using separate GPIO pins, here UART1 pins were put into

[6]The Beaglebone Black can be customized through various capes that can be connected to the two expansion headers with 92 pins

[7]see consumer.txt and gpio-legacy.txt on `https://www.kernel.org/doc/Documentation/gpio/`, accessed 2016-01-06

GPIO mode. Changing their mode at runtime is important, as they are being used by the UART driver for all other MP-Bus communication. The test setup in Chapter 4 was developed using the GPIO implementation though, as it is more stable at this point.

Code 3.5: Changing pin modes to GPIO through the control module registers

```
...
// definitions for GPIO control through memory
#define CONTROL_MODULE_START    0x44E10000
    ↪         // CONTROL_MODULE starting address in memory
#define CONTROL_MODULE_END              0x44E11FFF
    ↪         // CONTROL_MODULE end address in memory
#define CONTROL_MODULE_SIZE     (CONTROL_MODULE_END -
    ↪ CONTROL_MODULE_START)
#define GPIO1_17_OFFSET                 0x844
    ↪         // control offset for GPIO1_17
#define GPIO3_19_OFFSET                 0x9a4
    ↪         // control offset for GPIO3_19
...
// set pin modes to GPIO for bit banging
int mux_pins(void){
        uint32_t *control_module;
        int value;

        if (!(control_module = ioremap(CONTROL_MODULE_START,
            ↪ CONTROL_MODULE_SIZE))) {
            printk(KERN_ERR "GPIObbModule: unable to map control
                ↪ module\n");
            return -1;
        }

        // set both GPIOs to mode 27: Fast, Enable Receiver,
            ↪ Pulldown type enabled, mux mode 7.
        value = 0x27;

        control_module[GPIO1_17_OFFSET >> 2] = value;
        control_module[GPIO3_19_OFFSET >> 2] = value;

        iounmap(control_module);

        return 0;
}
...
```

As the method used here is not recommended by the Linux kernel developers, it can still be improved upon. On the beaglebone accesssing the memory registers for GPIO modules 1 to 3 leads to a segmentation fault if none of these GPIO pins have been registered first. Changing the pin mode to GPIO in the control module of is not sufficient to work around that. At least one GPIO pin situated on these modules therefore needs to be set up through the GPIO framework.

Bit	Field	Description
31–7	reserved	
6	conf_<module>_<pin>_slewctrl	Slew Control. Slew Rate: Fast is 0, Slow is 1
5	conf_<module>_<pin>_rxactive	Receiver Active. Input Enable: Receiver Disable 0, Receiver Enable 1
4	conf_<module>_<pin>_putypesel	Pad Pullup/Pulldown Type. Pulldown is 0, Pullup is 1
3	conf_<module>_<pin>_puden	Pad Pullup/Pulldown enable. Enabled is 0, Disabled is 1
2–0	conf_<module>_<pin>_mmode	Mode. Pad functional mux select. A number between 0 and 7 i.e. 000 and 111. This depends on which mode we require.

Table 3.5: Controle module values for GPIO/UART offsets [24]

3.3.4 KThread

Finally, as a fourth step, the `send_text` and `send_byte` functions were to be called in a separate kernel thread for better schedulability, as these are the timing sensitive methods. The method `kthread_run` is a wrapper for `kthread_creat` followed by `wake_up_process` and creates and starts a kernel thread. It takes a function, a data pointer for the function and the name of the thread as arguments. `kthread_run` is called using `send_text` as a function and a pointer to the message to be sent. `send_text` then calls `send_byte` for each byte in the message.

3.4 Real-Time Scheduling

The preempt-RT patch provides the `sched_setscheduler` function to set the scheduling policy and assign a priority to the process identifier (PID) of process calling the it. Possible scheduling policies are listed in Table 3.6. The first three are regular Linux scheduling policies, the last two are real-time scheduling policies added by preempt-RT. As only one process needs to run with high priority at any given time in this implementation SCHED_FIFO is used here. The real-time scheduler assigns priorities of 50 and lower to non real-time tasks including all interrupts. Kernel 3.14 added an additional scheduling policy with SCHED_DEADLINE[8]. which has to be set using `sched_setattr` and is not covered here.

[8]see Linux manpage at `http://man7.org/linux/man-pages/man7/sched.7.html`, accessed 2016-01-06

Value	Policy
SCHED_OTHER	the standard round-robin time-sharing policy
SCHED_BATCH	for “batch” style execution of processes
SCHED_IDLE	for running very low priority background jobs
SCHED_FIFO	a first-in, first-out policy
SCHED_RR	a round-robin policy

Table 3.6: Scheduling policies for `sched_setscheduler`[9]

As the most time critical task, the kernel module is assigned a high priority. The priority is set via `param->sched_priority`. The `struct param` is then passed to `sched_setscheduler` alongside the PID and the selected scheduling policy. Code 3.6 shows the priority assignment.

Code 3.6: Implementation of real-time scheduling

```
...
/* define realtime priority, we use 90 as PRREMPT_RT uses 50 as
 * the priority of kernel tasklets and interrupt handler by default
    ↪  */
#define MY_PRIORITY (90)
...
// set our thread to high rt priority
struct task_struct *TSK;
struct sched_param PARAM;
TSK = current;
PARAM.sched_priority = MY_PRIORITY;
sched_setscheduler(TSK, SCHED_FIFO, &PARAM);
...
```

3.5 Outlook: BACnet MS/TP

Integrating BACnet MS/TP into the setup presented in this chapter should not pose a problem. As mentioned in the section on the BACnet protocol in the background chapter(see Section 2.3.2) the timing requirements for BACnet are less strict than the timing needed for dominance arbitration with the MP-Bus. A software BACnet control module could therefore be assigned a lower real-time priority, still allowing the MP-Bus bit banging module to execute reliably. Furthermore so far BACnet and the MP-Bus, while both implemented, have not been used simultaneously by Ergon and Belimo.

[9]see Linux manpage at `http://man7.org/linux/man-pages/man2/sched_setscheduler.2.html`, accessed 2016-01-06

Chapter 4

Evaluation

The implementation and the soft real-time constraints of the MP-Bus were evaluated through different measurements. The kernel module implementing the time-critical parts of the MP-Bus protocol was tested with a MP-Bus connection using a host PC as a simulated MP-Master as further described in Section 4.1. In this setup, execution times of the bit banging module were measured as well as its collision detection tested. Roundtrip time on MP-Master side was also measured, can however not be seen as representative due to the test setup. These measurements are presented in Subsection 4.2.1. The performance of the patched Linux was measured through latency under load using cyclictest from the rt-tests framework[1], the results of which are listed in Subsection 4.2.2.

4.1 Test Setup

The test setup can be seen in Figure 4.1 and Figure 4.2. For the test, a host PC running a adapted MP-Master test implementation in Java is connected via USB to the Belimo ZIP-USB-MP gateway, which translates all commands sent by the MP-Master to the MP-Bus. The gateway's MP-Bus signal and ground wires are connected to an Ergon MP-Bus hardware driver. The MP-Bus hardware driver is additionally connected to a 24V 100mA power supply unit providing the power as well as high and ground voltage levels for the MP-Bus. The hardware driver transforms the the MP-Bus voltage down to 3.3V to be connected to the Beaglebone Black. The MP-Bus hardware driver furthermore separates the single-wire MP-Bus signal to the RX and TX connections for the beaglebone. On the microcontroller a MP-Slave simulator implementation in C, adapted from an earlier Ergon program for this thesis, polls the UART1 port for incoming MP-Bus commands. If a MP_Get_SeriesNo command is successfully detected, a response with the series number is sent to the more stable GPIO implementation of the bit banging kernel module via ioctl. The kernel module

[1]The rt-tests framework can be found here: `http://git.kernel.org/cgit/linux/kernel/git/clrkwllms/rt-tests.git`

transmits the response using bit banging on the GPIO pin if the bus is idle and breaks in case of collision with the incoming message. On the beaglebone, a small shell script was written and executed to load the bit banging module into the kernel and enable all GPIO modules to avoid segmentation faults as described in the Pin Multiplexing Section 3.3.3.

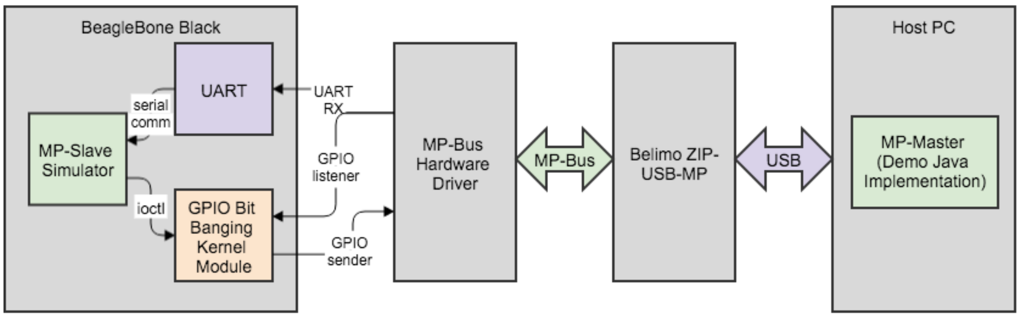


Figure 4.1: Overview over the setup used to test the implementation of the MP-Bus

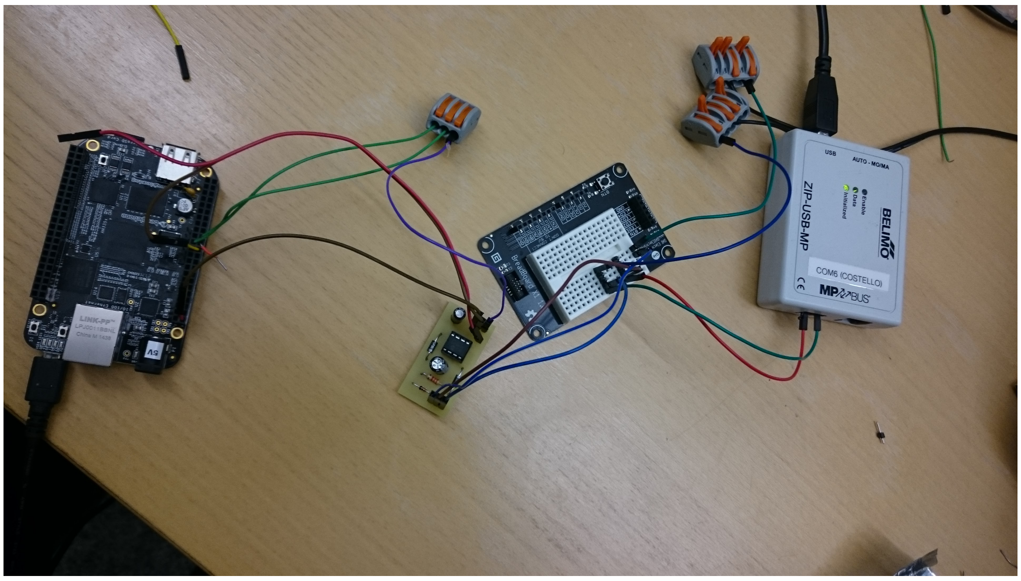


Figure 4.2: Picture of the test setup

4.2 Measurements

4.2.1 Byte and Roundtrip Times

The test setup described in the previous section is used to measure the timing of the bit banging module per byte, i.e. from right before setting the startbit to right after setting the stopbit within the kernel thread. The timing of the roughly 6000 bytes measured can be seen in Figure 4.3. The average duration per byte is 7.512ms, with the lower time bound for successful bit banging being 7.389ms[2] and the upper bound being 7.614ms[2]. The upper time bound, i.e. complying with the serial communication sending pattern, is violated in roughly 2% of all measurements. These violations are however clustered in five packets and correlated with collisions on the bus. In case of a collision, timing cannot and does not have to be guaranteed. The clusters have the additional effect, that the number of affected responses (i.e. packets of 10 bytes) is below those 2%, as often several bytes violating the upper time bound belong to the same response. Figure 4.4 shows a box-whiskers plot of the byte time measured. Again it can be seen that the most bytes are sent within a small timing band. The first quartile is at 7.502ms, the median at 7.507ms and the third quartile at 7.511ms.

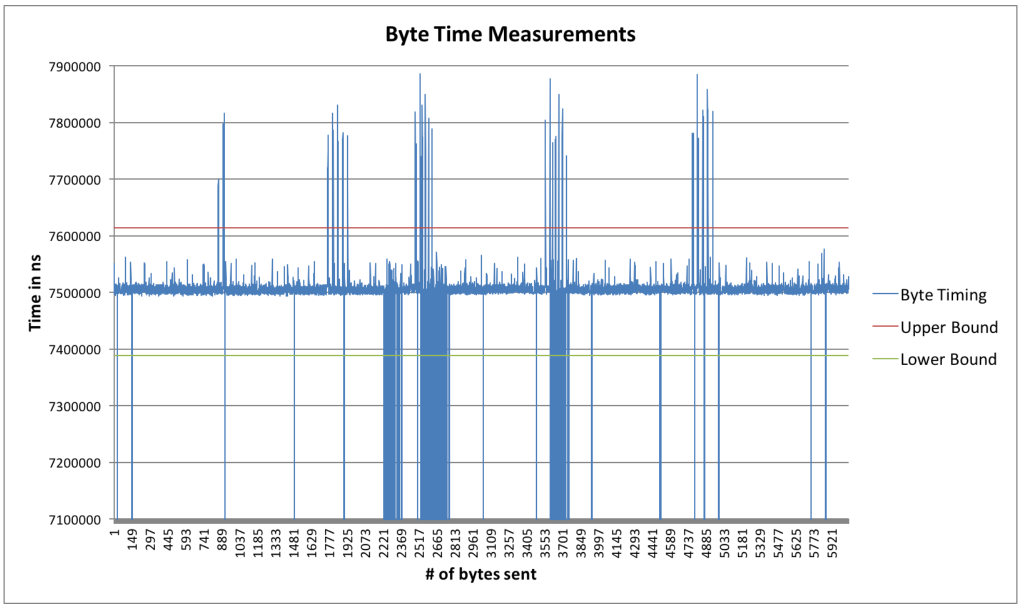


Figure 4.3: Byte sending time duration in the kernel module[3]

There are several possible explanations for these results and the high latency spikes in particular. The interrupt handling on kernel side could need more fine

[2]Nine times the min/max bit time as defined in Table 2.3. Nine, not ten, as the measurement is stopped right after setting the stopbit, thus not including its duration.

[3]Collisions were included as 0 values

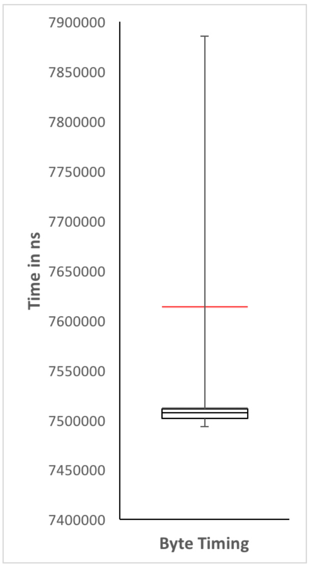


Figure 4.4: Box-whiskers plot of the byte time in the kernel module

tuning, although the bit banging kernel thread has a higher real-time priority than the interrupt handlers. Incoming messages could still lead to overhead an induce some latency. Another possible explanation comes from collisions between incoming messages and responses. In this case the MP-Slave will stop sending as soon as it detects a non-idle bus or bit mismatch according to dominance arbitration (see Section 2.3.1 for more details on dominance arbitration). The correlation between collisions and some high latency spikes supports the notion of collisions having an impact on latency. The `usleep` function, used in the kernel module to time the bit banging, may also be a source of unpredictability and additional latency.

The roundtrip time measured in the MP-Master implementation run on the host PC is depicted in Figure 4.5. It was measured from before calling the send method in the MP-Master to right after receiving and correctly identifying the response. The influence of the polling interval of the MP-Slave simulator can clearly be seen in the measurements. Roundtrip time is above the maximum answer delay[4] required by the MP-Bus. It was however not measured according to this requirement, as sending the command, transmission and processing of the response were within the time taken. The Java implementation and the host PC are furthermore not real-time optimized. The timing shows very little variation apart from the polling interval though, speaking in favour of a stable MP-Slave on the other side.

[4]See Table 2.3. The slave should start responding within this interval after receiving a command.

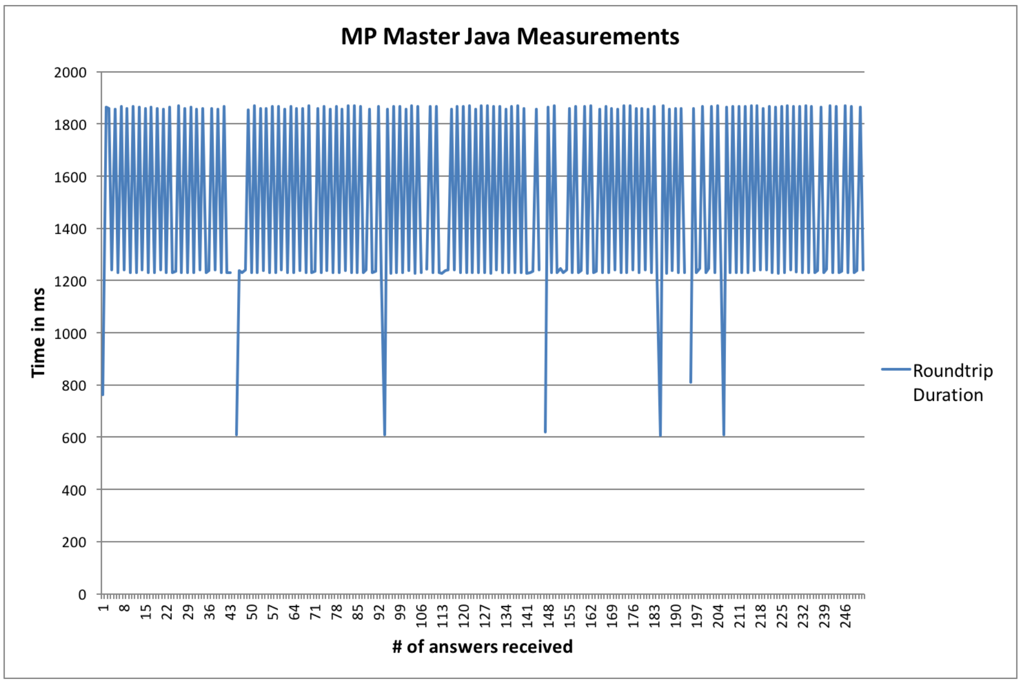


Figure 4.5: Roundtrip time measured in the MP-Master Java implementation run in eclipse[5]

4.2.2 System Latency

Latency measurements using cyclictest while running the Shared Logic framework on the Beaglebone Black produced the histogram showed in Figure 4.6. The cyclictest program, which is part of the rt-tests framework, was called with four threads, a real-time priority of 90, a thread interval of 1000μs and 100'000 iterations. The highest latency detected for threads 1–4 was 53 μs, 67 μs, 81 μs and 100 μs respectively. Another cyclictest with a similar setup, but detecting latency thresholds as shown in Code 4.1, needed 3 cycles to pass the threshold of 200 μs, 14 for 250 μs and 1575 for 300 μs with full load.

[5]Gaps show the end of a measurement block

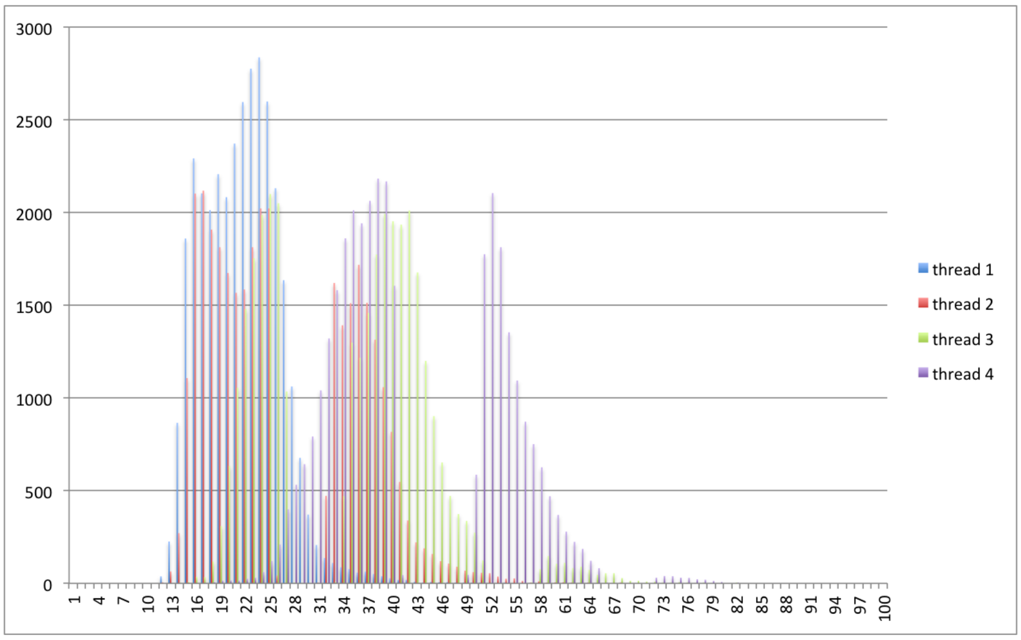


Figure 4.6: Histogram of latency under load in μs as measured by cyclictest

Code 4.1: Output by cyclictest (with 200, 250 and 300 μs max latency)

```
root@beaglebone:~/rt-tests# ./cyclictest --smp -p95 -f -b 200
# /dev/cpu_dma_latency set to 0us
INFO: debugfs mountpoint: /sys/kernel/debug/tracing/
policy: fifo: loadavg: 2.49 2.33 1.54 3/280 1785

T: 0 ( 1785) P:95 I:1000 C:      3 Min:    104 Act:  159 Avg:  140
    ↪ Max:     159
# Thread Ids: 01785
# Break thread: 1785
# Break value: 269

root@beaglebone:~/rt-tests# ./cyclictest --smp -p95 -f -b 250
# /dev/cpu_dma_latency set to 0us
INFO: debugfs mountpoint: /sys/kernel/debug/tracing/
policy: fifo: loadavg: 2.56 2.35 1.55 3/280 1789

T: 0 ( 1789) P:95 I:1000 C:     14 Min:    104 Act:  176 Avg:  131
    ↪ Max:     183
# Thread Ids: 01789
# Break thread: 1789
# Break value: 303

root@beaglebone:~/rt-tests# ./cyclictest --smp -p95 -f -b 300
# /dev/cpu_dma_latency set to 0us
INFO: debugfs mountpoint: /sys/kernel/debug/tracing/
policy: fifo: loadavg: 2.52 2.34 1.55 4/280 1792

T: 0 ( 1792) P:95 I:1000 C:   1575 Min:     81 Act:  137 Avg:  145
    ↪ Max:     261
# Thread Ids: 01792
# Break thread: 1792
# Break value: 304
```

Chapter 5

Conclusion and Future Work

A software based solution for the MP-Bus protocol based on Linux with the preempt-RT patch was possible. Replacing the dedicated hardware module for bus control with a kernel module is feasible using a real-time Linux framework. The preempt-RT patch adds the capability to reliably deal with real-time constraints and execute soft time-critical threads in kernel space. More sophisticated real-time frameworks for Linux offer hard real-time capability if additional adaptation efforts and, in some cases, reduced Linux functionality are not an issue.

The reliability of the implementation of the MP-Bus in this semester thesis can be further improved by replacing the `usleep` function used to time the bit banging with a more predictable method, optimising the byte time further to be closer to the lower bound with no load present on the system and additional fine tuning of the preempt-RT patched Linux. Separating application-level bus control into different non-Java modules with increased real-time priority would be advisable for better schedulability and reduced latency on the system. The pin multiplexing solution can also be further optimised, a better implementation in the Linux kernel is desirable.

All in all this semester thesis serves a successful proof of concept for the software solution sought for real-time field bus systems on Linux. This could allow for cost reductions in building automation as hardware requirements for control modules can be lowered.

Bibliography

[1] Martin Wild, Andreas Frey, *PP / MP Specifications*, Belimo, Hinwil CH, Rev. 26, November 5, 2010.

[2] Martin Wild, *MP Cooperation Documentation*, Belimo, Hinwil CH, Rev. 1.0, November 29, 2006.

[3] Texas Instruments, *AM335x ARM Cortex-A8 Microprocessors (MPUs) — Technical Reference Manual*, Texas Instruments, SPRUH73H, April 2013.

[4] BeagleBoard.org Foundation, *BeagleBone Black Schematics*, beagleboard.org, Doc.-Nr. 450-5500-001, March 21, 2014.

[5] Gerald Coley, *BeagleBone Black System Reference Manual*, beagleboard.org, Rev. C.1, May 22, 2014.

[6] American Society of Heating, Refrigerating and Air-Conditioning Engineers, Inc. (ASHRAE), *BACnet — A Data Communication Protocol for Building Automation and Control Networks*, AHSRAE, Atlanta, ISSN 1041-2336, 2008.

[7] Ergon Informatik AG, *Belimo Shared Logic — Software für ein besseres Klima*, 2013.

[8] Felipe Cerqueira, Björn B. Brandenburg, *A Comparison of Scheduling Latency in Linux, PREEMPT-RT, and LITMUS^{RT}*, Proceedings of the 9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2013), pp. 19–29, invited paper, July 2013.

[9] Carsten Emde, *Long-term monitoring of apparent latency in PREEMPT RT Linux realtime systems*, OSADL Project: Real Time Linux Workshops (RTLWS) 2012, October 2010.

[10] Carsten Emde, Robert Gleixner, *Quality assessment of real-time Linux*, boards & solutions / ECE, Reprint for OSADL, `https://www.osadl.org/uploads/media/ECE-2011-09.pdf`, Accessed: 2015-12-28, September 6, 2011.

[11] Linux Foundation, *The Linux Foundation Announces Project to Advance Real-Time Linux*, `http://www.linuxfoundation.org/news-media/announcements/2015/10/`

linux-foundation-announces-project-advance-real-time-linux, Accessed: 2016-01-02, October 5, 2015.

[12] RTwiki, *RT PREEMPT HOWTO*, https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO, Accessed: 2016-01-02, March 7, 2014.

[13] Ivan Cibrario Bertolotti, Gabriele Manduchi, *Real-time embedded systems: open-source operating systems perspective*, Boca Raton, Fl. : CRC Press, ISBN: 978-1-4398-4154-9, 2012.

[14] Daniel Bristot de Oliveira, Romulo Silva de Oliveira, *Timing analysis of the PREEMPT RT Linux kernel*, Software: Practice and Experience, DOI: 10.1002/spe.2333, May 25, 2015.

[15] Giorgio C. Buttazzo, *Hard RealTime Computing Systems — Predictable Scheduling Algorithms and Applications*, Pisa ITA, Springer Science and Business Media, ISBN 978-1-4614-0675-4, 2011.

[16] Hermann Härtig, Michael Roitzsch, *Ten years of research on L4-based real-time systems*, Proceedings of the 8th Real-Time Linux Workshop, 2006.

[17] XenWiki, *Xen Project Software Overview*, http://wiki.xen.org/wiki/Xen_Overview#Introduction_to_Xen_Architecture, Accessed: 2016-01-03, April 20, 2015.

[18] Arianna Avanzini, Paolo Valente, Dario Faggiolii, Paolo Gai, *Integrating Linux and the real-time ERIKA OS through the Xen hypervisor*, 10th IEEE International Symposium on Industrial Embedded Systems (SIES), 2015.

[19] Hasan Fayyad-Kazan, Luc Perneel, Martin Timmerman, *Linux PREEMPT-RT v2.6.33 versus v3.6.6: better or worse for real-time applications?*, ACM SIGBED Review — Special Issue on the 3rd Embedded Operating System Workshop (EWiLi 2013), Volume 11 Issue 1, p. 26-31, February 2014.

[20] Belimo AG, *A9-0001 — Introduction to MP-Bus Technology*, http://www.belimo.ch/pdf/e/MP_Technology_e.pdf, Accessed: 2015-12-27, v1.1, April 2010.

[21] H. Michael Newman, *BACnet — The Global Standard for Building Automation and Control Networks*, Momentum Press, New York, ISBN 978-1-60650-288-4, 2013.

[22] John Patrick, *Serial Protocols Compared*, Embedded Systems Programming, Vol. 15 No. 6, June 2012.

[23] Arianne Keller, *Kernel Space - User Space Interfaces*, http://people.ee.ethz.ch/~arkeller/linux/kernel_user_space_howto.html, Accessed 2015-11-20, Version 0.8, July 2008.

[24] Derek Molloy, *GPIOs on the Beaglebone Black using the Device Tree Overlays*, http://derekmolloy.ie/gpios-on-the-beaglebone-black-using-device-tree-overlays/, Accessed 2016-01-06.

[25] N.P. Mahalik, *Fieldbus technology: industrial network standards for real-time distributed control*, Springer, Berlin, ISBN: 3-540-40183-0, 2003.

[26] Hermann Merz, Thomas Hansemann, Christof H¸bner, *Building automation : communication systems with EIB/KNX, LON und BACnet*, Springer, Berlin, ISBN: 978-3-540-88828-4, 2009.