A Time-Triggered Architecture For an Attitude Control System With Space Technology

Jan Bredereke, Julian Greilich, Benjamin Hesseln, Jan Lehrke, Nils Müller, Jonas Pufahl, Jens Sager, Markus Salomon, Benjamin Schäfer, Tobias Schmitz, Maximilian Schönenberg, Nikolas Schreck, Peter Tschubij, and Mirco Wittrien

City University of Applied Sciences Bremen
Flughafenallee 10, D-28199 Bremen

http://homepages.hs-bremen.de/~jbredereke

Rico Thiele

Airbus Defence & Space
Airbus-Allee 1, D-28199 Bremen

Feb. 2017

The following pages were generated from the project wiki hosted by Airbus.

AIRBUS
DEFENCE & SPACE

Configuration v2 - A Time-Triggered Architecture For an Attitude Control System With Space Technology

Rico Thiele

Creation Date:	15-Sep-2016 09:03
Modification Date:	27-Jan-2017 13:28
Version:	18

Table of Contents

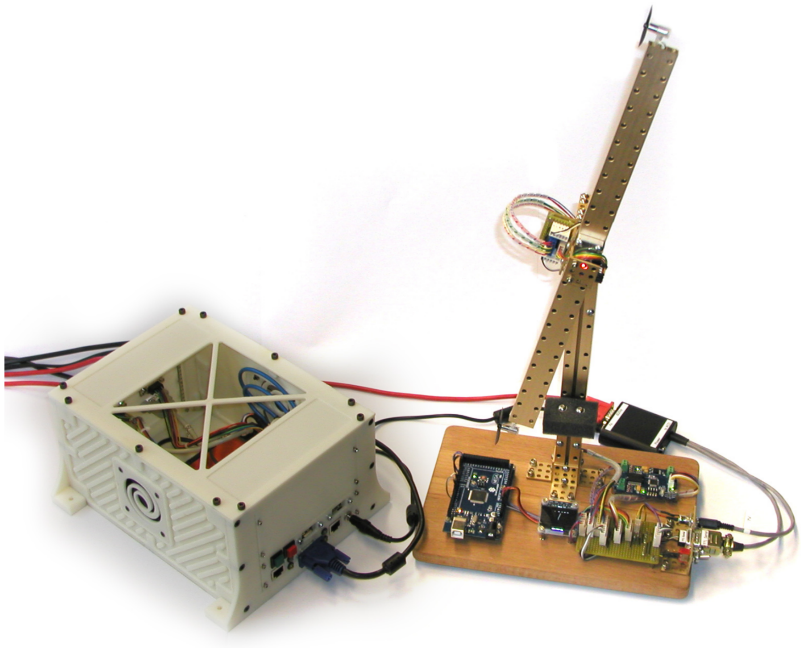
1 Introduction

Prof. Dr. Jan Bredereke, City University of Applied Sciences Bremen

This project demonstrates how to implement a time-triggered architecture for a real-time and distributed embedded system. The application is a simple attitude control system. The project was carried out as part of the course "Embedded Systems" at the City University of Applied Sciences Bremen, during the winter term 2016/17. It was supervised by Prof. Dr. Jan Bredereke. The chapters after the introduction were written by the students named there. The project was conducted in cooperation with Airbus Bremen.

1.1 The Attitude Control System Demonstrator

The demonstrator consists of an arm with a single degree of freedom, see figure. The arm is kept in a defined attitude by two propellers. The control algorithm for this is executed on a BeagleBone Black microcontroller running the QNX operating system. Originally, an e.Cube computer suitable for space missions was envisioned for this task. However, a missing board support package for the current version of the QNX operating system made us switch to the BeagleBone Black. The actuators and the sensor are controlled by a Sunfounder Mega microcontroller (Arduino-compatible). Both microcontrollers communicate over an RS-232 serial connection using a time-triggered communication protocol.



1.2 The Time-Triggered Architecture

1.2.1 Time-Triggered Processing for Hard Real-Time Systems

Using a time-triggered processing scheme helps to meet hard real-time constraints. Every task has a fixed time slot in the schedule, with a fixed period and a fixed length. This allows to prove that a system constructed in this way will meet its timing requirements under all circumstances. Such a proof can be done by checking that the worst-case execution time of every task in isolation does not exceed the length of its alotted time slot. Such a system does not use any interrupt mechanism. Each task runs to completion. The scheduling therefore is cooperative. If the worst-case execution times of all tasks have been proven to not exceed their deadlines, such a design is highly reliable.

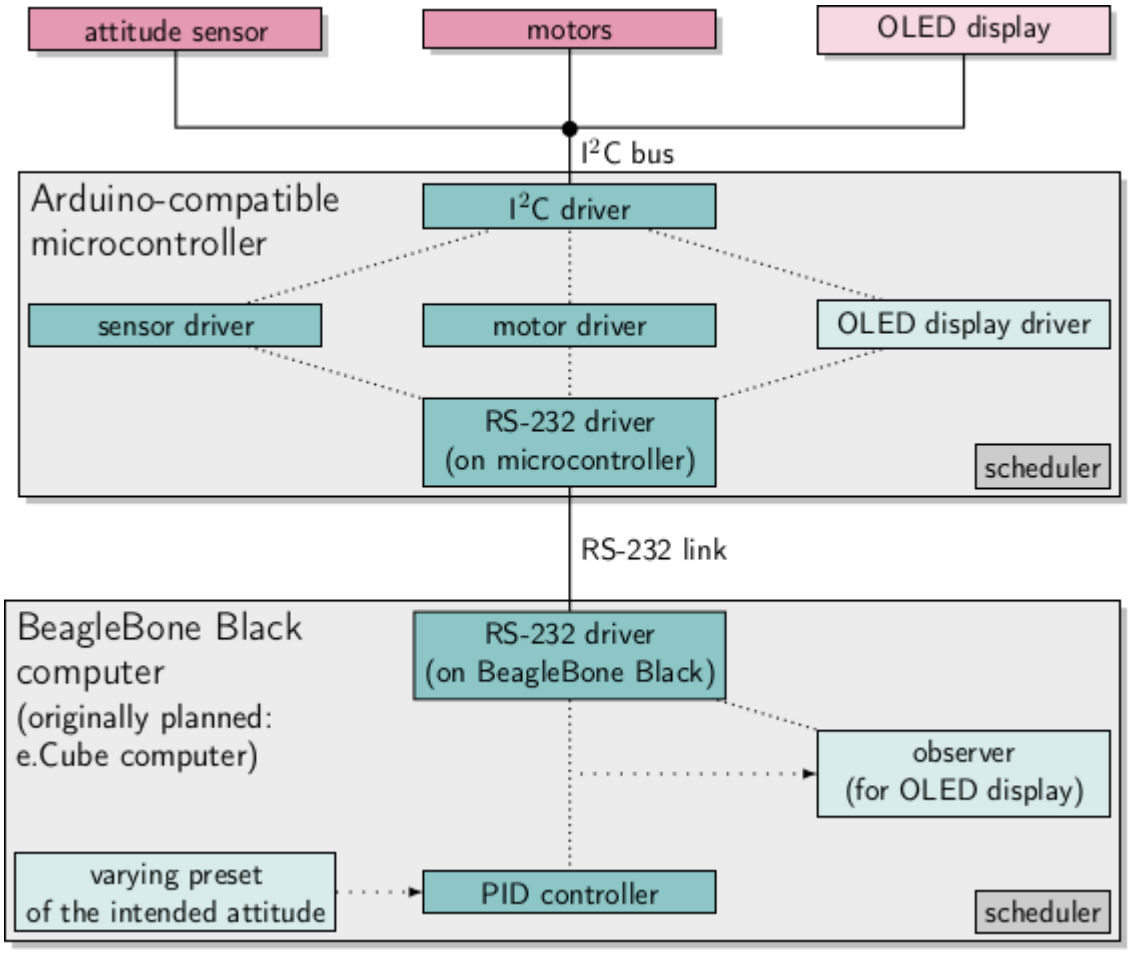
An alternative scheme would be event-triggered processing. Such a scheme uses interrupts and priorities for tasks. At any time, a higher-priority task may interrupt the execution of a lower-priority task. This allows for a quick response to an urgent issue. Also, the average response time of tasks in such a scheme often is considerably shorter than when using a time-triggered scheme. However, an interrupting task may be interrupted itself, and interrupts may be postponed for a long time or even lost due to other high-priority processing. Therefore, usually it is practically impossible to provide a proof that a specific task will meet a specific deadline in the worst case. Consequently, an event-triggered processing scheme usually is not suitable for a hard real-time system. Hard real-time system here means that guarantees for its reliability must be provided.

Please note that the documentation of the QNX real-time operating system systematically uses a substantially different meaning for the words "real-time". It uses them in the sense of "as fast as possible", and not in the sense of "guaranteed fast enough".

1.2.2 The Time-Triggered Architecture for the Attitude Control System Demonstrator

We use a distributed time-triggered architecture for our attitude control system demonstrator. There are two processing nodes, and they synchronize und communicate using an RS-232 serial interface connection. The BeagleBone Black microcontroller is the master. It generates the time ticks which determine the schedule of all tasks on all nodes. The Sunfounder Mega microcontroller is a slave and follows the time ticks. The communication schedule is defined by these regular time ticks, too. We defined a simple time-triggered communication protocol for this. The details follow in the sections below.

The following figure shows the system structure. Light blue and light red boxes denote optional components, to be realized only if time permits.



1.3 Original Project Description (in German)

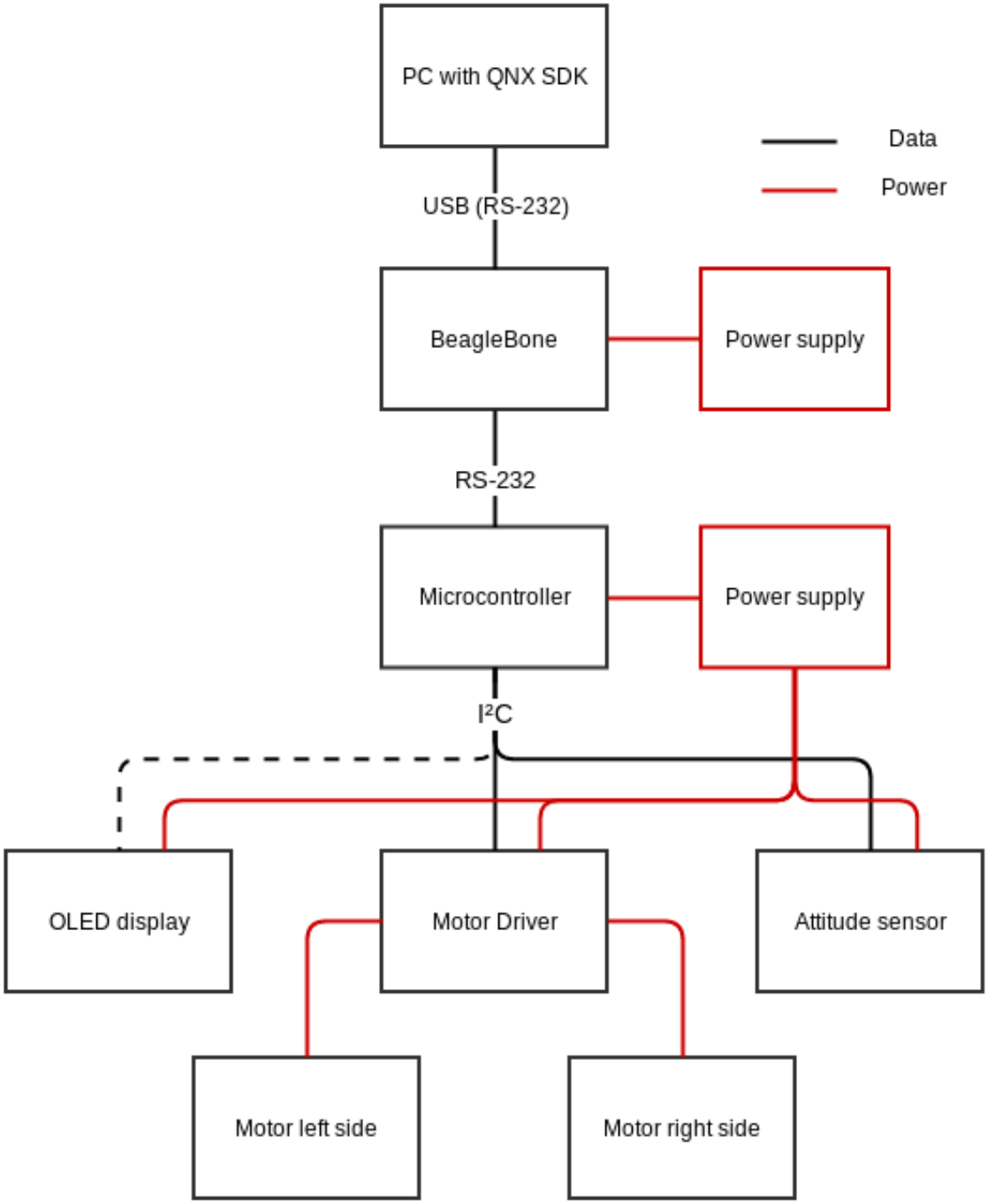
Projektbeschreibung-2016-09-23.pdf

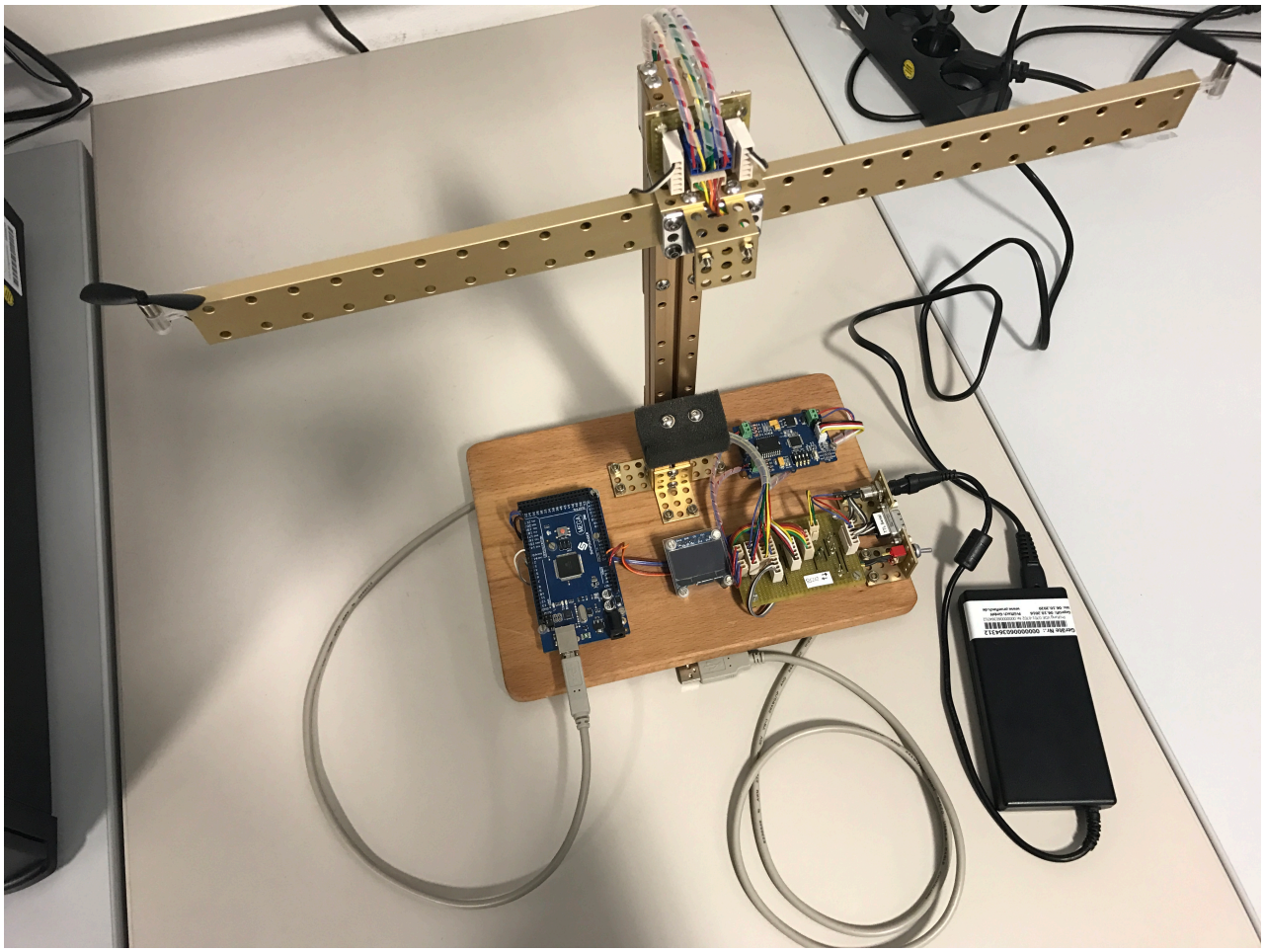
2 Hardware Architecture

Authors: Jan Lehrke, Jonas Pufahl

Related hardware specifications: Electronics

2.1 Overview


PC with QNX SDK
Data
Power
USB (RS-232)
BeagleBone
Power supply
RS-232
Microcontroller
Power supply
I²C
OLED display
Motor Driver
Attitude sensor
Motor left side
Motor right side



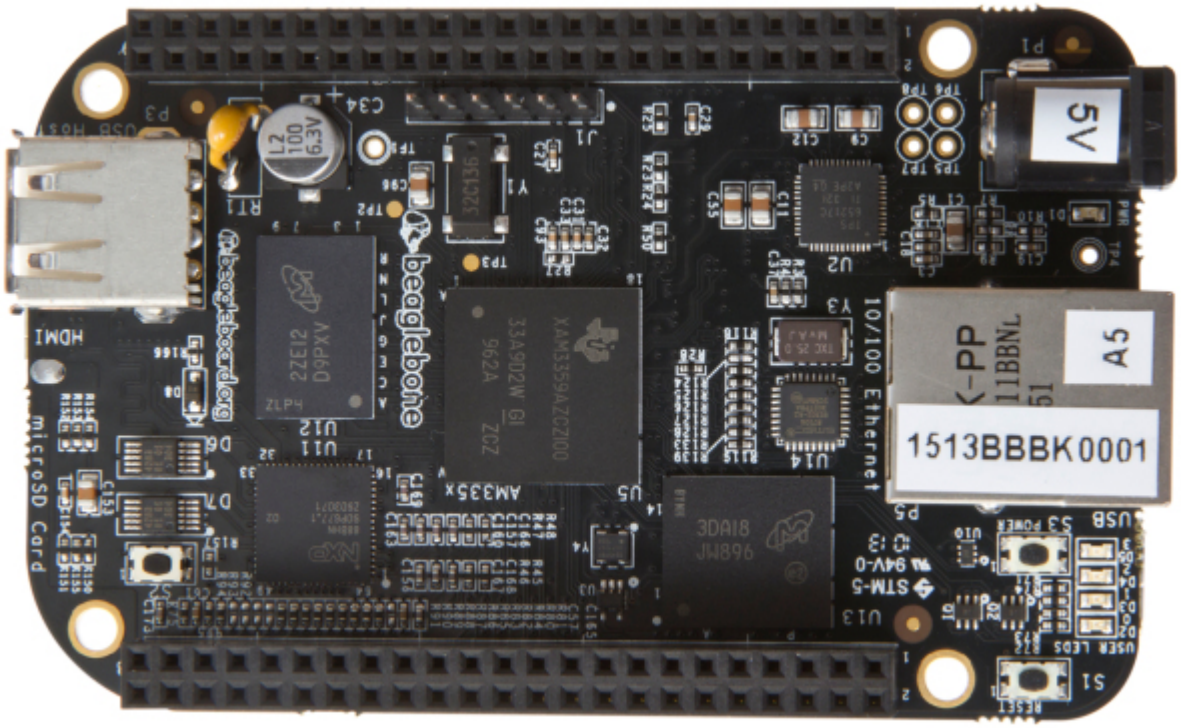
Assembly picture of all devices attached to I²C bus

2.2 Components

2.2.1 BeagleBone Black

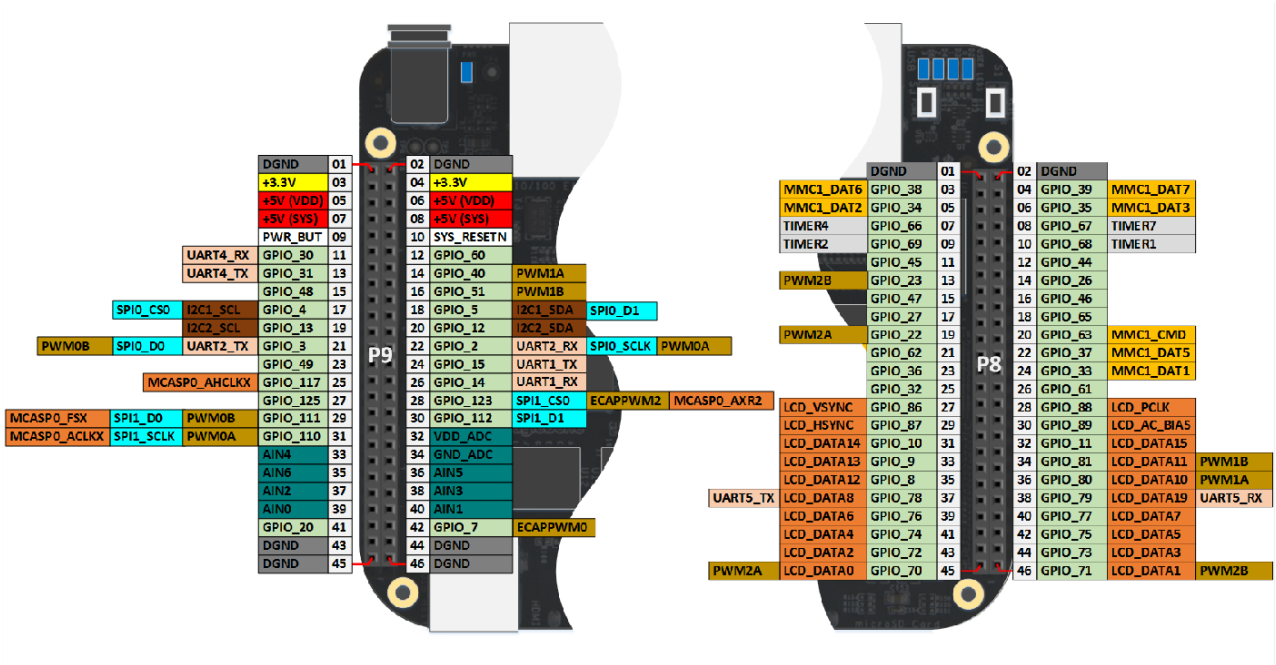
Name: BeagleBone Black

Data sheet: http://elinux.org/Beagleboard:BeagleBoneBlack



PINs

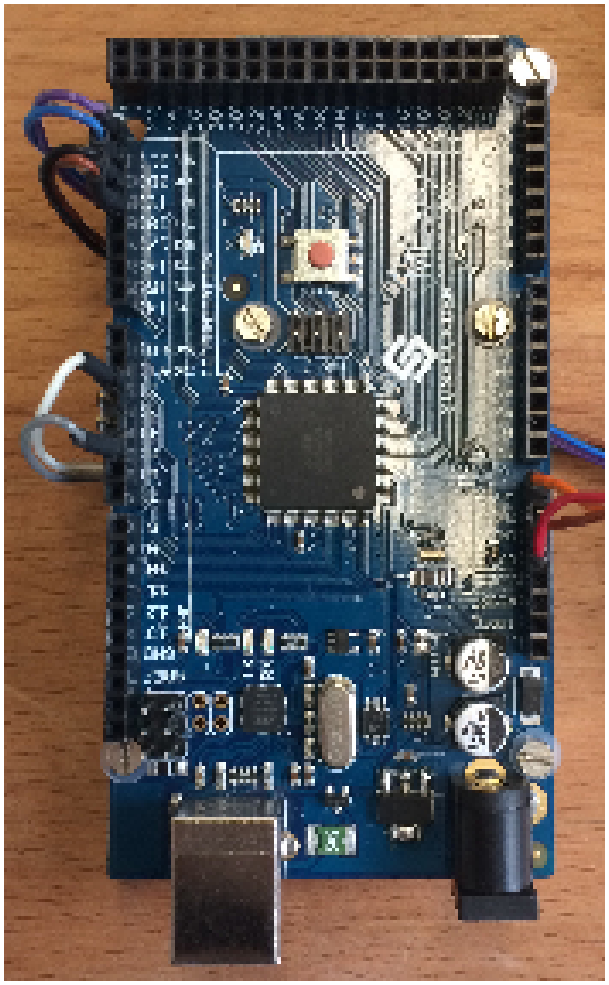
the following graphic shows the BeagleBone Black Serial Port Mapping



2.2.2 Microcontroller

Name: Microcontroller Sunfounder Mega 2560

Data sheet: http://www.robotshop.com/media/files/PDF/ArduinoMega2560Datasheet.pdf



2.2.3 Attitude sensor

Name: Invensense MPU-6050

Data sheet: https://www.invensense.com/products/motion-tracking/6-axis/mpu-6050/

The sensor provides the data in several registers which can be read by calling them seperately or pulling data frequently from the sensor so the sensor increments the register number by himself. Have a look at the document section of the website to get a description of the register mapping.

2.2.4 Motor

Motors

There are two equal motors (left and right). Both are controlled by one motor driver

Name: DC Motor Crazyflie Nano Quadcopter

Data sheet: http://www.watterott.com/de/Crazyflie-Nano-Quadcopter-6x15-mm-spare-motor-BC-CM-01-A

Motor driver

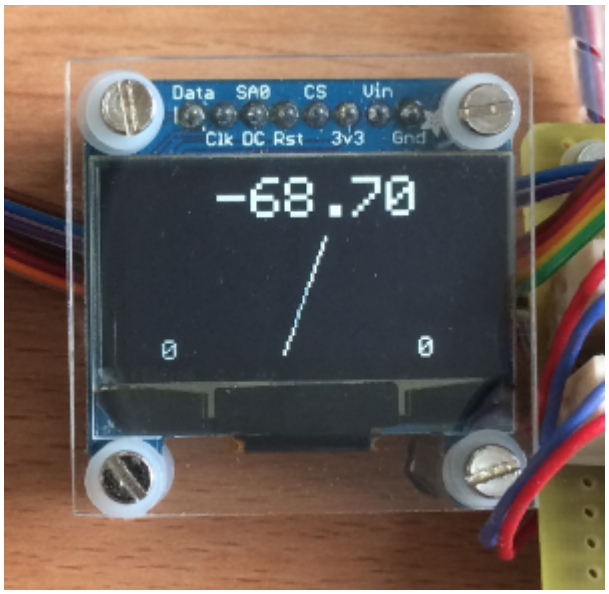
Name: Grove I^{2}C Motor Driver

Data sheet: http://wiki.seeed.cc/Grove-I2C_Motor_Driver_V1.3/

2.2.5 OLED-Display

Name: Adafruit Monochrome 1.3" 128x64 OLED graphic display

Data sheet: https://www.adafruit.com/product/938



3 Previous software versions

Authors: Jens Sager, Benjamin Schäfer

There are three previous software versions. Two of these versions can be found in the git repository on branch "config_v1_Control_loop_and_user_frontend". The arduino code can be found in the directory "ACD Arduino Code v1.0/sketches/eCubeTranslatorDisplay" and "ACD Arduino Code v2.0/Arduino Code/eCubeTranslatorDisplay". The third version "Echtzeitnachweis" can be accessed in AULIS (if you have no access to AULIS ask Jan Bredereke or Rico Thiele for the code).

The "Echtzeitnachweis" is a restructured version of the v1 code. It applies the design pattern for reactive real time systems. The v2 code is a continuation of v1.

The existing software is not suitable to implement a distributed real time system. Mainly because process real time properties have been neglected. Also the serial communication is unsuitable. It has to be examined if reading the sensor values can be done in a way more suitable for real time systems while keeping the sensor fusion.

In summary: technical concepts of the previous versions can be employed. A new structure should be created.

4 PID Controller

Authors: Jens Sager, Benjamin Schäfer

4.1 Structure

A PID controllers main purpose is, as with any other controller, to monitor and if necessary alter the operating conditions of a given dynamical system. [Wikipedia] A PID controller consists of a proportional, integral and a derivative term. It seeks to match the measured process variable to a desired setpoint. Each of the PID terms has a tuning parameter (proportional gain, integral gain, derivative gain), usually called $\mathbf{K_P}$, $\mathbf{K_I}$, $\mathbf{K_D}$.
The controller takes the current error value **e(t)**, that is the difference between the setpoint **SP** and the current process variable **PV(t)**, and uses it to compute each term which is again multiplied with its respective tuning parameter. The controllers output is the sum of the three resulting terms.

$$e(t) = SP - PV(t)$$

$$u(t) = K_P e(t) + K_I \int_0^t e(x)\,\mathrm{d}x + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t}$$

4.1.1 Proportional term (P)

$$P = K_P e(t)$$

The proportional term **P** produces an output that is proportional to the current error value, thus increasing the controllers output with increasing error. It makes up the main part of the control algorithm. A P-only controller is a sufficient controller in many cases. [Pont2001]

4.1.2 Integral term (I)

$$I = K_I \int_0^t e(x)\,\mathrm{d}x$$

The integral term **I** grows with both the magnitude of the error and its duration. This term can accelerate the movement of the process towards the setpoint if an error is not corrected over longer time periods. Because it accumulates the errors of the past it can result in overshooting the setpoint value. To mitigate this effect for certain circumstances windup protection can be used.

4.1.3 Derivative term (D)

$$D = K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t}$$

The derivative of the error is used to determine the slope of the error over time. This can be used to predict system behaviour and improves settling time as well as stability of the system.

4.2 Implementation

4.2.1 Basic algorithm

The basic algorithm as given by http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/ . A similar version can be found in [Pont2001]

```
/*working variables*/
unsigned long lastTime;
double Input, Output, Setpoint;
double errSum, lastErr;
double kp, ki, kd;

void Compute()
{
  /*How long since we last calculated*/
  unsigned long now = millis();
  double timeChange = (double)(now - lastTime);

  /*Compute all the working error variables*/
```

```
  double error = Setpoint - Input;
  errSum += (error * timeChange);
  double dErr = (error - lastErr) / timeChange;

  /*Compute PID Output*/
  Output = kp * error + ki * errSum + kd * dErr;

  /*Remember some variables for next time*/
  lastErr = error; lastTime = now;
}

void SetTunings(double Kp, double Ki, double Kd)
{
  kp = Kp; ki = Ki; kd = Kd;

}
```

4.2.2 Possible improvements

In the following possible improvements to the PID algorithm are listed. Most of these are taken from the arduino PID library http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction. Others are suggested by Pont. To see which of these improvements were implemented in the system please consult the code.

Windup reduction

In the basic approach the error is summed up for the integral term every time the algorithm is called. If the actuator is operating at its maximum or minimum limit changing the value of the sum serves no purpose because the actuator value can not be changed. If the summation continues it results in slowed system response. Hence Pont suggests stopping the summation when the output is at its limits.

Sample time

In order to get consistent behaviour from the PID it should be called in regular intervals. This also results in a simplification of the derivative and integral calculations.

Derivative kick

This steps goal is to eliminate spikes in output if the setpoint is changed which in turn results in an "instant" change in the error value and a spike for the derivative.
This change uses the fact that:

$$\frac{\mathrm{d}e(t)}{\mathrm{d}t} = -\frac{\mathrm{d}input(t)}{\mathrm{d}t}$$

This holds true when the setpoint is constant. Switching to observing the difference in input values instead of error changes smooths out the derivative kick.

Changing parameters on the fly

In the basic form sudden changes in the tuning parameters would lead to undesirable spikes or bumps in the output value because of the cumulated error sums. One way to deal with this is to rescale the current error sum based on the new $\mathbf{K_I}$ parameter. Additionally multiplying each sum term with the error individually instead of the cumulative sum smooths out the bump.

Source and sample code: http://brettbeauregard.com/blog/2011/04/improving-the-beginner%e2%80%99s-pid-tuning-changes/

4.2.3 Choosing the controller parameters

Pont suggests the following 6 steps to finding proper tuning parameters:

1. Set the integral and differential terms to 0
2. Increase the proportional term slowly, until you get continuous oscillations
3. Reduce $\mathbf{K_P}$ to half the value determined.
4. If necessary, experiment with small values of $\mathbf{K_D}$ to damp-out ‘ringing’ in the response
5. If necessary, experiment with small values of $\mathbf{K_I}$ to reduce the steady-state error in the system
6. Always use windup protection if using a non-zero $\mathbf{K_I}$ value

4.3 Sources

[Pont2001] : http://www.safetty.net/download/pont_pttes_2001.pdf

[Wikipedia] : https://en.wikipedia.org/wiki/PID_controller

5 Arduino IDE

The Arduino homepage offers a step-by-step guide on how to install the Arduino-IDE. It also offers an overview of a number of Arduino microcontrollers on the right side.

https://www.arduino.cc/en/Guide/HomePage

We are using for the Angle Control Demonstrator an Arduino MEGA2560. The link below offers a guide on how to start the IDE and how to create an image by writting a sketch.

https://www.arduino.cc/en/Guide/ArduinoMega2560

As a demo program we combined the two code samples down below to display the angle of the balanced beam with the OLED display.

https://pforge.eso-io.com/git/acd/arduino_demo/Arduino_demo_OLED_MPU6500.ino

Down below are some code samples for the hardware we are using:

Code sample for monochrome 1.3" 128x64 OLED graphic display:

https://github.com/adafruit/Adafruit_SSD1306

Code sample for MPU-6050 Accelerometer + Gyro:

http://playground.arduino.cc/Main/MPU-6050

Autoren:

NIls Müller, Tobias Schmitz, Mirco Wittrien

6 Time-Triggered Scheduling

Found a Whitepaper to the topic scheduling on QNX: The Joy of Scheduling

6.1 Cooperative Scheduling

Using cooperative scheduling each process will work uninterrupted until it finishes its task and meets its natural end. It requires every process to fulfill its real-time conditions, so that the CPU can handle other processes in time. For this purpose it is advantageous for every process to not use up too much time.

The opposite of cooperative scheduling is preemptive scheduling. Using preemptive scheduling, processes can be interrupted by the operating system, so that multiple processes can be executed concurrently. A priority control is needed for preemptive scheduling.

An advantage of cooperative scheduling is the low expense for implementation and not needing to implement the case of interrupted processes.

On the Arduino and the BeagleBone we use cooperative scheduling, even though QNX (the operating system of the BeagleBone) is intended to be used with preemptive scheduling. The reason for this decision is the easier validation of real-time conditions, even if it leads to us ignoring the inherent interrupt-features of QNX.

Extensive explanation of cooperative scheduling:
http://www.safetty.net/download/pont_pttes_2001.pdf (p. 246f)

Example code of cooperative scheduling with function pointers:
http://www.safetty.net/download/pont_pttes_2001.pdf (p. 256ff)

Example code of splitting tasks into simpler subtasks:
http://www.safetty.net/download/pont_pttes_2001.pdf (p. 316ff)

Authors:

Greilich, Salomon, Schreck, Tschubij

6.2 Multiprocessor Systems

A multiprocessor system is a system composed of multiple processors. Our system houses one master processor (the BeagleBone Black) and one slave (the Sunfounder Mega 2560, Arduino). Regarding those systems there are three main requirements:

1. clock synchronisation
2. data transfer (between processors)
3. error handling

6.2.1 Clock Synchronisation

Even if multiple processors have the same clock, the clock tolerances might lead to variations in execution times. To ensure synchronous executions clock synchronisation is required. A possible solution for this is the master sending periodical tick messages.

6.2.2 Data Transfer

The slaves respond to every tick message with an acknowledge message. Both message types can have appended payload data. No additional messages are permitted.

These facts allow the bandwidth to be predetermined, ensuring all messages are being delivered on time.

6.2.3 Error Handling

A slave can detect errors by measuring the timespan between two tick messages. When an error is detected the slave will fall into a safe state and will wait for another start sequence.

A master does error handling if an acknowledge message is missing: It stops sending tick-messages and therefore puts the slaves into a safe state and then shuts down; restarts the network by restarting itself; or engages a backup slave.

Drafted solutions for Clock Synchronisation:
http://www.safetty.net/download/pont_pttes_2001.pdf (p. 555ff)

Drafted solutions for Data Transfer:
http://www.safetty.net/download/pont_pttes_2001.pdf (p. 583ff)

Drafted solutions for Error Handling:
http://www.safetty.net/download/pont_pttes_2001.pdf (p. 596ff)

The above mentioned solutions save tasks in a query. The execution of the tasks is done in the main()-method.

An example of such a main()-method is here:

http://www.safetty.net/download/pont_pttes_2001.pdf (p. 259 and 267)

Authors:
Greilich, Salomon, Schreck, Tschubij

6.3 Data Flow Analysis

6.3.1 RS232 between Master and Slave

The master needs the following data from the slave:

- Rotation-Sensor-Value (2 Byte)

The slave need the following data from the master:

- Data to display on the OLED
  - Actual Rotation Value (2 Byte)
  - Target Rotation Value (2 Byte)
- PID-Value to control the motordrivers (1 Byte)

In the current protocol version, each tick-message from the master has be answered by an ack-message from the slave and each message consists of exactly 1 Byte (see Cooperative Scheduling, RS232 Communication Protocol.).

Because the PID-Value is more important, it is transfered more often (every second tick) than the other values for the slave.

Therefore it takes eight tick-messages to transfer all Data.

The details are described in RS232 Communication Protocol.

6.3.2 I²C between Slave, Driver of the motors, Driver of the OLED-Display and Sensor

To set a new motor value, the following data has to be send over I²C:

- I²C-device-address (1 Byte) (in "write-mode", see I²C-Standard for more details)
- data-register on the I²C-Device to write to (1 Byte)
- value to set for each motor (2 Bytes)

To read a new sensor-value from the rotation-sensor, the following data has to be send over I²C:

- I²C-device-address (1 Byte) (in "write-mode", see I²C-Standard for more details)
- data-register on the I²C-Device to read from (1 Byte)
- I²C-device-address (1 Byte) (in "read-mode", see I²C-Standard for more details)
- sensor-data (4 Bytes)

Because the driver for the OLED-Display is not implemented yet, it is unknown how much data is needed to display the values.

Author: Greilich

6.4 RS232 Communication Protocol

Communication between the BeagleBone and the Arduino is based on RS232 using TTL levels. Because of the multi processor system a time triggered protocol is needed. The protocol has to implement features listed on Multiprocessor Systems.

If the master would send all data in every tick-message, the slave-processor would be blocked for the whole receiving (at 9600 Bd, 8N1, 6 Byte Data: 10 ms).

To improve this, every Tick- and every Ack-Message consists of only a single byte, so that the hardware of the microcontroller can offload the receiving-process and transmission-process. (The Hardware generates and interrupt, when a byte is received. The slave only has to copy the received byte and from the receive-buffer and write the Ack-Message into the send-buffer.)

6.4.1 Message-Types

The following Message-Types exist. The order, in which these Messages are sent is described in below (Sequence).

Message-Type	Master (Tick)	Slave (Ack)
0	START	START_ACK
1	REQ_ROT_HIGH	ROT_HIGH
2	REQ_ROT_LOW	ROT_LOW
3	PID	ROT_HIGH
4	AV_ROT_HIGH	ROT_LOW
5	AV_ROT_LOW	ROT_LOW
6	TV_ROT_HIGH	ROT_LOW
7	TV_ROT_LOW	ROT_LOW

Message-Bytes

Message-Byte	Description
START	0xAB (magic value to be recognized by the slave)
REQ_ROT_HIGH	0x48 (magic value to request the first ROT_HIGH)

Message-Byte	Description
REQ_ROT_LOW	0x4C (magic value to request the first ROT_LOW)
PID	PID regulation output value
AV_ROT_HIGH	High-Byte of ACTUAL VALUE rotation for OLED[1]
AV_ROT_LOW	Low-Byte of ACTUAL VALUE rotation for OLED[1]
TV_ROT_HIGH	High-Byte of TARGET VALUE rotation for OLED[1]
TV_ROT_LOW	Low-Byte of TARGET VALUE rotation for OLED[1]
START_ACK	0xCD (magic value to be recognized by the master)
ROT_HIGH	High-Byte of Rotation sensor value
ROT_LOW	Low-Byte of Rotation sensor value

1) Values are mapped as following: 0 -> -180.00° ... 2^{16} -> +180.00°

6.4.2 Sequence

The Master repeatedly sends the start-byte START, until the Slave acknowledges it with START_ACK.

The first time the Master receives the START_ACK from the Slave, it will send REQ_ROT_HIGH (expecting the ROT_HIGH byte) and REQ_ROT_LOW (expecting the ROT_LOW byte). This is needed, because without the first sensor values, the Slave would not be able to properly calculate the PID-values. After this sequence the Master will send another START byte, after which he should get the START_ACK from the Slave. This means the first message-sequence should be:

0 -> 1 -> 2 -> 0.

After this first exchange of the sensor-values and the two start-handshakes, the following message-sequence will be endlessly repeated:

3 -> 4 -> 3 -> 5 -> 3 -> 6 -> 3 -> 7 -> ...

6.4.3 Error-Handling

If the master gets no acknowledge-message (within a time-slot, which has to be defined!), the master repeatedly sends the start-byte START, until the Slave acknowledges it with START_ACK.

If the slave get no tick-message (within a time-slot, which has to be defined!), the slave enters a safe state and waits for the start-byte START.

Authors:
Greilich, Salomon, Schreck, Tschubij

7 Arduino and Time-Triggered Drivers

7.1 Interface drivers

Authors: Jonas Pufahl, Jan Lehrke

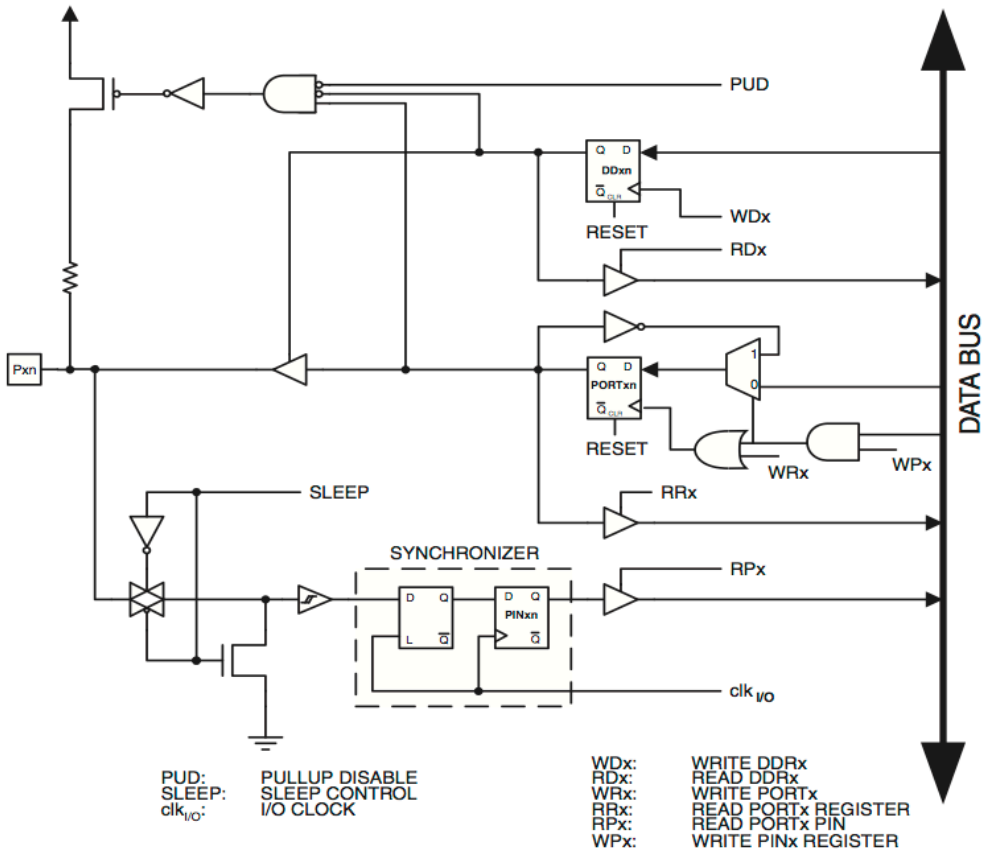
7.1.1 I2C

For our real-time requirements it is not possible to use the Wire Arduino library because it is not built for real-time use. It cannot provide a schedulable execution time and it is using interrupts.

We created a I2C library on our own and we worked on two versions. The first try/version failed due to limitations of the hardware. During the development and validation of the possible implementations of i2c the signals produced by the library were measured with the LogicPort logic analyzer.

First Version

In the first version the target was to create a I2C communication file to do a manual I2C connection with the general IO ports of the controller. For this approach the I2C library written by Prof. Dr. Jan Bredereke for the C515C microcontroller was ported to be used on the Mega 2560. During this approach an hardware limitation error was detected. The ATmega2560 is not able to to switch from weak 1 (Pull-Up signal) to a zero signal. This feature is mandatory to run I2C. The following graphic shows the structure of an IO port of the Mega 2560:



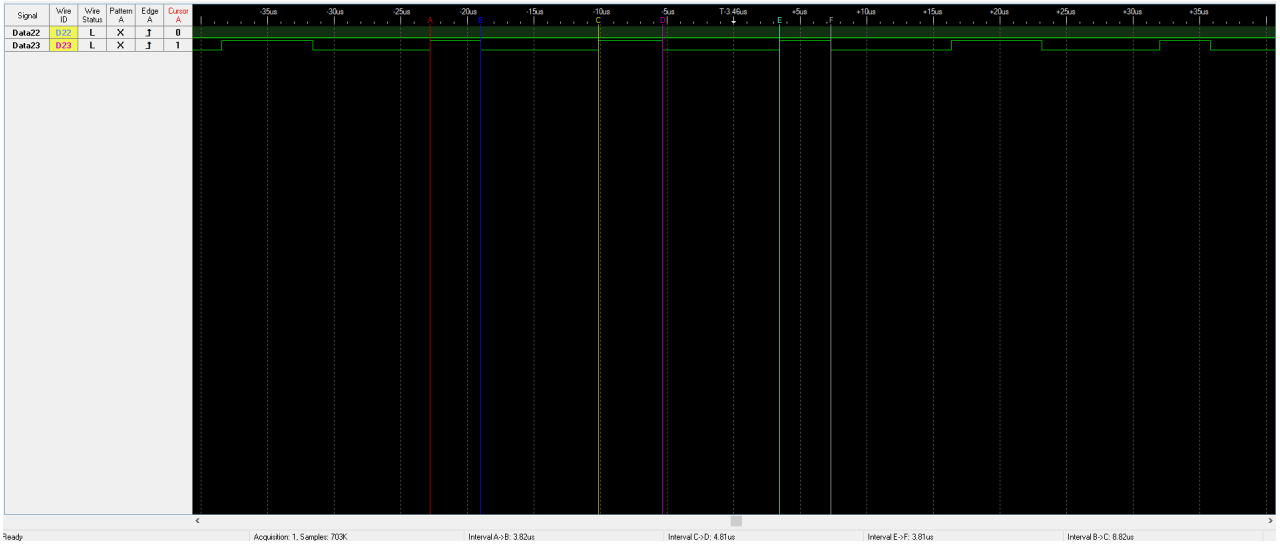
the used code is placed in the git repository in the folder interfaces/i2cBib/arduino_i2c_v1/.

It was necessary to do a timing analysis due to used and by the Mega 2560 not supported `nop()`-calls of the given library. For this reason the built-in Arduino function `delayMicroseconds()` was used to achieve the nearly the same delays in `i2cWait.cpp` as expected by the original library. The following code was used to produce some signals in order to start some measurements via LogicPort.

Code for timing analysis

```
/* Code from main loop() */
PORTE = 32;
delayMicroseconds(8);
PORTE = 0;
delayMicroseconds(10);
PORTE = 32;
delayMicroseconds(5);
PORTE = 0;
delayMicroseconds(10);
PORTE = 32;
delayMicroseconds(6);
PORTE = 0;
delayMicroseconds(10);
PORTE = 32;
i2cWaitClockHigh();
PORTE = 0;
delayMicroseconds(10);

/* Code from i2cWait.cpp */
void i2cWaitClockHigh() {
    /* Should wait approx. 4 microseconds */
    delayMicroseconds(5);
    return;
}
```



The screenshot shows that a call to `delayMicroseconds()` with a given number of microseconds will cause a delay for a time span that is around 1.2 microseconds smaller than the wished amount of microseconds. That should be considered if any use of `delayMicroseconds()` is needed. For the second version of the i2c library in this project it wasn't necessary to use this function.

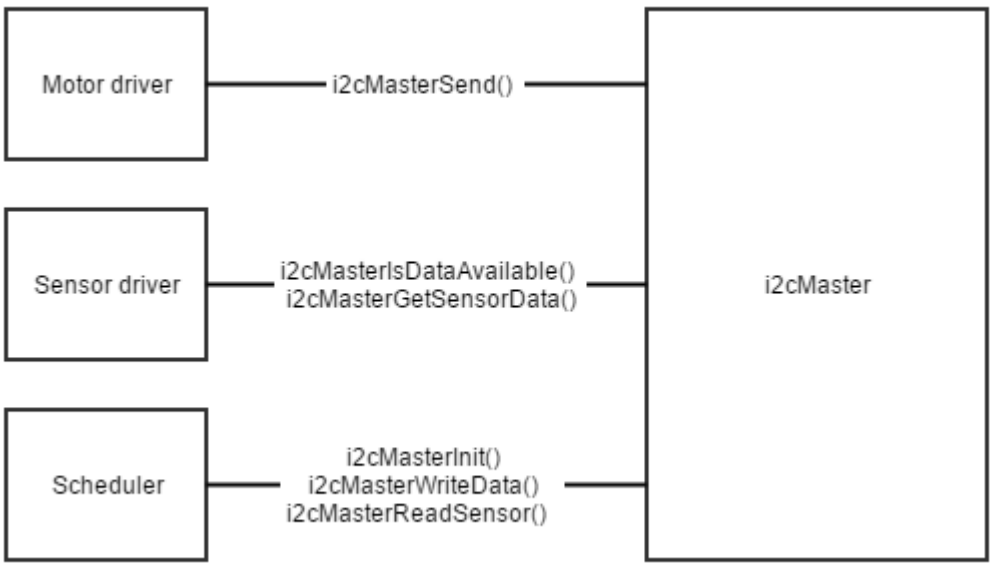
Second Version

In a second version the ATmega2560 embedded I2C functionality "AVR TWI" is used to send and receive data. This fixes the issue of version 1 because the controller delivers additional hardware for this usage.

The raw TWI call was first tested in the Arduino standard `setup()` method and tested with the LogicPort. After the successful test the interface API was planned and the following functions were planned and implemented in interfaces /i2cBib/arduino_i2c_v2/:

```
/*  Initialize i2c Master */
void i2cMasterInit();
/*  Send data to specific i2c address */
void i2cMasterSend(uint8_t address,uint8_t data[], uint16_t dataLength);
/*  Tell the Master to write data from buffer  */
void i2cMasterWriteData();
/*  Tell the Master to read data from altitude sensor  */
void i2cMasterReadSensor();
/*  Check for available sensor data  */
uint8_t i2cMasterIsDataAvailable();
/*  Get sensor data  */
uint16_t * i2cMasterGetSensorData();
```

The library is concentrating on the Master-functionalities and hasn't any Slave implementations. The main features are split up in as small packages as possible to garantee short and nearly constant execution times.



The i2cMaster driver is activated by the scheduler during the scheduling round by calling the write or read function.

If the i2c driver has any data left in the internal buffer that was previously provided by the motor driver and the write function is called by the scheduler, the i2c driver will either start the transmission by settings the write address and lock the bus as a preparation for the next transmissions or will continue sending on the locked bus by sending one byte per function call of the scheduler. The send function needs to get called as often as there are bytes in the array provided by the motor driver and one additional time für starting the transmission **in a row!**

Reading data is also triggered by the scheduler and will not lock the bus until all data is read. The i2c driver currently needs to get called three times (one time for the register configuration of the sensor and as often as described by `ALT_SENSOR_READ_DOUBLE_BYTES` in the header file) to read a complete set of data. One transmission starts with the read address and the next double byte from the sensor to prevent any errors due to changes done to one half byte between receive requests. The execution time is higher but it is built to deliver correct data. Additionally, there was no reason to build a modular receive function with support for an address because in this infrastructure, the sensor is the only device that provides data to the master The sensor driver can retrieve the data by checking for available data and getting the pointer to the internal data receive buffer of the i2c driver if the data set is complete.

There is an error handling with several error codes but it has no implementation for the analysis of the error code.

The library was successfully tested with the LogicPort logic analyzer and afterwards in combination with the scheduler.

7.1.2 RS232

Test of Serial Connection

RS-232 was tested via USB and the Serial1 interface on ports RX1 and TX1. Both were tested with baudrate 9600 using the following c-code:

```
void setup() {
    Serial1.begin(9600);
    Serial.begin(9600);
}

void loop() {
    Serial1.println("Hello");
    Serial.println("Hello");
    delay(200);
}
```

Sending data

"Hello" was detected correctly on both interfaces. Serial (Microcontroller USB Port) was tested via minicom, Serial1 via LogicPort.

There was no issue in reading the data from Serial

It was not possible to get any input data with the TTL-232R-5V adapter with minicom on the INCT_PC09 and with Putty on a private Dell Notebook (Windows 10 Pro 64bit).

There was a test with the RS232 Level Shifter and the internal RS232 connector of INCT_PC09 using a crosslinked cable. The incoming data was corrupted, some characters were dropped and in the end of the test the connection was lost and it was not possible to get a connection again.

Interface

In the real-time system it is not possible (it could be possible but it would be very hard to proof the real-time features) to use the Arduino Serial library, so it is the way to go to write a new RS232 interface.

This interface uses the ATmega2560 build in UART functionality.

7.2 Peripheral drivers

Authors: Tobias Schmitz, Mirco Wittrien

7.2.1 Motor

The existing motor driver also does not meet the real-time requirements, because it is not certain, if the used library functions might take to long, which would be against a real-time system. This means that we are using our own I2C library.

We rewrote the existing motor driver using our I2C library. The overall structure of the motor driver stayed the same.

Mapping of the motor values

The values for the motorspeed will be mapped in a way, that if the value is set to 255 the right motor runs at full speed and the left runs at idle speed and if the value is set to 0 the right motor runs at idle speed and the left motor runs at full speed.

```
const PIN_MIN =     0;
const PIN_MAX =     255;
const MOTOR_IDLE =     30;
const MOTOR_MAX =     254;
uint8_t motorspeed, motorSpeedCalc, motorSpeedLeft, motorSpeedRight;

motorSpeedCalc =     ((motorspeed - PIN_MIN) * (MOTOR_MAX - MOTOR_IDLE) / (PIN_MAX - PIN_MIN))
+ MOTOR_IDLE;
motorSpeedLeft =     (MOTOR_MAX + MOTOR_IDLE) - motorSpeedCalc;
motorSpeedRight =     motorSpeedCalc;
```

7.2.2 Sensor

The existing sensor driver already works with the sensor fusion of the MPU6050, though the functions of the Digital Motion Processor (DMP) used for the sensor fusion are using the default MPU library and the default I2C library.

This makes it impossible to estimate, whether the sensor driver manages to complete its task in the required time to fulfill the real-time requirements. That is why we decided to use the version without the sensor fusion, which reads a single set of values for the current position of the accelerometer from the I2C.

We rewrote the existing part of the sensor driver without the sensor fusion using our I2C library and removed the part with the sensor fusion. The overall structure of the used part of the sensor driver stayed the same.

7.3 Operating System

Authors: Jonas Pufahl, Jan Lehrke

During the development of the interfaces for I2C and RS232 no Arduino library was usable. The Arduino is focused on simplicity and it tries to provide a simple API for the developer. This is good for fast development but it is not good for real-time applications because every code executed in a real-time application must fulfill a maximum execution time and you cannot provide this information for the Arduino libraries.

In this project most parts of the code is using native ATMEGA code. It is not that much more code and it is simple to calculate the maximal execution time. On top of this the Arduino IDE is very limited in functionalities.

So this project showed that it would be a better way to choose a normal ATMEGA microcontroller having no ARDUINO bootloader installed and another IDE like the powerful Atmel Studio.

7.4 Problems during development

Authors: Jonas Pufahl, Jan Lehrke

While testing the i2cMaster driver there had been some unexpected behavior of the hardware. When using some simple test code to configure the motor driver, we noticed that the driver will stop working until restarting the driver by switching of the power (simply pressing the reset button of the driver hasn't solved the issue). To debug the i2c data stream the LogicPort logic analyzer was used and there was no issue in the data send by the Arduino. The motor driver send an acknowledge after receiving the first package but it doesn't control the motors. After some testing the issue was solved by doing some other i2c communication like the configuration of the sensor before configuring the motor driver.

At the end, the scheduler was able to communicate with all drivers and data was send via i2c correctly. Unfortunately there had been an issue about the sensor which stoped sending correct information. This was fixed by toggling the power but there had been no time to test the system again with a working sensor.

8 QNX Operating System

8.1 What is QNX?

QNX is a real-time embeddable POSIX operating system based on a microkernel architecture. It supports many processor families including x86 and ARM (4).

8.2 Using QNX on BeagleBone Black

The following steps were made for installing *QNX* on BeagleBone Black, executing a "Hello World" program and preparing the system for the project:

8.2.1 Requirements to run QNX on the BeagleBone Black

- BeagleBone Black with 5V power supply or mini-usb cable
- SD-Card (minimum 128MB)
- FTDI USB-to-TTL cable for a debugging connection
- *QNX Momentics IDE* Version 5.0.1
- BSP (Board Support Package) and additional files (MLO, u-boot.img) based on Version 6.6

8.2.2 Install QNX Momentics IDE

Follow the instructions of the *QNX Momentics IDE* installer.

8.2.3 Build the Board Support Package (BSP)

Follow the user guide from (1) and take care of the following changes:

- Chapter 3 - Building and installing the BSP - Connect the hardware
  - Step 2 is no needed, but install the drivers for the FTDI cable
  - You can use the *QNX Momentics IDE* terminal to establish a debugging connection
- Chapter 3 - Building and installing the BSP - Build the BSP
  - By default the BSP is built from the precompiled libraries. To build from your (modified) sources, modify the uppermost "makefile" in the root directory of the project. In the lines

    ```
    install: $(if $(wildcard prebuilt/*),prebuilt)
    $(MAKE) -Csrc hinstall
    $(MAKE) -Csrc
    ```

    you have to change "hinstall" to "install" (just remove the h) to perform a clean install from your sources. (2)
    Also note that the access rights of the BSP sources need to be executable. With "chmod -R +x *" this can be accomplished. (3)
    Make sure that it builds from your sources by editing the "build"-file under "/src/startup/boards/ti-am335x/beaglebone/" and adding "display_msg YOUR_TEXT" as last line. This should be displayed while boot.
  - When building from sources there occurred some failed imports to us. We could not fix them properly, but we changed the failing imports to use relative paths inside the project. Most of the libraries can be found under "/src/hardware/startup/lib" or in its subdirectories.
  - MLO and u-boot.img are found under (1).
  - Use the automated U-Boot commands described in the documentation
- Chapter 3 - Starting the screen graphics sample applications
  - Not needed

8.2.4 Disable the Watchdog-Timer

The BeagleBone will reset after running for 30 seconds because the watchdog timer kicks in. To disable this for testing, modify the "main.c"-file under "/src/startup/boards/ti-am335x/beaglebone/" and add a new case to the "int main()"-function like this:

```
case 'd':
    /* Disable WDT */
    wdt_disable();
    break;
```

Note: This case is documented (in the source code of the BSP), but not implemented by default.

Finally the "-d" option is added to the startup program of the BeagleBone Black in the "build"-file.

```
#########################################################################
## Startup arguments
## Use "-d" to enable watchdog timer support
##          please run "dm814x-wdtkick" with this option
#########################################################################
startup-ti-am335x-beaglebone -d
```

8.2.5 Enable execution of qconn

qconn is a program used to establish a network connection to the BeagleBone for debugging programs. When executing *qconn* it will fail because of missing libraries. Add "libtracelog.so.1" at the end of the "build"-file under "/src /startup/boards/ti-am335x/beaglebone/" and rebuild the project. Copy the generated binary to the sd card and reboot the BeagleBone. An IP address has to be assigned to the BeagleBone. This can be achived by executing "ifconfig dm0 IP_ADDRESS" on the target system. After that, start *qconn*. Now you can configure the BeagleBone as a target in *QNX Momentic IDE:* Right-click in *Project Explorer - New - Other - QNX/QNX Target System Project - Next* - Insert BeagleBone's IP address and the port you used for *qconn* (default: 8000) - *Finish.* The target can be used under "Run As..." or "Debug As..." to run or debug your project on the BeagleBone.

8.2.6 Mount the sd card in QNX

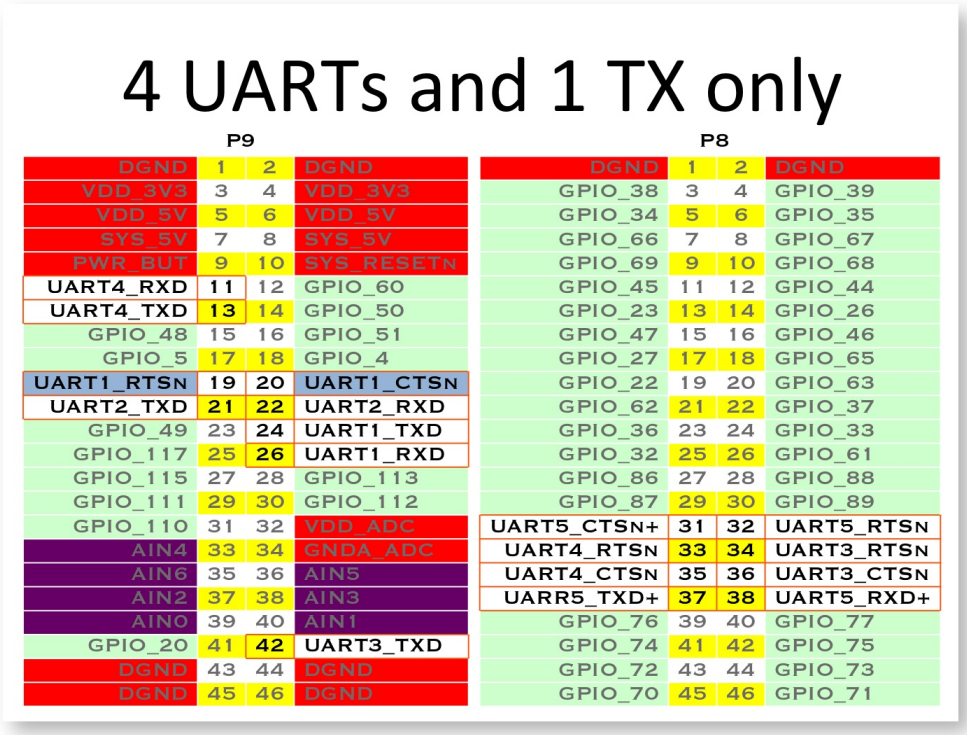
For development reasons a second partition is added to the sd card. On the first partition is the operating system installed and on the second partition is the user space to store programs and other files. To mount both partition on startup the "build"-file has been modified:

```
# HSB: mount both partitions on our SD card to get writable memory
waitfor /dev/hd0t11
mount -t dos /dev/hd0t11 /qnx
waitfor /dev/hd0t11.1
mount -t ext2 /dev/hd0t11.1 /sd
```

The first partition is mounted on /qnx while the second partition is mounted on /sd.

8.2.7 Initialize serial connection

The *BeagleBone Black* is able to control 5 serial ports (UART) on the GPIO pins, 1 serial port on the serial debug jack (J1) and the USB connector, which can also be used for a serial connection.



To enable the GPIO pins the correct function has to be uncommented in the /src/startup/boards/ti-am335x/beaglebone /init_pinmux.c file.

Note: The functions to enable the UARTs have the comment: `Untested - example configuration for cape uartX`

To bind these ports to a device-handler the program devc-seromap can be used:

```
# HSB: Initialize the configured UART1 connection
devc-seromap -e -F -b115200 -c48000000/16 0x48022000^2,46
```

The address for the UART register can be found in the documentation of the AM355x processor (see (5)).

Problem: It is possible to send data with the UART ports, but every attempt to receive data on the UART ports failed. Therefore the UART0 port (J1) is reconfigured for serial communications.

To enable shell access via USBtty you first have to change the baudrate to the desired speed:

```
sh -c '/bin/stty baud=115200 < /dev/serusb1'
```

Now you can start a listener who executes sh :

```
on -t serusb1 sh
```

Add these line to the build file to configure the port on startup.

8.3 Creating a QNX Project

To create a *QNX* project in the *QNX Momentics IDE* just click on *New - QNX C Project* - type in the name of the project and select the checkbox for ARM in the Build variance menu. Finally click on *Finish*.

Type in the following code for a "Hello world" program:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Welcome to the QNX Momentics IDE\n");
    return EXIT_SUCCESS;
}
```

To execute the program on the BeagleBone add a qconn target to a Run Configuration (described in *Enable execution of qconn*).

Problem: Debugging the program on the BeagleBone is not working.

8.4 Porting an old BSP to a higher Version

1. Download the newest BSP that you can get for your Board
2. Unpack the BSP
3. Remove the the folder "src/hardware/startup/**lib**"
4. Remove the contents of the folder "prebuilt/usr/include/sys/"
5. Download a BSP for the same architecture with the targeted version number.(QNX offers generic BSPs, you should use this one)
6. Unpack the new BSP
7. Copy the new "src/hardware/startup/**lib**" to the old BSP.
8. Copy all files and folder of the new "prebuilt" directory to the old BSP and overwrite existing files.
9. Edit the source.xml of the "old" BSP(change version number etc.)
10. Now pack the "old" package and import it into the QNX IDE.
11. Try to compile the BSP you should get errors like:

    ```
    make[6]: *** No rule to make target `libpm.a', needed by
    ```

    The libpm is no longer needed, you have to change the source like this:

    ```
    # LIBS+=io-char pm ps drvr

    LIBS+=io-char drvr
    ```

12. Try to compile again, if you get an imagefile: congratulations! If not go on...
13. Now its your turn to look for deprecated or replaced drivers, libraries and interfaces
    This process in undocumented and not repeatable for different boards, you can get hints on this link:

http://community.qnx.com/sf/wiki/do/viewPage/projects.bsp/wiki/Drivers

This is a try-and-error process, the other option is to ask the manufacturer for a BSP...

Good luck 

8.5 Conclusion

QNX is one of the most used embedded operating systems, has a verbose documentation on the main parts of the OS and seems to be a good decision as an embedded operating system. BUT the BSP for the BeagleBone is just "sloppy" created. It is not starting "out-of-the-box", there are many untested functions, some documented features are not implemented or just not working. After many hours of work we got the operating system prepared to use *qconn*, but even after many hours more we did not get the serial ports on the GPIO pins to work. The *QNX Momentics IDE* is working fine.

8.6 Sources

(1) http://community.qnx.com/sf/wiki/do/viewPage/projects.bsp/wiki/TiAm335Beaglebone

(2) http://www.qnx.com/developers/docs/660/topic/com.qnx.doc.neutrino.building/topic/bsp_BUILDSRC.html?cp=1_1_1_1_1_1

(3) https://groups.google.com/d/msg/beagleboard/mNA8DbL0GfE/1Q3XWH3rBQAJ

(4) http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/intro.html

(5) http://phytec.com/wiki/images/7/72/AM335x_techincal_reference_manual.pdf

Authors: Maximilian Schönenberg, Benjamin Hesseln and Nils Müller

9 Master Implementation

Authors: Jens Sager, Markus Salomon

9.1 General Software Design

The master implementation is done within four files. These files are PID.c, RS232.c, Scheduler.c and Main.c (or on Arduino ACD_Master_Arduino.ino).

9.1.1 PID

The PID controller was created with the structure described in PID Controller. Before use the sample time, PID parameters and setpoint have to be set via PID_setSampleTime() and PID_setTunings() and PID_setSetpoint(). Setting a sample time is necessary because the controller assumes regular intervals in which PID_comput() is called. This function takes the current sensor value and returns the newly calculated PID value. It is possible to change the sample time while the controller is running. The sample time is set in milliseconds while K_I and K_D are set in units of 1/s and s.

Note: The PID controller has only undergone basic tests and is very likely not bug free. Actually controlling the ACD and searching for proper tuning parameters has not been done.

9.1.2 Scheduler

The scheduler itself is split into the files Scheduler.c and Main.c. In Scheduler.c is the state machine of the designed protocol (see RS232 Communication Protocol), a thread for generating tick messages and the logic for updating the task list. In Main.c is an endless loop which checks if a task is ready to run. If so, it executes these task.
We decided not to use the solution with function pointers and structs descripted in http://www.safetty.net/download/pont_pttes_2001.pdf (p. 596ff) to increase readability. To descripe a task in the current solution there are three variables needed. One for the delay, one for the period and one for checking if the task is ready to run. These variables have to be updated in the updateTaskList() method of scheduler and in the Main.c file checked.

9.2 Platform Specific Implementations

9.2.1 Arduino

Because of some trouble with the QNX operating system, the master was also implemented on an Arduino. This way we were able to test parts of the logic of the master (PID, state machine of the scheduler and scheduler), although some problems on QNX were not fixed. To achieve this, we had to change the code for RS232-Communication and the code for getting the timer interrupt. The code for the RS232-Communication was taken from the slave's group (just removed the flag for serial interrupt) and the code for the timer was taken from http://playground.arduino.cc/Code/Timer1.

9.2.2 QNX

The Implementation in QNX is split into two threads. The Main-thread calls all initialization methods and then constantly executes tasks when their corresponding flags are set. The timer-thread waits for an interrupt on the timer. When an interrupt happens it reads any Messages received from the Slave and updates the state machine. Afterwards it sends the corresponding tick message to the slave and updates the task list for the main thread.

Since the PID-Implementation was made with a constant sample time in mind its sample time has to be initialized. This time is the same as the period of the timer (with a constant factor).

Setting up the timer

The timer notifies the timer-thread via a pulse over a preddefined communication channel. This channel is created via the use of the method "ChannelCreate(int)". The timer event then gets initialized with the macro "SIGEV_PULSE_INIT" and the timer gets created with the "timer_create()" method. The timer can now be given a time of initial execution as well as an interval time. Thus

```
timer.it_value.tv_sec = 1;
timer.it_value.tv_nsec = 0;
timer.it_interval.tv_sec = 5;
timer.it_interval.tv_nsec = 0;
```

would first send a pulse after 1 second and then another pulse every 5 seconds after that. Finally the timer needs to be started with the "timer_settime" method.

Setting up the timer thread

To make a thread we first create a variable of the type "pthread_attr_t". This variable gets initialized by "pthread_attr_init". If attributes are not manually set then the created thread will inherit the priority of the parent thread. The thread is then created with "pthread_create" where it is given its attributes as well as a function pointer for the function it should execute. In this case we are using a simple loop to wait for timer interrupts and handle them when they happen:

```
while(1){
rcvid = MsgReceive (chid, &msg, sizeof (msg), NULL);

if (rcvid == 0) {
gotAPulse();
}
}
```

Problems with QNX SDP

During the implementation of the Master for QNX we were unable to to run or debug the code on the machine we wrote it on. Because of this we had to resort to running each iteration on the Beaglebone using basic console text outputs for debugging purposes.

9.3 Open Issues

Because we ran out of time, there are still some known bugs.
The first open issue is, that the values of the sensors will be encoded on the slave's side before sending to the master but not decoded by the master.
The second issue is, that no optimization of the PID values was done. An introduction about how this can be done, can be found in PID Controller.

10 Conclusions

Author: Prof. Dr. Jan Bredereke

This project achieved its main goals fully. However, due to lack of time, we could not complete the actual implementation. The main goals of the project were:

- Demonstrate the application of a time-triggered architecture to a simple embedded distributed, hard real-time system. (This was the learning goal set for the students.)
- Collect practical experience on using the QNX real-time operating system on space hardware and for a time-triggered architecture. (This was an additional goal of the project's organizers.)

We summarize our experiences with regard to these aspects in the following subsections.

10.1 Using a Time-Triggered Architecture

As expected, using a time-triggered architecture is suitable for designing a hard real-time system, that is, a system which is "guaranteed fast enough". In contrast, the alternative scheme of an event-triggered system would have provided an execution "as fast as possible" only, which very well might be too slow. However, it turned out that using Custom Off-The-Shelf drivers in a time-triggered architecture often is not possible.

10.1.1 Suitable for Hard Real-Time

The students learned how to use a time-triggered architecture for designing a hard real-time system. Section Time-Triggered Scheduling provides an overview and references, in particular to the textbook by Pont. As a bonus, the students refer to a whitepaper by Schaffer and Reid from QNX which discusses many different common scheduling approaches and their pros and cons in different settings.

10.1.2 Using Custom Off-The-Shelf Drivers in a Time-Triggered Architecture

A practical experience from the project is that using Custom Off-The-Shelf (COTS) drivers in a time-triggered architecture often is not possible. All the COTS driver libraries we wanted to use are based on interrupts. Therefore they are based on the event-triggered paradigm. This kind of interrupts does not integrate into the scheduling scheme of a time-triggered system. Furthermore, the time-triggered, cooperative approach demands that each task must yield the processor after a a fixed and, in particular, after a short period of time. If necessary, the task must be organized such that it continues its work in the next time slot. None of the COTS drivers used was designed in this way. This required us to redesign them from scratch.

This concerned the driver for the I2C bus (provided by the Wire library for Arduino), the driver for the RS-232 link on the Arduino (also provided by the Wire library), and the driver for the attitude sensor (provided by the sensor's manufacturer). See Section Arduino and Time-Triggered Drivers for details, in particular its Subsections 7.1.1, 7.1.2, and 7.2.2.

We did not have the time to fully investigate the driver for the RS-232 link on the BeagleBone Black (provided by the QNX board support package).

As a consequence, using the Arduino development environment for a time-triggered architecture is of no particular help. The Arduino development environment and its simplicity is great to aid beginners. But in our context, using a normal IDE for the same ATmega microcontroller would have been better.

10.2 Using QNX on Space Hardware

Our project provides a practical crib sheet for using QNX in Section QNX Operating System. One example for these steps is how to disable the watchdog timer, which otherwise resets by default any system after 30 seconds. Beyond that, we made the experience that the availablity of a board support package (BSP) for QNX is critical.

Originally, we intended to use Airbus's e.Cube computer suitable for space as the master node. However, it turned out that there the most current BSP by the board manufacturer is for QNX version 6.4.1, while currently QNX already is at version 6.6.0. There does not appear to be hope for a more current BSP by the manufacturer.

Nevertheless, the students didn't give up, and in Subsection 8.4 they provide a recipe for porting a BSP to a current version of QNX. However, this solution can be seen as a last resort only. It is no way to deliberately design a reliable system.

Similarly, the BSP for the BeagleBone Black did not have the quality we expected, either. The code for the UART (RS-232) driver bears the comment "Untested", and the code for receiving data on the UART ports didn't run out-of-the-box. (We didn't have time to investigate further.)

The lesson learned is that before choosing QNX (or any other commercial real-time operating system) for a project, we should have a deep look at the quality of the applicable board support package. Likewise, we should ensure that the board manufacturer will provide updates of the BSP for future releases of QNX.

10.3 Using QNX for a Time-Triggered Architecture

Using the QNX real-time operating system for a time-triggered architecture appears to be feasible without difficulties. This is so even though we noted in the introduction (Subsection 1.2.1), that the documentation of QNX systematically uses a substantially different meaning for the words "real-time" than us. It uses them in the sense of "as fast as possible", and not in the sense of "guaranteed fast enough". The introductory documentation insinuates an event-triggered, not a time-triggered, system architecture to its readers.

Nevertheless, it turned out that designing a time-triggered architecture with QNX appears to be very well feasible. You only have to know well the concepts you want to apply. Otherwise, the QNX documentation can be misleading easily.

10.4 Incomplete Implementation of the Angle Control Demonstrator

We did not complete the implementation of the Angle Control Demonstrator in the (short) time available. Several work packages required more time than anticipated. In particular, we under-estimated the time necessary for the drivers, which we had to rewrite from scratch to make them fit into the time-triggered architecture, instead of reusing existing code. Some loose ends therefore remain, see the previous sections. Similarly, we had no opportunity yet to find suitable parameters for the PID controller. This might need some additional time. Section PID Controller, Subsection 4.2.3, describes how it could be done.

Nevertheless, this project achieved its main goals fully, as stated in the beginning of this section.

11 Appendix: Structure of the Oral Presentation (in German)

- Es wurde sich darauf geeinigt, dass die Präsentation auf deutsch gehalten wird und keine explizite Präsentation hierfür erstellt wird. Lediglich das Wiki soll als Präsentationsgrundlage dienen.
- 1. Einführung/Aufgabenstellung (Jan Bredereke)
- 2. Genutzte Hardware und angestrebte Architektur (Jan Lehrke, Jonas Pufahl)
- 3. PID (Jens Sager, Benjamin Schäfer)
  - 3.1. Was ist das?
- 4. Zeitgesteuertes Scheduling (Markus Salomon, Nikolas Schreck)
  - 4.1. Allgemein
  - 4.2. Verteilte Systeme
  - 4.3. Unser System und Protokoll
- 5. Arduino und zeitgesteuerte Treiber (Jan Lehrke, Jonas Pufahl)
- 6. QNX (Nils Müller, Maximilian Schöneberg, Benjamin Hesseln)
  - 6.1. Was ist das? Was stellt es zur Verfügung?
  - 6.2. Warum nutzt man es?
  - 6.3. Praktische Erfahrungen (Installation, BSP, Dokumentation)
  - 6.4. QNX und zeitgesteuerte Verarbeitung + Treiber
- 7. Zusammenfassung und Ausblick (Julian Greilich)