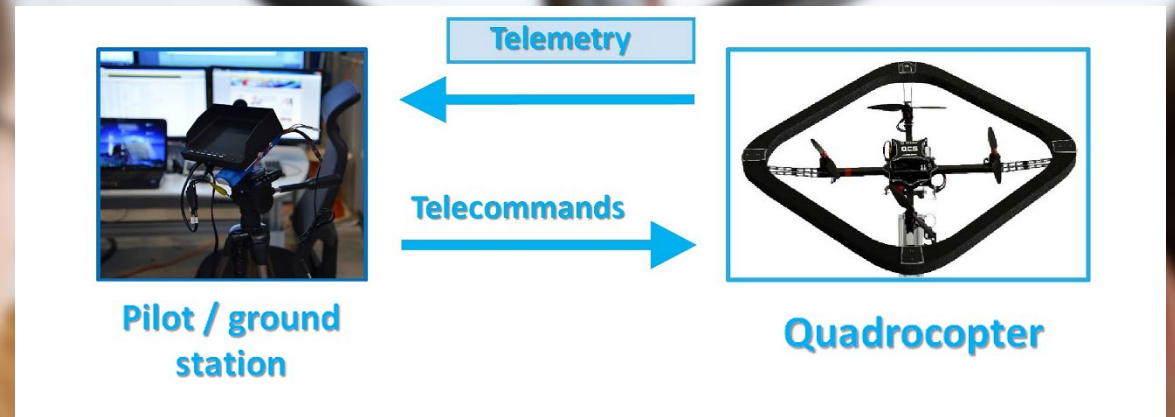


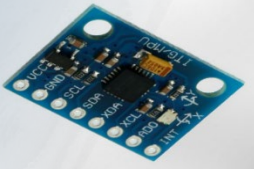
Communication

Telemetry

Lesson 5

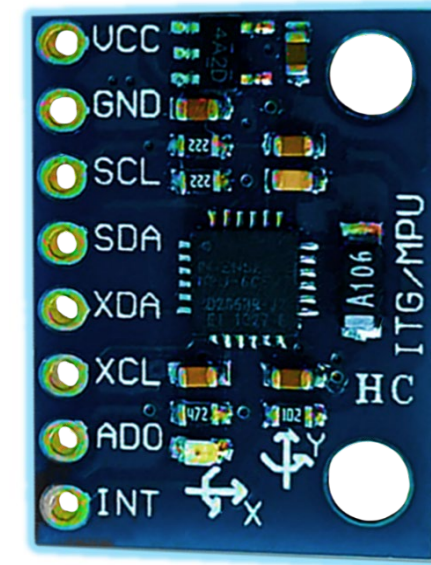


Content

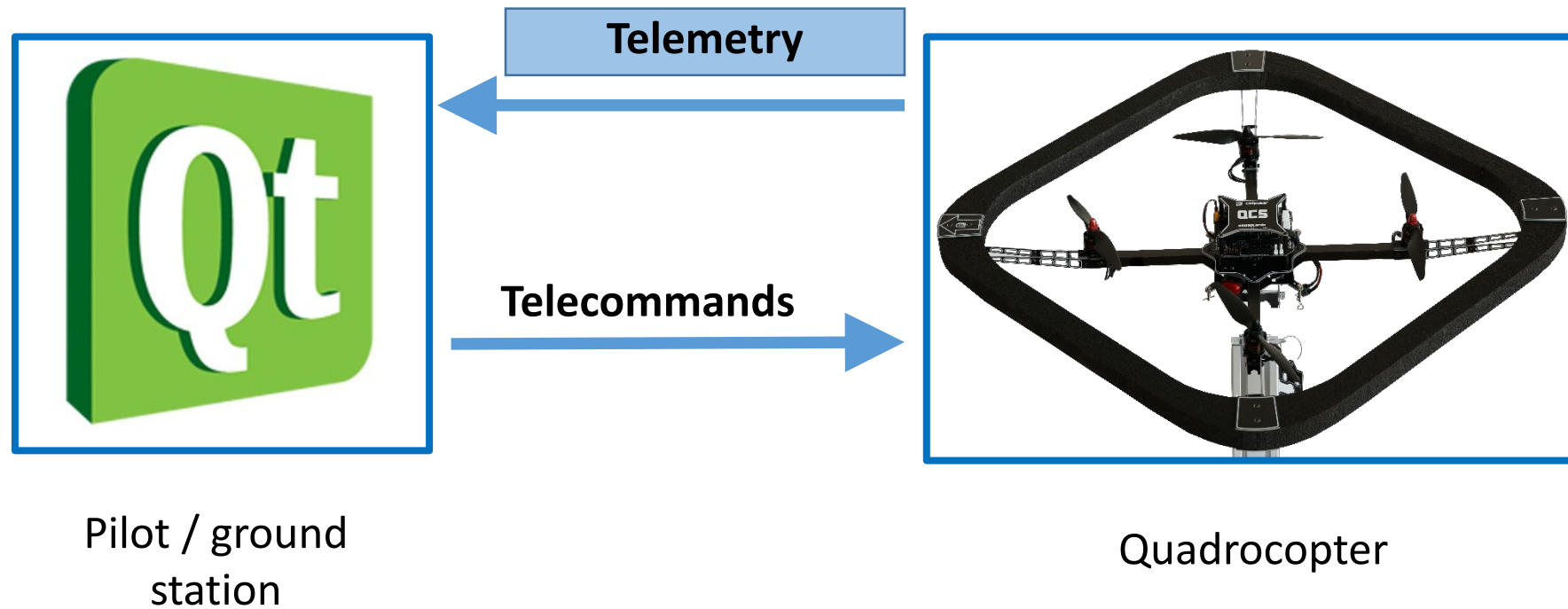
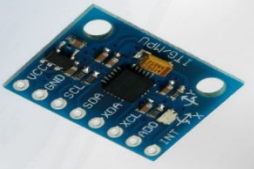


- Introduction Telemetry
- Application framework Qt
- Telemetry framework Quatplay
- Quatplay User Interface
- Telemetry implementation
 - Send Telemetry
 - Receive Telemetry
- Exercises

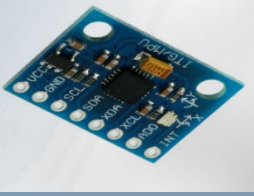
Time scope: approx. 2-4 hours



Introduction Telemetry



Introduction Telemetry

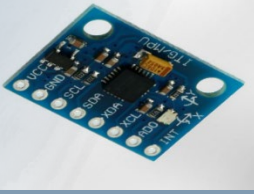


Condition for successful communication:

The sender of a message must speak a language that the recipient understands.

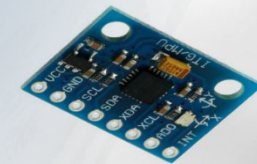
- Introduction of the communication protocol in the *protocol.h* file
 - Same file in AVR software und Qt software!
 - **Contains:**
 - Start-, separator- and end signs
 - Telecommand- and Telemetry-IDs
- Determination of the language / the format for telemetry and telecommands
- Changes in the protocol must be considered at sender and receiver!

Application framework Qt



Qt:

- C++ class library
- GUI programming
- Qt Creator -> Drag & Drop (simple, ergonomic)
- GPL license, open source, freely available
- **Platforms:** Windows, Unix, Max, Android, ...
- Extends C++ standard with MOC (meta object compiler):
 - **More functions:** Signal & Slots
 - Fully compatible to C++ compiler



Using Signal & Slots:

(Qt Reference Documentation: „Signals & Slots“)

- 1.) Signal and slots are methods
- 2.) The signal and slots are defined in the method declaration in the class definition.
- 3.) A signal is to be connected to a slot, then the slot is called each time when the signal is called.

This is done via connect: `connect(sender, signal, receiver, slot)`

- **Sender:** Sending object
- **Signal:** Method that initiates another method when called
- **Receiver:** Received object
- **Slot:** Initiated method

- 4.) In the example, the signal function `quaternion_received()` of the object `QSerialPortobj->conn_uart` calls the slot function `QUAT_Update()` of the object (this), in which this code is executed.

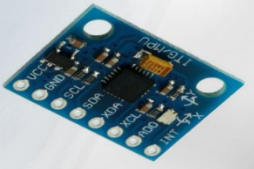
```
signals:
    void quaternion_received();
    void data_received(char *buff, int anzahl_Bytes);

public slots:

private slots:
    void receiveMsg();
```

```
connect(QSerialPortobj->conn_uart, SIGNAL(quaternion_received()), this, SLOT(QUAT_Update()));
```

Telemetry framework Quatplay

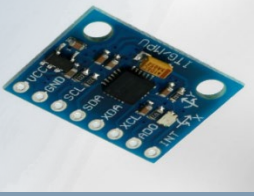


Quatplay:

- Telemetry framework
- Qt based
- Contains 5 ready-made modules:

Module	Description
glwidget	3D Display (airplane)
serialDriver	serial communication and parser
myGraph	Realizes a graph
myCurve	Realizes „ curves “ per graph three „ curves “ can be generated
quaternion	Representation / Realization of quaternions

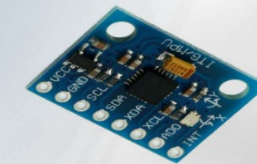
Telemetry framework Quatplay



Quatplay:

- **QwtPlot:**
 - QwtPlot is a widget / UI element
 - Extension of the Quatplay project
...\\EMQt_Framework\\qwt-6.0.1
- No standard Qt widget → by default not as UI layout in the UI editor
- **Usage:**
 - Add a Qframe → right click → Set as placeholder for user-defined classes
 - Select QwtPlot

Telemetry framework Quatplay



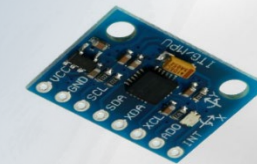
myGraph:

- **QwtPlot:** UI element which is drawn on is passed to the constructor of the myGraph
- `add_value(x, y, ID-Kurve)` → add new data
- `clear_all()` → deletes all data
- `update_graph()` → updates graphs

myCurve:

- Ring buffer, values are stored twice internally for efficient simultaneous reading + writing
- `update_graph(bool data_source)` → updates graphs with data (realdata = true) or default values (realdata = false)
- `add_value(x, y)` → adds value
- `clear_all()` → delete data

Telemetry framework Quatplay



1. Open *EMQt_Framework* project:

File -> Open file or project ->

EMQt_Framework -> quatplay -> quatplay.pro

2. Set build path:

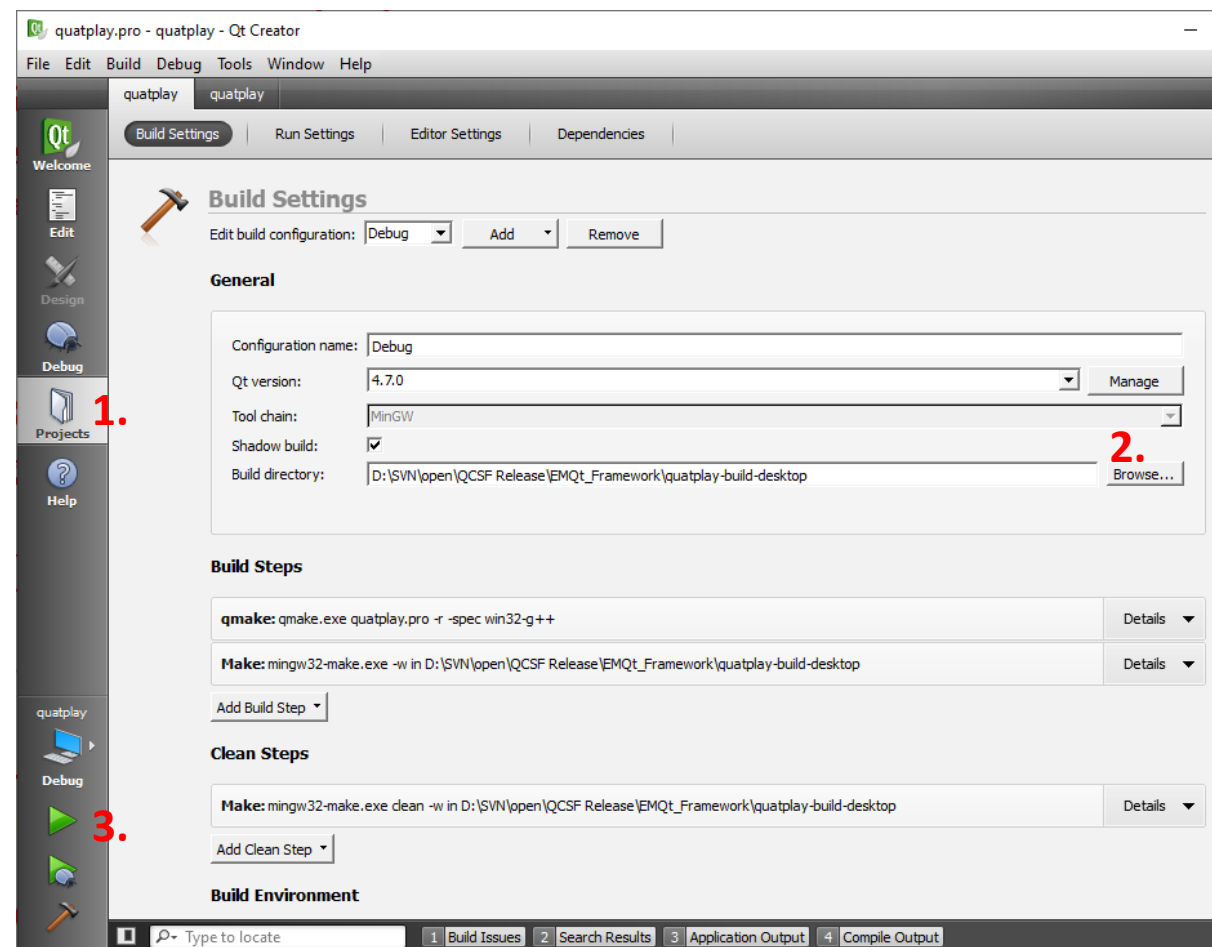
Projects -> Build directory -> Select ->

„...\\EMQt_Framework\\quatplay-build desktop“

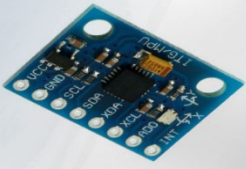
3. Execute

IMPORTANT:

The path must not contain spaces!

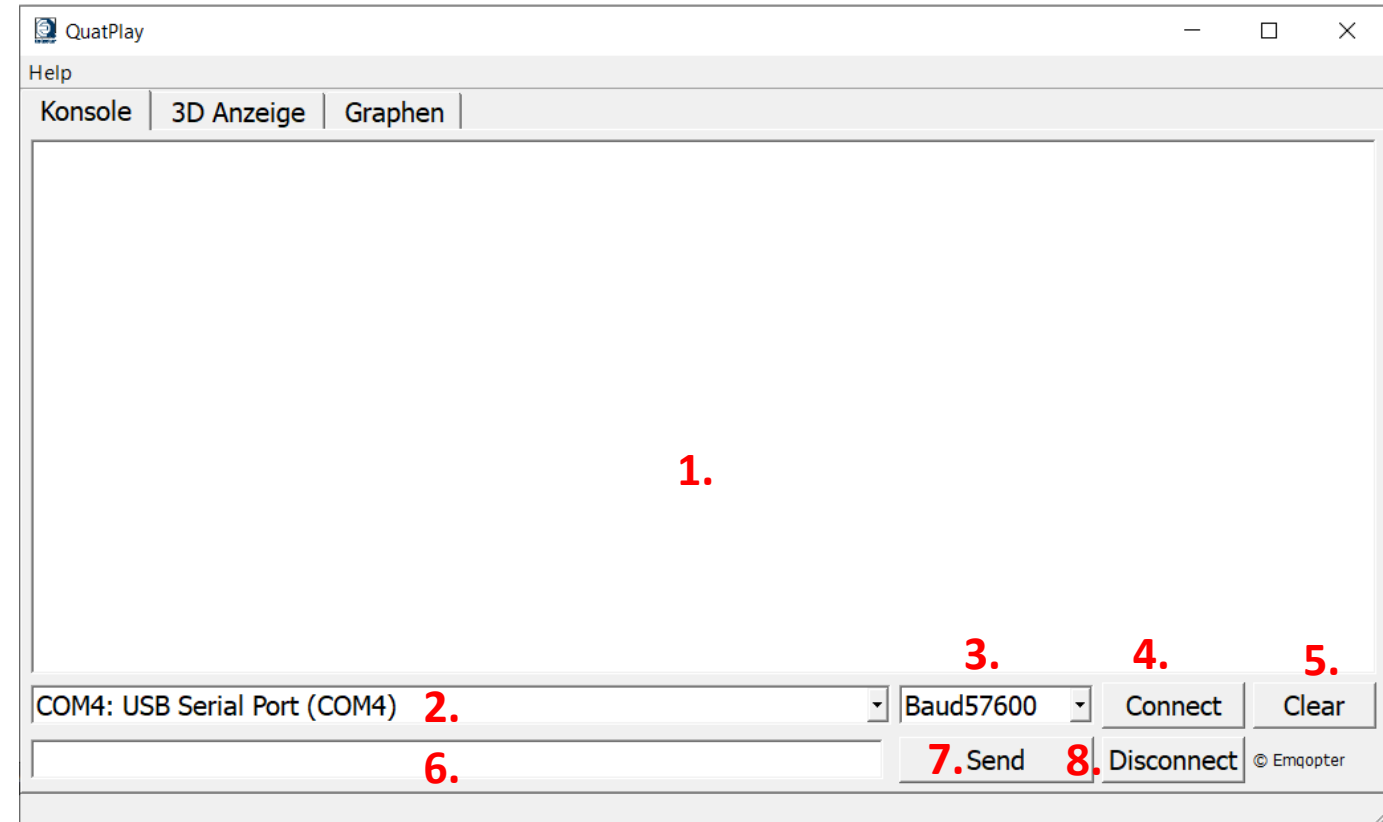


Quatplay – User Interface

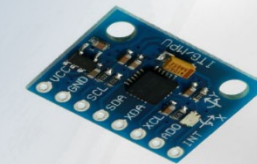


Start view (Console tab)

1. Console
2. COM port selection
3. Baud rate selection
4. Connection setup
5. Delete console
6. Text field for telecommands
7. Send telecommands
8. Disconnect

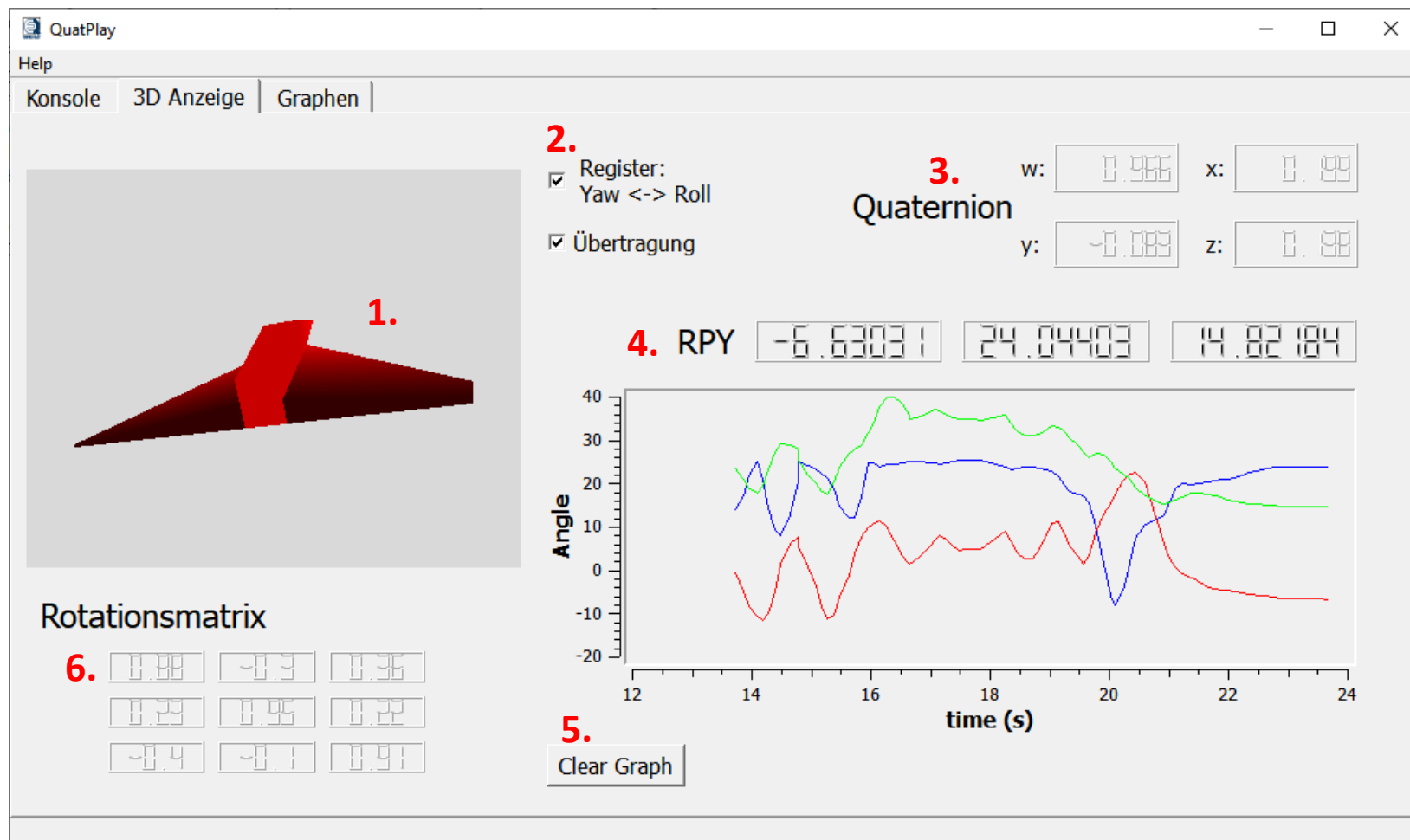


Quatplay – User Interface

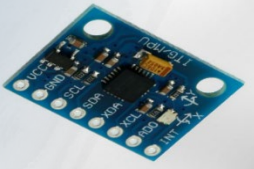


3D display

1. Attitude indicator
2. Swaps yaw and roll
3. Quaternion representation
4. Roll, pitch & yaw
5. Delete graphs
6. Rotation matrix

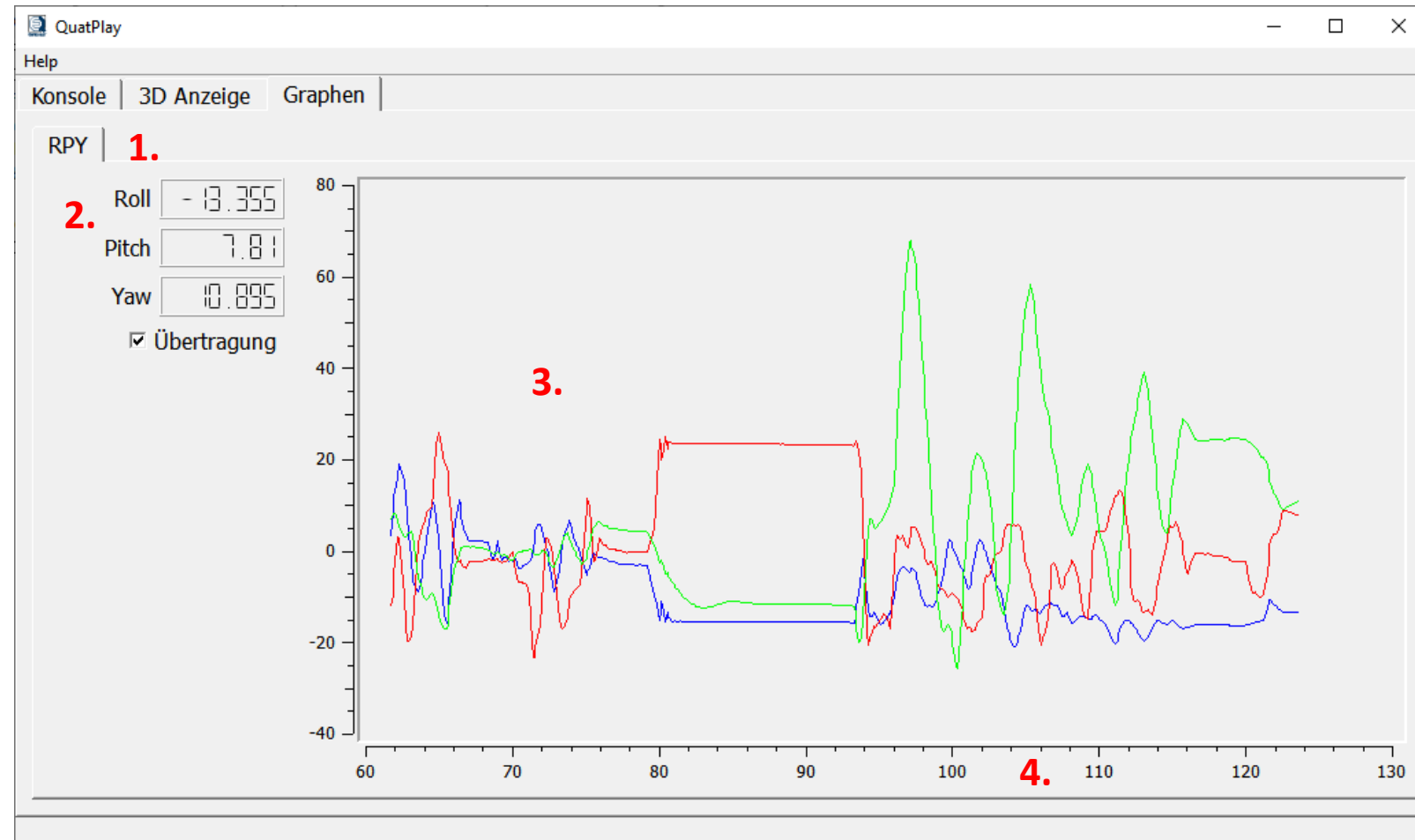


Quatplay – User Interface

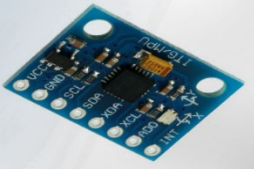


Graphs

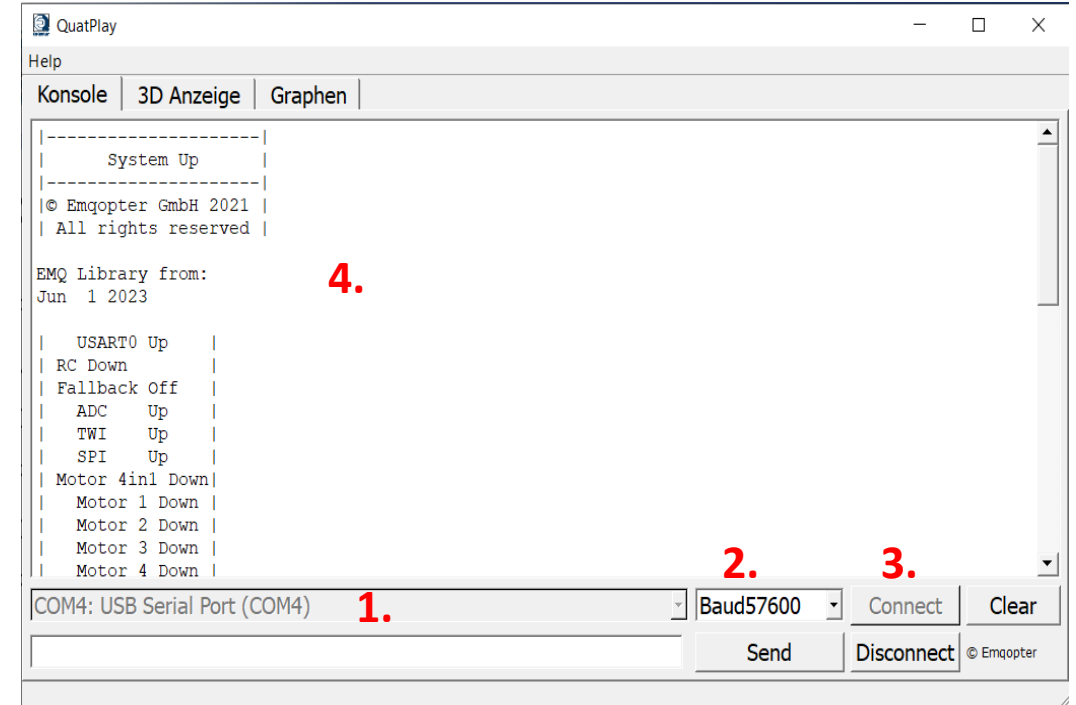
1. Tab for various graphs
2. Number display for 3 values
3. Graph display for 3 values
4. Time axis in seconds



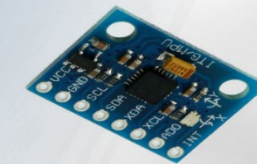
Quatplay – Set up connection



1. Set COM port to the corresponding port of the USB – RS232 converter
2. Set the baud rate to **Baud57600**
3. Establish connection by clicking on **Connect**
4. Receive messages



Telemetry implementation



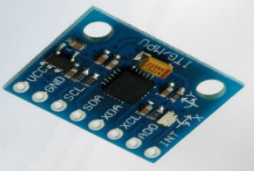
In the AVR software, data packages („**Frames**“) are created and sent, which can be read and represented on the PC side with Qt.

General procedure:

1. Assemble data into frames on the AVR side
2. Formulate frames into a message according to a protocol
3. Send the created message via USART
4. Receive message on Qt page
5. Interpret the message based on the protocol and frames
6. Display data from frames

} AVR
} Qt

Send Telemetry



The required functions can be found in the *telemetry.h/.c* files:

`void mySendMessage (Frame* f, int messageType) :`

Here a frame is filled for the transmission of the data of messageType with data.

`void startFrame (Frame* f) :`

Takes over the creation of a new frame and sets start signs for the frame according to protocol.

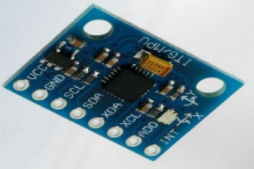
`void addValuesToFrame (Frame* f, float* values, int numOfValues) :`

Adds data from values to frame f according to protocol #numOfValues.

`void endAndSendFrame (Frame* f) :`

Ends the frame f according to the protocol and sends it.

Send Telemetry



1. Assemble data into frames on AVR side

- *protocol.h* provides **Frame** – struct:

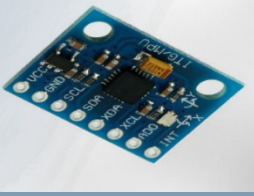
```
17 typedef struct {  
18     char data[MAX_FRAME_LENGTH];  
19     int data_pos;  
20 } Frame;
```

- Initialize frame with `void startFrame (Frame* f)`

```
// Legt den Telemetrie Frame an.  
Frame f; // USART Frame entsprechend definiertem Protokoll  
startFrame (&f);
```

- Data can be added to an existing frame with `void addValuesToFrame (Frame* f, float* values, int numOfValues)`.

Send Telemetry



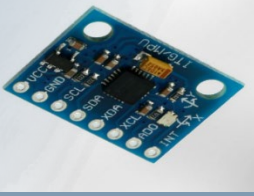
2. Formulate frames into a message according to a protocol

- `void addValuesToFrame(Frame* f, float* values, int numOfValues)` already implements the necessary procedure to combine the data into a message according to a defined protocol.

3. Send the created message via USART

- With `void endAndSendFrame(Frame* f)` the frame is completed and sent via USART.

Send Telemetry

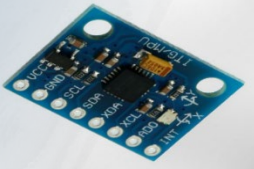


- The entire procedure is programmed in `void mySendMessage(int messageType)`.
- `int messageType` identifies the message type – depending on which data should be sent (quaternions, gyrometer data, accelerometer data,...). This data is created in the *protocol.h* file via define:

```
39// Telemetrie: IDs
40#define      TM_QUATERNION      0
41#define      TM_RPY            1
42#define      TM_MESSAGES       2
```

- Here, an ID is created for each message type.

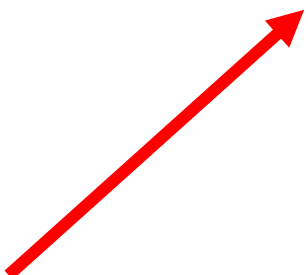
Send Telemetry



For each `int messageType`, the data is added to the frame in `void mySendMessage(int messageType)`.

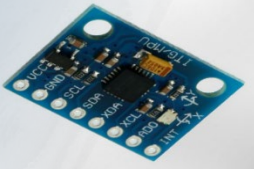
```
switch (messageType) {  
    default:  
        break;  
  
    case TM_QUATERNION:  
        add_Quaternion_Values(f);  
        break;  
}
```

```
void add_Quaternion_Values(Frame* f) {  
    float values[16]; // Array of values to be sent.  
    values[0] = (float) imu.q0;  
    values[1] = (float) imu.q1;  
    values[2] = (float) imu.q2;  
    values[3] = (float) imu.q3;  
    addValuesToFrame(f, values, 4); // Add 4 values to Frame frame.  
}
```



The addition of data is implemented in the respective functions.

Receive Telemetry

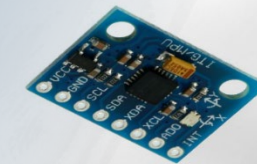


4. Receive message on Qt page

Basic functionality (*signals* and *slots*):

Modules in Qt provides ***slots*** (functions) that are called when certain ***signals*** (events) are triggered – provided that signals and slots are connected.

Receive Telemetry



4. Receive message on Qt page

Slot *receiveMsgFrames()* in *qserialconnection_uart.h* for receiving messages

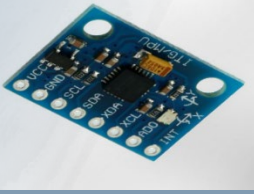
```
44     private slots:  
45         void receiveMsgFrames();
```

is connected in *qserialconnection_uart.cpp* by the function `bool connect(...)` with a Signal *timeout()* of the object *timer_usart*:

```
12     // Frage alle 50ms nach neuen Daten am USART  
13     timer_usart.setInterval(50);  
14     connect(&timer_usart, SIGNAL(timeout()), this, SLOT(receiveMsgFrames()));
```

-> As soon as *timeout()* is triggered, the function *receiveMsgFrames()* is executed.

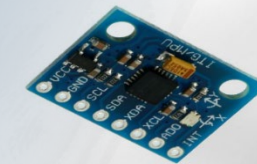
Receive Telemetry



4. Receive message on Qt page

- ***receiveMsgFrames()*** checks whether new messages have arrived at the USART
- If yes, process ***processUARTFrames()*** is called in ***QSerial.cpp*** where the message is processed.

Receive Telemetry



5. Interpret the message based on the protocol and create frames

```

8  extern Quaternion Quat;
9  extern PacketData receivedPacket_Quat;
10 extern PacketData receivedPacket_RPY;

119 if(frameStart != -1) {
120     msg = msg.right(frameStart - 1);
121     while(msg.length() > 1) {
122         QString data = entnehmePaket();
123         if (data != FEHLERSTRING){
124             PacketData receivedPacket = this->parsePacket(data);
125             4. if(receivedPacket.typ == TM_QUATERNION) {
126                 6. receivedPacket_Quat = receivedPacket;
127                 5. } else if (receivedPacket.typ == TM_RPY) {
128                     6. receivedPacket_RPY = receivedPacket;
129                     6. frameReceived_RPY();
130                 }
131             }
132             msg = "";
133         }else{
134             // Framestart nicht gefunden -> Message über Console ausgeben.
135             console_output(msg);
136             msg = "";
137         }
138     }
139 }
140
141
142
143

```

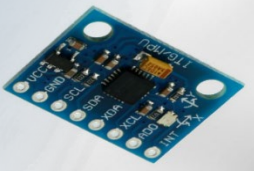
1.

2.

3.

1. **receivedFrame_Quat**, **receivedFrame_RPY** and **Quat** are global variables from mainwindow.cpp to temporarily store the data until it is displayed in the graph
2. As long as there are still unprocessed messages, packets are searched for and extracted according to the protocol
3. A packet is always taken and processed before the next one is taken
4. **PacketData.typ** obtains the packet type
5. The interpretation of the data depends on the packet type
6. **frameReceived_Quat()** and **frameReceived_RPY()** are signals that indicate the arrival of a new frame of the respective type

Receive Telemetry

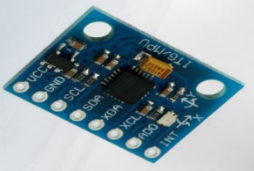


5. Interpret the message based on the protocol and create frames

Notes to *PacketData* in *QSerial.h*:

- **Contains:**
 - *typ* see TelemetryTypes in *protocol.h*
 - *values* Data values
 - *value_count* Number of the data values

```
19 struct PacketData {  
20     float typ;  
21     float * values;  
22     int value_count;  
23 };
```



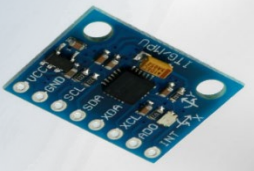
6. Display data from frames

- Packet data is stored separately for each type in *PaketData* – structures in ***mainwindow.cpp***

```
34 // Framedaten
35 PacketData receivedPacket_Quat; // Framedaten zu Quaternionen
36 PacketData receivedPacket_RPY; // Framedaten zum Accelerometer
```

- Reminder: Structures were filled with data in ***processUARTFrames()*** in *QSerial.cpp*.
- Signals for the arrival of new frame data created in ***QSerial.h***:

```
48 signals:
49 void frameReceived_Quat(); // Neue Quaternionendaten angekommen.
50 void frameReceived_RPY(); // Neue RPY - Daten angekommen.
```



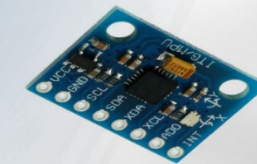
6. Display data from frames

- In *mainwindow.cpp*, the **signals** for incoming Data are connected to the functions for representing the data in graphs:

```
87      // ***** FRAMES & GRAPHEN *****
88
89      connect(QSerialPortobj,SIGNAL(frameReceived_Quat()),this,SLOT(frameUpdate_Quat()));
90      connect(QSerialPortobj,SIGNAL(frameReceived_RPY()),this,SLOT(frameUpdate_RPY()));
```

- If new quaternion data arrives, **frameUpdate_Quat()** is called in *mainwindow.cpp*.
- If new RPY data arrives, **frameUpdate_RPY()** is called in *mainwindow.cpp*.

Receive Telemetry



6. Display data from frames

- In the ***frameUpdate_RPY()*** function in *mainwindow.cpp*, the RPY data stored in *receivedFrame_RPY* is transferred to the *rpy_graph* and the LCD displays.

```
200 // Stellt die ankommenden RPY - Daten dar
201 void MainWindow::frameUpdate_RPY() {
202     // Daten in Graph schreiben
203     rpy_graph->add_value_on_time(receivedPacket_RPY.values[0], 1);
204     rpy_graph->add_value_on_time(receivedPacket_RPY.values[1], 2);
205     rpy_graph->add_value_on_time(receivedPacket_RPY.values[2], 3);
206     // Daten in LCDs anzeigen
207     ui->lcdNumber_RPY_X->display(receivedPacket_RPY.values[0]);
208     ui->lcdNumber_RPY_Y->display(receivedPacket_RPY.values[1]);
209     ui->lcdNumber_RPY_Z->display(receivedPacket_RPY.values[2]);
```

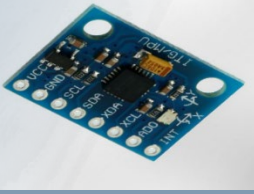
**Initialization of the graphs
in *mainwindow.cpp*:**

```
38 // Graphen
39 myGraph* rpy_graph;
40 myGraph* quat_graph;

92 rpy_graph = new myGraph(ui->qwtPlot_Graph_RPY);
93 quat_graph = new myGraph(ui->qwtPlot_Quat);
```

***myGraph receives the UI object when
it is created!***

Receive Telemetry



6. Display data from frames

myGraph module:

- Contains up to 3 independent curves
- Relevant functions:

myGraph(*QwtPlot* *p)

void add_value_on_time(y, Kurven_ID)

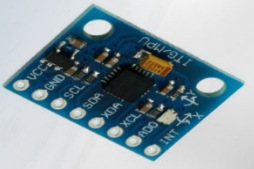
void clear_all()

→ Constructor with *QwtPlot* *p as drawing layer

→ adds new data pair to the curve with Kurven_ID

→ deletes all data from the graph

Exercises



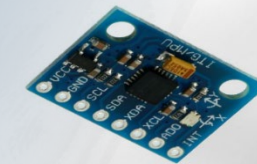
Necessary hardware:

- EMQ3000
- Micro USB cable for power and flashig
- Micro USB cable for communication: USART to PC
- QCS / QCSF

Necessary software:

- AVR Studio 32 installed (with toolchain and flip driver)
- Qt Creator 2.0.1 and Qt 4.7.0 (32bit)
- EMQt_Framework\quatplay\quatplay.pro (code)

Exercises



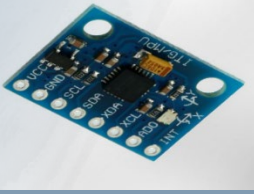
Exercise 1:

Create a communication link with the Qt control program. Test the transmission with the functions already implemented from the episodes 2+4 (IMU, quaternions).

Exercise 2:

Use the function `void my_send_telemetry()` in the file *telemetry.c* to send the quaternions of the QCS via USART to the Qt control program. The sending of message packets is done via `void mySendMessage(int messageType)`. Use as parameter ***messageType*** the entry *TM_QUATERNION* according to the define from the file *protocol.h*. When the task has been solved, an airplane model can be seen under the tab ***3D Display***, which reacts according to the movements of the IMU.

Exercises



Exercise 3:

Create a frame in the AVR software, analogous to the quaternion example, which sends the RPY data of the QCS to Qt via telemetry. To do this modify the function `void mySendMessage(int messageType)`. Use and implement the function `void my_add_RPY_Values(Frame * f)`. The RPY data is obtained via the external struct `imu_data imu`. You should now see the position of the QCS in relation to the roll, pitch and yaw axes in a diagram in the ***Graphs/RPY*** tab.