Lucerne University of
Applied Sciences and Arts
HOCHSCHULE
LUZERN

Hochschule Luzern - Technik & Architektur

Industriearbeit

Hardware in the Loop Autopilot

Pascal Häfliger

betreut durch:
Prof. Dr. Christoph Eck
Prof. Dr. Thierry Prud'homme

Industriepartner:
Aeroscout GmbH, Horw

16. Januar 2016

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigte und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Ort, Datum

Unterschrift

Abstract

Pixhawk is an embedded autopilot project with a tremendous market potential. This Product had just been upgraded to a new version. A flight controller has to run stable under any circumstances. Malfunctioning may cause severe damage to the aircraft and its surroundings. The main objective for this paper is to compare and implement a hardware in the loop test bench for such product. Utilizing this procedure, several test cases can be executed without the risk of damage. Also, a Simulink plugin was reviewed for stated requirements. However, that expansion was unable to fulfill the specificaion and therefore not persued. Thus, a different approach was evaluated. The source code for the Pixhawk firmware was altered to integrate a feasibility to access the internal data such as sensor values or actuating variables. Moreover, an enviroment simulation was designed with Matlab/Simulink to be run on a host PC. This paper offers a slim, expandable, reuasable hardware in the loop solution for Pixhawk.

Inhaltsverzeichnis

Glossar

B

Baud

Die Symbolrate einer Datenübertragung. Durch Modulation kann mehr als 1 Bit pro Baud übertragen werden, deshalb die Unterschiedlichen Begriffe.

C

cmake

Ein Programmiertool, welches Code generiert, basierend auf den Makefile Parametern.

Code Composer

Simulink Plugin welches aus Blockschaltbildern lauffähigen Code in unterschiedlichen Sprachen generieren kann.

E

ESC

Electronic Speed Controller.

F

FCS

Flight Management Unit.

FMU

Flight Management Unit.

H

HiL

Hardware in the Loop.

I

I2C

Inter-Integrated Circuit.

N

nsh

Nuttx Shell.

Nuttx

RTOS with multiple Threads.

P

Pixhawk

Kommerzieller Autopilot zweiter Generation. Entwickelt als Forschungsprojekt an der ETH Zürich.

POSIX

Portable Operating System Interface.

PPM

Pulsphasenmodulation, parts per million.

PSP

Pilot Support Package. MathWorks Plugin welches nicht in der Basis Verison ausgelifert wird. Erweiterung für Simulink.

PSP

Pixhawk Pilot Support Package.

PWM

Pulsweitenmodulation.

R

RTOS

Real Time Operating System.

S

SPI

Serial Peripheral Interface.

U

UART

Universal Asynchronous Receiver Transmitter.

uORB

Micro Object Request Broker.

1 Einführung

Diese Arbeit handelt von einer bidirektionalen Datenstromrealisierung zwischen Simulink und einem embedded Board namens Pixhawk. Diese Daten sollen in einer Hardware in the Loop Simulation verwendet werden.

Im Kapitel 2 und 3 werden die Grundlagen von Pixhawk sowie der Programmierumgebung erklärt und aufgezeigt. Weiter wird das Konzept der Hardware in the Loop Simulation an einigen Beispielen veranschaulicht, welches anschlissend mit dem Pilot Support Package versucht wurde zu realisieren.

Als Lösungskonzept der Hardware in the Loop Simulation wurde schlussendlich der Programmcode vom Pixhawk geändert. Auf der Gegenseite wurde im Simulink eine Datenstromverarbeitung und Simulation erstellt. Dieser Ansatz wird im Kapitel 6.1 sowie 6.2 aufgezeigt.

1.1 Aufgabenstellung

Bei der folgenden Aufgabenstellung handelt es sich um eine Kurzzusammenfassung. Die komplette Anforderungsliste ist in Kapitel 8 ersichtlich.

- Einarbeitung in die Pixhawk Firmware und Designpatterns
- Eigene Test-App mit Datenstromverarbeitung demonstrieren
- Kommunikationsmöglichkeiten zwischen Pixhawk und Simulink ausarbeiten
- Hardware in the Loop Simulation verwirklichen
- Eigene App schreiben, welche die Komunikation mit Simulink übernimmt
- Einfache Hardware in the Loop Simulation auf Seiten Simulink programmieren und demonstrieren.

2 Was ist Pixhawk

Das open-source Projekt Pixhawk ist ein Hochleistungsautopilot für Modellflugzeuge, Multicopter, Helikopter, Autos und Boote. Es wurde an der ETH Zürich entwickelt und ist nun auf dem Markt verfügbar. Das Pixhawk ist die zweite Version des Flugreglers. Für die Verwendung sowie Weiterentwicklung stehen zahlreiche Tools, Anleitungen und eine aktive Community zur Verfügung .
Für die Arbeit dient das Pixhawk zur Regelung eines Quadrocopters oder Modellhubschraubers.

Abbildung 1: Pixhawk

Die wichtigsten Features des Pixhawks sind:

- 32-bit 168 MHz Cortex M4F (floating point unit)
- 256 KB RAM, 2 MB Flash
- MPU6000 als Accelerometer und Gyrometer
- Power Controller mit Ausfallsicherung
- ESC protection und überstrom Schutz
- 5x UART, I2C, SPI, CAN, SD Karte, PWM, PPM, AD Wandler

2.1 POSIX

POSIX (Portable Operating System Interface) ist eine Erweiterung zum IEEE 1003, welche die API, Shell und Utilities von UNIX standartisierte. Ursache für POSIX war, dass UNIX Software nicht auf allen UNIX Systemen lief. Durch diese Standardisierung soll ein Skript oder Programm portierbar sein, ohne das eine Änderung im Sourcecode vorgenommen werden muss. Der Vorteil liegt darin, dass es auf jedem System läuft. Dadurch wird jedoch der technische Fortschritt stark abgebremst, weil es auch auf dem ältesten System laufen soll, welches z.B. ein 8 Bit Prozessor besitzt. Wenn man z.B. die Shell betrachtet, muss alles für *sh*, der originalen Bourne Shell, kompatibel sein. (?, ?)

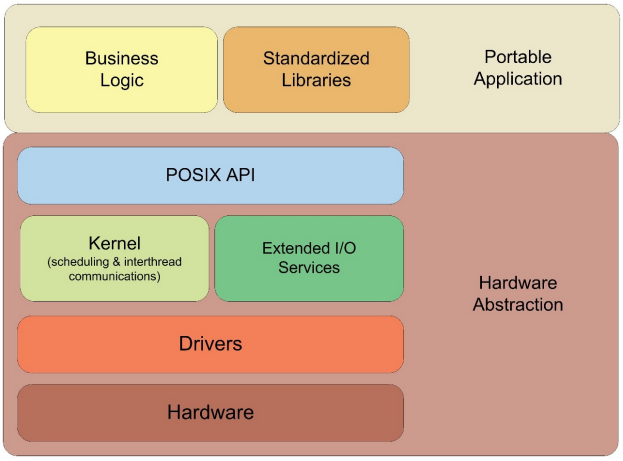


Abbildung 2: POSIX API
(?, ?)

In der Abbildung 2 ist ersichtlich, wie die Applikation durch den POSIX Layer von der Hardware getrennt wird. Dieses POSIX API stellt einfache Interfaces zur Verfügung, welche im Kernel oder dem erweiterten I/O Service implementiert wurden.
Die Applikation kann nun auf jede Hardware portiert werden, da sie klar definierte Funktionen verwendet.

2.2 Nuttx

Nuttx ist ein RTOS (Real Time Operating System), welches auf 8 bis 32 Bit Architekturen läuft. Nuttx verwendet POSIX, damit das Operating System portierbar ist. Es ist open source, dadurch also hoch flexibel und anpassbar an die Bedürfnisse. Das Betriebssystem wird oft für kleine Embedded Systeme verwendet.
Pixhawk verwendet Nuttx als Grundelement für ihre gesamte Struktur. Auf dem OS können dann mehrere Threads, sogenannte Apps, laufen. Diese verwenden das Preemtive Prinzip, also nicht kooperativ. Der Scheduler kümmert sich grösstenteils um die Einhaltung der Echtzeitkriterien.
Der Cortex M4F besitzt nur 1 Kern, dadurch kann jeweils nur ein Thread laufen. Durch das Task switching können jedoch alle Tasks ihre Berechnungen durchführen. Der Scheduler, auch als Idel Task bekannt, gibt jedem Thread ein Zeitfenster, welches voll ausgenützt werden kann. Falls ein Task nicht das komplette Zeitfenster belegt, soll dieser die Ressourcen freigeben für andere Apps. Falls kein Thread Rechenzeit benötigt, wartet der Scheduler in einer Sleep ähnlichen Funktion. Im Hintergrund läuft dann der Garbage Collector und verwaltet die Speicherblöcke.

Über eine UART (Universal Asynchronous Receiver Transmitter) Schnittstelle kann man auf die nsh (Nuttx Shell) zugreiffen. Dies ermöglicht die Überwachung und Ausführung des Systems wie bei einem Terminal. Mit dem Befehlt top kann der Taskmanager angezeigt werden. Wie man aus der Abbildung 3 entnehmen kann, ist der Idel Task mit einer CPU Auslastung von 76% stark vertreten. Die anderen Apps wie z.B. gps, px4io, sdlog2 laufen alle auch auf dem System.

```
Processes: 20 total, 3 running, 17 sleeping
CPU usage: 22.17% tasks, 0.86% sched, 76.97% idle
Uptime: 77.365s total, 59.350s idle

 PID COMMAND                  CPU(ms) CPU(%)  USED/STACK PRIO(BASE) STATE
   0 Idle Task                 59350 76.974     0/    0   0 (  0)  READY
   1 hpwork                     1011  1.332   596/ 1592 192 (192)  w:sig
   2 lpwork                      352  0.380   572/ 1592  50 ( 50)  READY
   3 init                       1593  0.000  1348/ 2496 100 (100)  w:sem
   6 nshterm                       9  0.000   860/ 1496  70 ( 70)  w:sem
 112 commander_low_prio          255  0.000   732/ 2592  50 ( 50)  w:sem
  83 dataman                      23  0.000   684/ 1192  90 ( 90)  w:sem
 101 sensors                    2081  2.664  1636/ 1992 250 (250)  w:sem
 104 gps                         256  0.761   692/ 1192 220 (220)  w:sem
 106 commander                   401  0.380  2964/ 3392 215 (215)  w:sig
 108 px4io                      2538  3.235   940/ 1496 240 (240)  w:sem
 185 top                         514  2.854  1292/ 1696 100 (100)  RUN
 114 mavlink_if0                 688  0.856  2100/ 2392 100 (100)  READY
 116 mavlink_rcv_if0               6  0.000   756/ 2096 175 (175)  w:sem
 127 sdlog2                      181  0.190  2052/ 2992  70 ( 70)  READY
 156 attitude_estimator_q       3417  4.471  1796/ 2096 250 (250)  w:sem
 158 position_estimator_inav    1260  1.617  4572/ 4992 250 (250)  w:sem
 167 mc_att_control             1769  2.378  1068/ 1496 250 (250)  w:sem
 171 mc_pos_control              381  0.475  1092/ 1496 250 (250)  w:sem
 175 navigator                   464  0.570   836/ 1496 105 (105)  w:sem
```

Abbildung 3: Befehl top

2.3 uORB

Die uORB (Micro Object Request Broker) wird auf dem Pixhawk verwendet um Datenstrukturen, sogenannte Topics, zwischen Apps, also Threads, auszutauschen. Dies ermöglicht eine ressourcenarme Möglichkeit, auf interne Daten zu warten per Betriebssystem Interrupt. Man kann einen Filedescriptor auf ein gewünschte Topic anlegen, wie in folgendem Codebeispiel ersichtlich ist.

```
1  //File descriptor erzeugen und zuweisen
2    struct sensor_combined_s raw;
3    int sensor_sub_fd = orb_subscribe(ORB_ID(sensor_combined));
4    struct pollfd fds_uorb[]= {
5        { .fd = sensor_sub_fd,   .events = POLLIN },
6    };
7
8    [...]
9
10   //Warte auf neue Daten per OS Interrupt
11   poll_ret = poll(fds_uorb, 1, TIMEOUT_MS);
12
13   if (poll_ret <= 0)
14     //Keine Daten erhalten
15   else
16     //Neue Daten erhalten
```

Hier wird ein Filedescriptor erzeugt, welcher alle Sensordaten abonniert. Falls während dem Timeout 'TIMEOUT_MS' neue Daten auf die uORB geschrieben wurden, wird ein Interrupt ausgelöst und der return Wert wird grösser 0 sein.

2.4 cmake

Die Applikationsentwicklung auf dem Pixhawk ist modular aufgebaut. Durch Verwendung von cmake können Kompilerparameter einfach geändert oder Codesegmente hinzugefügt werden. Pixhawk führte cmake gegen Ende 2015 neu ein. Vorher basierte der Build Prozess auf make.
Durch folgende *CMakeLists.txt* Datei werden zwei neue Sourcedateien namens *my_app.c* und *crc.c* zum Build Prozess hinzugefügt. Die Priorität wurde sehr hoch angesetzt. Der Stack liegt mit 1200 Bytes auf der sicheren Seite. Stackoverflows sollten dadurch nicht vorkommen, solange keine rekursiven Methoden verwendet werden.
Falls die App programmiert wurde, kann ihre main Methode mit dem Kommando my_app in der nsh gestartet werden. Dies ist auf Zeile 3 definiert.

```
1  px4_add_module(
2      MODULE modules__my_app
3      MAIN my_app
4      PRIORITY "SCHED_PRIORITY_MAX-30"
5      STACK 1200
6      COMPILE_FLAGS
7          #${MODULE_CFLAGS}
8          #-Os
9      SRCS
10         my_app.c
11         crc.c
12     DEPENDS
13         #platforms__common  )
```

3 Toolchains

3.1 Installation

Die Installation der Pixhawk Toolchain wird auf www.pixhawk.org/dev/quickstart für alle Betriebssysteme sehr detailliert und ausführlich erklärt.

3.2 Verwendung

Mit dem Kommando **make px4fmu-v2_default** wird die Firmware kompiliert und die .elf Datei erzeugt. Mit **make px4fmu-v2_default upload** wird die Firmware kompiliert, .elf Datei erzeugt, auf das Pixhawk per USB hochgeladen und dort programmiert.

3.2.1 Linux

Für Linux Benützer ist es zu empfehlen, ein alias für die Kommandos zu erstellen. Dies kann erzeugt werden durch:

```
cd ~
vim .bashrc
```

Anschliessend die Taste 'a' drücken für den Eingabe Modus. Jetzt folgende Zeilen eingeben:

```
mk () {
  cd ~/path/to/Firmware/
  make px4fmu-v2_default
}

mkup () {
  cd ~/path/to/Firmware/
  make px4fmu-v2_default upload
}
```

Der Pfad zur Firmware muss vorher angepasst werden. Durch ESC kann anschliessend in den Navigationsmodus gewechselt werden. Mit ':wq' werden die Änderungen gespeichert und das Programm verlassen. Der Pfad zur Firmware muss vorher angepasst werden. Zum Schluss muss die **.bashrc** Datei neu kompiliert werden mit:

```
source .bashrc
```

Nun kann über das Terminal die Firmware kompiliert und hochgeladen werden mit dem Befehl 'mkup'. Falls man diesen nicht immer neu eingeben möchte, kann mit der Pfeiltaste ↑ oder !! das letzte Kommando erneut ausgeführt werden.

Für die Programmierung empfiehlt sich die IDE Code::Blocks. Auch hierzu gibt es eine Anleitung unter https://pixhawk.ethz.ch/toolchain/codeblocks. In der Anleitung sollten jedoch nur die stable builds verwendet werden und keine nightly builds.

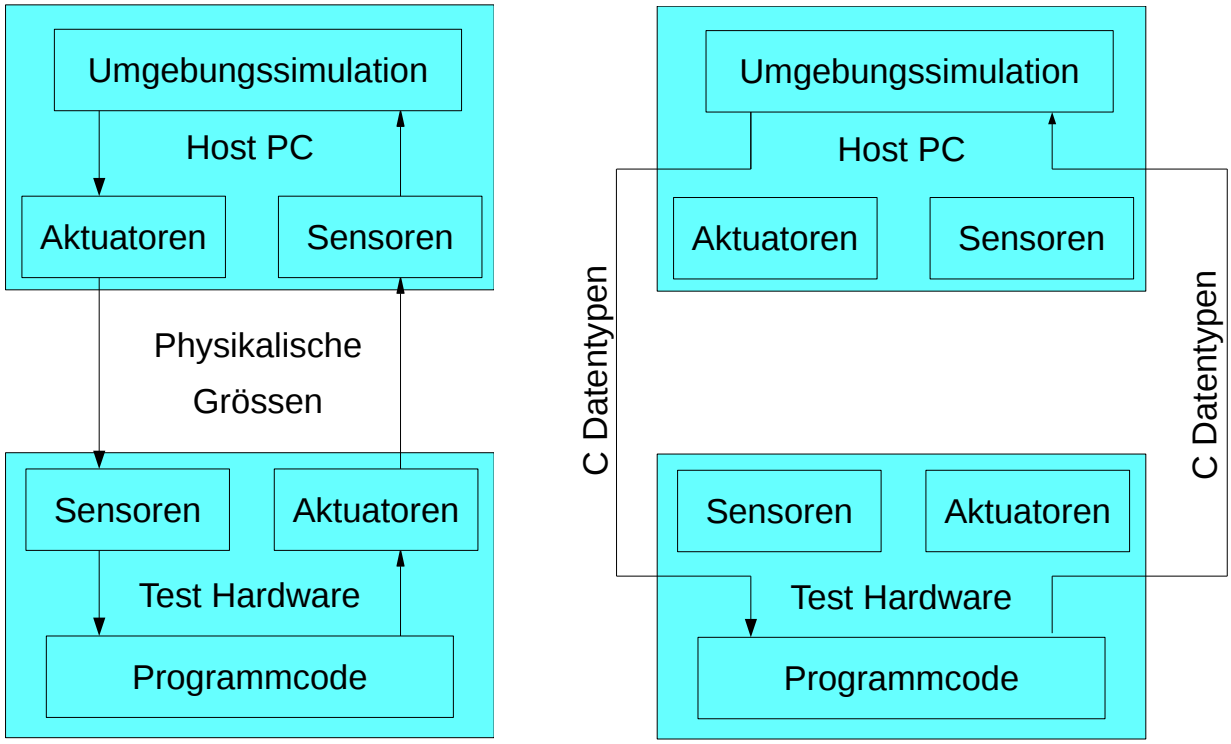
3.3 Troubelshooting

Während der Arbeit mit dem Pixhawk traten keine Fehler auf. Das px4 v1 (Pixhawk Version1) hatte im Vergleich einige zeitkritische Probleme beim Löschen und Programmieren der Firmware.

4 Hardware in the Loop

4.1 Einleitung

Bei HiL (Hardware in the Loop) handelt es sich um eine Simulation, bei welcher die Sensoren und Aktuatoren des Systems getestet werden sollen. Diese Simulationen werden oft für eingebettet Systeme verwendet. Der Vorteil liegt darin, dass bei einem Modellabsturz keine Schäden entstehen und der Vorgang zu diesem Ereignis klar verfolgbar ist durch die Log Dateien.



(a) Allgemeiner Hil Aufbau

(b) Pixhawk Hil Aufbau

Abbildung 4: Vergleich Hil Aufbau

Für die Pixhawk Simulation (Abbildung: 4b) werden die elektrischen Signale abgegriffen, welche die Aktuatoren ansteuern würden. Dies senkt die Simulationskomplexität stark. Bei einer echten HiL Simulation (Abbildung: 4a) müssten die Aktuatoren und Sensoren der Test Hardware auch miteinbezogen werden.

Beispiel Rotoren:
Bei einer echten HiL Simulation würde der Rotorenschub der Test Hardware durch ein Anemometer gemessen. Dieses Signal würde dann in dem Umgebungssimulator ausgewertet und die Aktuatoren Stellgrössen berechnen. Bei der Pixhawk HiL Simulation wird das PWM der Frequenzumrichter direkt an den Host PC per UART übermittelt. Die Aktautoren und Sensoren werden jeweils überbrückt.

4.2 Pixhawk and HiL

4.2.1 Sensorrauschen und Kalmann Filter

Hierbei handelt es sich **nicht** um eine HiL Simulation, sondern um einen ersten, einfachen Testaufbau mit funktionierender Datenübertragung zwischen Pixhawk und Simulink als Host PC.

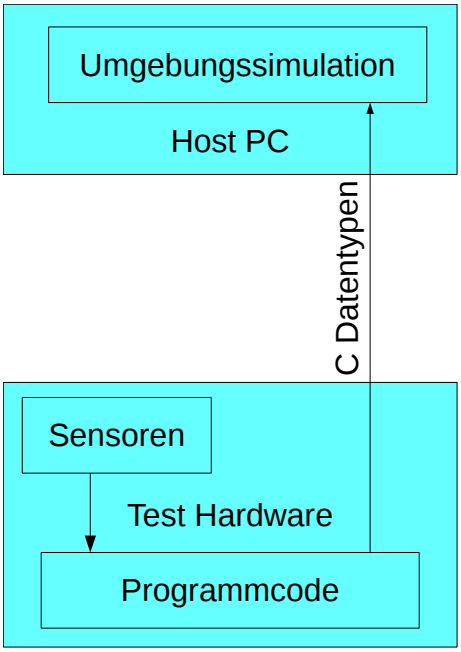


Abbildung 5: Messung Sensorrauschen

Die Sensordaten wie Gyro-, Accelero-, Barometer und Temperaturfühler werden von der Test Hardware entgegengenommen. Diese Daten durchlaufen einen Kalmann Filter und liegen dann als roh-, sowie gefilterte Werte auf der uORB zur Weiterverarbeitung. Diese Werte werden dann im Programmcode abgegriffen und über eine serielle Schnittstelle an den Host PC übermittelt.
Auf dem Umgebungssimulator kann nun ein Kalmann Filter auf die roh Werte angewendet werden und mit den gefilterten Werten verglichen.

4.2.2 Systemtest 1

In einem weiteren Test könnte die Flugsteuerung vom Pixhawk getestet werden wie in Abbildung 4b aufgezeigt. Die Datenübertragung wäre weiterhin auf einer seriellen Schnittstelle. Daruch wäre eine komplette HiL Simulation möglich.

4.2.3 Systemtest 2

Eine weitere Möglichkeit wäre, vorhandene Sensorwerte eines aufgezeichneten Flugs in the uORB einzuspeisen. Die berechnete Rotorenstellgrösse vom Pixhawk könnte man anschliessend mit den vorhandenen Aktuatorenwerten vergleichen.

5 Pilot Support Package

5.1 Einführung

Das PSP (Pixhawk Pilot Support Package) ist eine offizielle Toolbox von MathWorks. Diese Erweiterung für Simulink und Pixhawk Firmware ermöglicht es, eigene Flugregler für Pixhawk in Simulink zu entwerfen, welche in Echtzeit auf dem Pixhawk laufen. Um dies zu realisieren, wurden neue Simulink Blöcke hinzufügt (siehe Abbildung 6). Diese Blöcke können entweder Sensordaten ausgeben oder Aktuatorendaten empfangen.
In Simulink kann die eigene Flugsteuerung als Blockschaltbild aufgebaut werden. Diese wird dann mit dem Code Composer als C Code generiert und in die Pixhawk Firmware eingebunden.
Für die Verwendung muss zuerst die Toolbox unter https://ch.mathworks.com/hardware-support/forms/pixhawk-downloads.html heruntergeladen werden. Auf derselben Seite steht eine Anleitung mit Beispielen zur Verfügung.

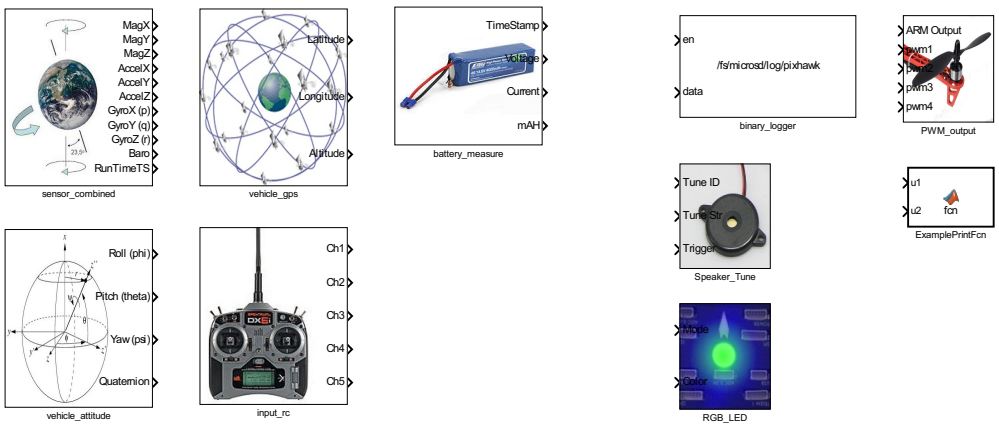


Abbildung 6: PSP Blöcke

5.2 Ausführung

Für die Ausführung und Verwendung der Simulink Modelle sollte vorzugsweise die PSP Anleitung konsultiert werden. Auf mehreren Seiten ist der Vorgang sowie Projekteinstellungen detailliert und ausführlich erklärt.
Anbei wird eine Flugregelung erzeugt, welche auf dem Pixhawk läuft und über das Simulink gesteuert werden kann. Das Vorgehen basiert auf den folgenden Schritten:

Modellentwurf
Das Modell in Abbildung 7 enthält den Flugregler, welcher später auf dem Pixhawk laufen soll. Der Regler berechnet anhand den Sensordaten die Stellgrössen der Aktuatoren.

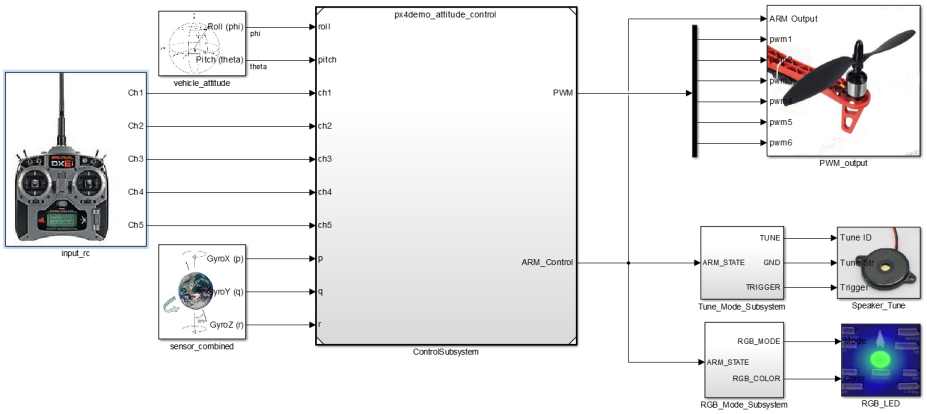


Abbildung 7: Simulink Modell

Projekteinstellungen
Eine Vielzahl von Einstellungen muss für den Code Composer vorgenommen werden. Je nach Verwendungszweck weichen diese ab.

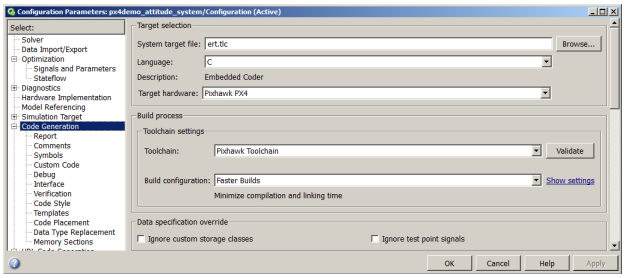


Abbildung 8: Projekteinstellungen

Codegeneration
Durch bestätigen des Knopfs in Abbildung 9 wird aus dem Simulink Blockschaltbild lauffähigen C Code generiert. Dieser Programmcode enthält mehrere Apps, welche automatisch auf das Pixhawk geladen werden. In Abbildung 10 ist der Simulink Diagnostic Viewer ersichtlich. Hier wird aufgezeigt, dass der C Code erfolgreich zu einer .elf Datei kompiliert wurde und auf das Pixhawk geladen würde.

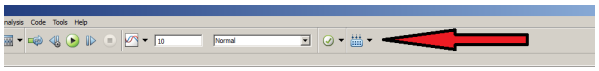


Abbildung 9: Code generieren und hochladen

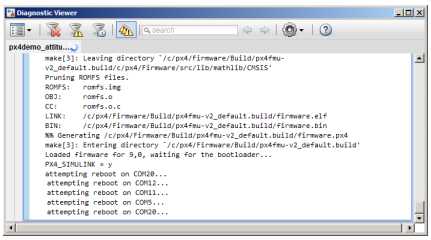


Abbildung 10: Diagnostic Viewer

Simulation / realer Test
Der Flugregler aus Abbildung 7 ist nun auf dem Pixhawk implementiert. Das Modell kann auf zwei Arten gestartet werden:

- Durch den 'run' Knopf im Simulink
- Mit dem nsh Kommando px4_simulink_app start

5.3 Auswertung

Das PSP dient dazu eigene Flugregler mit einfachen Mitteln und ohne Programmierkenntnissen zu entwerfen, realisieren und auf dem Fluggerät in realer Umgebung zu testen.
Weiter können Parameter (z.B. P-, I- Anteil, Matritzen) während der Laufzeit geändert werden. Dies sollte jedoch aufgrund der Performance nicht während dem Flug vorgenommen werden.

Im Simulink Modell können auch die Signale der PSP Blöcke zur Laufzeit angezeigt werden. Dadurch hat man einen Überblick der Gyro- Accel-, Batterie-Werte. Dies ist jedoch nur während der kabelgebundenen Ausführung möglich.

Das Tool hat jedoch nicht nur Vorteile. Durch die Struktur der Blöcke ist **keine** HiL Simulation, wie in Abbildung 4b aufgezeigt, möglich. Für die Umgebungssimulation auf Seiten Simulink (Host PC) stehen nur die Sensorenwerte zur Verfügung und nicht jene der Aktuatoren. Dadurch findet der Kreislauf in reversierter Abfolge statt.

Als weiterer Negativpunkt sind umständlichen Projekteinstellungen in Simulink aufzuführen. Diese weichen in jedem Beispielprojekt voneinander ab und falls ein Häcken bei einer Option falsch gesetzt wurde, funktioniert der Code Composer nicht. Die Fehlerausgabe dieses Tool ist dann bei der Fehlersuche nicht hilfreich.

6 Eigene HiL Simulation Entwicklung

Aufgrund des Misserfolgs vom PSP, wurde ein neuer Weg eingeschlagen. Es gibt bereits fertige, komplexe Pixhawk HiL Simulatoren, jedoch basieren diese auf mehreren, teils kostenpflichtigen, Programmen und der Support für diese Applikationen ist nicht gewährleistet.
Deshalb wurde eine eigene App für das Pixhawk entwickelt, welche die Daten auf dem Mikroprozessor modifizieren, auslesen oder einfügen kann. Dadurch ist dieses Lösungskonzept sehr mächtig, offen und erweiterbar. Auf der Gegenseite wurde eine Umgebungssimulation in Simulink programmiert, welche ohne Verzögerung ihre Aufgabe erfüllen soll.

6.1 App Entwicklung

Dieses Kapitel beschreibt der Aufbau der Pixhawk App. Dieses Programm kann völlig losgelöst vom Simulink programmiert werden.

6.1.1 Statemachine

Für diese App wurden mehrere Statemachines eingesetzt. Eine übernimmt die komplette Businesslogik für den Ablauf des Programms. Eine zweite Statemachine sorgt dafür, dass die serielle Datenkommunikation korrekt interpretiert wird.

Businesslogik
Die App besitzt zwei Threads. Der Erste ist nur für die Kommunikation und Interpretation der nsh Kommandos verantwortlich. Dieser Thread läuft nur ein paar Zyklen lang und steuert den zweiten Task. Die Steuerkommandos sind start, stop, status, help.

Der zweite Thread übernimmt die gesammte Businesslogik, welche in Abbildung 11 aufgezeigt wird. In dieser Logik werden jeweils Daten zwischen Interfaces ausgetauscht.

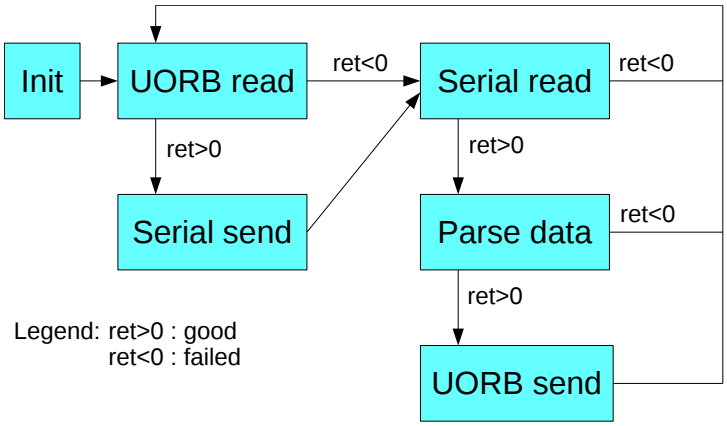
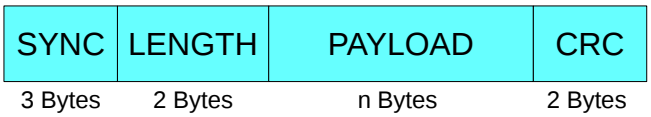


Abbildung 11: Statemachine Businesslogik

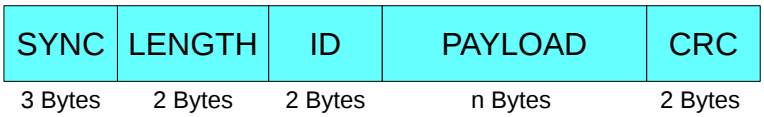
Nach der Initialisierung werden zwei 2 Hauptaufgaben ausgeführt. Zum einen die uORB Pakete auf die UART übertragen, zum anderen die UART Pakete in die uORB weiterleiten. Dieses Vorgehen ist in obiger Abbildung detailliert erklärt. Die Zustandsfunktionen haben jeweils einen return Wert (ret). Anhand dieses Wertes wird jeweils das weitere Vorgehen bestummen.

Parser

Die Daten werden jeweils als Paket übertragen. Diese Struktur ist in Abbildung 12a aufgezeigt. Eine Erweiterung wäre die ID der Payload zu übermitteln. Dadurch wüsste man, um welches uORB Topic es sich handelt.



(a) Packet Struktur ohne ID



(b) Packet Struktur mit ID

Abbildung 12: Packetstruktur

Die genaue Paketstruktur ist in folgender Tabelle aufgezeigt. Es handelt sich um das 'little Endian' Format.

Name	Inhalt	Länge (Byte)
SYNC	Synchronisationszeichen ASCII: #:_	3
LENGTH	Länge Bytelänge der Payload	2
ID	Identifikation: Bit Array, bei welchem jedes Bit einem definierten uORB Topic entspricht.	2
PAYLOAD	Nutzdaten Daten im roh Format	n
CRC	Checksumme CRC16 über die SYNC, LENGTH, (ID) und PAYLOD Daten	2

Tabelle 1: Paketstruktur Beschreibung

Durch die ID kann festgelegt werden, welche uORB Topics in der PAYLOAD vorhanden sind. Damit alle Permutation möglich sind, ist jedem ID-Bit ein uORB Topic zugewiesen.

Bits [MSB ... LSB]	uORB Topic Name
1xxx'xxxx xxxx'xxxx	Heartbeat
x1xx'xxxx xxxx'xxxx	Sensor combined
xxxx'xxxx 1xxx'xxxx	Actuator armed
xxxx'xxxx x1xx'xxxx	Actuator controls

Tabelle 2: Paket ID

Bei den empfangenen Daten muss zuerst der Beginn festgelegt werden. Für dies wird eine Parser (Abbildung 13 verwendet.

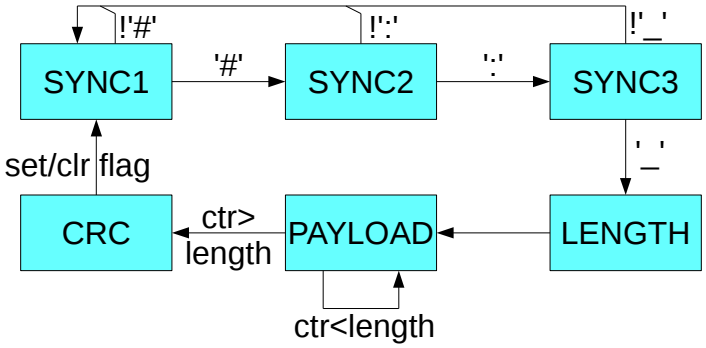


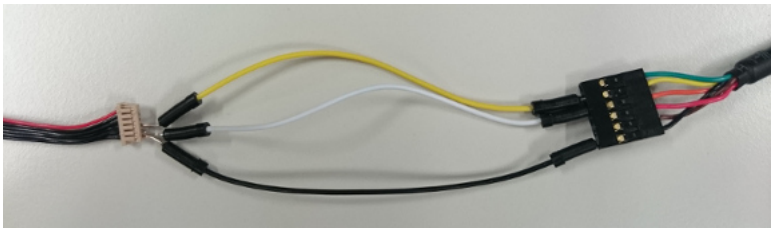
Abbildung 13: Statemachine Parser

Die Synchronisationsreihenfolge #:_ wurde desshalb gewählt, da diese eine zufällige Reihenfolge an Bits enthält. Weiter ist die Chance, dass dieses Bit-Reihenfolge erscheint $P = \frac{1}{2^{3 \cdot 8Bits}} \approx \frac{1}{16 \cdot 10^6} \approx 6 \cdot 10^{-8}$. Weiter sollte diese Reihenfolge im ASCII Format leicht ersichtlich sein, um das Debuggen zu vereinfacht. ASCII Steuerzeichen wurden bewusst nicht verwendet, da diese ja nach Betriebssystem, sowie Programm anders interpretiert werden können.
Am Ende des Pakets wird eine CRC16 mit dem Polynom $0x8005 = x^{16} + x^{15} + x^2 + 1$ angehängt. Durch Verwendung von nur einer Checksumme kann der Fehler im Paket nicht lokalisiert und korrigiert werden. Für die HiL Simulation steht die Geschwindigkeit jedoch im Vordergrund. Falls die Checksumme nicht übereinstimmt, wird das gesamte Paket verworfen.

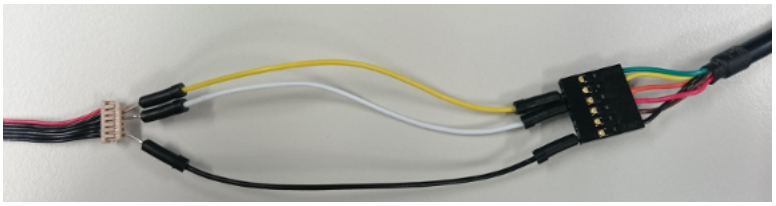
6.1.2 Serial

Die serielle Datenübertragung wird per UART realisiert. Das Pixhawk besitzt fünf serielle Interfaces. Einige dieser sind bereits reserviert, z.B. das GPS. Drei Interfaces stehen für die Entwickler zur Verfügung, AMC0 (USB Anschluss), ttyS5 (Serial 4) und ttyS6 (Serial5). Diese Bezeichnungen basieren auf GNU/Linux. Auf AMC0 und ttyS5 sind bereits belegt durch eine nsh. Die Implementation der Lese- und Schreib-Methode sind nicht reentrent. Deshalb sollte die Schnittstelle nur von einem Thread verwendet werden. Aus diesem Grund wurde **ttyS6** für den Datenstrom verwendet, damit keine anderen App das Interface belegt.

Auf der Host Seite wird ein USB $\leftrightarrow$ UART Konverter verwendet. Für diese PAIND Arbeit wurde ein offizielles FTDI Kabel verwendet. Je nach Anschluss (siehe Abbildung 14) wird eine andere Schnittstelle verwendet. Für diese Arbeit wurde ttyS6 belegt.



(a) ttyS5 mit nsh



(b) ttyS6 ohne nsh

Abbildung 14: Serielle Schnitstelle

Anbei die Farbcodierung der Signalleitungen:

Farbe	Verwendung Pixhawk	Verwendung Host PC
gelb	TX	RX
weiss	RX	TX
schwarz	GND	GND

Tabelle 3: UART Farbcodierung

Baud
Das uORB Datenpaket 'Sensor combined' enthält 722 Bytes. Dies könnte eine hohe Bandbreite erfordern. Es ist nicht genau klar, wie oft dieses in der uORB bereitsteht. Deshalb wurde diese Bandbreite gemessen. Dafür wurde die Baudrate sehr hoch angesetzt, damit die Daten sofort gesendet werden und der Buffer kein Überlauf erfährt. Die Messung ergab folgende Werte: $\frac{390240 Bytes}{16s} = 24390 \frac{Byte}{s}$.
Jedes Byte enthält zusätzlich ein Start- und Stopbit, somit ergibt sich:
$1 \frac{Nutzbyte}{s} = 1 \frac{Startbit}{s} + 8 \frac{Nutzbits}{s} + 1 \frac{Stopbit}{s} = 10 \frac{Bit}{s} = 10 Baud.$
Dadurch wird der Umrechnungsfaktor 10 bestummen. Falls nun die gemessen Byterate umgerechnet wird:

$24390\frac{Byte}{s} = 243900Baud$.

Die verfügbaren Baud sind [..., 115200, 230400, 460800, 921600]. Damit die Daten keine Verzögerung erfahren und aufgrund bidirektionaler Kommunikation, wurde die Baud von $460800Baud$ gewählt. Bei $921600Baud$ ist die Störungsanfälligkeit noch höher.

Code
Die Implementierung des POSIX Interfaces der seriellen Schnittstelle ist nicht multithread fähig. Dadurch kann nur 1 Thread auf den Filedescriptor zugreifen. Falls ein zweiter Versucht den selben Filedescriptor zu verwenden, kommt es zu Laufzeitfehler.

Die Verwendung des Interfaces ist einfach gehalten.

```
1  /** open Serial port **/
2  fd_serial = open(PORT_TTYS, O_RDWR);
3  if (fd_serial < 0) {
4    thread_should_exit = true;
```

Durch dieses Kommando wird dem Filedescriptor fd_serial eine neue Indexnummer zugewiesen. Alle weiteren Aktionen basieren nun auf diesem Index. Falls der Index kleiner 0 ist, konnte der Port nicht geöffnet werden.

```
1  write(fd_serial, buf_send, i);
```

Das Senden erfolgt durch Übergabe eines char Pointers und der Datenlänge. Hierbei ist zu beachten, dass die Länge in Form des Datentyps angegeben wird. i=1 entspricht für ein char Pointer 8 Bits.

```
1  /** poll-file-descriptor for serial interface **/
2  struct pollfd fds_serial[]= {
3    {.fd = fd_serial,   .events = POLLIN },};
4
5  [...]
6
7  poll_ret = poll(fds_serial, 1, TIMEOUT_RCV_POLL_MS);
8  if (poll_ret > 0){
9    //We got new data on serial Interface
10 }
```

Die uORB kann auch Filedescriptors aufnehmen. Durch obigen Code wird die serielle Schnittstelle per uORB überwacht. Falls neue Daten anliegen oder das Timeout auf Zeile 7 ausgelaufen ist, wird ein Betriebssystem Interrupt ausgeführt. Falls der return Wert grösser 0 ist, liegen neue Daten an der seriellen Schnittstelle.

```
1  int ctr = read(fd_serial, buf_tmp, BUFFER_SIZE_RCV);
```

Das Auslesen der Daten erfolgt ressourcenschonend. Man übergibt der read Funktion den Filedescriptor, die Adresse des char Buffers und die maximale Anzahl Zeichen, welche man empfangen möchte. Als return Wert bekommt man die Anzahl empfangener Daten, welche nun im char Buffer bereitstehen.

6.1.3 Auswertung

Mit dieser App ist eine HiL Simulation auf seitens Pixhawk möglich. Es können die Stellgrössen der Aktuatoren über die UART gesendet werden und generierte Sensorwerte den Reglern "vorgehalten"werden. Die App ist sehr ressourcenschonend mit nur 12% CPU Auslastung, trotz den grossen Datenmengen. Falls die komplette Bandbreite benützt würde, wäre die CPU Auslastung voraussichtlich 25%.

6.2 Simulink

Auf Simulink Seite wird die Umgebungssimulation vorgenommen. Dafür müssen die empfangenen Datenpakete auch auf dieser Seite zuerst geparst werden. Anschliessend durchlaufen die Werte einen Algorithmus, welcher neue Werte an das Pixhawk sendet.

6.2.1 Kommunikation

Simulink stellt in der 'Instrumental Control Toolbox' TCP/UDP, IP und serielle Kommunikationsblöcke zur Verfügung. Diese Toolbox ist nicht in der HSLU Lizenz einbezogen.

Für die Datenübertragung per Serial Port muss dieser zuerst korrekt konfiguriert werden im Block 'Serial Configuration' (Abbildung 15). Die Baud, Kontrollbits, Symbolgrösse, Byte-Reihenfolge sowie ein Timeout müssen angegeben werden.

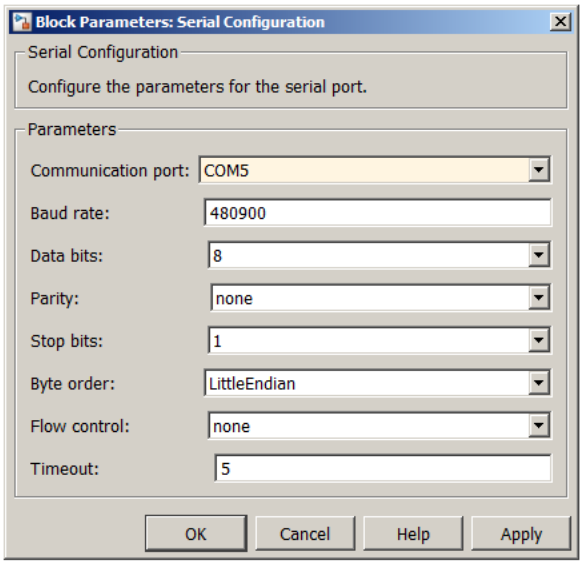


Abbildung 15: Einstellungen Konfigurationsblock

Der Empfängerblock (Abbildung 16), welcher die Daten ins Simulink einspeist, muss auch konfiguriert werden. Hierbei gilt zu beachten, dass der 'Blocking Mode' ausgeschaltet wird. Dieser Parameter erlaubt, dass die Simulation weiterlaufen kann, obwohl keine neuen Daten empfangen wurden. Der Parser kann auch gleich übergeben werden im Header Feld. In diesem Block wird versucht, alle 50ms ein Packet von 772 Bytes zu empfangen. Diese Paketstruktur entsprechen dem Format von Abbildung 12a und muss für die Verwendung zuerst noch geparst werden. Würde man jedes Byte einzeln empfangen, wäre dies sehr ineffizient. Der Block enthält ein Sleep Befehl von 1ms. Dies würde bei einer kurzen Abtastzeit stark ins Gewicht fallen. Bei den 50ms im Projekt ist dies jedoch vernachlässige.

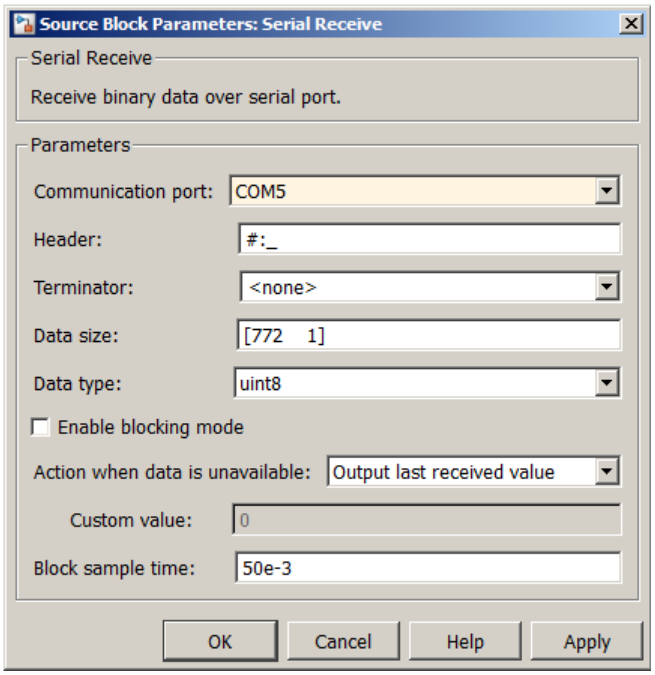


Abbildung 16: Einstellungen Empfänger Block

Der Sendeblock (Abbildung 17) extrahiert die Simulationsdaten und sendet diese ans Pixhawk. Die drei Synchronisationsbyte können als Paketkopf übergeben werden.

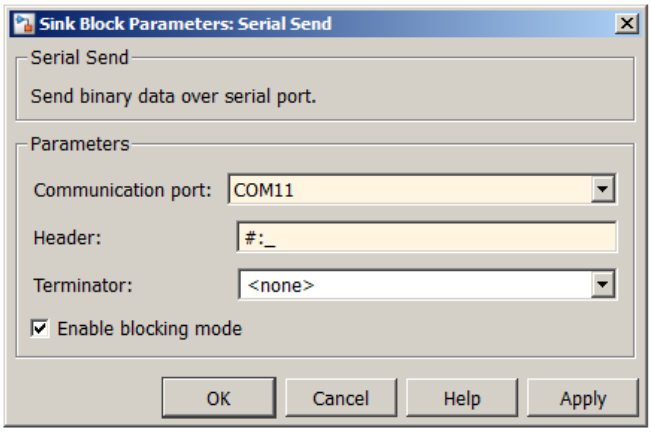


Abbildung 17: Einstellungen Sender Block

Diese drei Blöcke übernehmen die gesamte Kommunikation auf Seite Simulink. Der Inhalt des Datenpakets müssen nun gegliedert und einer Typenumwandlung unterzogen werden.

6.2.2 Parser

Damit die Daten im Simulink in sinnvoller Struktur verfügbar sind, muss der Inhalt entpackt werden. Dazu ist momentan die Struktur der Daten nötig. Falls die Paket ID implementiert wäre, könnte man anhand der gesetzten ID-Bits diese entpacken. Diese Aufgabe übernimmt ein 'Interpreted MATLAB Function' Block.

```
1  function output_args = parser_small(input_args)
2
3  global pck_length;
4
5  % Parse package length
6  pck_length = input_args(1,:);
7  pck_length = pck_length + 256*input_args(2,:);
8
9  % Parse package ID
10
11
12 % Parse package payload and convert
13 time = double(typecast(uint8(input_args(3:10)), 'uint64'))/1e6;
14 g_x = double(typecast(uint8(input_args(207:210)), 'single'));
15 g_y = double(typecast(uint8(input_args(211:214)), 'single'));
16 g_z = double(typecast(uint8(input_args(215:218)), 'single'));
17
18 % crc detector
19
20
21 % output data
22 output_args = [time ; g_x ; g_y ; g_z ];
```

Der Parser muss folgende Aufgaben übernehmen:

Zeile	Aktion
6...7	Paketlänge zusammensetzten
10	ID auslesen
13..16	Nutzdaten auslesen und casten
19	CRC16 Überprüfung der Nutzdaten
22	Vektorausgabe der Werte

Tabelle 4: Simulink Parser

6.3 Simulation

Durch Verwendung der Pixhawk App sowie der Simulink Blöcke kann eine einfache Datenstromverarbeitung erfolgen. Die digitalen Dateien sind im Kapitel 8 vorhanden.

Das Subsystem in Abbildung 18 konfiguriert den COM Port und öffnet diesen. Falls dies fehlschlägt wird die Simulation automatisch abgebrochen.

Abbildung 18: Serielle Schnittstelle

Die Pakete werden im System verarbeitet. Dazu wird eine 'Interpreted MATLAB Function' verwendet. Dieser Block unterstützt nur das double Format. Aus diesem Grund wurde ein Datentyp Konverter hinzugefügt. Der Parser liefert ein Vektor, welcher mit einem Demultiplexer die Kanäle separiert.

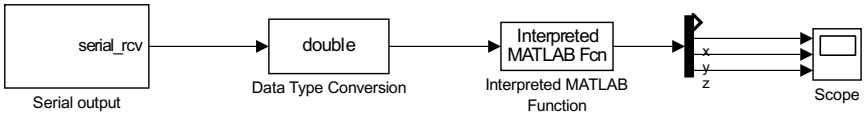


Abbildung 19: Serielle Schnittstelle

In der Abbildung 20 ist der Scope ersichtlich. Auf der x-Achse befindet sich die Zeit in Sekunden. Die y-Achse beschreibt die Beschleunigung in x, y und z-Richtung in $\frac{m}{s^2}$. Der Scope kann während der Simulation geöffnet werden und zeigt die Daten zur Laufzeit an.

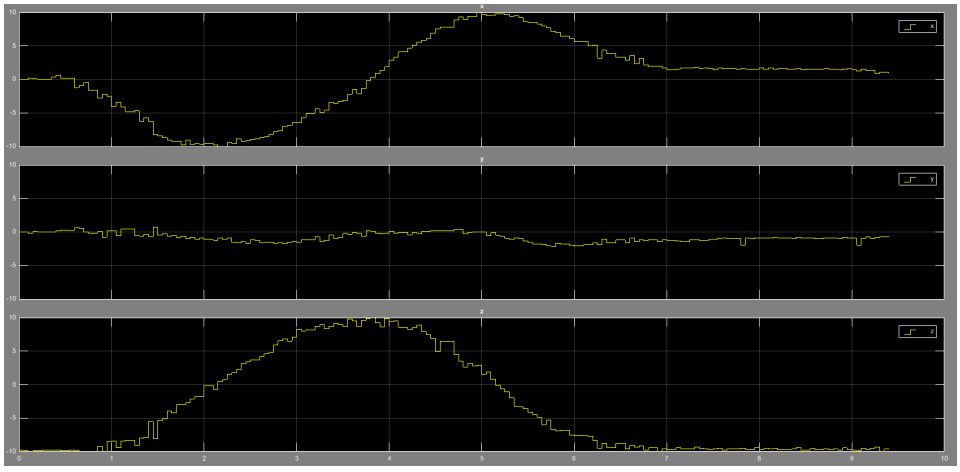


Abbildung 20: Accelorometer während eines Looping

7 Auswertung

Fachliches Fazit

Durch die Pixhawk App wurde ein mächtiges Tool entwickelt, um Daten auszulesen, zu modifizieren und einzuspeisen. Dieses kann auch als Basis für andere Aufgaben dienen. Selbes gilt auch für die Simulink Blöcke.
Durch das Zusammenspiel der Pixhawk App sowie dem Simulinkmodell wurde erfolgreich eine stabile Datenstromverarbeitung realisiert. Die Daten können auf der jeweiligen Seite mit einer hohen Baud gesendet, empfangen und interpretiert werden. Die CPU Auslastung ist auf beiden Seiten sehr gering in Anbetracht der grossen Datenmengen.
Eine HiL Simulation konnte in der vorgegebenen Zeit nicht realisiert werden.

Persönliches Fazit

Durch dieses spannende Arbeit erhielt ich Einblicke in die Aviatik, Flugregelung, C sowie C++ Programmierung Die Arbeit war in drei Aufgaben aufgeteilt und jede Arbeit erforderte unterschiedliche Tools. Zum einen musste man sich in das PSP und den Code Composer einarbeiten, zu anderen in die Pixhawk Firmware und zum Schluss noch ins Simulink mit einer Datenstrom-Verarbeitung und -Ausgabe. Jedes dieser Tools benötigte eine Einarbeitungszeit. Das erlernte Hochschul-Wissen konnte teilweise angewendet werden, jedoch haben Elektroniker ein sehr kleines Programmierwissen im Vergleich zu den Informatikern. Aus meiner Sicht hätte ein Interdisziplinäres Team zu einem besseren Ergebnis geführt.
Durch die Einzelarbeit konnte ich jedoch einen grösseren, persönlichen Nutzen erarbeiten. Es ermöglichte einen Einblick in die anderen Fachgebiete.

8 Anhang

Literatur

Hambarde P., V. R. (2014). *The survey of real time operating system: Rtos* [Report].

Kuznicki, S. (2015). *Pixhawk pilot support package (psp) user guide.*

Meier, L., Honegger, D. & Pollefeys, M. (2015a). *Git tutorial.* Zugriff am 2015-10-27 auf `https://pixhawk.org/dev/git`

Meier, L., Honegger, D. & Pollefeys, M. (2015b). *Pixhawk autopilot.* Zugriff am 2015-10-27 auf `https://pixhawk.org/modules/pixhawk`

Meier, L., Honegger, D. & Pollefeys, M. (2015c). *Pixhawk daemon app.* Zugriff am 2015-10-27 auf `https://pixhawk.org/dev/px4_daemon_app`

Meier, L., Honegger, D. & Pollefeys, M. (2015d). *Pixhawk hil for developer.* Zugriff am 2015-10-27 auf `https://pixhawk.org/dev/hil/start`

Meier, L., Honegger, D. & Pollefeys, M. (2015e). *Pixhawk lab.* Zugriff am 2015-10-27 auf `https://pixhawk.ethz.ch/dev/tutorials/introduction`

Meier, L., Honegger, D. & Pollefeys, M. (2015f). *Pixhawk quickstart.* Zugriff am 2015-10-27 auf `https://pixhawk.org/dev/quickstart`

Meier, L., Honegger, D. & Pollefeys, M. (2015g). *Pixhawk simple app.* Zugriff am 2015-10-27 auf `https://pixhawk.org/dev/px4_simple_app`

Meier, L., Honegger, D. & Pollefeys, M. (2015h). *Pixhawk wiring.* Zugriff am 2015-10-27 auf `https://pixhawk.org/dev/wiring`

Meier, L., Honegger, D. & Pollefeys, M. (2015i). *Pixhawl hil simulation setup.* Zugriff am 2015-10-27 auf `https://pixhawk.org/users/hil`

Meier, L., Honegger, D. & Pollefeys, M. (2015j, Mai). PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *Robotics and automation (icra), 2015 ieee international conference on.*

Meier, L., Honegger, D. & Pollefeys, M. (2015-10-27). *uorb.* Zugriff am 2015-10-27 auf `https://pixhawk.org/dev/shared_object_communication`

Nuttx rtos, overview. (2015). Zugriff auf `http://www.nuttx.org/Documentation/NuttX.html#overview`

Rowebots. (2015). *Posix and linux compatible.* Zugriff am 2015-10-27 auf `http://www.rowebots.com/img/Unison_POSIX_RTOS_Model.jpg`

Shotts, W. (2012). *Linux command line.* No Starch Press.

Abbildungsverzeichnis

Tabellenverzeichnis

A CD ROM

B Code Pixhawk

my_app.h

src/my_app/my_app.h

```
#ifndef MY_APP_H
#define MY_APP_H


/*
 * Serial port
 */
#include <stdio.h>   /* Standard input/output definitions */
#include <string.h>  /* String function definitions */
#include <unistd.h>  /* UNIX standard function definitions */
#include <fcntl.h>   /* File control definitions */
#include <errno.h>   /* Error number definitions */
#include <termios.h> /* POSIX terminal control definitions */


/*
 * For uORB and topics
 */
#include <nuttx/config.h>
#include <nuttx/sched.h>
#include <poll.h>
#include <uORB/uORB.h>
#include <uORB/topics/sensor_combined.h>


/*
 *
 */
#include <systemlib/systemlib.h>
#include <systemlib/err.h>


/*
 *
 */
#include <drivers/drv_hrt.h>
#include <px4_time.h>
#include <px4_config.h>


/*
 * Prototypes
 */
int my_app_main(int argc, char *argv[]);
int serial_task_main(int argc, char *argv[]);


int threads_start(int argc, char *argv[]);
```

```
49 int threads_status(void);
50 int threads_help(void);
51 int threads_stop(void);
52
53 int uart_config(void);
54
55 void parse_buf(const uint8_t b);
56
57
58
59 #endif /* MYAPP_H */
```

my_app.c

src/my_app/my_app.c

```
1  //Default includes
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6  #include <crc32.h>
7
8
9  #include "my_app.h"
10 #include "crc.h"
11
12
13 /*
14  * Defines
15  */
16 #define deamon_task_stack_size  (4096)  // could be smaller, but set higher
       to reduce potential nasty problem
17 #define BUFFER_SIZE_SEND        (1024)
18 #define BUFFER_SIZE_RCV         (1024)
19 #define BUFFER_SIZE_CRC         (2)     // 4 bytes - 32 bit - crc32
20
21 #define TIMEOUT_SEND_POLL_MS    (200)  // max waiting time for poll function
22 #define TIMEOUT_RCV_POLL_MS     (200)  // max waiting time for poll function
23 #define TIMEOUT_RCV_US          (20e3)  // wait after successful poll for
       more data
24 #define TIMEOUT_TASK_US         (50e3)  // 50 ms wait
25
26 #define PORT_TTYS               "/dev/ttyS6"
27 #define BAUD                    (460800)            //115200   460800
28
29 #define SEND_DATA               1
30 #define RCV_DATA                0
31
32
33 /*
34  * Global Variables
35  */
```

```
36 static bool thread_should_exit = false;     /** daemon exit flag **/
37 static bool thread_running =     false;     /** daemon status flag **/
38 static bool data_rcv_good =      false;
39 static int daemon_task =         0;         /** Handle of daemon task/thread
       **/
40
41 static int fd_serial =           ERROR;
42
43 static uint16_t crc_calc =       0;
44 static uint16_t length_payload = 0;
45
46
47 static char buf_send[BUFFER_SIZE_SEND];
48 static char buf_rcv[BUFFER_SIZE_RCV];
49
50
51 typedef enum {
52     SYNC1 = 1,
53     SYNC2,
54     SYNC3,
55     LENGTH_1,
56     LENGTH_2,
57     PAYLOAD,
58     CHKSUM_GET_1,
59     CHKSUM_GET_2,
60     CHKSUM_CALC,
61 } state_t;
62
63
64 /*
65  *
66  * uORB <--> Serial converter
67  *
68  */
69 int
70 serial_task_main(int argc, char *argv[])
71 {
72     warnx("Thread␣starting\n");
73     thread_running = true;
74
75     int error_counter = 0;
76     int poll_ret = 0;
77     uint16_t i = 0;
78
79     fd_serial = open(PORT_TTYS, O_RDWR);
80
81     /** open Serial port **/
82     if (fd_serial < 0) {
83         warnx("Failed␣to␣open␣serial␣port,␣fd␣=␣%i", fd_serial);
84         thread_should_exit = true;
85     } else {
86         warnx("Serial␣port␣opened␣successful,␣fd␣=␣%i", fd_serial);
```

```
87      }
88
89      /** configure Serial port **/
90      if (uart_config() == ERROR) {
91          warnx("Serial port configure failed successful");
92          thread_should_exit = true;
93      } else {
94          warnx("Serial port configured successful");
95      }
96
97      warnx("Creating file descriptors");
98  #if SEND_DATA == 1
99      /* poll-file-descriptor for uORB topic(s) */
100     struct sensor_combined_s raw;
101     int sensor_sub_fd = orb_subscribe(ORB_ID(sensor_combined));
102     struct pollfd fds_uorb[]= {
103         { .fd = sensor_sub_fd,   .events = POLLIN },
104     };
105 #endif // SEND_DATA
106
107 #if RCV_DATA
108     /* poll-file-descriptor for serial interface */
109     struct pollfd fds_serial[]= {
110         {.fd = fd_serial,   .events = POLLIN },
111     };
112 #endif // RCV_DATA
113     warnx("File descriptors created");
114
115
116     /** For debug purpose **/
117     memcpy(buf_rcv,  "a", 1);
118     memcpy(buf_send, "a", 1);
119
120
121     while (!thread_should_exit) {
122
123 #if SEND_DATA == 1
124         {
125
126             /*
127              *
128              * Get uORB messages and send them to serial
129              *
130              */
131             /* wait for sensor update of 1 file descriptor for 1000 ms (1
                    second) */
132             poll_ret = poll(fds_uorb, 1, TIMEOUT_SEND_POLL_MS);
133
134             /* handle the poll result */
135             if (poll_ret == 0)
136             {
137                 /* this means none of our providers is giving us data */
```

```
138                 warnx("Got␣no␣data␣within␣%i␣milli␣seconds\n",
                        TIMEOUT_SEND_POLL_MS);
139             } else if (poll_ret < 0)
140             {
141                 /* this is seriously bad - should be an emergency */
142                 if (error_counter < 10 || error_counter % 50 == 0) {
143                     /* use a counter to prevent flooding (and slowing us
                            down) */
144                     warnx("ERROR␣return␣value␣from␣poll():␣%d\n"
145                     , poll_ret);
146                 }
147                 error_counter++;
148             } else {
149                 if (fds_uorb[0].revents & POLLIN)
150                 {
151                     i = 0;
152 
153                     /* copy sensor raw data into local buffer */
154                     orb_copy(ORB_ID(sensor_combined), sensor_sub_fd, &raw);
155 
156                     /** Add Parser item **/
157                     char parser[3] = "#:_";
158                     memcpy(&buf_send[i], &parser, SYNC3);
159                     i += SYNC3;
160 
161 
162                     /** Add length **/
163                     uint16_t tmp = sizeof(raw);
164                     memcpy(&buf_send[i], &tmp, sizeof(tmp));
165                     i += sizeof(i);
166 //                  warnx("Payload length is %i",sizeof(raw));
167 
168 
169                     /** Add payload **/
170                     memcpy(&buf_send[i], &raw, sizeof(raw));
171                     i += sizeof(raw);
172 //                  warnx("g1 is %2.4f\n",
    (double)raw.accelerometer_m_s2[0]);
173 //                  warnx("g2 is %2.4f\n",
    (double)raw.accelerometer_m_s2[1]);
174 //                  warnx("g3 is %2.4f\n",
    (double)raw.accelerometer_m_s2[2]);
175 
176 
177                     /** Add checksum **/
178                     crc_calc = gen_crc16((uint8_t*)buf_send + 5*sizeof(char)
                            , (i - SYNC3 - 2));// crc of payload, sync or length
                            is not included
179                     memcpy(&buf_send[i], &crc_calc, BUFFER_SIZE_CRC);
180                     i += BUFFER_SIZE_CRC;
181 
182 
```

```
183                     /** print crc for debug **/
184                     char tmp2[2];
185                     sprintf(tmp2,"%d",crc_calc);
186 //                  warnx("crc is %s \n", tmp2);
187
188
189                     /** Send data over serial **/
190                     write(fd_serial, buf_send, i);
191 //                  warnx("Sending data!\n\n");
192                 }
193             }
194         }
195 #endif // SEND_DATA
196
197 #if RCV_DATA == 1
198         {
199             /*
200              *
201              * Get serial messages and send them to uORB
202              *
203              */
204             /* poll for new data */
205             poll_ret = poll(fds_serial, 1, TIMEOUT_RCV_POLL_MS);
206
207
208             /* handle the poll result */
209             if (poll_ret == 0)
210             {
211                 /* this means none of our providers is giving us data */
212                 warnx("Got␣no␣data␣within␣%i␣milli␣seconds\n",
                        TIMEOUT_RCV_POLL_MS);
213             } else if (poll_ret < 0)
214             {
215                 /* this is seriously bad - should be an emergency */
216                 if (error_counter < 10 || error_counter % 50 == 0) {
217                     /* use a counter to prevent flooding (and slowing us
                            down) */
218                     warnx("ERROR␣return␣value␣from␣poll():␣%d\n", poll_ret);
219                 }
220                 error_counter++;
221
222             } else if (poll_ret > 0)
223             {
224                 /* if we have new data from GPS, go handle it */
225                 if (fds_serial[0].revents & POLLIN) {
226                     //warnx("Waiting for Data\n");
227                     usleep(TIMEOUT_RCV_US);
228
229                     char buf_tmp[BUFFER_SIZE_RCV*2];
230                     /*
231                      * We are here because poll says there is some data, so
                            this
```

```
232                      * won't block even on a blocking device.  If more bytes
                             are
233                      * available, we'll go back to poll() again...
234                      */
235
236                     int ctr = read(fd_serial, buf_tmp, BUFFER_SIZE_RCV);
                           //only saved data on newline
237
238                     for (i = 0; i < ctr; i++) {
239                         parse_buf(buf_tmp[i]);
240                     }
241                     //warnx("Data received");
242
243                     if(data_rcv_good) {
244                         data_rcv_good = 0;
245                         warnx("Good data received, writing to uORB");
246
247                         // .....
248
249                     }
250                 }
251             }
252         }
253 #endif // RCV_DATA
254
255         // Slow down the whole app
256         usleep(TIMEOUT_TASK_US);
257
258     }
259     warnx("Thread: exiting.\n");
260     thread_running = false;
261
262     close(fd_serial);
263     return 0;
264 }
265
266
267 /**
268  * The daemon app only briefly exists to start
269  * the background job. The stack size assigned in the
270  * Makefile does only apply to this management task.
271  *
272  * The actual stack size should be set in the call
273  * to task_create().
274  */
275 int
276 my_app_main(int argc, char *argv[])
277 {
278     if (argc < 2) {
279         warnx("missing command");
280         return 1;
281     }
```

```
282
283     if (!strcmp(argv[1], "start")) {
284         return threads_start(argc, argv);
285     }
286
287     if (!strcmp(argv[1], "stop")) {
288         return threads_stop();
289     }
290
291     if (!strcmp(argv[1], "status")) {
292         return threads_status();
293     }
294
295     if (!strcmp(argv[1], "help") || !strcmp(argv[1], "--help") ||
        !strcmp(argv[1], "-h")) {
296         return threads_help();
297     }
298
299     // Wrong input detected
300     warnx("my_app:␣unrecognized␣command,␣try␣--help\n");
301     return 1;
302 }
303
304
305 int
306 threads_start(int argc, char *argv[])
307 {
308     if (thread_running) {
309         warnx("thread␣is␣already␣running\n");
310         /* this is not an error */
311     } else {
312         thread_should_exit = false;
313         daemon_task = px4_task_spawn_cmd(
314                               "hil_simulation",
315                               SCHED_DEFAULT,
316                               SCHED_PRIORITY_DEFAULT,
317                               deamon_task_stack_size,
318                               serial_task_main,
319                               (argv) ? (char *const *)&argv[2] : (char *const
                                  *)NULL);
320     }
321     return 0;
322 }
323
324
325 int
326 threads_status(void)
327 {
328     if (thread_running) {
329         warnx("\tthread␣is␣running\n");
330     }  else  {
331         warnx("\tthread␣not␣started\n");
```

```
332     }
333     return 0;
334 }
335 
336 
337 int
338 threads_help(void)
339 {
340     warnx("options:\n");
341     printf("\t␣start:␣Start␣the␣application\n");
342     printf("\t␣stop:␣Stop␣the␣application\n");
343     printf("\t␣status:␣Request␣current␣application␣status\n");
344     return 0;
345 }
346 
347 
348 int
349 threads_stop(void)
350 {
351     thread_should_exit = true;  /** Set static variable */
352     warnx("stop");
353     return 0;
354 }
355 
356 
357 int
358 uart_config(void)
359 {
360     struct termios uart_config;
361 
362     int termios_state;
363 
364     /* fill the struct for the new configuration */
365     tcgetattr(fd_serial, &uart_config);
366 
367     /* clear ONLCR flag (which appends a CR for every LF) */
368     uart_config.c_oflag &= ~ONLCR;
369     /* no parity, one stop bit */
370     uart_config.c_cflag &= ~(CSTOPB | PARENB);
371 
372     /* set baud rate */
373     if ((termios_state = cfsetispeed(&uart_config, BAUD)) < 0) {
374         warnx("ERR:␣%d␣(cfsetispeed)\n", termios_state);
375         return ERROR;
376     }
377 
378     if ((termios_state = cfsetospeed(&uart_config, BAUD)) < 0) {
379         warnx("ERR:␣%d␣(cfsetospeed)\n", termios_state);
380         return ERROR;
381     }
382 
383     if ((termios_state = tcsetattr(fd_serial, TCSANOW, &uart_config)) < 0) {
```

```
384         warnx("ERR:␣%d␣(tcsetattr)\n", termios_state);
385         return ERROR;
386     }
387     return 0;
388 }
389 
390 void
391 parse_buf(const uint8_t b)
392 {
393     static state_t state;
394     static int ctr_payload;
395     static uint32_t crc_sent;
396 
397 //  warnx("char is %i", b);
398 
399     switch (state) {
400     case SYNC1:
401 //      warnx("SYNC 1");
402 
403         if(b == 0x23)           // #
404             state = SYNC2;
405         break;
406 
407 
408     case SYNC2:
409 //      warnx("SYNC 2");
410         if(b == 0x3A)           // :
411             state = SYNC3;
412         else
413             state = SYNC1;
414         break;
415 
416 
417     case SYNC3:
418 //      warnx("SYNC 3");
419         if(b == 0x5F){          // _
420             ctr_payload = 0;    // reset values
421             crc_sent    = 0;
422             crc_calc    = 0;
423             state = LENGTH_1;
424 
425         }
426         else
427             state = SYNC1;
428         break;
429 
430 
431     case LENGTH_1:
432 //      warnx("LENGTH_1");
433         length_payload += 256*(uint8_t)b;
434         state = LENGTH_2;
435         break;
```

```
436
437
438     case LENGTH_2:
439 //      warnx("LENGTH_2");
440         length_payload += (uint8_t)b;
441 //      warnx("length is %i", length_payload);
442         state = PAYLOAD;
443         break;
444
445
446
447     case PAYLOAD:
448         if(ctr_payload < length_payload-1) {
449             buf_rcv[ctr_payload] = b;
450 //          warnx("payload char is %c", buf_rcv[ctr_payload]);
451             ctr_payload++;
452         } else if (ctr_payload == length_payload-1) {           //change
               state without losing a byte
453             buf_rcv[ctr_payload] = b;
454 //          warnx("payload char is %c", buf_rcv[ctr_payload]);
455             state = CHKSUM_GET_1;
456         }
457         break;
458
459
460     case CHKSUM_GET_1:
461         crc_sent += ((uint8_t)b)<<8;
462         state = CHKSUM_GET_2;
463         break;
464
465
466     case CHKSUM_GET_2:
467         crc_sent += (uint8_t)b;
468         state = CHKSUM_CALC;
469 //      warnx("crc value is %i", (uint8_t)b);
470         break;
471
472
473     case CHKSUM_CALC:
474         /** checksum is calculated for everything payload only **/
475         crc_calc = gen_crc16((uint8_t*)buf_rcv , length_payload);
476
477 //      warnx("crc sent is %d", crc_sent);
478 //      warnx("crc calc is %d", crc_calc);
479
480         if(crc_sent == crc_calc) {              //checksum of payload
481             warnx("Data␣received␣is␣ok");
482             data_rcv_good = 1;
483         } else {
484             warnx("Data␣received␣is␣bad");
485         }
486         state = SYNC1;
```

```
487             break;
488
489
490         default:
491             state = SYNC1;
492             break;
493         }
494 }
```

crc.h

src/my_app/crc.h

```
1  #ifndef CRC_H
2  #define CRC_H
3
4
5  #include <stdio.h>   /* Standard input/output definitions */
6
7
8  uint16_t gen_crc16(const uint8_t *data, uint16_t size);
9
10 #endif /* CRC_H */
```

crc.c

src/my_app/crc.c

```
1
2  #include "crc.h"
3
4  /*
5   * From:
        http://stackoverflow.com/questions/10564491/function-to-calculate-a-crc16-checksum
6   * Access:  16 Nov. 2016
7   *
8   * CRC-16 , 0x8005 , x^16 + x^15 + x^2 + 1
9   *
10  *
11  * Korrekt, überprüft mit payload "123456789" unter verwendung von
        "http://www.lammertbies.nl/comm/info/crc-calculation.html"
12  * Es entspricht dem einfachen CRC-16, ohne CCITT oder sonstigem.
13  */
14 #define CRC16 0x8005
15
16 uint16_t gen_crc16(const uint8_t *data, uint16_t size)
17 {
18     uint16_t out = 0;
19     int bits_read = 0, bit_flag;
20
21     /* Sanity check: */
22     if(data == NULL)
23         return 0;
```

```
24
25      while(size > 0)
26      {
27          bit_flag = out >> 15;
28
29          /* Get next bit: */
30          out <<= 1;
31          out |= (*data >> bits_read) & 1; // item a) work from the least
                significant bits
32
33          /* Increment bit counter: */
34          bits_read++;
35          if(bits_read > 7)
36          {
37              bits_read = 0;
38              data++;
39              size--;
40          }
41
42          /* Cycle check: */
43          if(bit_flag)
44              out ^= CRC16;
45
46      }
47
48      // item b) "push out" the last 16 bits
49      int i;
50      for (i = 0; i < 16; ++i) {
51          bit_flag = out >> 15;
52          out <<= 1;
53          if(bit_flag)
54              out ^= CRC16;
55      }
56
57      // item c) reverse the bits
58      uint16_t crc = 0;
59      i = 0x8000;
60      int j = 0x0001;
61      for (; i != 0; i >>=1, j <<= 1) {
62          if (i & out) crc |= j;
63      }
64
65      return crc;
66  }
```

CMakeLists.txt

src/my_app/CMakeLists.txt

```
1  px4_add_module(
2      MODULE modules__my_app
3      MAIN my_app
4      PRIORITY "SCHED_PRIORITY_MAX-30"
```

```
    STACK 1200
    COMPILE_FLAGS
        #${MODULE_CFLAGS}
        #-Os
    SRCS
        my_app.c
        crc.c
    DEPENDS
        #platforms__common
    )
# vim: set noet ft=cmake fenc=utf-8 ff=unix :
```

C Code Simulink

script.m

src/simulink/script.m

```
%% hil v1
close all; clear all; clc;




global pck_length;
global pck_ctr;
pck_ctr = 0;

global time_old;
global g_x_old;
global g_y_old;
global g_z_old;

sim('sys_main');

display('simulation␣done')
pck_ctr

%Time of simulation
%serial.time(length(serial.time))
```

parser_small.m

src/simulink/parser_small.m

```
%function output_args = parser_small(input_args)

function output_args = parser_small(input_args)


%MAT_PARSER_1 Summary of this function goes here
%   Detailed explanation goes here

```

```
global pck_length;
global pck_ctr;

global time_old;
global g_x_old;
global g_y_old;
global g_z_old;

%Debug purpose
pck_ctr = pck_ctr+1;

THRESHOLD = 50;

%Parse package
pck_length = input_args(1,:);
pck_length = pck_length + 256*input_args(2,:);

clc;

time = double(typecast(uint8(input_args(3:10)), 'uint64'))/1e6;
g_x = double(typecast(uint8(input_args(207:210)), 'single'));
g_y = double(typecast(uint8(input_args(211:214)), 'single'));
g_z = double(typecast(uint8(input_args(215:218)), 'single'));


% crc detector




% crc workaround for legal data
if isnan(time)   time = time_old;       end

if isnan(g_x)          g_x = g_x_old;    end
if (g_x<-THRESHOLD)    g_x = g_x_old;    end
if (g_x>THRESHOLD)     g_x = g_x_old;    end

if isnan(g_y)          g_y = g_y_old;    end
if (g_y<-THRESHOLD)    g_y = g_y_old;    end
if (g_y>THRESHOLD)     g_y = g_y_old;    end

if isnan(g_z)          g_z = g_z_old;    end
if (g_z<-THRESHOLD)    g_z = g_z_old;    end
if (g_z>THRESHOLD)     g_z = g_z_old;    end

g_x_old = g_x;
g_y_old = g_y;
g_z_old = g_z;


%output data
output_args = [time ; g_x ; g_y ; g_z ];
```

D Aufgabenstellung

Die Aufgabenstellung enthält 5 Punkt, a bis e.

Punkt	Aufgaben
a	· Einarbeitung in die Pixhawk Firmware · Die Programmiertechniken, Designpatterns sollen vertsanden werden · Eigene Test-App mit Datenstomverarbeitung erstellen · Eigene Test-App mit Datenstomverarbeitung demonstrieren
b	· In Simulink soll eine Übersicht an Möglichkeiten erstellt werden um mit dem Pixhawk-Modul zu kommunizieren, dass die Hardware-in-the-loop Simulation verwirklicht werden kann
c	· Pixhawk Firmware erweitern, dass die Anbindung an Simulink durch starten einer einzelnen App möglich wird
d	· Eine einfache Hardware-in-the-loop Simulation soll auf Seite Simulink programmiert werden · Die Simulation muss demonstriert werden
e	Optional: · Programmieren verschiedener Tests · Einzelne Testcases in Testscenarien zusammenfassen · Automatisierte Ablauf von Testscenarien soll erfolgen

Tabelle 5: Aufgaben

Die Punkte a bis d müssen vor dem Abschlusstermin erfüllt sein.

E Projektplan

Meilenstein a: 26.10.2015 bis 20.11.2015
Meilenstein b: 26.10.2015 bis 20.11.2015
Die Meilensteine werden in Form einer kurzen Präsentation abgehalten.

Abgabe Schlussbericht: 18.12.2015, 16:00 im Raum D311
Abschlusspräsentation: Zwischen 14.12.2015 bis 22.01.2016

Untitled Gantt Project

Dec 13, 2015

2

Tasks

Name	Begin date	End date
(a) Einfache Hardware-in-the-loop Simulation erstellen, Testapp mit Datenstromverarbeitung	9/14/15	10/25/15
Einarbeitung in Aviatik	9/14/15	9/19/15
Firmware, Designpattern verstehen, Hello Sky App	9/14/15	9/28/15
Testapp mit Datenstromverarbeitung	9/27/15	10/7/15
Über UART? *Von Simulink direkt auf Pixhawk flashen?*		
Meilenstein (a)	10/26/15	10/26/15
Einarbeitung in die Pixhawk Firmware. *Die Programmiertechniken (Designpatterns) sollen verstanden werden und eine eigene Test-App mit Datenstromverarbeitung soll demonstriert werden.*		
(b) Aus Simulink bidirektionale Kummnikationslösungen evaluieren mit PixHawk	10/10/15	11/19/15
Kommnikationsmöglichkeiten erörtern	10/10/15	10/21/15
Geschwindigkeit (Sensoren habe 2kHz!) *Verlässlichkeit* *Einfachheit*		
Kommnikationsmöglichkeiten implementieren	10/16/15	11/8/15
Kommnikationsmöglichkeiten testen	10/16/15	11/8/15
Meilenstein (b)	11/20/15	11/20/15
Auf der Seite Simulink soll eine Überischt an Möglichkeiten erstellt werden um mit dem Pixhawk-Modul zu kommunizieren und die geforderten Aufgaben (Hardware-in-the-loop Simulation) verwirklichen zu können.		
(c) PixHawk Firmware erweitern, dass Anbindung an Simulink durch eine App möglich ist	11/9/15	11/22/15
Die Pixhawk Firmware sill so erweitert werden, dass die Anbindung an Simulink durch starten einer einzelnen App möglich wird		
App für PixHawk programmieren	11/9/15	11/16/15
App testen	11/17/15	11/22/15
(d) Einfache Hardware-in-the-loop Simulation programmieren und demonstrieren	11/23/15	12/4/15
Eine einfache Hardware-in-the-loop Simulation soll auf Seiten Simulink programmiert und demonstriert werden.		
Einfache Hardware-in-the-loop Simulation erörtern	11/23/15	11/25/15
Simulation in Simulink programmieren	11/25/15	12/4/15
(e) Unittests, Integrationstests, Systemtests automatisiert testen	12/7/15	12/18/15
Programmieren verschiedener Tests oder ganzer Testszenarien welche automatisiert ablaufen		

Untitled Gantt Project

Tasks

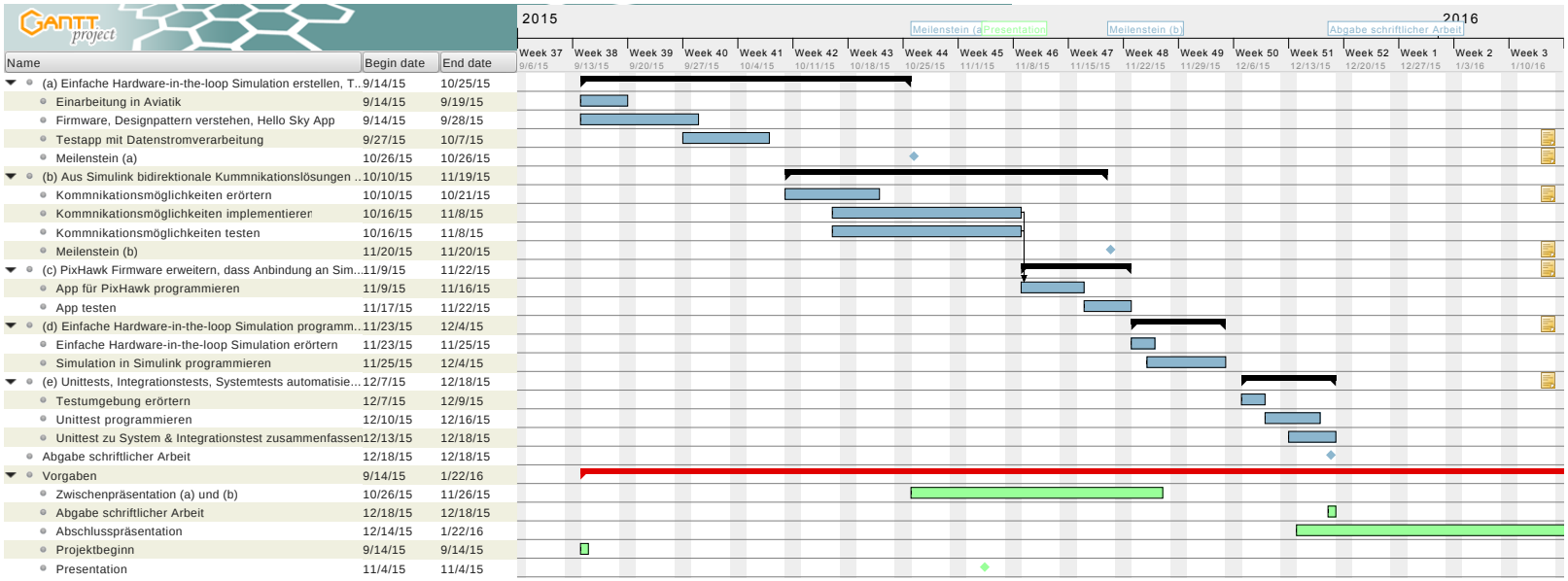
Name	Begin date	End date
Testumgebung erörtern	12/7/15	12/9/15
Unittest programmieren	12/10/15	12/16/15
Unittest zu System & Integrationstest zusammenfassen	12/13/15	12/18/15
Abgabe schriftlicher Arbeit	12/18/15	12/18/15
Vorgaben	9/14/15	1/22/16
Zwischenpräsentation (a) und (b)	10/26/15	11/26/15
Abgabe schriftlicher Arbeit	12/18/15	12/18/15
Abschlusspräsentation	12/14/15	1/22/16
Projektbeginn	9/14/15	9/14/15
Presentation	11/4/15	11/4/15

Untitled Gantt Project

Dec 13, 2015

5

Gantt Chart



Name	Begin date	End date
(a) Einfache Hardware-in-the-loop Simulation erstellen, T...	9/14/15	10/25/15
Einarbeitung in Aviatik	9/14/15	9/19/15
Firmware, Designpattern verstehen, Hello Sky App	9/14/15	9/28/15
Testapp mit Datenstromverarbeitung	9/27/15	10/7/15
Meilenstein (a)	10/26/15	10/26/15
(b) Aus Simulink bidirektionale Kummnikationslösungen ...	10/10/15	11/19/15
Kommnikationsmöglichkeiten erörtern	10/10/15	10/21/15
Kommnikationsmöglichkeiten implementieren	10/16/15	11/8/15
Kommnikationsmöglichkeiten testen	10/16/15	11/8/15
Meilenstein (b)	11/20/15	11/20/15
(c) PixHawk Firmware erweitern, dass Anbindung an Sim...	11/9/15	11/22/15
App für PixHawk programmieren	11/9/15	11/16/15
App testen	11/17/15	11/22/15
(d) Einfache Hardware-in-the-loop Simulation programm...	11/23/15	12/4/15
Einfache Hardware-in-the-loop Simulation erörtern	11/23/15	11/25/15
Simulation in Simulink programmieren	11/25/15	12/4/15
(e) Unittests, Integrationstests, Systemtests automatisie...	12/7/15	12/18/15
Testumgebung erörtern	12/7/15	12/9/15
Unittest programmieren	12/10/15	12/16/15
Unittest zu System & Integrationstest zusammenfassen	12/13/15	12/18/15
Abgabe schriftlicher Arbeit	12/18/15	12/18/15
Vorgaben	9/14/15	1/22/16
Zwischenpräsentation (a) und (b)	10/26/15	11/26/15
Abgabe schriftlicher Arbeit	12/18/15	12/18/15
Abschlusspräsentation	12/14/15	1/22/16
Projektbeginn	9/14/15	9/14/15
Presentation	11/4/15	11/4/15