Wie klappt das mit dem I2C-Bus (das ist das gleiche wie der TWI-Bus) ?

Zunächst das Datenblatt des anzuschließenden Moduls. Daraus kann ich entnehmen :

I2C Communication

I2C communication protocol with the compass module is the same as popular eeprom's such as the 24C04. First send a start bit, the module address with the read/write bit low, then the register number you wish to read. This is followed by a repeated start and the module address again with the read/write bit high. You now read one or two bytes for 8bit or 16bit registers respectively. 16bit registers are read high byte first. The compass has a 28 byte array of registers. organized as below:

Register	Function
0	Command register (write) / Software version (read)
1	Compass Bearing 8 bit, i.e. 0-255 for a full circle
2,3	Compass Bearing 16 bit, i.e. 0-3599, representing 0-359.9 degrees. register 2 being the high byte
4	Pitch angle - signed byte giving angle in degrees from the horizontal plane, Kalman filtered with Gyro
5	Roll angle - signed byte giving angle in degrees from the horizontal plane, Kalman filtered with Gyro
6,7	Magnetometer X axis raw output, 16 bit signed integer with register 6 being the upper 8 bits
8,9	Magnetometer Y axis raw output, 16 bit signed integer with register 8 being the upper 8 bits
10,11	Magnetometer Z axis raw output, 16 bit signed integer with register 10 being the upper 8 bits
12,13	Accelerometer X axis raw output, 16 bit signed integer with register 12 being the upper 8 bits
14,15	Accelerometer Y axis raw output, 16 bit signed integer with register 14 being the upper 8 bits
16,17	Accelerometer Z axis raw output, 16 bit signed integer with register 16 being the upper 8 bits
18,19	Gyro X axis raw output, 16 bit signed integer with register 18 being the upper 8 bits
20,21	Gyro Y axis raw output, 16 bit signed integer with register 20 being the upper 8 bits
22,23	Gyro Z axis raw output, 16 bit signed integer with register 22 being the upper 8 bits
24,25	Temperature raw output, 16 bit signed integer with register 24 being the upper 8 bits
26	Pitch angle - signed byte giving angle in degrees from the horizontal plane (no Kalman filter)
27	Roll angle - signed byte giving angle in degrees from the horizontal plane (no Kalman filter)

Oben steht : Sende einen **Startbefehl**, dann die **Moduladresse (192, steht weiter unten), mit dem letzten Bit = 0**, dann die gewünschte **Registernummer** (z.b. 1 für den Kompasswinkel 0…255), dann nochmal nen **Start** und dann die **Moduladresse mit dem letzten Bit = 1**, das ist dann die 193. Dann kann man den **Wert lesen**.

Jetzt das Prozessordatenblatt, damit ich die verschiedenen Befehle schreiben kann. Seite 246 bringt mich weiter. Ich brauche eigentlich gar nicht in die Registererklärungen reinzuschauen, weil hier schöne Beispiele angegeben sind :

Zum Beispiel der Startbefehl :

C Example	Comments
`TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN)`	Send START condition

In meiner späteren I2C-Klasse also der Startbefehl :

```
TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN) ;
```

Natürlich kann man das auch aus den Registererklärungen rauslesen (Seite 248) :

A START condition is sent by writing the following value to TWCR:

TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE
value	1	X	1	0	X	1	0	X

TWEN must be set to enable the 2-wire Serial Interface, TWSTA must be written to one to transmit a START condition and TWINT must be written to one to clear the TWINT Flag. The TWI will then test the 2-wire Serial Bus and generate a START condition as soon as the bus becomes free. After a START condition has been transmitted, the TWINT Flag is set by hardware, and the status code in TWSR will be 0x08 (see Table 24-2 on page 249). In order

Da steht also, dass nach dem Startbefehl von der Hardware das TWINT-Flag als Ausführungsbestätigung gesetzt wird. Also können wir das nutzen, um zu warten daß der Befehl wirklich angekommen ist :

```
while ((TWCR & (1<<TWINT)) == 0)          //Schleife, solange kein TWINT-Flag
{
}
```

Nach dem selben Muster werden nun auch die anderen nötigen Befehle (aus dem Datenblatt des Kompassmoduls ersichtlich, welche man braucht) mit Hilfe des Prozessordatenblatts geschrieben.

Schwieriger wird es bei der einzustellenden Taktrate. Hier steht im Datenblatt (Seite 242) :

The SCL frequency is generated according to the following equation:

$$\text{SCL frequency} = \frac{\text{CPU Clock frequency}}{16 + 2(\text{TWBR}) \cdot 4^{TWPS}}$$

Das Register TWBR wird einfach mit einem Wert 0..255 beschrieben, dazu kommt noch der sogenannte „Prescaler“ mit dem Wert TWPS (PS wohl für PreScale). Dafür muß man suchen, wo dieser Wert einzuschreiben ist, und findet dann auf Seite 262 :

24.9.3 TWSR – TWI Status Register

Bit	7	6	5	4	3	2	1	0	
(0xB9)	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0	TWSR
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	1	1	1	1	1	0	0	0	

..hier die beiden möglichen Bit (also 0..3)

Das Problem ist jetzt, das man nicht weiß, welcher Wert für den Takt der Optimale ist. Beim Quadrocopter hatten wir (aus dem Internet übernommen) 400 khz eingestellt, damit gabs dann aber Probleme. Der I2C machte Fehler, stürzte sogar manchmal komplett ab.

Also : geringer ! Ich habe dann einfach den Prescaler mit 3 belegt, damit wird's langsamer. (Machen sie das besser : überlegen Sie sich Werte für einen Takt von z.b. 100 khz !)

Für die Basisbefehle am I2C habe ich dann folgenden Code als Klasse i2c :

```
#include "i2c.h"
#include <util/twi.h>

i2c::i2c()
{
    TWBR = 72;                    //Bitrate 400 khz (bei Prescale=0
    TWSR = 3;                     //Prescaler 64, weil 400khz zu schnell

}

void i2c::start()
{
        TWCR=(1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
        while ((TWCR & (1<<TWINT)) == 0)
        {
        }
}

void i2c::stop()
{
        TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO); //stop
}


unsigned char i2c::read_end()
{
        TWCR = (1<<TWINT)|(1<<TWEN);                //read ohne ACK
        while ((TWCR & (1<<TWINT)) == 0)
        {
        }
        return TWDR;
}

void i2c::write(unsigned char wert)
{
        TWDR = wert;
        TWCR = (1<<TWINT) | (1<<TWEN) ;
        while (!(TWCR & (1<<TWINT)))               //data und warte
        {
        }
}
```

(Den Read mit Acknowledge aus meinem alten Beispiel brauchts hier nicht)

Damit schreibe ich eine zweite Klasse, weil ich das hier für übersichtlicher halte, da kommen nun direkt die Befehlszyklen rein, die der Kompass braucht. Die nötigen Register stehen ja im Datenblatt des Kompassmoduls :

```
#include "kompass.h"
#include "i2c.h"

i2c twi;

void kompass::lies_winkel()
{
        twi.start();                //start
        twi.write(192);             //moduladresse (default 0xc0 = 192)
        twi.write(1);               //register 1 bitte (winkel 0..255)
        twi.start();
        twi.write(193);             //read-modus
        winkel = twi.read_end();    //Wert kommt aus dem Datenregister TWDR
        twi.stop();
}
```

Und dann benutze ich das einfach (hier in der Klasse zur Regelung der Fahrtrichtung) :

```
#include "bewegung.h"
#include "kompass.h"
#include "antrieb.h"
#include "pid_regler.h"

kompass richtung;
antrieb motoren;
pid_regler regler;


void bewegung::vor()
{
    regler.sollwert = zielwinkel;
    richtung.lies_winkel();
    regler.istwert = richtung.winkel;
```