A Distributed System with Guaranteed Temporal Properties For Attitude Control

Jan Bredereke, Linus Andrae, Patrick Delventhal, Dennis Hilmer, Lennart Lindenberg, Sören Stingl, Michael Sved, Niklas Wahrenberg, and René Wolter

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

Oct. 2018

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

A Distributed System with Guaranteed Temporal Properties For Attitude Control

e.Cube based infrastructure

Exported on 09/26/2018

Table of Contents

1 Introduction 6
1.1 The Attitude Control System Demonstrator 6
1.2 The Time-Triggered Architecture 7
1.2.1 Time-Triggered Processing for Hard Real-Time Systems 7
1.2.2 The Time-Triggered Architecture for the Attitude Control System Demonstrator 8
1.3 Original Project Description (in German) 9
2 Basics / Structure 10
2.1 Introduction 10
2.2 Overview 10
2.3 Hardware components 12
2.3.1 NUCLEO-32L152RE 12
2.3.2 Arduino Mega 2560 14
2.3.3 Motor driver 15
2.3.4 Motors 16
2.3.5 Attitude sensor 17
2.4 Wiring 19
2.4.1 Ardunio side 19
2.4.1.1 Arduino 19
2.4.1.2 AltIMU-10 v3 (I2C Address: 0b00011101) 19
2.4.1.3 Grove I2C Motor Driver V1.3 (I2C Address: 0b00001111) 20
2.4.1.4 TTL Serial 21
2.4.1.5 Power Barrel Jack 21
2.4.1.6 Power Barrel Jack Switch 21
2.4.2 STM32 side 22
2.4.2.1 STM32 Nucelo board 22
2.4.2.2 Logic converter (LV side) 22
2.4.2.3 Logic converter (HV side) 23
2.4.2.4 Analog Potentiometers 23
3 Initial evaluation of the available Hardware 24
3.1 Summary 24
3.2 Meassurments 25

4 Interface definition 26
4.1 Introduction 26
4.2 General 26
4.3 Communications 26
4.3.1 RS-232 26
4.3.2 I2C Motor Driver 27
4.3.3 I2C Sensor Board 28
4.3.3.1 Init Protocol 28
4.3.3.2 Read register 28
4.4 Globalvariables 29
4.4.1 STM32 29
4.4.2 Arduino 31
5 Overview of the Code Repository in Git 32
6 Component Arduino 34
6.1 Introduction 34
6.2 Scheduler 34
6.3 RS232 35
6.4 Message Dispatcher 35
7 Drivers for Arduino-compatible Microcontroller 37
7.1 I2C driver 37
7.2 InvenSense MPU-6050 37
7.3 Attitude Sensor AltIMU-10 v3 37
7.4 Grove - I2C Motor Driver V1.3 38
7.4.1 Re-engineering of existing Motor Driver 38
7.4.2 Interfaces for Motor Operation 39
7.4.2.1 Usage 39
7.4.2.2 Timeout 39
7.4.3 Mapping of the motor values 40
8 Component STM32 41
8.1 Introduction 41
8.1.1 Setting up the Development Enviroment 41
8.2 Main 41

8.2.1 Initalisation 41
8.2.2 Main Loop 42
8.3 Scheduler.... 42
8.3.1 Diagram 42
8.3.2 Schedule without Pots and Serial 42
8.3.3 Schedule with Pots and Serial....43
8.3.4 Scheduling algorithm 43
8.4 RS232 driver of the STM32.... 43
8.4.1 Message dispatcher 44
8.5 Onboard LED 44
8.6 Attitude Control (PID) 44
8.6.1 Control loops in general 44
8.6.2 PID Controller....45
8.6.2.1 Proportional term (P)....46
8.6.2.2 Integral term (I) 46
8.6.2.3 Derivative term (D)....47
8.6.2.4 Combination of all terms: PID controller....48
8.6.3 Implementation 48
8.6.4 Requirements for Scheduler....49
8.6.5 Improvements of the algorithm 49
8.6.5.1 Derivative kick problem....49
8.6.5.2 Windup Reduction 51
8.6.6 Adjustable PID parameters during runtime....52
8.7 Changing Setpoint in defined interval 52
9 The SysTick-Timer of the FreeRTOS....53
9.1 What does the SysTick-Timer do? 53
9.2 How to Configure the SysTick-Timer?.... 53
9.3 What are side effects of changeing the systick time?.... 53
10 Using the STM32 SysTick as a timebase for the Arduino board 54
11 Other components....55
11.1 Introduction 55
11.2 Software components 55
11.2.1 I2C driver 55

11.2.2 Attitude Sensor AltIMU-10 v3 55
11.3 Grove - I2C Motor Driver V1.3 55
11.4 Hardware components 57
11.4.1 Motors 57
11.4.2 Motor driver 57
11.4.3 Attitude Sensor AltIMU-10 v3 58
12 Conclusion 59

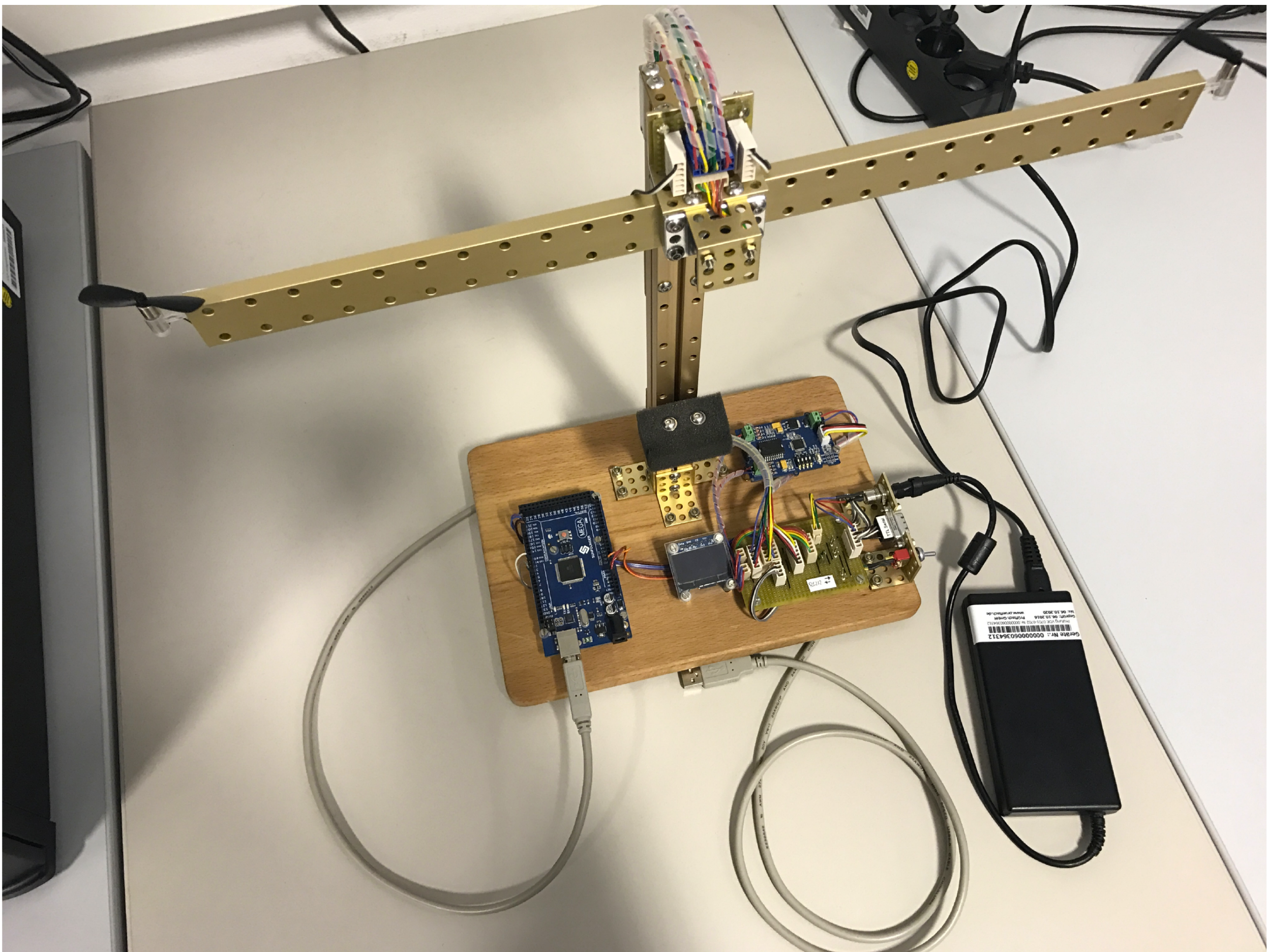
1Introduction

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

This project demonstrates how to implement a distributed system with guaranteed temporal properties using a time-triggered architecture. 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 summer term 2018. It was supervised by Prof. Dr. Jan Bredereke. This project builds on a similar project in the preceding term. The previous project realized many components already. However, in several places matching interfaces are missing still. Therefore, the emphasis of the current project is on the relations among the concepts, and on implementing a coherent system. The chapters after the introduction were written by the students named there. The project was conducted in cooperation with Airbus Bremen.

1.1The 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 STM32 microcontroller running the FreeRTOS operating system. The actuators and the sensor are controlled by an Arduino-compatible microcontroller (Sunfounder Mega). Both microcontrollers communicate over an RS-232 serial connection using a time-triggered communication protocol.



1.2The Time-Triggered Architecture

1.2.1Time-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.

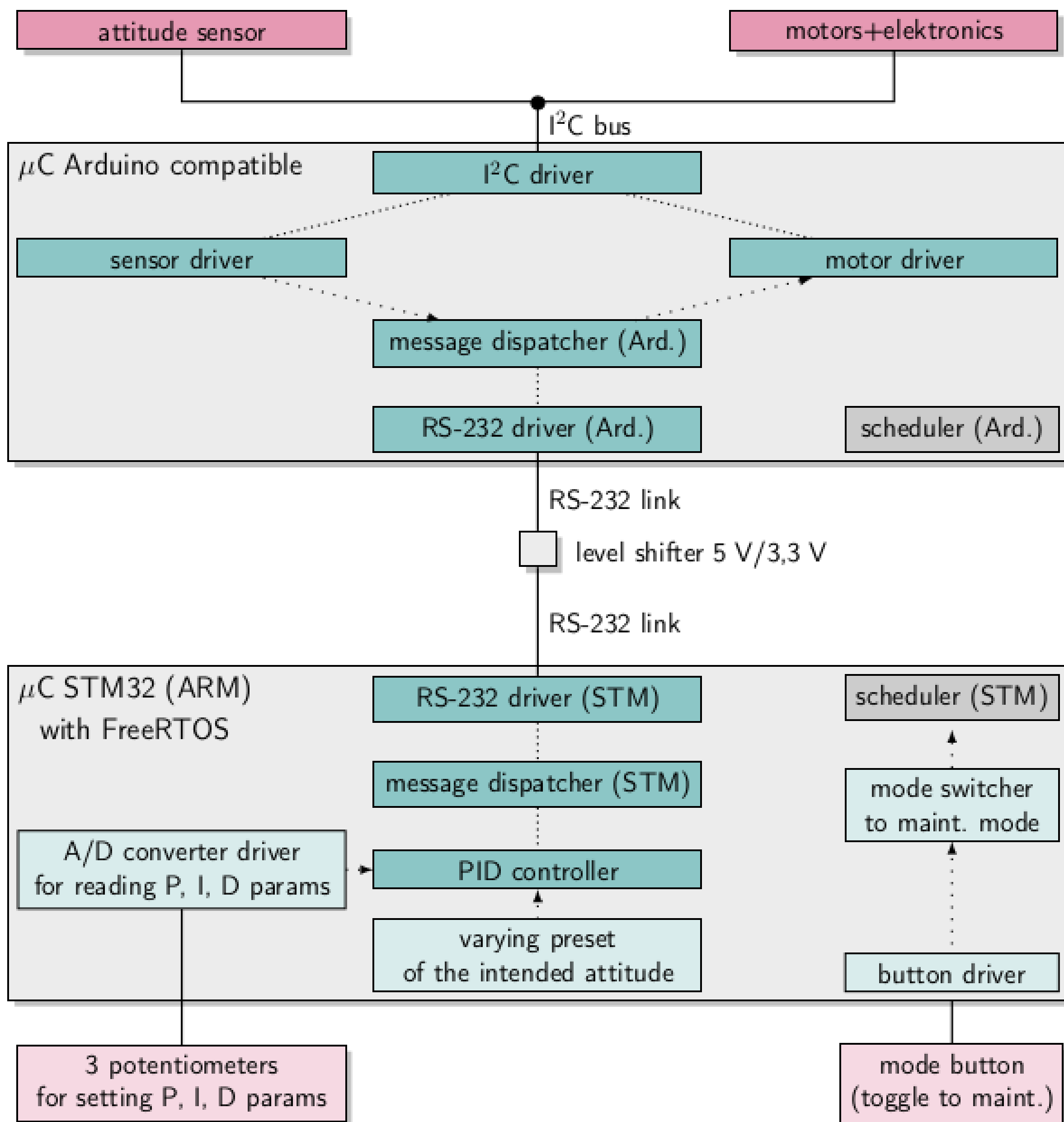
An alternative scheme would be event-triggered processing. Such a scheme uses interrupts and priorities for tasks. At any time, a higher-priority task may interrupt the execution of a lower-priority task. This allows for a quick response to an urgent issue. Also, the average response time of tasks in 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.

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 STM32 microcontroller is the master. It generates the time ticks which determine the schedule of all tasks on all nodes. The Arduino-compatible 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. One optional component is a varying preset of the intended attitude. It is intended to make the behaviour of the controlled arm look more interesting. Furthermore, optionally the parameters of the PID controller shall be controllable by three potentiometers. This can help to find suitable values for these parameters much faster than by recompiling the software for every iteration. In order to be able to find out the current values of these parameters, optionally there is also a button that toggles the STM32 node into a maintenance mode. In the maintenance mode, the PID controller does not run. Instead, the parameter values are sent by a maintenance program to a PC on a second RS-232 link. (This mode and the second link are not shown in the figure.)



1.3Original Project Description (in German)

projektbeschreibung-2018-03-08.pdf[1]

1 https://pforge.eso-io.com/confluence/download/attachments/115818039/projektbeschreibung-2018-03-08.pdf?api=v2&modificationDate=1526026389671&version=1

2 Basics / Structure

Authors: Tim Niebuhr, Andre Sarich (SoSe 2017),

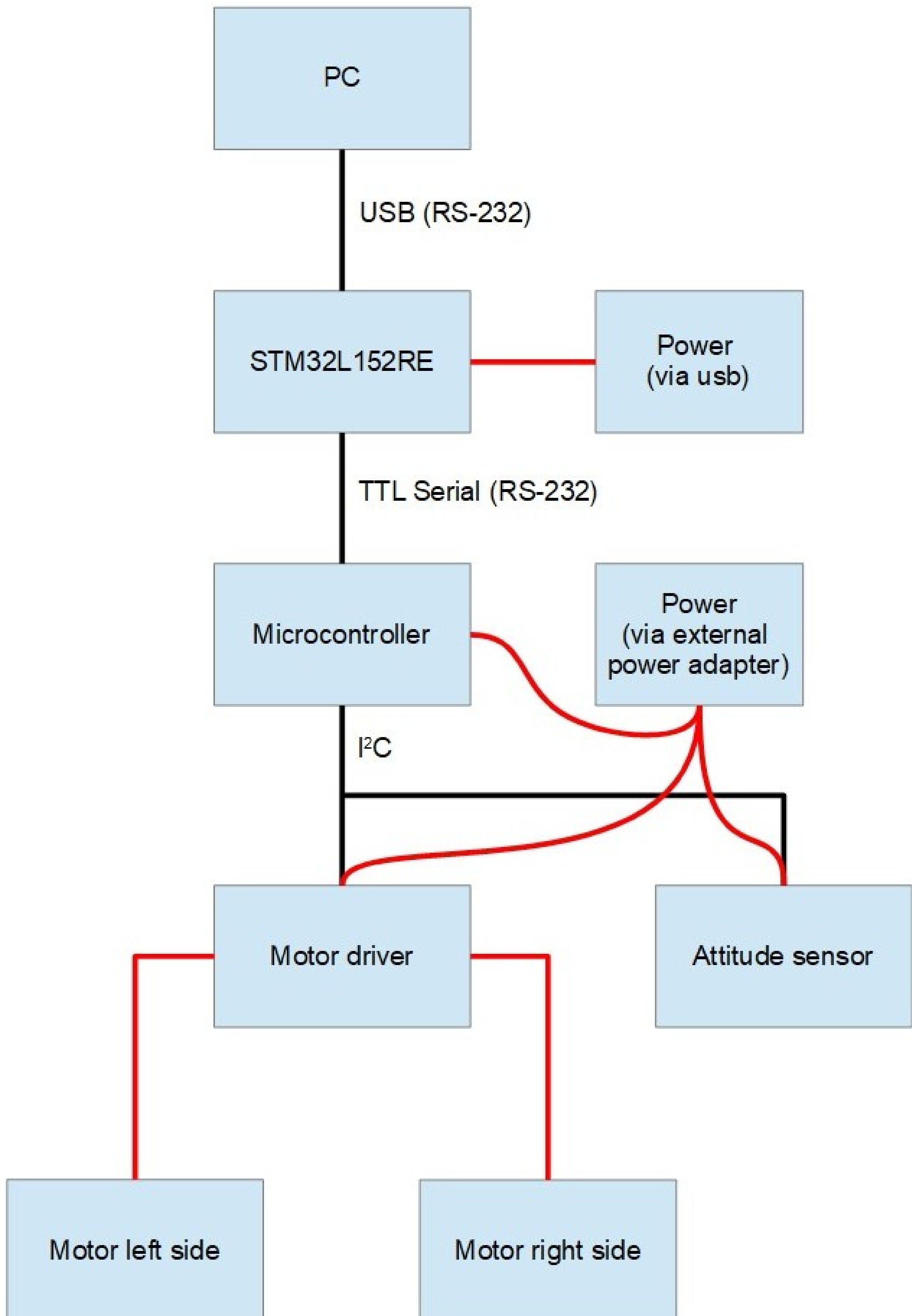
modified by Michael Sved, Niklas Wahrenberg, Linus Andrae

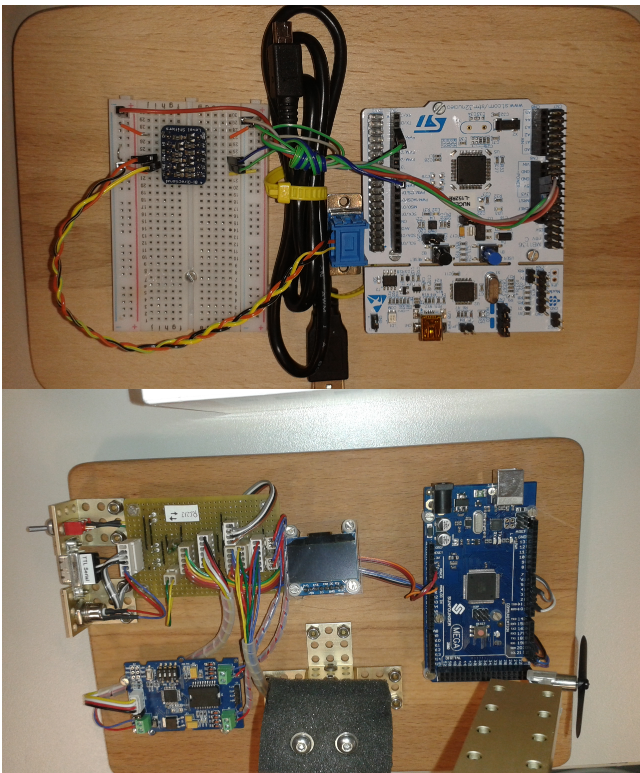
2.1 Introduction

Author: Sören Stingl

This chapter describes the components and the technical details. It contains the datasheets, further information and the setup of the hardware.

2.2 Overview


PC
USB (RS-232)
STM32L152RE
Power
(via usb)
TTL Serial (RS-232)
Microcontroller
Power
(via external
power adapter)
I²C
Motor driver
Attitude sensor
Motor left side
Motor right side

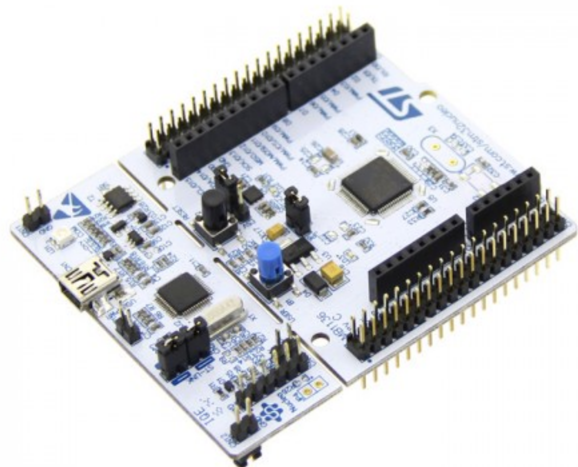


2.3 Hardware components

2.3.1 NUCLEO-32L152RE

Author: Sören Stingl, Michael Sved

The board NUCLEO-L15RE included the microcontroller STM32L15RE with the Firmware STM32CUBEL1. STM32CUBEL1 collects different embedded software to develope applications for the microcontroller STM32L15RE. There are included among others the operating system FreeRTOS, various utilities and drivers.



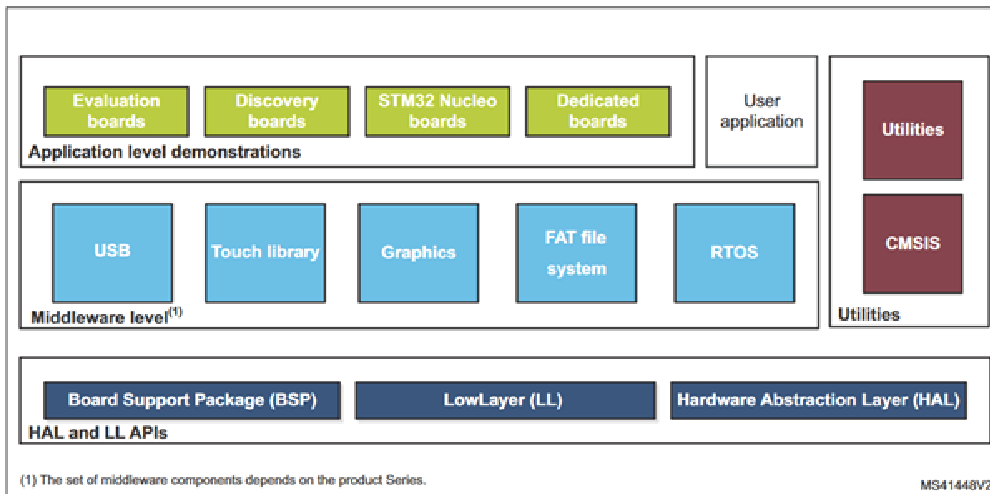


Figure: STM32Cube firmware components from the datesheet

- The NUCLEO-L15RE board offers the following features:
- Three LEDs (USB communication, user LED, power LED)
- Two push-buttons: USER and RESET
- Two types of extension resources
    - Arduino™ Uno V3 connectivity
    - ST morpho extension pin headers for full access to all STM32 I/Os
- etc.

Datasheet STM32CUBEL1:

http://www.st.com/content/ccc/resource/technical/document/user_manual/b2/e2/c0/27/b3/2a/40/ee/DM00127090.pdf/files/DM00127090.pdf/jcr:content/translations/en.DM00127090.pdf (April 2017)

Datasheet NUCLEO-L15RE:

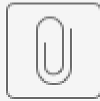
http://www.st.com/content/ccc/resource/technical/document/user_manual/98/2e/fa/4b/e0/82/43/b7/DM00105823.pdf/files/DM00105823.pdf/jcr:content/translations/en.DM00105823.pdf (December 2017)

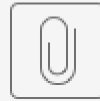
site: https://www.st.com/en/evaluation-tools/nucleo-f334r8.html

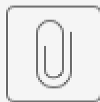
Mircocontroller used: STM32L152RE

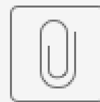
MCU datasheet: www.st.com/resource/en/datasheet/stm32l151qe.pdf[2]

2 http://www.st.com/resource/en/datasheet/stm32l151qe.pdf


stm32l151qe.pdf

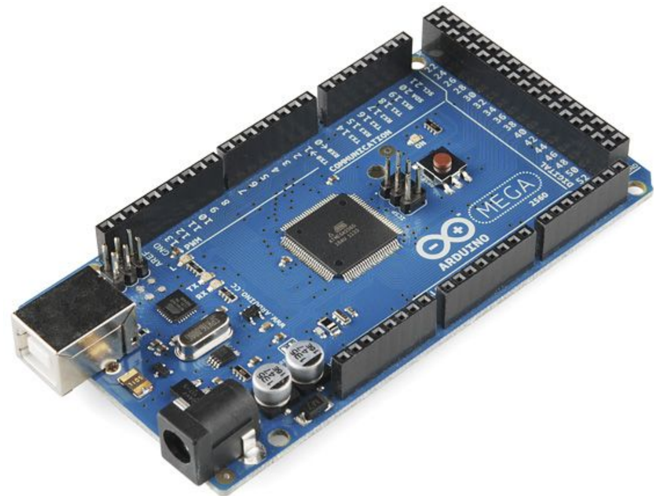

en.DM00105823.pdf


MB1136.pdf


en.DM00127090.pdf

2.3.2Arduino Mega 2560

Author: Michael Sved

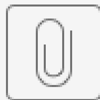


Name: Arduino mega 2560

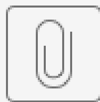
Data sheet:https://www.arduino.cc/en/uploads/Main/arduino-mega2560_R3-sch.pdf

Microcontroller used: Atmega2560

MCU datasheet: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf



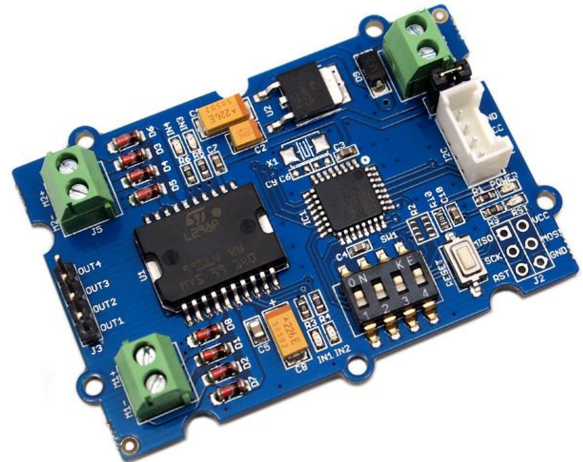
Atmel-2549-8-bi...1_datasheet.pdf



arduino-mega2560_R3-sch.pdf

2.3.3Motor driver

Author: Michael Sved

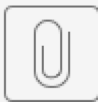


Name: Grove I^{2}C Motor Driver V1.3

Site: http://wiki.seeed.cc/Grove-I2C_Motor_Driver_V1.3/

driver IC used:L298N

Datasheet: https://www.sparkfun.com/datasheets/Robotics/L298_H_Bridge.pdf



Specifications

Item	Min	Typical	Max	Unit
Working Voltage	6	-	15	VDC
Max Output Current per channel	0.5			A
Maximum Total current	1.0			A
Input/output voltage on I2C bus	5			V
Communication protocol	I2C			/

Features

- Grove Compatible
- I2C Interface
- Adjustable motor speed and rotation direction
- Changeable slave address by hardware

2.3.4Motors

Author: Michael Sved

Name: DC Motor Crazyflie Nano Quadcopter (BC-CM-01-B)

Data sheet: https://media.digikey.com/pdf/Data%20Sheets/Seeed%20Technology/316030003_Web.pdf

There are two equal motors (one left and one right on the rotating beam). Both are controlled by the Motor driver Grove I^2C Motor Driver V1.3.

2 x Motor Crazyflie Nano Quadcopter with each 1 x Propeller

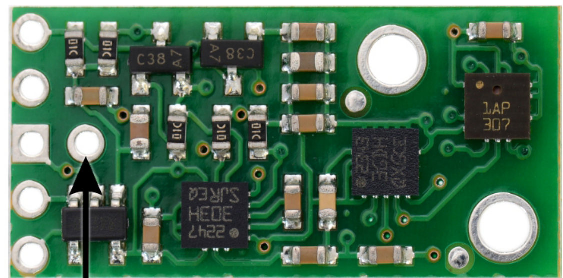
Description	Spare 6x15 mm DC coreless motor for the Crazyflie Nano Quadcopter	Spare counter rotating propellers for the Crazyflie Nano Quadcopter
Specifications	• Diameter: 6 mm • Length: 15 mm • Shaft length: 3.5 mm • Shaft diameter: 0.8 mm • Weight: 1.7 g • Kv: 12000 rpm/V • Rated voltage: 4.2 V • Rated current: 810 mA • Wire length: 67 mm	• Size: 45 mm • Fits shaft: 0.8 mm

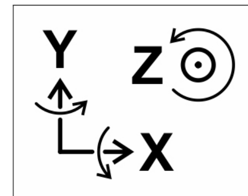
2.3.5Attitude sensor

Authors: Michael Sved, Patrick Delventhal

In this semester it was decided to use a different angle sensor. This is due to the difficulties that the previous semester had with the MPU6050 sensor.

The replacement sensor supplies a simple means to read the acceleration values and to convert them to angles values in degrees.





The sensor module actually contains 3 seperate sensors.
these are:

LSM303: magnetometer and accelerometer

L3GD20H: gyroscope

LPS331AP: barometer

Of these sensors, only the LSM303's accelerometer is needed.

AltIMU-10 v3

Site: https://www.pololu.com/product/2469

Sensor IC used

Name: STmicroelectronics LSM303D

Data sheet: www.st.com/resource/en/datasheet/lsm303d.pdf[3]

[3] http://www.st.com/resource/en/datasheet/lsm303d.pdf

2.4Wiring

2.4.1Ardunio side

Authors: Michael Sved, Patrick Delventhal

2.4.1.1Arduino

Arduino 2560 Pin	**usage**
5V	5V for system
GND	GND for system
Vin	7V for system
5	OLED Reset pin (Rst)
18 (TX1)	TTL Serial TX (2)
19 (RX1)	TTL Serial RX (3)
20 (SDA)	H-Bridge Data (SDA) OLED Data (DATA) Sensor Data (SDA)
21 (SCL)	H-Bridge CLK (SCL) OLED CLK (Clk) Sensor CLK (SCL)
Barrel jack	7V Power Supply input
USB	Firmware upload interface

2.4.1.2AltIMU-10 v3 (I2C Address: **0b00011101**)

AltIMU-10 v3	**usage**
SCL	Arduino pin 21 (SCL)
SDA	Arduino pin 20 (SDA)

AltIMU-10 v3	usage
GND	GND
VIN	5V
VDD	NC
SA0	NC

2.4.1.3 Grove I^{2}C Motor Driver V1.3 (I2C Address: 0b00001111)

Communication Port

Motor Driver	usage
GND	GND
VCC	5V
SDA	Arduino pin 20 (SDA)
SCL	Arduino pin 21 (SCL)

Power input

Motor Driver	usage
VS	7V for System
GND	GND

Motor1

J1	usage
M1	Motor Left

Motor2

J5	usage
M2	Motor Right

2.4.1.4TTL Serial

D-SUB 9	usage
1	NC
2	Arduino pin 18 (TX1)
3	Arduino pin 19 (RX1)
4	NC
5	GND
6	NC
7	NC
8	NC
9	NC

2.4.1.5Power Barrel Jack

Barrel	usage
inner pin	7V for system (powers STM32)
outer shield	GND

2.4.1.6Power Barrel Jack Switch

Switch	usage
Top	NC
center	Barrel pack inner pin (7V)
Bottom	7V for system

2.4.2STM32 side

Authors: Michael Sved, Linus Andrae

The STM32 MCU has 3 USART and 2 UART interfaces. The USART interface can also be used as a UART interface.

It was at first attempted to use the UART interface on Pins D0/D1 as these pins were also labeled RX/TX on the silkscreen on the PCB. This would have been USART2 as per the Datasheet of the Nulceo board. However it was not possible to establish any kind of connection. The reason for this was that the solder bridges SB62 and SB63 were not in place thereby essentially disabling the interface. This was determined only after in depth research of both the MCU datasheet and the Nucleo datasheet. Following this discovery it was determined that USART1 could possibly be used, as it did not have any solder bridges and was directly connected to pins D8(TX) and D2(RX). Once the firmware and the hardware connections were correctly configured we could finally send data from the STM32 which could be received by the Arduino.

In conclusion we must stress that the documentation for establishing the UART connection is practically non-existent and must be painstakingly pieced together manually through cross reference of both the MCU and Nucleo datasheets. Had the Nucleo board datasheet been more detailed it would have been much easier and fast to establish the needed UART connection between the STM and Arduino MCUs.

2.4.2.1**STM32 Nucelo board**

Pin	usage
D8	UART1 TX
D2	UART1 RX

2.4.2.2**Logic converter (LV side)**

Pin	usage
LV	3.3V
A1	NC
A2	UART1 RX
A3	UART1 TX
A4	NC
GND	GND

2.4.2.3 Logic converter (HV side)

Pin	usage
HV	5V
B1	NC
B2	UART1 RX
B3	UART1 TX
B4	NC
GND	GND

2.4.2.4 Analog Potentiometers

Pot1	usage	Pot2	usage	Pot3	usage
left		left		left	
center		center		center	
right		right		right	

3Initial evaluation of the available Hardware

Author: Michael Sved

Before any software design work for this project was started, a short overview of the available hardware was established.

This included determining the capabilities of the provided hardware, in the case of this first evaluation, the propeller motors for changing the angle of the arm.

Initially the provided Arduino compatible board was powered solely by the PC USB port. The Arduino board then supplied the dual H-bridge driver with a voltage which in turn would drive the motors.

To test the H-Bridge driver we used the Arduino library for the H-bridge driver as described in chapter Hardware. The example sketch was loaded on the Arduino and executed.

Our initial finding were as follows:

Running the driver, motors and Arduino off the PC USB port proved to be insufficient. As there seemed to be no clear documentation from the previous semester as to what voltage was needed to run the motors, it was expected that 5V from the USB port would also be sufficient, given that the motors are rated at 4.2V as per the chapter Hardware.

However, 5V proved to be insufficient to allow a single or even two motors operating in opposite directions to move the arm from the fully tilted position. Stranger still was the fact that we only measured ~2.2V across each motor. Rather than 5V.

After consulting the datasheets of the H-bridge driver, the motors and Arduino the cause of these large voltage drop was found.

There are numerous diodes across the system that drop the voltage by roughly 2.5V

The entire system is to be supplied with 7V via the Arduino barrel jack. The voltage input to the Arduino is routed directly to the Vin pin of the Arduino headers. However, there is a reverse polarisation protection diode placed between the barrel jack and the Vin pin. This results in a voltage drop of ~600mV.

Further investigation resulted in finding the next voltage drop on the H-bridge driver. This driver board also has a reverse polarisation protection diode with a ~600mV voltage drop.

This however does not provide the total voltage drop measured across the motors. For this the datasheet of the actual driver IC was consulted. The internal schematic of the driver was depicted in the datasheet. The H-Bridge component is constructed using 4 conventional BJTs per channel. The datasheet specifies a voltage drop between 1.8 and 3.2V. This corresponds to the voltage drop measured across the motors when these were running at 100%.

3.1Summary

5V supply voltage is insufficient to power the motors. Instead 7V must be used using the provided 7V power supply which connects to the Arduino barrel connector. Although the motors are rated for 4.2V, due to the voltage drops of the Arduino, the H-bridge driver board and the H-bridge driver IC itself. The total voltage drop of ~3V is present across the system. As a result the motors are then supplied with ~4V which is within the design specifications.

When supplied with a voltage of 4V it is possible for a single motor to change the angle of the arm. Is should also be noted that then attempting to run both motors in opposite direction to increase thrust, the motor rotating in the negative direction does not supply any meaningful additional thrust. This is likely due to the orientation of the propeller blades. They are likely optimised for one rotation direction only. As the datasheet specifies them for use in small remote controlled quad-copters this theory would be most probable. The motors would have no reason to rotate in the opposite direction to change the altitude of the quad-copter as gravity would supply the downward force to lower the altitude instead. Only the roatation speed would need to be adjusted.

Source code of the example sketch used the test the motors is available in the Git-repository.

3.2Meassurments

Votlage drop Arduinio: ~600mV

Voltage drop H-bridge driver board: ~600mV

Voltage drop H-bridge driver IC: 1.8 - 3.2V (per datasheet of L298N)

total measured voltage drop: ~3V

4Interface definition

4.1Introduction

Author: Sören Stingl

In this chapter the interfaces are defined and described. This includes the structure and content of the messages to be transmitted.

4.2General

Data inside the microcontroller is passed using global variables.

“Shall” indicates a mandatory requirement

“Should” indicates a recommended but not mandatory requirement

“May” indicates a lesser used optional interface

4.3Communications

4.3.1RS-232

Author: Linus Andrae

The RS232 interface shall run with 9600 Baud, 8Bit Wordlength, 1 Stopbit, no parity.

Sender	**Receiver**	**length**	**data**	**dataformat**
STM32	Arduino	1 Byte	motor control data	signed char: data (valid values (-100,...,100)) -100 = Motor 1 at 100% +100 = Motor 2 at 100% 0 = all Motors off values larger than 100 or smaller than -100 shall be considered invalid
Arduino	STM32	1 Byte	angle	signed char degrees; (+-128 degrees with one degree precision); positive values = mathematical positive turn of the rotor, 0 = horizontal rotor position

4.3.2 I²C Motor Driver

Author: Michael Sved

receiver	transmitter	length	data	dataformat
MPU6050	ATmega2560	2 Byte min	Adress byte, Data byte(s)	8bit char

To communicate with the Motor Driver, 1 Adress byte and 3 Data bytes shall be sent

Address byte: 0x0F

A7	A6	A5	A4	A3	A2	A1	A0(R/¬W)
0	0	0	0	1	1	1	0

The R/W bit must be set to write.

Data byte 1: command

Name	Value
MotorSpeedSet	0x82
PWMFrequenceSet	0x84
DirectionSet	0xAA
MotorSetA	0xA1
MotorSetB	0xA5
Nothing	0x01

D7	D6	D5	D4	D3	D2	D1	D0
-	-	-	-	-	-	-	-

Data byte 2:

contains data relevent to the command

If this byte not used it shall contain 0x01 (Nothing)

D7	D6	D5	D4	D3	D2	D1	D0
-	-	-	-	-	-	-	-

Data byte 3:

contains data relevent to the command

If this byte not used it shall contain 0x01 (Nothing)

D7	D6	D5	D4	D3	D2	D1	D0
-	-	-	-	-	-	-	-

4.3.3 I²C Sensor Board

Author: Michael Sved, Patrick Delventhal

receiver	transmitter	length	data	dataformat
AltIMU	ATmega2560	2 Byte min	Adress byte, Data byte(s)	8bit char

4.3.3.1 Init Protocol

3 control messages shall be sent to correctly initialise the sensor:

Each message must consist of the adressbyte and 2 data bytes. The first databyte is the control register to access, while the second data byte defines the value of the register.

Adressbyte: (0b00011101)

A0	A1	A2	A3	A4	A5	A6	A7(R/¬W)
0	0	0	1	1	1	0	1

The R/W bit must be set to read.

Data byte 1:

- 1. message: 0x20 → Register 1
- 2. message: 0x24 → Register 5
- 3. message: 0x26 → Register 7

Data byte 2:

- 1. message: 0x31 → X-axis enabled, Y-axis disabled, normal power mode, 25Hz refresh rate of acceleration
- 2. message: 0x08 → Latch interrupt request on INT1_SRC register, with INT1-SRC regsiter cleared by reading INT1_SRC itself
- 3. message: 0x00 → Refernece value for interrupt generation. Default value: 0000 0000

4.3.3.2 Read register

To read the x-axis value 2 registers must be accessed. The 16bit signed int value is stored in 2 8 bit registers representing the high and low byte.

This time 2 messages with 2 bytes must be sent with 1 data byte representing the register to be read.

Data byte 1:

1. message: 0x28 → OUT_X_L_A, Low Byte Acceleration Axis X

2. message: 0x29 → OUT_X_H_A, High Byte Acceleration Axis X

X-axis acceleration data is expressed in two's complement.

Each message returns the value of the register selected. These values shall then be stored in a 16bit signed int called senX as describes in the section below. The high byte must be bitshifted by one byte when the low byte is ORed with the 16bit signed int.

Further functions may be implemented in the future to read status of error registers or other sensor values. This is however not planned at this time.

To convert the raw 16bit signed integer to a degree value between -90° and 90° one can use the following fomular:

'Raw data'/(2^14/90) = degree

4.4Globalvariables

Author: Linus Andrae, René Wolter

4.4.1STM32

Name	Meaning	Domain
p	proportional part of the pid controller: Kp	float
i	integrative part of the pid controller: Ki	float
d	derivative part of the pid controller: Kd	float
rotorControl	Value for the motor control (set by the PID) to be read as percent values when controlling the rotors the sign (+/-) defines which rotor to control	signed char (+/- 100)
setpoint	setpoint for the angle in degrees (0°=horizontal, value is set by a script or fixed inside a header)	float
reading	actual reading of the angle in degrees (specified by messages of the arduino over RS232 Serial)	float

Name	Meaning	Domain
rs232DataIn	a char holding the sensor readings, received over RS232 interface	char
hasError	Indicates that an error occured	int, 0 if not in error state
CALL_INTERVAL_IN_MS	the time each cycle runs	int
currentScheduler	variable to let the scheduler know what functions to call. See: STM32-Scheduler (see page 41)	{SCHEDULE_1, SCHEDULE_2, ERROR_SCHEDULE }
defaultScheduler	The scheduler selected by the user (fixed at runtime) the scheduler should call. See: STM32-Scheduler (see page 41)	{SCHEDULE_1, SCHEDULE_2 }

The values are declared and initialized in acd/stm32/globalVars.h .

4.4.2Arduino

Name	Meaning	Domain
rotorControl	value for the motor control (specified by the messages from the STM32 over RS232)	signed char (+/-100)
dataReceivedRS232	indicates if new data is received by the RS232	boolean
dataReceivedI2C	indicates if new data is received by the I2C	boolean
senX	x value from the accelerometer	signed short int
mot1	value for motor 1 regulation	signed char
mot2	value for motor 2 regulation	signed char
reading	angle calculated from x,y and z sensor values	float
rs232DataIn	a character array with the sensor data float	char[4]

Communication between motordriver and the I2C driver as well as the sensordriver and the I2C driver should be made by function calls and not by the global scheduler.

5 Overview of the Code Repository in Git

- adc
    - doc
        - tex-documents
        - etc.
    - src
        - arduino/arduino/ardunio/sketch_jun05a
            - RS232
            - I2C
            - lsm303
            - MotorDriver
            - ArduinoScheduler
            - GlobalVar
            - Dispatcher
            - sketch (Main)
        - stm32
            - RS232
            - PID
            - ...
    - lib/cots

6Component Arduino

6.1Introduction

Author: Sören Stingl

This chapter describes how the Arduino microcontroller is used in the project. Which functions will be uses, how the scheduler works and how everything was implemented.

6.2Scheduler

Author: Lennart Lindenberg

According to the requirement, the Arduino should wait for the SysTick of the STM32 to first synchronize with it and finally to respond to it after a certain time. If the Arduino does not receive a SysTick, he may turn off the motors or simply wait for the next SysTick. Depending on the communication to the STM32 and the state of the motors, the scheduler must therefore allocate computation time to certain tasks.

This requirement is handled in the files "ArduinoScheduler.h" and "Sketch.cpp", which are contained in the folder "acd\src\arduino\Arduino\Arduino\sketch_jun05a".

After completing the cooperative method for receiving the SysTick, one of three schedule plans is selected by the scheduler method "setSchedule":

1. If the SysTick is received within a designated timeslot, the following tasks are consecutively executed: "setSchedule", "reciveMsg", "motorUpdate", "sensorUpdate", "emitMsg", "rs232SendBytes", "rs232ReadByte". "setSchedule" immediately follows "rs232ReadByte" to synchronize the Arduino (the slave).The other tasks follow at a specific time.
2. If the SysTick is not received and the motors are not turned off, the tasks "setSchedule", "motorUpdate" and "rs232ReadByte" will be executed immediately after each other. (To turn off the motors)
3. If the SysTick is not received and the motors are turned off, the tasks "setSchedule" and "rs232ReadByte" will be executed immediately after each other. (To wait for the next SysTick)

For the call of tasks at a certain time, timerTicks are provided, which are set by the interrupt routine of the timer 1 of the Arduino. An array lists the tasks that can be performed by the Arduino. For each task is specified how many TimerTicks it will take to execute it. The scheduler routine is timed and cooperative.

Since communication between the STM32 and the Arduino via the RS232 interface was not yet possible, only sample values for interrupt triggers and delays of the tasks were selected.

Author: Niklas Wahrenberg

The call to modules that were realized using classes, those were motor driver and sensor driver, had to be wrapped inside a function belonging to no class. That was necessary to avoid a type error because the struct taskInfoElem_t used by the scheduler has it’s function pointer of type “void (*)(void)” but a function pointer to a class member function would be a e.g. “void(MotorDriver::*)(void)” and cause an error.

6.3RS232

Author: Niklas Wahrenberg

This module wraps calls to the following functions of the Serial Library:

- begin()
- setTimeout()
- readBytes()
- write()
- end()

A detailed explanation of those methods can be found here: https://www.arduino.cc/reference/en/language/functions/communication/serial/

The RS232 driver can be initialized using „rs232Init(int baud, int timeout)“, as the parameter names suggest this method sets the used baud rate and the time to wait for data to become available when trying to read. The method „rs232ReadByte()“ reads the motor control values and presents them to the message dispatcher via the global variable „rotorControl“. The bool „dataReceivedRS232“ is filled with the return value of the bulk read method readBytes(), which is the amount of bytes received. Even though only one byte is sent by the STM32 bulk read is used because only that method returns how many bytes have been read. By checking the global variable „dataReceivedRS232“ the message dispatcher can determine whether the values in „rotorControl“ are new or from a previous round. After the other modules have actuated the motors and read the new angle, that value has to be written into the global variable “angleOut“ so that “rs232SendBytes()” writes it back to the STM32 as input for the next PID round.

We have confirmed, that values send using this module are received by the STM32. Receiving values from the STM32 seemed to not work, for „dataReceivedRS232“ remained 0 all the time, indicating that zero bytes have been read. Commenting out the scheduler and putting a print of „Serial.available()“ directly in the main loop revealed that the Serial Library of the Arduino was unaware of any traffic. A measurement with an oscilloscope proved however that there actually was data on the line, this ensured that a writing problem on the STM32’s side was not the cause and narrowed down the search. We were unable to find out what the reason for this behavior was. It could have been a defective contact or a loose connection, which already had been a problem in other parts of this project before. However, we checked and did not find anything in that regard.

6.4Message Dispatcher

Author: Dennis Hilmer

The software component Message Dispatcher on the Arduion is use to interpred the received data from the r232 and to interpred the data from the sensor that will be send to the R232. For this the component have two functions that will be invoke from the Scheduler. The function receiveMsg() and emitMsg(). receiveMsg() reads the global varibale roteControl and and interpred this. The variable roterControl is a signed char array with the lenght 1. The values are interpreted as follows:

- roterControl > 0 => mot1 = rotereControl
- roterControl < 0 => mot2 = -rotereControl

The value from roterControl can not be over 100 or less then -100. The variable mot1 and mot2 will be use from the motor driver to set the left or the right motor speed. In this case the values were misinterpreted. The value 100 from mot1 or mot2 does not match 100%motor power. The value 255 corresponds to 100%. This mistake shoud be fix in a new version. The function emitMsg() takes the variable senX that was set from the sensor before and compute a angle. float var = senX*360/65536 The value 65536 is the resolution from the sensor. The value from the variable var

can not be higher as 100 oder less then -100 and will be cast to a signd char. This is to reduce the data traffic to the r232. A higher value as 100 or less then -100 have no recognizable effect to the functionality.

7Drivers for Arduino-compatible Microcontroller

In this section we describe the arduino drivers for the two motors "Crazyflie Nano Quadcopter", the attitude sensor "AltIMU 10 v3" and the RS232 Driver.

In this semester we resuse most of the code and documentation from the last semester with some small changes for our needs. this included adding an additional function to set the speeds of the motors.

7.1I2C driver

The I2C library remains unmodified from the previous semester and is available in the Git repository.

7.2InvenSense MPU-6050

Author:Patrick Delventhal

Using the same properties and the same setting as the previous group we found discrepancies in the called registers. According to previous group, two registers contain the required values. According to the datasheet however, the required X-axis values are shifted by one register address.

The previous group had calculated the angle of the arm from two accelerometer sensors with trigeometric functions. But this only produces usable vales within a very small range (arm is horizontal to an angle of plus minus about 20 degrees). In addition the orientation sensor value presented extreme amounts of noise. At constant position it varried between + - 160000. Maximum values were + - 32000. According to the data sheet of the sensor, an internal calibration is necessary for the best measurement result. This requires its own internal clock. But the project requirement was that the clock signal comes from outside, so that an answer is only given if the data is requested. The decision was then made to switch to an alternative sensor that only provides the required values when requested and meets the requirements. With a small test program, the required angle could be read out very quickly and easily with the alternative sensor.

7.3Attitude Sensor AltIMU-10 v3

Authors: Michael Sved, Patrick Delventhal

The Ardunio library written for the AltIMU-10 v3 was not used due to the fact that it would have been to difficult and time consuming to rewrite the library to use the modifief I2C library. As a result a simple custom driver library was written using the modified I2C library.

library only contains two functions:

```
void lsm303Init();
void lsm303Read();
```

These two function are all that is required to initiate the sensor and to read the x-axis value. Timeout and error handeling is controlled and monitored via the I2C bus, rather than the sensor library. due to the time contraints it

was decided to only implement a library for the functions we actually require. Reading the acceleration values for y and z axis are therefore not needed. Nor are magnetometer, gyroscope or barometric readings needed for this project. Even if the sensor board offers these values.

Actually obtaining the values is then as easy as executing the init function followed by the read function and declaring the global variable as per the interface design document.

full code is stored in the Git repository

7.4Grove - I2C Motor Driver V1.3

Authors: Christian Zöller, Eike Diekmann, Hermann Wafo

Modified by: Michael Sved

7.4.1Re-engineering of existing Motor Driver

Since the existing motor driver doesn't meet realtime systems requirements, it has been necessary to think about using appropriate I2C library with functionalities that have got optimized for our purpose. The selected I2C library needs to fit the structure and functionalities of the existing motor driver, so it brings us to the point of writing needed functions within it. Then we rewrote the existing motor driver using fitted I2C library.

The resulting structure of the motor driver differs just a little bit from the existing one, but the most important functionalities are well-covered.

7.4.2Interfaces for Motor Operation

```
...
class MotorDriver {
    private:
/*...*/

    public:
            MotorDriver(I2C &i2c, const uint32_t timeout);
            void initMotors(void);
            // Initialize I2C with an I2C address you set on Grove - I2C Motor Driver v1.3
            // default i2c address: 0x0f
            void begin(unsigned char i2c_add);
            // Set the direction of both motors
            // _direction: BothClockWise, BothAntiClockWise
            uint8_t direction(unsigned char _direction);
                        uint8_t MotorDriver::set(uint8_t motor, uint8_t speed);
                        //Sets speed for Motors.
                        uint8_t set(uint8_t motor, uint8_t speed);
                        uint8_t setSpeed(uint8_t speedLeft, uint8_t speedRight);
            //Update speed of Motor with set speed
            void update();
            //get returnStatus: returns ErrorCode: 0 -> No Error, >0 Error
            uint8_t getReturnStatus();
            // Set the frequence of PWM(cycle length = 510, system clock = 16MHz)
            // F_3921Hz is default
            // _frequence: F_31372Hz, F_3921Hz, F_490Hz, F_122Hz, F_30Hz
            uint8_t frequence(unsigned char _frequence);
            // Stop motors
            void stop();
};
```

7.4.2.1Usage

This class is optimized for using within a scheduler.
For instantiation there will be needed the I2C and a maximum time (timeout) this task is allowed to get sensor data.
The scheduler first calls ***update()*** to get current sensor values.
After updateing, sensor values can set by calling ***setSpeed(uint8_t motor, int16_t speed*** where motor and speed is speciefied as the speed of correspondig motor 1 or 2 from speeds of (-100 to +100).
Calling ***getReturnStatus()*** reveals whether an error occured or not. 0 = no Error.

7.4.2.2Timeout

The scheduler runs the sensor driver and is awaiting an answer within a certain time. If the sensor driver can't receive an answer it must not block the software.
Therefore there is a timeout. The timeout is defined by the scheduler and passed to the Motor Driver Constructor.
If the defined time is exceeded, the Motor Driver stops the attempt sending any further data.
If a timeout or an error occured it can be read by calling the method ***getReturnStatus().***

7.4.3Mapping of the motor values

The speed range of the two motors(Motor Crazyflie Nano Quadcopter) is the same 0 to 255. Both motors speeds can be accessed from outside and setted directly using the speed function, that returns a status 0 when the function successful completed without timeout occured. Statuses 1 to 7 are also returned when waiting for completion/ACK/NACK while adressing slave, sending/receiving data to/from slave. It's also possible to get other statuses 8 to 0xFF which are well-documented in the datasheet of the used 8-bit Mikrocontroller ATmega2560.

Path for motor driver example code
An example is located at acd2.1/Arduino/ExampleMotorDriver/ExampleMotorDriver/ExampleMotorDriverData/ExampleMotorDriverData.cpp

8Component STM32

•

8.1Introduction

Author: Sören Stingl

This chapter describes how the STM32 microcontroller is used in the project. Which functions will be used, how the scheduler works and how everything was implemented.

8.1.1Setting up the Development Enviroment

Author: Linus Andrae

To program the STM microcontroller we used the System Workbench for STM32. To downlaod and install: http://www.openstm32.org/System%2BWorkbench%2Bfor%2BSTM32 (Registration required and **NO HTTPs** so use a password you don't use anywhere else!)
A good guide how to setup a stm32 project can be found

here[4]

8.2Main

Author: Linus Andrae

The main function is the first function that is called after a system reset.

8.2.1Initalisation

The main will fist call the `HAL_Init()` function of the STM32 controller to initialize the Hardware Abstraction Layer functionality such as timers, drivers and GPIO.
after that the `initHW()` function is called that also is part of the `main.c` file. The `initHW()` function enables timers for the different GPIO banks of the stm, initialize the onboard LED and call the `rs232init()` function that sets up the UART communication ports for usage.

After the initialisation is complete the SystemTick interrrupt is started by calling `HAL_InitTick()` and the onboard LED is turned on (since it is Green proper operation is indicated by the LED beeing on, an error state will turn of the LED).
The `resetValues()` function will then be called to initialize the PID and scheduler values to defaults, this function is defined in `globalVars.c` and if a user /developer wants to change the device operation this should be the first place to look.

Since the device couldn't be tested with the arduino the PID Values probably need adjustment and also the time for each schedule iteration is set to 2.5 seconds wich is for testing purposes only and should be adjusted to just above the WCET and be in the range of some hundreds ms.

4 https://www.carminenoviello.com/2015/06/04/stm32-applications-eclipse-gcc-stcube/

8.2.2Main Loop

After the initailisation the main loop is started. Each iteration a new end time `t` is set by adding the `CALL_INTERVAL_IN_MS` to the current system time.

After that the `runSchedule()` function is called (see below).

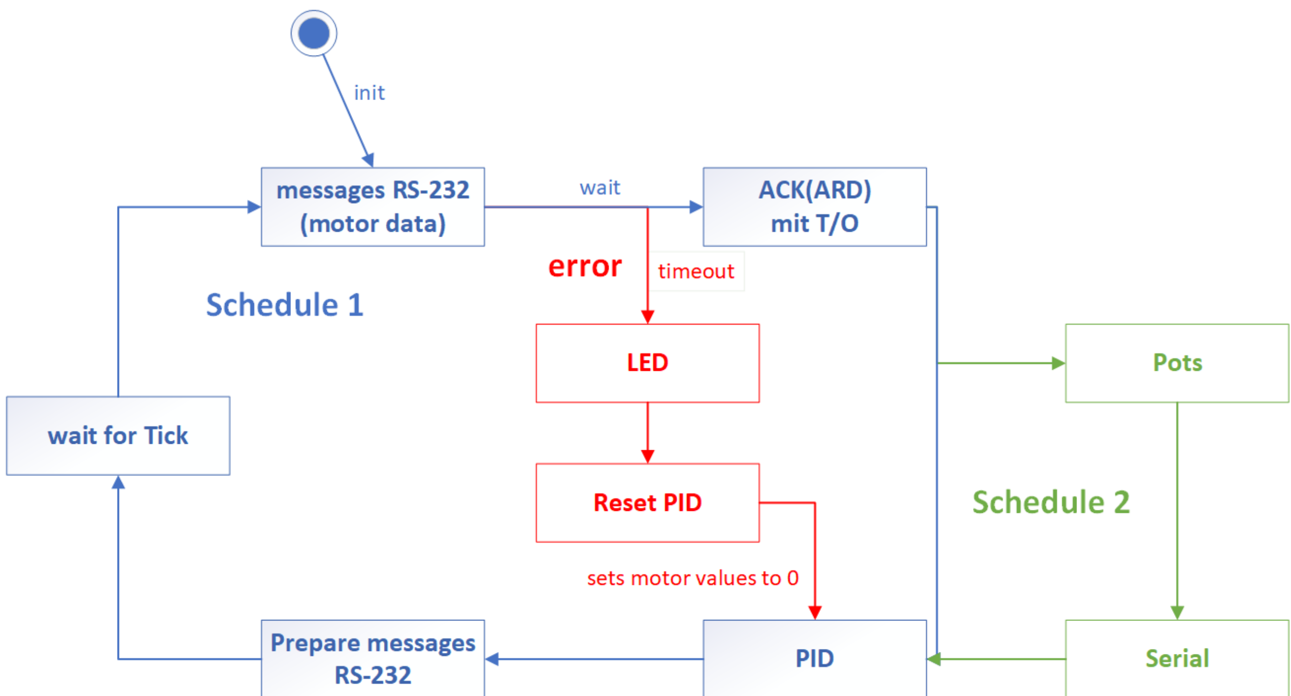
The last step in each loop is to wait for the system time to cross the `t` threshold

8.3Scheduler

Author: Sören Stingl

The following diagram shows a model of the STM32-Scheduler:

8.3.1Diagram



8.3.2Schedule without Pots and Serial

The messages that contain the motor control data that are sent via the RS-232 link are initialized with the start values and sent to the Arduino. This first message also determines the Tick for the Ardunio. The STM32 shall wait for the acknowledge (ACK/ARD) message from the Arduino with the sensor data. Once the sensor data is received the PID regulation can be processed. The PID process shall produce new motor data values and a new message shall be prepared. A new system tick must occur before the new message is sent. Once this system tick is received, the prepared message shall be sent to the Ardunio.

In the the event of a timeout while waiting the the ACK message from the Ardunio, an LED on the STM32 board shall be activated to give a visual indication that an error as occurred. In conjunction to this, the PID regulator shall be reset and the motor control data will be reset to zero as well.

8.3.3Schedule with Pots and Serial

Schedule 2 shall be executed when the PID vales are to be set via external potentiometers, rather then fixed values set in code. The digital vales determined by the position of the potentiometers shall be transfered to the PID regulator as well as an external serial interface.

8.3.4Scheduling algorithm

Author: Linus Andrae

The Scheduler uses the same principle that was taught in the lecture.
Inside the `globalVars.h` header the two variables `currentScheduler` and `defaultScheduler` are defind.

The `defaultScheduler` defines if Scheduler one or two is used, this has to be set befor compiling the programm and can't.
The `currentScheduler` is used by the scheduler to switch into the error mode and back.
The Scheduler can use three modes which are `SCHEDULE_1`, `SCHEDULE_2` and `ERROR_SCHEDULE`, that are also defined in `globalVars.h` and that are set as default values in `globalVars.c`

For indepth documentation see the source code documentation in `globalVars.h`, `globalVars.c`, `scheduler.h` and `scheduler.c`

8.4RS232 driver of the STM32

Author: Linus Andrae

The RS232 protocol is used to connect the STM32 to the Arduino microcontroller, to send and receive telemetry, and telecommand packets.
Since the full communication of couln't be tested since the arduino couln't receive data the communication bewteen the two boards couln't be tested.
But we used an ocilloscope to see if data could be send and we send data from the arduino board using different software to test if data could be received both was verified.

The RS232 of the STM is not that easy to initalised since the L-Series of STM boards aren't that well documented (see Basics / Structure (see page 10)#STM32Side).
To initailize the UART pins of the board a `UART_InitTypeDef` stucture has to be created and all values need to be set (see rs232.c).
Since the GPIO pins need to be initialized correctly the user has to reimplement the `HAL_UART_MspInit` function and add a part that enables the clock for the ports in question, the GPIO pins them self.
After that is done a `UART_HandleTypeDef` needs to be created to choose between the different ports (we choose to use `USART1` since it is acually connected to a GPIO Pin).Calling `HAL_USART_INIT` will initialize the ports to be used as uart pins.

The rs232.h will provide the functions `rs232readByte` and `rs232writeByte` that both work with a timeoutand will return if an error occured.

`rs232readByte` reads data to the global variable `globalVars.rs232dataIn.`
`rs232writeByte` sends the data contained in `globalVars.rotorControl.`

8.4.1Message dispatcher

The Message dispatcher is the component that calls the rs232 functions, converts data and puts the scheduler into an error state if the data couldn't be send or received.

The function waitForAck is called by the scheduler after the new control data has been sent to the arduino and waits to receivce the sensordata from the arduino. If an error occured the scheduler is set to `ERROR_SCHEDULE` and the `globalVars.hasError` variable is set to 1.
If no error has occured and the system is in an error state the function will set the scheduler to normal and change the error variable.
After data has been received the angle is calculated (`calcAngle`)since the PID controller needs a floatingpoint angle and the data is received as a signed char.

8.5Onboard LED

Author: Linus Andrae

The onboard LED shows errors by turning off. The functions `showErrorOnLED` turns the LED off. The `turnOffErrorOnLED` function turns it back on if the system has been recovered from an error by receiving data from the arduino.
Both can be found in `controlLED.h` and `controlLED.c`.

The scheduler will turn the LED off. The message dispatcher will turn it back on if it received data and was in an error state before.

8.6Attitude Control (PID)

Author: René Wolter

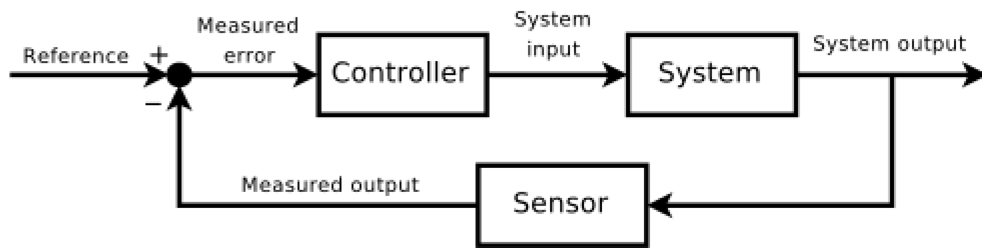
The following paragraphs "Control loops in general" and "PID Controller" were copied from the previous documentation[5], but modified and expanded.

8.6.1Control loops in general

Basics

A control loop is a mechanism that allows a system to be controlled dynamically. By means of a comparison of target value and actual measured value. It adjusts the current value of a system using an actor and takes the error value (difference between target and measured value) into account. Picture 1 shows the systematic structure and context of the components which are explained in more detail below.

5 https://pforge.eso-io.com/confluence/pages/viewpage.action?pageId=109610901



Picture 1: Standard control loop

Controller

This is the part of the control loop that computes how to take action to correct the system deviation, taking into account the dynamic properties of the controlled system.

System

The System is the actual system which has to be controlled. It has a connection to at least one actor to influence the system in the desired way. Moreover it is observed by at least one sensor which measures the status of the system.

Sensor and Measured output / Process variable (PV)

The sensor measures the current value of the system. The measured value (measured output) is passed on in a suitable form.

Reference / Setpoint (SP)

The Reference, also called Setpoint, is the target value that is desired to be reached by the control loop. It is defined by an external source.

Measured error

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

The measured error is the deviation of the reference value (SP) and the measured value (PV(t)). It is the value by which the system still has to be changed at the current point in time to reach the target value.

System input / Controller output (CO)

The System input a value calculated by the controller. Its function is to define how strongly the actors are to influence the system to correct its status quo.

System output

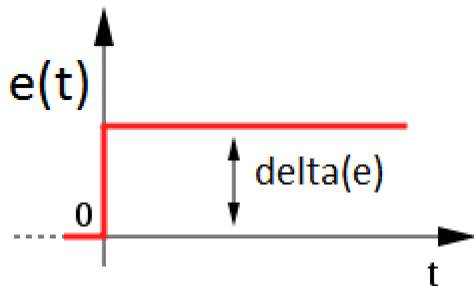
The System output are the values set by the system to control the peripherie. This is mostly the System input values converted into control commands for the actors.

8.6.2PID Controller

The following images describe the behaviour of a PID controller based on an example of a sudden jump (step) in the error value and the controllers response. Picture 2 shows the sudden jump of the error value.

The PID controller is a single-input, single-output control loop. It uses a single value of measurement to compare with the target value and computes a single value of reaction for the actors to work towards the desired status. This is repeated continuously.

Its advantages are that it is easy to implement, usually effective and can be used for all SISO systems (single-input single-output). Its disadvantages are that it can't be used for MIMO systems (multi-input multi-output) and the tuning of parameters can be time consuming.



Picture 2: error graph of a step

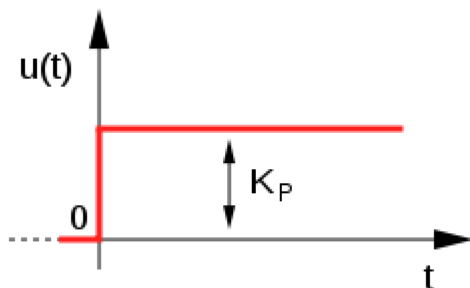
8.6.2.1 Proportional term (P)

$$P = K_P e(t)$$

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. The controller multiplies the error e(t) by the proportional gain Kp to get the controller output. The step response of the proportional term is shown in picture 3.

Advantage: The proportional term tries to reach the setpoint as fast as possible. In many cases this term alone is sufficient to create a working controller (P-only controller).

Disadvantage: If the proportional gain is too large, the process variable will begin to oscillate around the setpoint. If Kp is increased further, the oscillations will become larger and the system will become unstable and may even oscillate out of control.



Picture 3: proportional term, step response graph

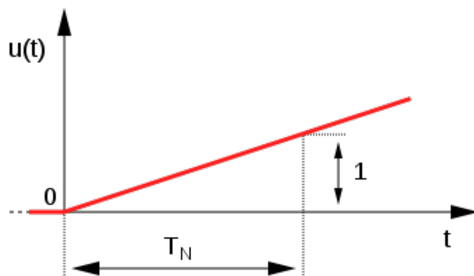
8.6.2.2 Integral term (I)

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

The integral term **I** grows with both the magnitude of the error and its duration, as shown in picture 4. Ki is multiplied with the integral value of the error.

Advantage: This term can accelerate the movement of the process towards the setpoint, if an error is not corrected over longer time periods.

Disadvantage: 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. The integral term also takes a lot of time to take influence. Therefore it can be time consuming to find a fitting value for Ki.



Picture 4: integral term, step response graph

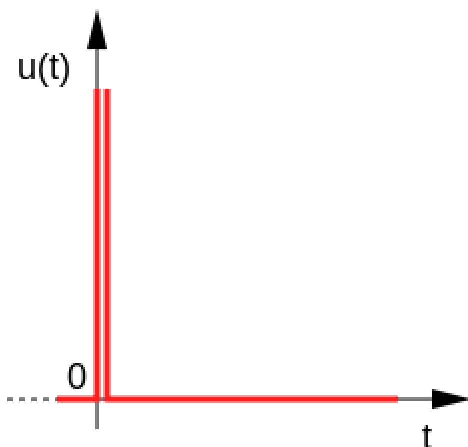
8.6.2.3Derivative term (D)

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

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

Advantage: Increasing the *derivative time (Kd)* parameter will cause the control system to react more strongly to changes in the error term and will increase the speed of the overall control system response.

Disadvantage: The derivative doesn't consider if e(t) is postive, negative or how much time has passed, just how fast e(t) is changing.



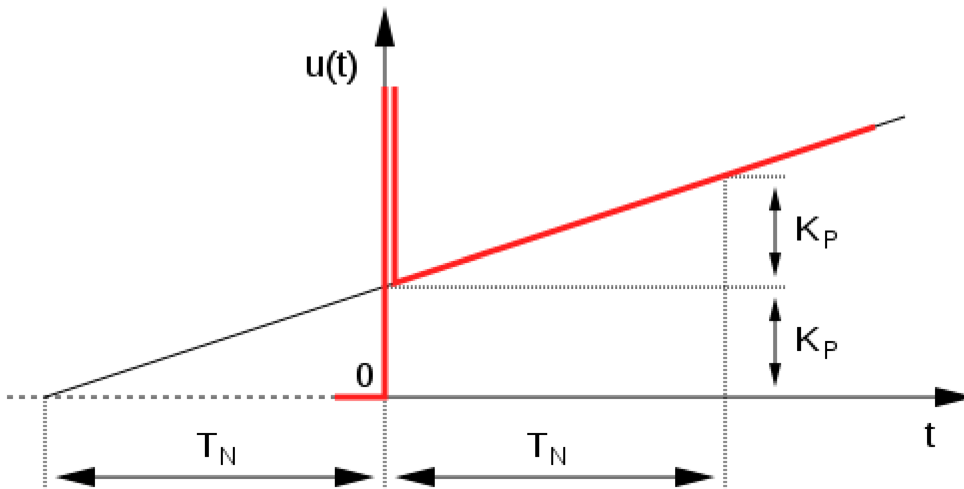
Picture 5: derivative term, step response graph

8.6.2.4Combination of all terms: PID controller

A PID controller is build by combining the proportional, integral an derivative term and summing them up. The resulting value is the controller output that the system

The proportional term considers how far the current process variable is from the setpoint and changes the output to reach the setpoint as fast as possible. This makes up the main part of the controller, the derivative and integral term are mainly used for 'fine-tuning' the controller. The derivative term acts against the oscillations caused by the proportional term, because it considers how fast e(t) is changing. The integral term considers how long and how far PV has been from SP by continually summing e(t) over time, so it acts like a moving bias that eliminates the offset over a long period of time.

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



Picture 6: PID step response graph

8.6.3Implementation

As the implementation of the previous group based upon different interfaces and assigned different responsibilities to the devices, the development group decided not to use the code as a base for the STM32 side. Rather it was decided to start a new project from scratch and integrate old code only where needed.

The PID controller features an interface consisting of 2 functions: compute() and resetPID(). Both functions have neither a return value nor required parameters as they use to global variables defined in globalVars.h for communication with each other and other functions. See the Interface definition (see page 26) document for detailed information.

compute() calculates the output of the PID-controller. The result will be a value inside the bounds of [-100, 100]. The function reads from:

- globalVars.setPoint
- globalVars.reading

- globalVars.p
- globalVars.i
- globalVars.d

and writes to:

- globalVars.rotorControl

resetPID() is supposes to be called on start and whenever there is no proper input data at hand. This function will reset all values in order to reach a result of 0 which means that the rotors will turn off when calling compute() afterwards. It reads from:

- globalVars.setPoint

and writes to:

- globalVars.readings

For indepth documentation see `globalVars.h`, `pid.h` and `pid.c`

8.6.4Requirements for Scheduler

Sample time awareness is important for the PID controller.

The implementation requires the function compute() to be called in a fixed time interval to function correctly. This is due to the fact that the integral and differential terms are time-based terms and those terms usually need information about the time difference since the last calculation. This implementation does not take the time difference to the previous call into consideration as it is bound to be a constant value. The derivative and integral term function exactly the same without the time difference value, but only if it is the same value each time. The only difference in the calculation is the constant time value that is missing in the equation, but this can be accounted for by altering the Ki and Kd values. By integrating the constant time value into Ki or Kd, it is possible to effectively remove the time delta value from the computation.

The PID controller does not have a hard time restriction for the interval, in which it is called, besides for it to be constant. But the results and the reaction time improve accordingly to a fast call interval. Therefore the main loop should run as fast as it can guarantee to finish each round.

8.6.5Improvements of the algorithm

The code for the PID controller was created completely from scratch, but it attacked the same challenges as the previous group and used the same improvement methods, as those are limited for a PID controller. Therefore it features the description of the previous group with a few changes.

8.6.5.1Derivative kick problem

This was copied from the previous group (Documentation[6]) as it was implemented in the same way.

The formula for the derivative term is the derivation of the error value.

$$\frac{de(t)}{dt}$$

Since the error value is defined by:

[6] https://pforge.eso-io.com/confluence/pages/viewpage.action?pageId=109610901

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

it means the following for the derivative:

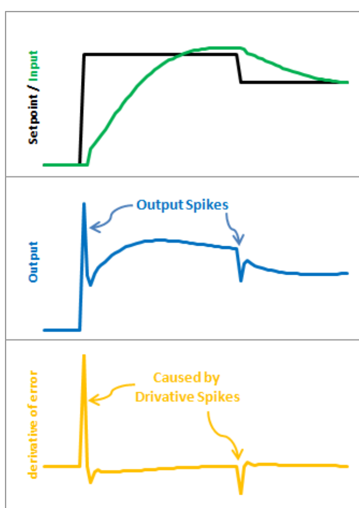
$$\frac{de(t)}{dt} = \frac{d(SP - PV(t))}{dt}$$

When it comes to a setpoint change, the error increases rapidly and results in a spike of the derivative, as shown in picture 7. It is called the derivative kick.

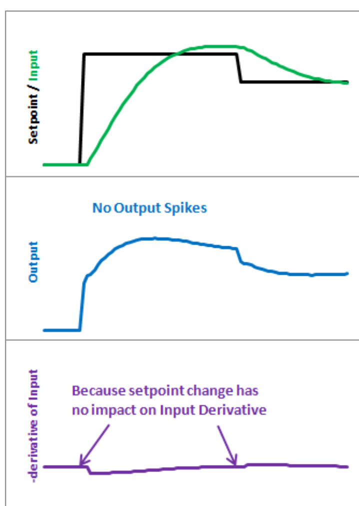
Due to the fact, that the derivative part only needs to consider the change of the process variable, the derivative kick can be prevented by eliminating SP from the formula.

$$SP = 0 \rightarrow \frac{de(t)}{dt} = \frac{d(-PV(t))}{dt} = \frac{dPV}{dt}$$

That means that the derivative depends only on the last input. This is called "Derivative on Measurement". In picture 8 you can see that the setpoint change doesn't cause spikes anymore.



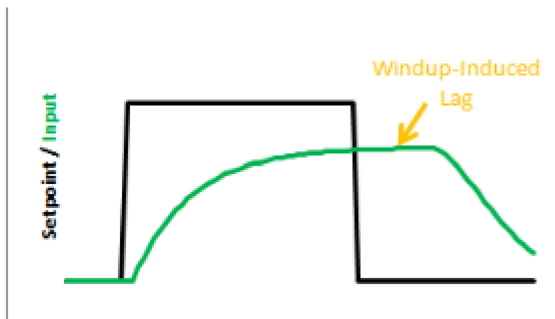
Picture 7: derivative kick



Picture 8: Correction of derivative kick

8.6.5.2Windup Reduction

One possible problem during controlling is the windup. If the setpoint is not reachable by the system (e.g. due a movement restriction in the system), the integral will continue to grow in absolute value and after a while it adds up to a very large number. If the setpoint is reached after some time (e.g. by removing a restriction or changing the setpoint), the desired value will be passed due to the cumulated value of the integral term. This is called windup. The controller can't regulate the process until the error changes sign and the integral term shrinks sufficiently which might take a lot of time. Therefore the controller reacts much later on a setpoint change, so it leads to a windup-induced lag as shown in picture 9.



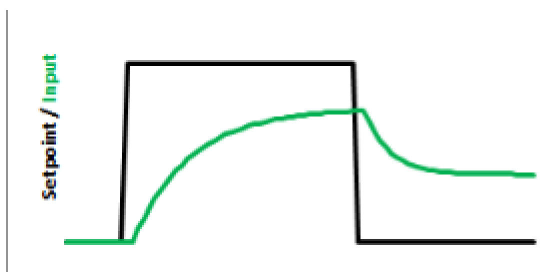
Picture 9: windup lag

To prevent the windup it is necessary to let the controller know about the restriction in output and react accordingly to it.

This problem was adressed differently by the previous group than we would have thought. Although it will have a similiar effect, it is not the same and we believe it to be less effective. The previous group defined a limit for the integral term and clamped the value to it, so the term could not take a value outside these boundaries. This solution does not prevent a windup, but limits it to a certain value, which we considered not to be the optimal solution.

Therefore we decided to implement it exactly the way it is described in "Patterns for Time-Triggered Embedded Systems (2001), page 869ff" by Michael J. Pont. Whenever the output is at its maximum (or minimum) and therefore the rotor is operating at its limit, the integral term will not be changed in that turn. For the computation of the rotor control value the integral term will contain the current error value, it is only afterwards that it will be checked whether the new integral term will be saved or the old one. This prevents a windup due to an unreachable setpoint, but still enables the integral term to work as designed.

An additional advantage of this implementation over the previous one is that no limits need to be defined and the optimal value of these limits would even be unknown. The limit must not be too small in order to not prevent the integral term from taking any effect and it must not be too big in order to efficiently prevent a windup effect. Moreover this solution takes effect directly when the output is at its limit value. This results in a faster reaction time, than waiting for the integral term to reach a defined limit.



Picture 10: Correction of windup lag

8.6.6Adjustable PID parameters during runtime

There was an optional feature request for adjusting the PID parameters during runtime using potentiometers. For this function, we decided to use the solution of the previous group.

The code was copied and slightly altered, but not in any way to affect the function. We did not succeed in reading analog values from ADC. This is likely due to the hardware not being correctly configured as we did not create our project with the STM-32 project generator, but instead created it using the Eclipse plugin. The documentation did not help us to resolve this problem.

Everything apart from the connection to the ADC is already prepared. This includes the conversion of the read values and their storage.

The Analog Reader features an interface consisting of one function: readTuningsFromAnalog()

The function has neither a return value nor required parameters as it uses global variables defined in globalVars.h for communication with other functions. See the Interface definition (see page 26) document for detailed information.

readTuningsFromAnalog() reads the potentiometer values and converts them into new parameters for Kp, Ki and Kd. The function does not read any parameters and writes to:

- globalVars.p
- globalVars.i
- globalVars.d

For indepth documentation see `globalVars.h`, `pidRead.h` and `pidRead.c`

8.7Changing Setpoint in defined interval

There was an optional feature request for adjusting the setpoint during runtime to different values in order to demonstrate the functionality of the system. Those changes should happen in an unspecified, but fixed interval.

The Setpoint updater features an interface consisting of one function: setPointUpdate()

setPointUpdate() is required to be called every iteration of the main loop in order to function correctly, as it counts the iterations in order to change the setpoint after the interval time has run out. It reads from:

- globalVars.CALL_INTERVAL_IN_SEC

and writes to:

- globalVars.setpoint

For indepth documentation see `globalVars.h`, `setpoint.h` and `setpoint.c`

The change can occur every x calls or, if the call interval is known, after the running off of a time value. The functionality is not provided by running in an own thread because we desire to build a guaranteed real-time system und this is hardly achievable using multithreading.

The function uses a few pre-defined setpoints that will be used in that order. These setpoint can be changed, but will require a rebuild of the affected source files. The pre-defined setpoints are:

1. -50°
2. -25°
3. 0°
4. 25°
5. 50°

9The SysTick-Timer of the FreeRTOS

Author: Linus Andrae

9.1What does the SysTick-Timer do?

The SysTick Timer is a Component that increases a variable and provides a timer signal every 1 ms. SysTick is part of the Hardware Abstraction Layer (HAL).
The Timer must be initialized when the system is started and can be disabled and enabled at any time.

The timer also provides a time base for other functions for example for timeouts.
For providing a tick an interrupt is used. Every ms the ISR of the systick is called and the internal tick variable `uwTick` is increased.

9.2How to Configure the SysTick-Timer?

To Configure the SysTick timer the function `HAL_RCC_ClockConfig()` (see documentation) is used. After changing the clock configuration the `HAL_InitTick()` function must be called.
Since we use the default ClockConfig all that has to be done is to call `HAL_InitTick()` with a interrupt priority (lower priority means higher priority), since the timer should have the highst priority we use -1 as the timer priority.

9.3What are side effects of changeing the systick time?

If the frequency of the systick is changed, the programmer needs to change the timebase to another timer than SysTick, because other components (eg. `PPP_TIMEOUT_VALUE`) uses 1ms as a base, not changing the time base would result in non accurate timeout periods.

The full documentation of the SysTick-API be found inside the manual that can be downloaded from the ST website http://www.st.com/resource/en/user_manual/dm00132099.pdf (visited 2018-05-19)

10Using the STM32 SysTick as a timebase for the Arduino board

Author: Linus Andrae

The System is designed in a master- slave manner.
The STM32 acts as the master device providing a global time to sync the two devices by sending a message over the RS232 Interface.
The STM32 will send the motor controll data every scheduling round. This can be used as a global time base since it is send right after the scheduler is called each round after the global systick has been emited.
The Arduino-Scheduler (see page 54) is designed in a way that if the tick is not provided the device goes into a wait state until a packet is received.

The STM32 will continue to send out packets with motor control information if the acknowlage packets are missing, but it will turn off the green onboard LED to notify the user about the error state.

Since the SysTick operates at a 1 ms base (see The SysTick-Timer of the FreeRTOS (see page 53)) the timeinterrupt stays enabled and increases the time since system start every 1ms.
Each iteration of the main loop sets the end time for the current scheduler round (t > then the WCET) and after the scheduler is finished it waits till that time has arrivered to start the next iteration and send the motor control data packet.

11Other components

11.1Introduction

This chapter describes how the other components (sensors, motors etc.) are used in the project

Author: Sören Stingl

11.2Software components

11.2.1I2C driver

The I2C library remains unmodified from the previous semester and is available in the Git repository.

11.2.2Attitude Sensor AltIMU-10 v3

Authors: Michael Sved, Patrick Delventhal

The Ardunio library written for the AltIMU-10 v3 was not used due to the fact that it would have been to difficult and time consuming to rewrite the library to use the modified I2C library. As a result a simple custom driver library was written using the modified I2C library.

Interfaces for sensor operation

```
void lsm303Init();
void lsm303Read();
```

These two function are all that is required to initiate the sensor and to read the x-axis value. Timeout and error handling is controlled and monitored via the I2C bus, rather than the sensor library. due to the time contraints it was decided to only implement a library for the functions we actually require. Reading the acceleration values for y and z axis are therefore not needed. Nor are magnetometer, gyroscope or barometric readings needed for this project. Even if the sensor board offers these values.

Actually obtaining the values is then as easy as executing the init function followed by the read function and declaring the global variable as per the interface design document.

full code is stored in the Git repository

11.3Grove - I2C Motor Driver V1.3

Authors: Christian Zöller, Eike Diekmann, Hermann Wafo

Modified by: Michael Sved

Re-engineering of existing Motor Driver

Since the existing motor driver doesn't meet realtime systems requirements, it has been necessary to think about using appropriate I2C library with functionalities that have got optimized for our purpose. The selected I2C library needs to fit the structure and functionalities of the existing motor driver, so it brings us to the point of writing needed functions within it. Then we rewrote the existing motor driver using fitted I2C library.

The resulting structure of the motor driver differs just a little bit from the existing one, but the most important functionalities are well-covered.

Interfaces for motor operation

```
...
class MotorDriver {
    private:
/*...*/

    public:
            MotorDriver(I2C &i2c, const uint32_t timeout);
            void initMotors(void);
            // Initialize I2C with an I2C address you set on Grove - I2C Motor Driver v1.3
            // default i2c address: 0x0f
            void begin(unsigned char i2c_add);
            // Set the direction of both motors
            // _direction: BothClockWise, BothAntiClockWise
            uint8_t direction(unsigned char _direction);
                        uint8_t MotorDriver::set(uint8_t motor, uint8_t speed);
                        //Sets speed for Motors.
                        uint8_t set(uint8_t motor, uint8_t speed);
                        uint8_t setSpeed(uint8_t speedLeft, uint8_t speedRight);
            //Update speed of Motor with set speed
            void update();
            //get returnStatus: returns ErrorCode: 0 -> No Error, >0 Error
            uint8_t getReturnStatus();
            // Set the frequence of PWM(cycle length = 510, system clock = 16MHz)
            // F_3921Hz is default
            // _frequence: F_31372Hz, F_3921Hz, F_490Hz, F_122Hz, F_30Hz
            uint8_t frequence(unsigned char _frequence);
            // Stop motors
            void stop();
};
```

Usage

This class is optimized for using within a scheduler.
For instantiation there will be needed the I2C and a maximum time (timeout) this task is allowed to get sensor data.
The scheduler first calls ***update()*** to get current sensor values.
After updateing, sensor values can set by calling ***setSpeed(uint8_t motor, int16_t speed*** where motor and speed is speciefied as the speed of correspondig motor 1 or 2 from speeds of (-100 to +100).
Calling ***getReturnStatus()*** reveals whether an error occured or not. 0 = no Error.

Timeout

The scheduler runs the sensor driver and is awaiting an answer within a certain time. If the sensor driver can't receive an answer it must not block the software.
Therefore there is a timeout. The timeout is defined by the scheduler and passed to the Motor Driver Constructor.
If the defined time is exceeded, the Motor Driver stops the attempt sending any further data.
If a timeout or an error occured it can be read by calling the method ***getReturnStatus()***.

Mapping of the motor values

The speed range of the two motors(Motor Crazyflie Nano Quadcopter) is the same 0 to 255. Both motors speeds can be accessed from outside and setted directly using the speed function, that returns a status 0 when the function successful completed without timeout occured. Statuses 1 to 7 are also returned when waiting for completion/ACK/NACK while adressing slave, sending/receiving data to/from slave. It's also possible to get other statuses 8 to 0xFF which are well-documented in the datasheet of the used 8-bit Mikrocontroller ATmega2560.

An example is located at acd2.1/Arduino/ExampleMotorDriver/ExampleMotorDriver/ExampleMotorDriverData/ExampleMotorDriverData.cpp

full code is stored in the Git repository

11.4Hardware components

11.4.1Motors

Author: Michael Sved

Name: DC Motor Crazyflie Nano Quadcopter (BC-CM-01-B)

The motors are mounted to both ends of the the arm with the axis of the motors at 90° towards the arm axis.

The motors provide thrust via a propeller attached to the motor to lift one side of the arm higher than the other, thereby rotating the arm around its center axis.

11.4.2Motor driver

Author: Michael Sved

Name: Grove I^2C Motor Driver V1.3

The Motor driver controls the funtion of the motors. It determines the speed and rotational direction of the motors.

11.4.3Attitude Sensor AltIMU-10 v3

Author: Michael Sved

Name: AltIMU-10 v3 Attitude sensor

The Attitude sensor sensor provides the orientation of the arm in respect to the ground. where 0° corresponds to the arm being horizontal.

12Conclusion

Author: Niklas Wahrenberg

The project started off with everyone reading into the essentials and Prof. Bredereke giving the important hint that interfaces are a crucial part of the system which caused last year's group a lot of problems. That's why we had them discussed and defined very early. Following that was a rather lengthy developing phase with some confusions as to how things work or why they (don't) work / compile. In this phase we also made ourselves acquainted with the hardware and checked with some quick and dirty test code whether one rotor is sufficient to pull up the arm. About as the exams started so did our deployment which was exceptionally bad timing because we could not invest as much time as we wanted to, leading to a not completely finished system.

Nevertheless, there were some lessons learned, in the positive way and in the negative way as well.

Everyone participated actively and worked his part through. There was a lot of teaming up in groups to tackle certain problems. Working in the university laboratory proved to be an effective way of collaborating, for helping each other and explaining and discussing problems, are easiest conducted in person. Also, the presence of Prof. Bredereke and thus our access to his vast knowledge helped us a lot. He observed our approach and pointed out things we did not consider and gave advice and hints.

The testing of the code on the hardware should have started earlier, so that there is enough time for fixing and debugging in case something does not work as expected. It is unrealistic to expected everything to work flawlessly together when deployed onto the hardware for the first time. We did not expect everything to work on the first attempt and it did not entirely, however as mentioned afore there was just not enough time to find out the sources of major flaws and to correct them. The STM32 worked but there were problems on the Arduino side, for further information refer to the chapter of that component. Of course, we could have stretched developing and testing as long as it would take to finish the project, whole summer if need be, but that would have been an overkill for this course. Our group was heterogeneous, some had problems with Git, others with the Airbus Wiki, some with C/C++, and so on… We are convinced to have put our group potential to good use and complemented each other well.

The project was unfortunately not as successful as we wanted it to be but on the other hand by encountering and tackling problems we learned a lot. We hope that all stakeholders are content with the outcome of this year's round of the project and want to thank Prof. Bredereke for this fun, exciting and also demanding project.