# Sprint 3

## Domain model

In the picture **Abbildung 2 Domaine Model** the layer orientated structure of our flight controller can be seen. From the top to the bottom, each layer uses the software form the layer below. This subjection is shown by the arrows pointing to the bottom. The very bottom layer is layer 1 and the very top layer is layer 8. The flight controller acts as an auto stabilizer when the UAV is in the air so that the user doesn’t have to steer against wind gusts and other external influences that disturb the flight.

## Layer 1

The very bottom layer is the layer that's closet to the hardware and coloured light red in **model1.png**. In this layer there are the drivers for the general I/O communication on the board such as I²C and PWM.

## Layer 2

In the second layer which is coloured yellow, the drivers for the sensors and actors the X-Copter uses to communicate with the real world. Here four or six motors with rotors are used depending on the configuration of the X-Copter. In addition, there is an accelerometer, a gyroscope and a compass to detect the three dimensional lay and alignment. To share data between each other, they use the orange layer drivers. To receive the command signals from the remote control a RC driver is also implemented in this layer.

## Layer 3

To get useful data from the sensors it has to be filtered first to handle the disturbances. These filters are coloured dark green and are in layer three.

## Layer 4

In the fourth layer the RC and sensor interpreters are placed and coloured blue. The interpreters are used to get human readable and understandable data from the sensors.

## Layer 5

One layer above and coloured pink, there are the controllers for the sensors, motors and the remote control. Each controller manages the specific device. This includes accessing the driver to get or set the raw data from the drivers. This raw data is then, if it’s one of the sensors, filtered and interpreted. All this is done automatically to provide an easy access by the controllers to higher layers.

## Layer 6

The PIDs are located between the PID\_Controller (orange) and the other controllers (pink) in the sixth layer. Each motor has its own PID regulator that is controlled by the orange coloured PID\_Controller.

## Layer 7

The PID controller in layer seven collects all necessary data from the controllers in the pink layer five for the PID regulators. This includes the set point from the remote control and the actual value from the sensors. The value the PIDs return is then used by the motor controllers to set the needed speed to let the XCopter fly stable.

## Layer 8

The very top layer is the flight controller that manages the timing and general flow of the application in the main method.

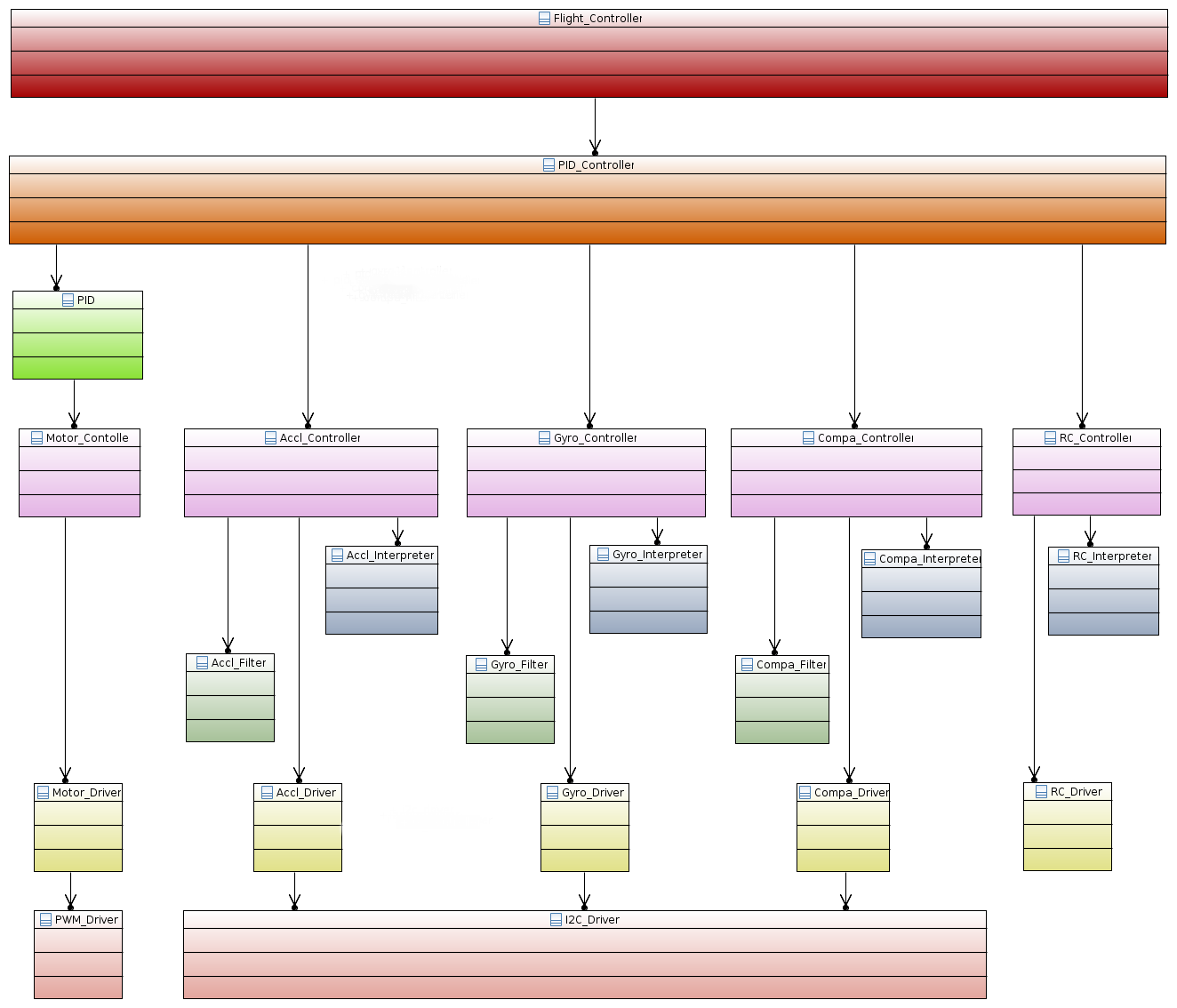


Abbildung 2 Domaine Model

## Graupners HoTT-SUMD-Signal

### Why SUMD

The “GR-16” receiver supports two different sum signals, “SUMO”- and “SUMD”- Signal.

The SUMO- Signal is an analogue sum signal and is equal to a pulse position modulation (PPM) whereas the SUMD- Signal is a digital sum signal.

So the big advantage of SUMD is, that it is easy decode able.

### Definition

HoTT SUMD is implemented with an 115200 bit/s serial data stream. The data stream is generated by HoTT receivers. The transmitter generates data frames at a data rate of 100Hz (10ms). Each data frame consists of a header followed by a data section representing the channel data and is concluded by a CRC checksum.

### Time Requirements

The serial connection needs to be set to 115200 Bit/s baud rate, 8 Databits, no Paritybit, 1 Stopbit. Each data frame is sent as a consistent data burst leaving minimal gaps less than 50µs between transmitted data bytes.

### Structure of a HoTT- SUMD frame

A single SUMD data frame comprises of three consecutive sections. SUMD\_Header, SUMD\_Data, SUMD\_CRC.

The SUMD\_Data section contains the channel data in sequential order. The number of channels to be transmitted can be up to 32. Each channel data is represented by an unsigned 16 bit word.

### SUMD\_Header section description

***Byte Byte\_Name Byte\_Value***

Byte 0 Vendor\_ID 0xA8

Byte 1 Status 0x01 or 0x81

Byte 2 Number of channels

### SUMD Data section description

Byte n\*2+1 High Byte of channel n

Byte n\*2+2 Low Byte of channel n

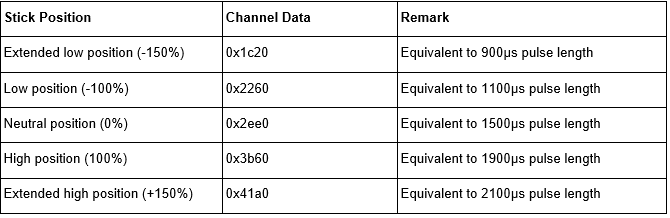
### SUMD\_CRC section description

Byte (N\_Channels+1 )\*2+1 High Byte of CRC

Byte (N\_Channels+1 )\*2+2 Low Byte of CRC derived

### Channel data interpretation

Each channel data is represented by an unsigned 16 Bit Word. The data range is derived from the pulse length for standard servos.



## Implementation of the SUMD Parsing

SUMD is a serial format and can be read directly from the receiver that’s connected via UART.

Luckily Altera is offering an RS232 UART IP Core, which can be added to our SoPC using Qsys.

It only requires two additional GPIO Pins, for receiving or transmitting serial data.

Reading and controlling the UART will be part of the UART driver.

The UART has to be initiated with the following settings, to receive a SUMD-Frame:

- 115200 Baud

- No Parity

- 1 Stop Bit

Every received Byte has to be interpreted according to the definitions of the SUMD signal format, which is described in the previous section.

Following Steps are executed by the RC interpreted Controller:

### Saving raw SUMD-Frame Bytes from the UART

The SUMD-Controller has to wait for a new SUMD-Frame. A frame starts if the value of a received Byte equals the VendorID. After that, the following Bytes will be saved in an Array.

The size of the Array will be equal to the frame this can be calculated with:

SUMD-Frame length = SUMD Header length + Number of Channels \* 2 + CRC length)

### Interpreting the received SUMD-Frame

According to the SUMD format description, every Byte has its own specific purpose.

The actual received RC-commands are sliced into a High Byte and a Low Byte,

thus it is necessary to unroll both Bytes to a 16 Bit Integer. Every Channel value will be stored in an Array, which is accessible in a c struct including all additional Data of the SUMD-Frame.

### UART Driver

This driver will offer functions to initiate and read the RS232 UART IP Core.

It is also possible to check if a new Byte was read. This is highly recommended if only one Byte will be read from the UART. The driver is divided in a source file "b\_uartriver.c" and a header file "b\_uartriver.h". The UART can be selected with an enumeration, which is defined in the header file of the driver.

### SUMD-Frame-high



Abbildung 3 SUMD-Frame-high

### SUMD-Frame-low



Abbildung 4 SUMD-Frame-Low

## Second Flight Test

### Organisation

The Organisation was similar to the first flight test. For security reasons we went to a Model flying site. Again the control was taken over by Mr. Hoeft because his Model flying insurance.

### Changes at the X-Copter since the last Flight test

To mount the motors without the rubber vibration damper the mount points had to be modified slightly.

The motor direction of rotation has been checked.

### Attempt 1

Even though the conditions were bad (it was very windy this day) the X-Copter flew very well. The Test was a success.

### Conclusion

In quadcopter form the X-Copter does only use about 50% of its power, so there is a lot of capacity for payload left.

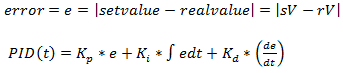
Now it’s proven that the Model is able to fly and we can start to work seriously with our own flight controller.

## PID Regulator

To guarantee a stable hovering UAVs often employ PID regulators and so do we in the X-Copter.

The general functionality works like this: The PID has two input parameters and one return value. The input parameters are the set point and the real value whereby the set point is the value, the to-be-regulated-part (in our case one of the motors), should reach and the real value is the value, the motor has currently. These two values are subtracted and the resulting difference is known as the error. Now each part of the PID regulator manipulates the error respectively and returns it. The acronym PID stands for proportional, integral and differential and means that all of the parts are accumulated into one PID regulator. The factors have to be set individually for each project and setting by hand and trial and error. The mathematical formula is shown in Formel 1 PID-Formula.

Formel 1 PID-Formula



In the following these three parts of the PID regulator will be explained.

## The Proportional Part

This is not a real regular for that matter. It's more a proportional amplifier that enforces the error by the factor. This is the most influencing part of the PID because it reacts very fast. For better understanding here is an example:

Let's say we want to have a motor run at 100 RPM. When we start the P-Regulator and the motor is not running the error will be

and it will return to the motor.

## The Integral Part

This is slowest part of a PID because it's an integral meaning its return value increases or decreases linear over time depending on the error being positive or negative. The bigger the positive errors are the faster the return value increases, the bigger the negative errors are the faster the return value decreases.

For comparison to the P-Part: If the error is 0, the P-Part will return 0 but the I-Part return value won't change. So the I-Part serves mostly to correct the residual error of the P-Part.

## The Differential Part

The D-Part is the most nervous part of the PID because it reacts to temporal changes. So the faster the real value (or sensor data) changes the higher is the return value. This part does not depend on the RC signals. This is useful for fast response what is important in the system because it has to be stabilized at all the time. Let's say the XCopter is hovering stable and a wind gust is hitting the XCopter. The P-Part will recognize and steer against it in proportion to the error – but this to slow and it would move the UAV. The I-Part won't do anything helpful in this case because the time period is too short. The D-Part will detect a fast and big change of the real value so it will accelerate the motor very fast to correct the error and steer against the wind gust.

Evaluation by the team

In the last three sprints we reached some important milestones like the first successful flight or the successful transmission from the RC Transmitter to the NiosII-processor. On the way to these goals, our team worked very well together and everybody solved his own tasks very carefully. So the team was able to finish a lot of small tasks in a short time. However, at some tasks, we have also lost a lot of time. For example, the integration of the additional USB-controller, the error analysis of the existing power supply and the installation of the Wi-Fi-Driver on the embedded Linux. Another problem was our development methodology - Scrum was a loss of time. We decided to focus on the software development and not on scrum anymore. Thus we should be able to get more work done, in a shorter period of Time in order to reach our goal.

# Lessons Learned

## Scrum

Scrum is a very good agile Software development tool. But for our team was Scrum a big overhead because of his planning for the future. We needed too much time to build up User stories or tasks. In a big team of 9 people it was always a hard discussion until every person was clear. To maintain our Jira tool a lot of time was wasted. And we decided to continue without Scrum.

## Long term calendar management

Planning of team meetings was very hard because every person had a lot of work to do during the semester. To find a date where all team members had time was impossible. Because we were discussing in small groups, communication problems occurred like the results and problems of the single groups wasn’t clear to all members.

## Quick task cannot be made quick

We thought some task could be done quickly. But in a hardware environment there is more than one thing to consider. So we needed more time to finish some tasks. In the future we are planning in smaller steps.