**DIPLOMARBEIT**

Gesamtprojekt

**FPV-Drohne**

**Elektronik FPV-Drohne**

Marcel Bieder 5AHEL

**Softwareentwicklung FPV-Drohne**

Maximilian Lendl 5AHEL

**CAD-Entwicklung & Datenübertragung**

Ben Heinicke 5AHEL

**Entwicklung einer APP für Smartphone & Videoübertragung**

Sebastian Hinterberger 5AHEL

Betreuer: Dipl.-Ing. Josef Reisinger

Schuljahr 2023/24

Abgabevermerk:

Datum: 02.04.2024 übernommen von:

****

**Höhere Technische Bundeslehranstalt Hollabrunn**

**Höhere Lehranstalt für Elektronik und Technische Informatik**

**EIDESSTATTLICHE ERKLÄRUNG**

**Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.**

**Marcel Bieder**

**Maximilian Lendl**

**Ben Heinicke**

**Sebastian Hinterberger**

Hollabrunn, am 02.04.2024

**HINWEISE**

Die vorliegende Diplomarbeit wurde in Zusammenarbeit mit der Firma **Dronetech Austria** ausgeführt.

oder

Die vorliegende Diplomarbeit wurde für die Abteilung Elektronik und Technische Informatik der HTL Hollabrunn ausgeführt.

Die in dieser Diplomarbeit entwickelten Prototypen und Software-Produkte dürfen ganz oder auch in Teilen von Privatpersonen oder Firmen nur dann in Verkehr gebracht werden, wenn sie diese selbst geprüft und für den vorgesehenen Verwendungszweck für geeignet befunden haben.

Es wird keinerlei Haftung übernommen für irgendwelche Schäden, die aus der Nutzung der hier entwickelten oder beschriebenen Bestandteile des Projekts resultieren.

Für alle Entwicklungen gilt die GNU General Public License [http://www.gnu.org/licenses/gpl.html] der Free Software Foundation, Boston, USA in der Version 3.

Die Diplomarbeit erfüllt die “Standards für Ingenieur- und Technikerprojekte” entsprechend dem Rundschreiben Nr. 60 aus 1999 des BMBWK (GZ.17.600/101-II/2b/99).

[https://www.bmb.gv.at/ministerium/rs/1999\_60.html]

SCHLÜSSELBEGRIFFE

DANKSAGUNGEN

Reisinger, Wihsböck, Kauer, Stoll, Dronetech Austria (Daniel Stoiber),

**DIPLOMARBEIT**

**DOKUMENTATION**

|  |  |
| --- | --- |
| Namen der  Verfasser/innen |  |
| Jahrgang  Schuljahr |  |
| Thema der Diplomarbeit |  |
| Kooperationspartner |  |

|  |  |
| --- | --- |
| Aufgabenstellung |  |

|  |  |
| --- | --- |
| Realisierung |  |

|  |  |
| --- | --- |
| Ergebnisse |  |

|  |  |
| --- | --- |
| Typische Grafik, Foto etc.  (mit Erläuterung) |  |

|  |  |
| --- | --- |
| Teilnahme an Wettbewerben,  Auszeichnungen |  |

|  |  |
| --- | --- |
| Möglichkeiten der Einsichtnahme in die Arbeit | HTL Hollabrunn  Anton Ehrenfriedstraße 10  2020 Hollabrunn |

|  |  |  |
| --- | --- | --- |
| Approbation  (Datum / Unterschrift) | Prüfer/Prüferin | Direktor/Direktorin  Abteilungsvorstand/Abteilungsvorständin |

**DIPLOMA THESIS**

**Documentation**

|  |  |
| --- | --- |
| Author(s) |  |
| Form  Academic year |  |
| Topic |  |
| Co-operation partners |  |

|  |  |
| --- | --- |
| Assignment of tasks |  |

|  |  |
| --- | --- |
| Realisation |  |

|  |  |
| --- | --- |
| Results |  |

|  |  |
| --- | --- |
| Illustrative graph, photo  (incl. explanation) |  |

|  |  |
| --- | --- |
| Participation in competitions  Awards |  |

|  |  |
| --- | --- |
| Accessibility of  final project thesis | HTL Hollabrunn  Anton Ehrenfriedstraße 10  2020 Hollabrunn |

|  |  |  |
| --- | --- | --- |
| Approval  (Date / Signature) | Examiner/s | Head of Department / College |

DA Antrag und unterschriebene Erklärung aus der Diplomarbeitsdatenbank einfügen

Inhaltsverzeichnis

[1 Einleitung 12](#_Toc162121306)

[1.1 Projektziel 12](#_Toc162121307)

[1.2 Gesamtüberblick 12](#_Toc162121308)

[2 Grundlagen Drohnenflug 13](#_Toc162121309)

[2.1 Lagewinkel 14](#_Toc162121310)

[2.1.1 Throttle 14](#_Toc162121311)

[2.1.2 Pitch 14](#_Toc162121312)

[2.1.3 Roll 14](#_Toc162121313)

[2.1.4 Yaw 14](#_Toc162121314)

[2.1.5 Komplementärfilter 15](#_Toc162121315)

[2.2 PID-Regler 17](#_Toc162121316)

[2.2.1 Proportionalglied (P-Glied) 18](#_Toc162121317)

[2.2.2 Integralglied (I-Glied) 18](#_Toc162121318)

[2.2.3 Differenzialglied (D-Glied) 19](#_Toc162121319)

[2.2.4 PID-Regler 19](#_Toc162121320)

[3 Mechanischer Aufbau 20](#_Toc162121321)

[4 Elektronik FPV-Drohne (BIE) 20](#_Toc162121322)

[5 Steuerungssoftware 21](#_Toc162121323)

[5.1 Hauptprogramm 21](#_Toc162121324)

[5.1.1 Umgang mit Initialisierungsfehler 25](#_Toc162121325)

[5.1.1.1 Übersicht Fehlercodes 26](#_Toc162121326)

[5.2 Bestimmen der Akkuspannung - DS2438 28](#_Toc162121327)

[5.2.1 One-Wire Protokoll 28](#_Toc162121328)

[5.2.1.1 One-Wire Schreibzyklen 29](#_Toc162121329)

[5.2.1.2 One-Wire Lesezyklen 32](#_Toc162121330)

[5.2.1.3 Initialisierungssequenz 34](#_Toc162121331)

[5.2.1.4 ROM-Funktionsbefehl 35](#_Toc162121332)

[5.2.1.5 Memory-Funktionsbefehl 35](#_Toc162121333)

[5.2.2 Registerübersicht DS2438 36](#_Toc162121334)

[5.2.3 Initialisierung DS2438 37](#_Toc162121335)

[5.2.4 Spannungsüberwachung 38](#_Toc162121336)

[5.3 Real Time System Interrupt (MAIN\_ISR) 40](#_Toc162121337)

[5.4 Einlesen der Daten von Fernsteuerung 42](#_Toc162121338)

[5.4.1 Konfiguration Fernsteuerung 44](#_Toc162121339)

[5.4.2 Unterstützte Protokolle 46](#_Toc162121340)

[5.4.2.1 PPM (Pulse Position Modulation) 46](#_Toc162121341)

[5.4.2.2 S.Bus 46](#_Toc162121342)

[5.4.2.3 I.Bus 48](#_Toc162121343)

[5.4.3 Initialisierung Empfangssoftware 50](#_Toc162121344)

[5.4.4 Empfangssoftware 54](#_Toc162121345)

[5.5 Inertial Measurement Unit (IMU) 62](#_Toc162121346)

[5.5.1 I²C Protokoll 62](#_Toc162121347)

[5.5.1.1 Schreibzyklus IMU 64](#_Toc162121348)

[5.5.1.2 Lesezyklus IMU 65](#_Toc162121349)

[5.5.2 IMU-Verbindungstest 67](#_Toc162121350)

[5.5.3 Bestimmen der Lagewinkel - MPU9250 69](#_Toc162121351)

[5.5.3.1 Registerübersicht MPU9250 69](#_Toc162121352)

[5.5.3.2 Initialisierung Accelerometer und Gyroskop 70](#_Toc162121353)

[5.5.3.3 Einlesen der Accelerometer- und Gyroskop-Daten 71](#_Toc162121354)

[5.5.3.4 Berechnen der Lagewinkel 73](#_Toc162121355)

[5.5.4 Bestimmen der Höhe - BMP280 75](#_Toc162121356)

[5.5.4.1 Registerübersicht BMP280 75](#_Toc162121357)

[5.5.4.2 Initialisierung Barometer 76](#_Toc162121358)

[5.5.4.3 Einlesen der Barometer-Daten 78](#_Toc162121359)

[5.5.4.4 Berechnung der Höhe 80](#_Toc162121360)

[5.6 Motorregelalgorithmus 81](#_Toc162121361)

[5.7 Motoransteuerung 83](#_Toc162121362)

[5.7.1 DShot Protokoll 83](#_Toc162121363)

[5.7.2 Initialisierung Motoransteuerung 84](#_Toc162121364)

[5.7.3 Motoransteuerung Software 85](#_Toc162121365)

[5.8 PID-Regler 89](#_Toc162121366)

[5.8.1 Initialisierung PID-Regler 89](#_Toc162121367)

[5.8.2 PID-Algorithmus 91](#_Toc162121368)

[5.9 Terminal Übertragung und Status LEDs 94](#_Toc162121369)

[5.10 Terminal Kommunikation 96](#_Toc162121370)

[5.11 Status – LEDs 98](#_Toc162121371)

[6 Datenübertragung der Mess- und Videodaten (Ben) 100](#_Toc162121372)

[7 Visualisierungs-App 101](#_Toc162121373)

[8 Videostreaming 101](#_Toc162121374)

[9 Videoserver-Port Forwarding 101](#_Toc162121375)

[10 Ergebnisse 101](#_Toc162121376)

[11 Anhang 102](#_Toc162121377)

[11.1 Einführung CAD – Software (Fusion 360) (Ben) 102](#_Toc162121378)

[*11.1.1* UI und Projekterstellung 102](#_Toc162121379)

[*11.1.2* Skizze anfertigen 102](#_Toc162121380)

[*11.1.3* Körper erstellen 102](#_Toc162121381)

[*11.1.4* Schrift und Bilder einfügen 102](#_Toc162121382)

[11.2 3D – Druck (Ben) 102](#_Toc162121383)

[*11.2.1* 3D – Drucker 102](#_Toc162121384)

[*11.2.2* Filamente 102](#_Toc162121385)

[11.2.3 3D – Drucker Software (Ultimaker Cura) 102](#_Toc162121386)

[11.3 Inbetriebnahme Anleitung 102](#_Toc162121387)

[11.4 Projektplan 102](#_Toc162121388)

[11.5 Projektkosten 102](#_Toc162121389)

[11.6 Projekttagebuch 102](#_Toc162121390)

[11.7 Einführung Entwicklungsumgebung - Steuerungssoftware 103](#_Toc162121391)

[12 Quellen 110](#_Toc162121392)

[12.1 Gedruckte Medien 110](#_Toc162121393)

[12.2 Online 110](#_Toc162121394)

[13 Verzeichnis der Abbildungen 112](#_Toc162121395)

[14 Begleitprotokoll 114](#_Toc162121396)

[15 Anhang 115](#_Toc162121397)

[15.1 — Projektdokumentation (Kostendarstellung, Besprechungsprotokolle etc.) 115](#_Toc162121398)

[15.2 — Technische Dokumentation (technische Beschreibungen, Berechnungen, 115](#_Toc162121399)

[15.3 Konstruktionszeichnungen, Versuchsberichte, betriebswirtschaftliche Kalkulationen etc.) 115](#_Toc162121400)

[15.4 Schaltungen, Zeichnungssätze, sonstiges 115](#_Toc162121401)

# Einleitung

## Projektziel

## Gesamtüberblick

# Grundlagen Drohnenflug

## Lagewinkel

### Throttle

### Pitch

### Roll

### Yaw

### Komplementärfilter

Der Komplementärfilter hat die Aufgabe, die Messwerte von einen Beschleunigungssensor und Gyroskop in Lagewinkel umzuwandeln.

**Accelerometer Winkel**:

Durch die Kombination der terrestrischen Gravitation und Trigonometrie ermöglicht es die Accelerometer-Werte in Neigungswinkel umzuwandeln:

accelerometerPitch/Roll … jeweiliger berechneter Winkel in Radiant (rad)

accelerometerX/Y/Z … Accelerometer Messwert der jeweiligen Achse in g (Erdbeschleunigung, ungefähr 9,81m/s²)

Der Yaw-Winkel kann nicht berechnet werden, da die Yaw-Bewegung in der Ebene, um die Z-Achse, stattfindet und daher unabhängig von dem Werten der X- und Y-Achse ist.

Im Gegensatz zum Gyroskop besitzt ein Accelerometer keinen Drift in der Messung und kann daher für längerfristige Messungen verwendet werden.

Für den Filter wird der berechnete Winkel in Grad statt in Radiant benötigt. Daher müssen die Ergebnisse mit einem Faktor von multipliziert werden.

Problem:

Wenn der Accelerometer bewegt wird, kann dieser nicht zwischen den Bewegungsbeschleunigungen und der Erdbeschleunigung unterscheiden und liefert daher verfälschte Werte.

**Gyroskop Winkel**:

Durch die Integration der Winkelbeschleunigungswerte ist es möglich, Lagewinkel zu bestimmen:

gyroscopePitch/Roll/Yaw … jeweiliger berechneter Winkel in Radianten (rad)

gyroscopeX/Y/Z … Gyroskop Messwert der jeweiligen Achse in Grad pro Sekunde (°/s)

dt … Zeitbereich in Sekunden (s)

Problem:

Durch die Integration werden Messungenauigkeit aufsummiert, die zu einen Wertedrift führen. Daher kann das Gyroskop nur für kurzfristige Messungen verwendet werden.

**Sensorfusion**:

Der Komplementärfilter kann als Kombination von zwei Filter gesehen werden: Ein Hochpassfilter für das Gyroskop und einen Tiefpassfilter für den Accelerometer.

Das Accelerometer liefert eine gute Indikation der Orientierung bei konstanten Bedingungen, und das Gyroskop liefert eine gute Indikation bei schnellen Neigungsänderungen.

Winkel … Pitch/Roll-Winkel in Grad (°)

α … Filterkoeffizient

gyroscopeData … Gyroskop Messwerte der jeweiligen Achse in Grad pro Sekunde (°/s)

dt … Abtastzeit in Sekunden (s)

accelerometerWinkel … berechneter Roll/Pitch nur mit Accelerometerdaten in Grad (°)

Der Wert für α ist typischerweise 0,98. Das bedeutet, dass die Gyroskop-Messung zu 98% und die Accelerometer-Messung zu 2% gewichtet wird. Daraus folgt:

Dadurch, dass mit dem Accelerometer kein Yaw-Winkel bestimmt werden kann, wird der Winkel nur mit dem Gyroskop berechnet. Wegen dem Gyroskop-Drift wird dieser Winkel über die Zeit immer ungenauer und muss für einen genauen Wert nach einer gewissen Zeit zurückgesetzt werden.

## PID-Regler

Im Allgemeinen wird zwischen einer Steuerung und einer Regelung unterschieden. Bei einer Steuerung wird der Sollwert direkt in einen Steuerungsalgorithmus geleitet, der immer nach demselben Schema abläuft. Bei einer Regelung wird der Istwert des Systems rückgekoppelt und mit dem Sollwert verglichen. Dieser Vergleich ergibt eine Regelabweichung, die das in den Regelalgorithmus geleitet wird. Das Ziel einer Regelung ist, die Regelabweichung zum Verschwinden zu bringen – Soll- und Istwert auf denselben Wert regeln.

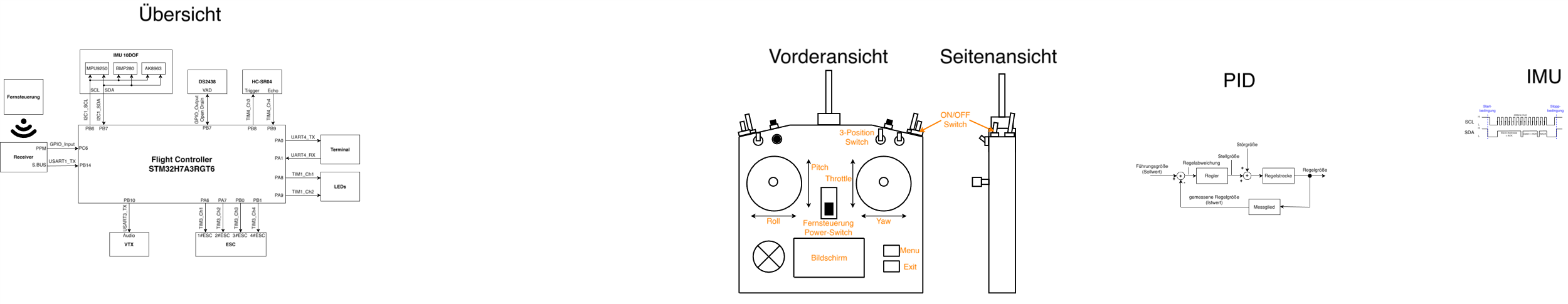


Abbildung 1: Regler-Blockschaltbild

Führungsgröße … eingestellter Sollwert, der vom Regler erreicht werden soll

Regelabweichung … Differenz zwischen Führungsgröße und gemessener Regelgröße

Regler … erstellt Stellgröße proportional von der Regelabweichung

Stellgröße … Reglerausgang

Störgröße … externe Einflüsse, die die Stellgröße beeinflussen

Regelstrecke … wandelt Stellgröße in die Regelgröße um

Regelgröße … gemessener Istwert, meistens durch Sensoren bestimmt

Es gibt verschiedene Arten für Regler, die für unterschiedliche Anwendungen Vor- und Nachteile liefern. Für den Quadrokopter wurde ein PID-Regler verwendet. Dieser hält während des Fluges die Drohne stabil. Bei Störgrößen, wie zum Beispiel Windstößen, soll die Drohne sich automatisch wieder in die richtige Lage ausrichten, um weiter einen stabilen Flug zu gewährleisten.

Ein PID-Regler besteht aus drei verschiedenen Gliedern (Proportional-, Integral-, Differenzialglied), die unterschiedliche Aufgaben erfüllen und Funktionsweisen vorweisen:

### Proportionalglied (P-Glied)

Das Proportionalglied setzt das Ausgangssignal proportional zum Eingangssignal.

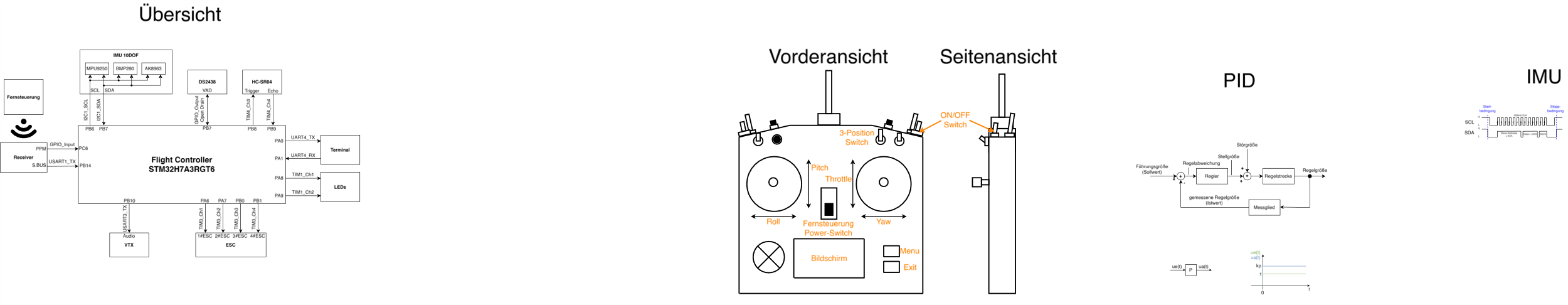


Abbildung 2: P-Glied Sprungantwort

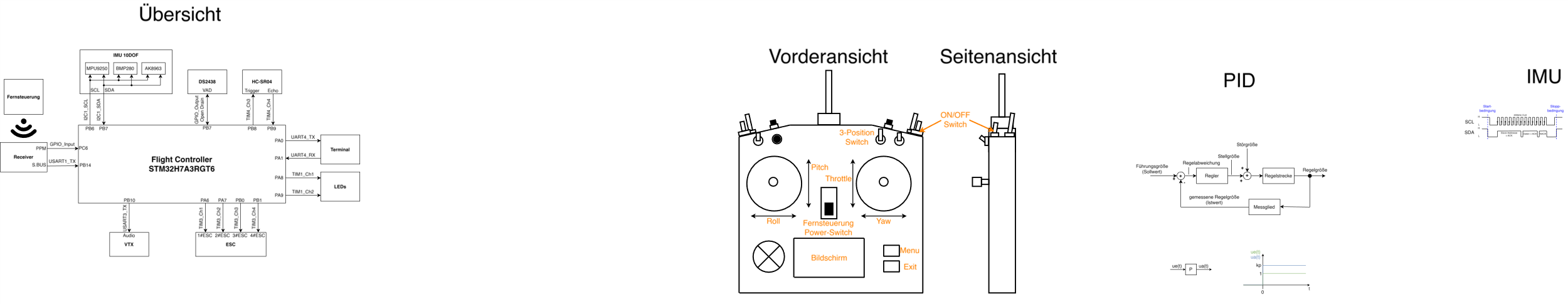


Abbildung 3: P-Glied Schaltsymbol

**Vor- (+) und Nachteile (-):**

+ kann Regelabweichung zum Verschwinden bringen

+ schnelles Anregeln

+ keine Phasenverschiebung

- kann Regelabweichung = 0 nicht halten

### Integralglied (I-Glied)

Das Integralglied setzt das Ausgangssignal zum Integral vom Eingangssignal.

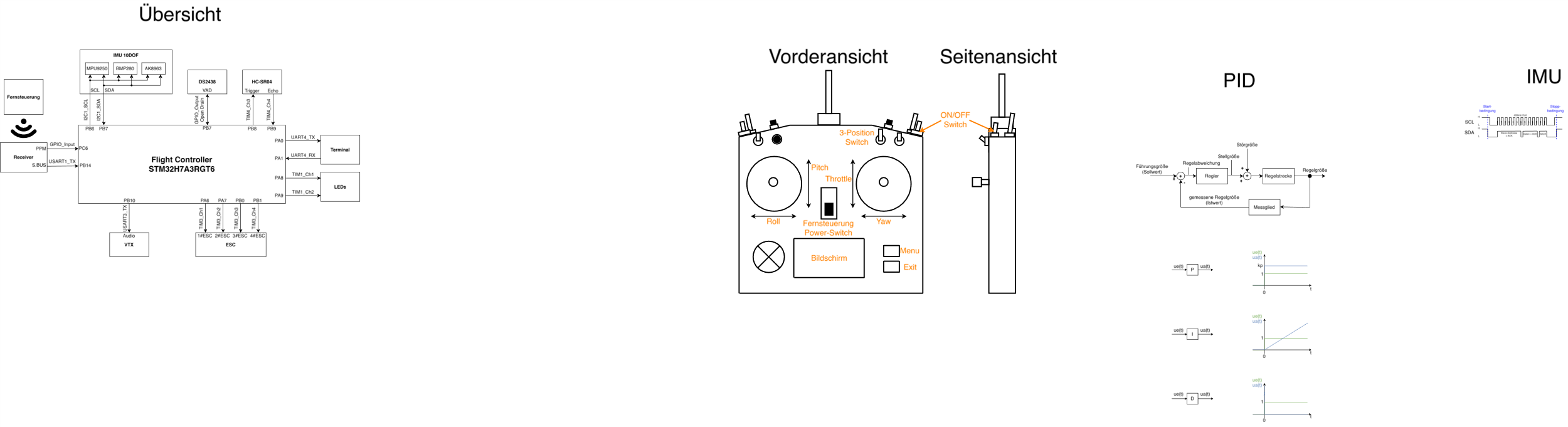


Abbildung 4: I-Glied Schaltsymbol

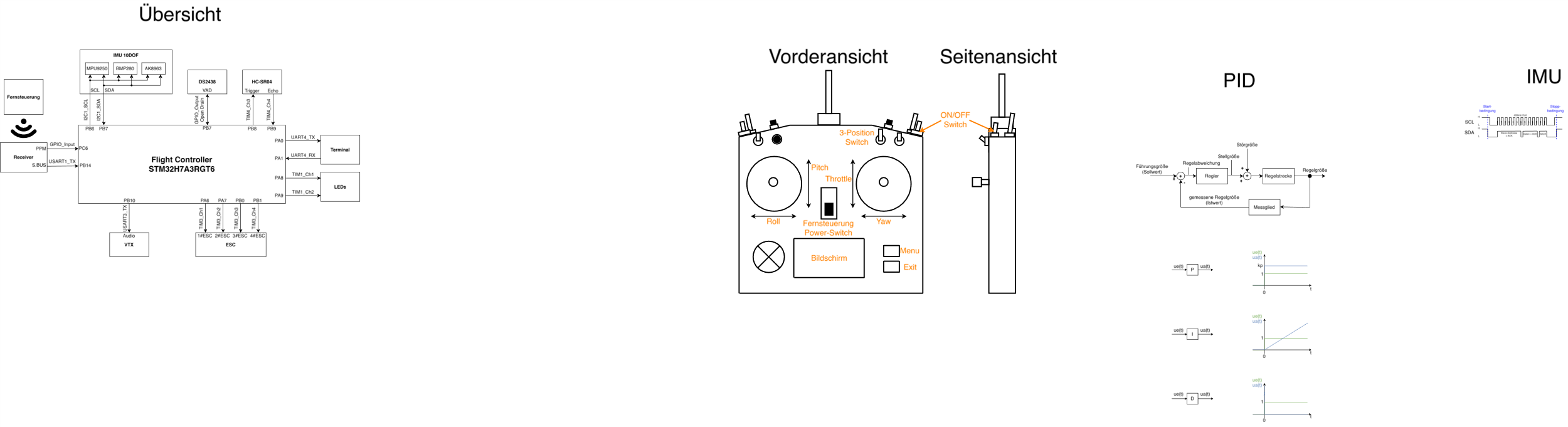


Abbildung 5: I-Glied Sprungantwort

**Vor- (+) und Nachteile (-):**

+ kann Regelabweichung zum Verschwinden bringen

- langsames Anregeln

- verringert die Stabilität (φ = -90°)

- neigt zum Überschwingen

### Differenzialglied (D-Glied)

Das Differenzialglied setzt das Ausgangssignal zur momentanen Änderung des Eingangssignals.

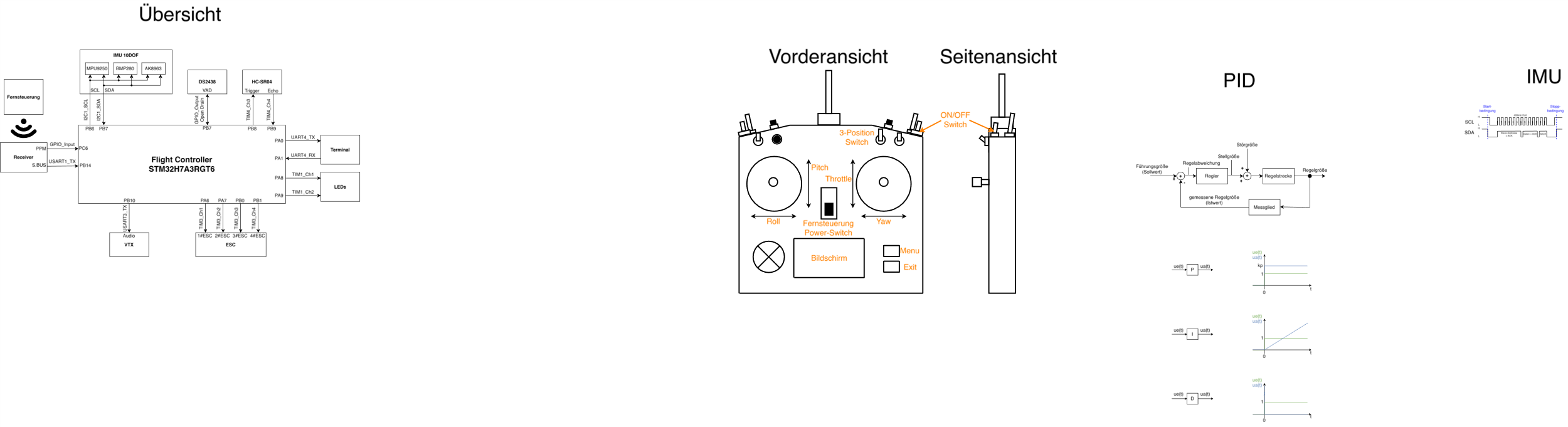


Abbildung 6: D-Glied Schaltsymbol

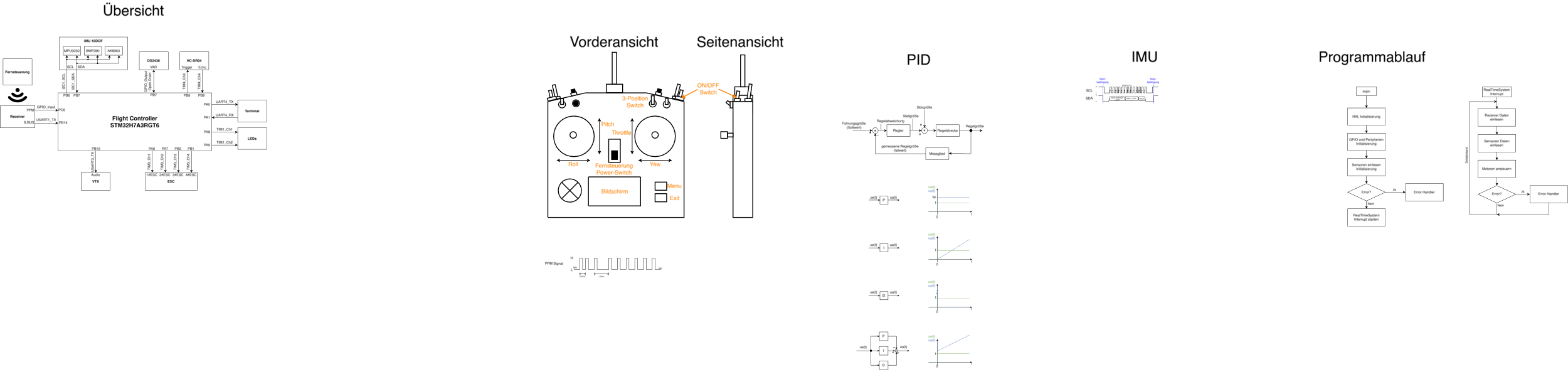


Abbildung 7: D-Glied Sprungantwort

**Vor- (+) und Nachteile (-):**

- kann allein nicht Regeln

In Kombination mit anderen Gliedern:

+ schnelles Anregeln

+ verbessert die Stabilität (φ = +90°)

### PID-Regler

Der PID-Regler kombiniert alle Ausgangssignale der einzelnen Glieder.

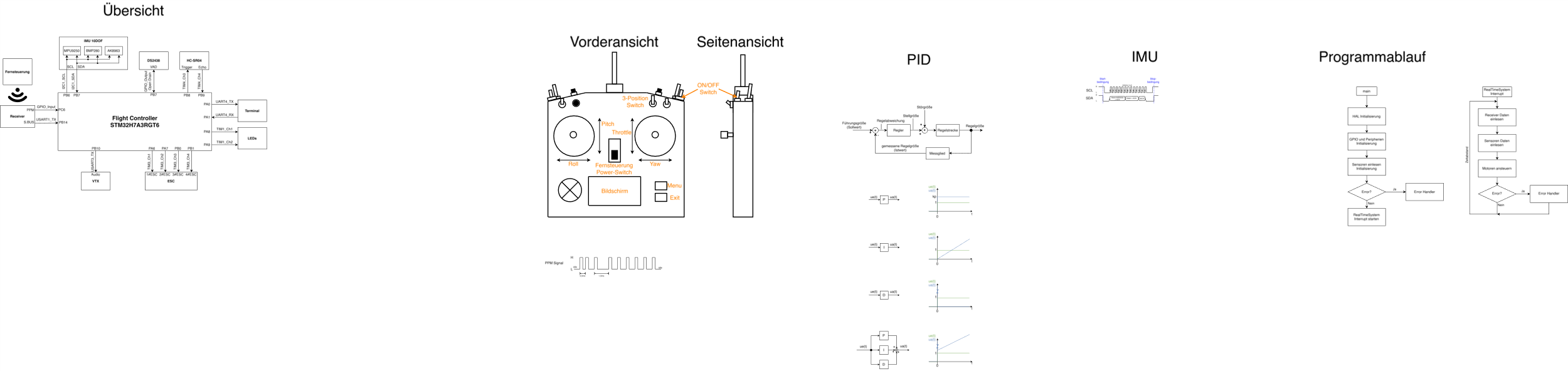


Abbildung 8: PID-Regler Schaltsymbol

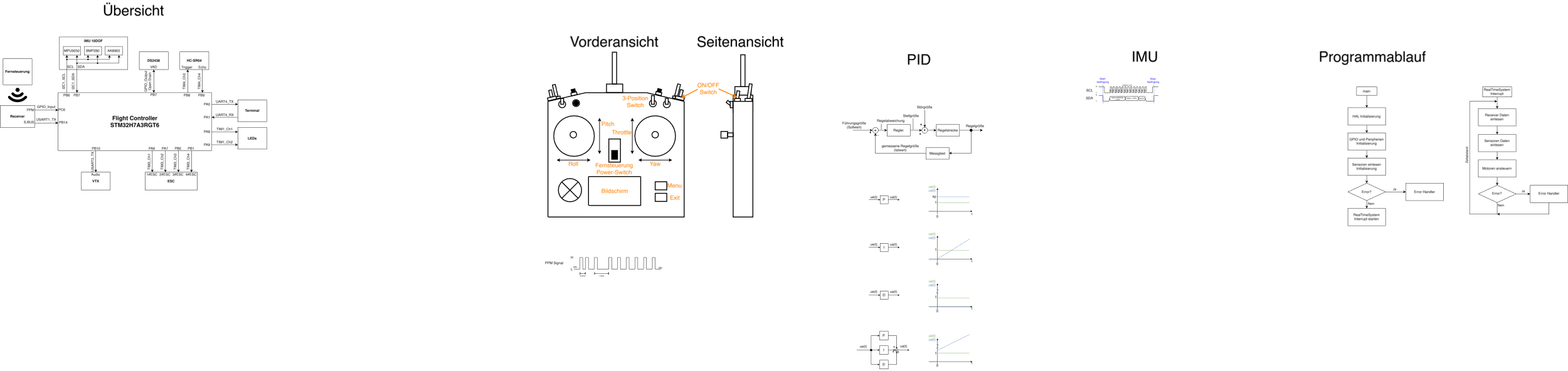


Abbildung 9: PID-Regler Sprungantwort

**Vor- (+) und Nachteile (-):**

+ schnelles Anregeln (durch P- und D-Glied)

+ kann Regelabweichung zum Verschwinden bringen (P und I) und halten (I)

+ verbessert die Stabilität (D)

- kann Überschwingen (I)

# Mechanischer Aufbau

# Elektronik FPV-Drohne (BIE)

# Steuerungssoftware

Die Software des Flight Controllers hat die Aufgabe, die Daten der Fernsteuerung über einen Receiver und die Messwerte der Sensoren einzulesen und diese umzuwandeln, damit diese an die ESC, VTX, Status-LEDs und Terminal senden zu können.

Die Architektur ist sehr zeitkritisch und benötigt genaue Interrupts, damit ein stabiler und sicherer Flug möglich ist. Daher wird das gesamte System von einem Real Time System Interrupt gesteuert, welcher in einem Zeitbereich von ein paar Millisekunden auslöst und mithilfe der Sensorik die gesamte FPV-Drohne steuert.

## Hauptprogramm



Abbildung 10: Flussdiagramm Programmablauf

Am Beginn des Hauptprogramms (main) wird die gesamte HAL-Struktur initialisiert. Danach wird die Taktversorgung, GPIO-Pins und die einzelnen Peripherien mit den gewünschten Angaben aus STM32CubeMX eingestellt. Der nachfolgende Ausschnitt wird von STM32CubeMX automatisch generiert.

Dateiname: main.c | Ausschnitt aus main() Funktion

/\*\*

  \* @brief  The application entry point.

  \* @retval int

  \*/

int main(void)

{

  /\* Reset of all peripherals, Initializes the Flash interface and the Systick. \*/

  HAL\_Init();

  /\* Configure the system clock \*/

  SystemClock\_Config();

  /\* Initialize all configured peripherals \*/

  MX\_GPIO\_Init();

  MX\_DMA\_Init();

  MX\_TIM3\_Init();

  MX\_USART1\_UART\_Init();

  MX\_USART3\_UART\_Init();

  MX\_I2C1\_Init();

  MX\_UART4\_Init();

  MX\_TIM4\_Init();

  MX\_TIM15\_Init();

  MX\_TIM16\_Init();

  MX\_TIM17\_Init();

  MX\_TIM14\_Init();

  MX\_TIM2\_Init();

  MX\_TIM1\_Init();

...

}

Bemerkung:

Die Punkte ... bedeuten, dass sich nach dem Ausschnitt noch weiterer Programmteile befinden. Im weiteren Verlauf werden die Punkte immer für diesen Kontext verwendet.

Als nächsten Schritt werden alle Sensoren, Ein- und Ausgänge mit den angegebenen Einstellungen initialisiert.

Dateiname: main.c | Ausschnitt aus main() Funktion

int main(void)

{

  ...

  int8\_t errorCode;

  Terminal\_Print("Program start\n\r");

// initialize data transmission

DATA\_INIT(&huart3);

  // initialize DS2438 (battery monitoring)

  Terminal\_Print("DS2438 start ... ");

  errorCode = DS2438\_Init(&htim16, GPIOC, GPIO\_PIN\_0);

  if(errorCode != DS2438\_OK)

    Sensor\_ErrorHandler(DS2438, errorCode);

  Terminal\_Print("DS2438 OK\n\r");

  // initialize PID

  Terminal\_Print("PID start ... ");

  PID\_Init(&huart4);

  Terminal\_Print("PID OK\n\r");

  // initialize output via DSHOT protocol

  Terminal\_Print("DShot start ... ");

  errorCode = DShot\_Init(&htim3, DSHOT300, &htim14);

  if(errorCode != DSHOT\_OK)

    Sensor\_ErrorHandler(DSHOT, errorCode);

  Terminal\_Print("DShot OK\n\r");

  // initialize IMU 10DOF

  Terminal\_Print("IMU start ... ");

  IMU\_InitTypeDef imuInit;

  imuInit.hi2c = &hi2c1;

  imuInit.gyroFS = GYRO\_500DPS;

  imuInit.accelFS = ACCEL\_16G;

  imuInit.gyroDLPF = GYRO\_DLPF\_10HZ;

  imuInit.accelDLPF = ACCEL\_DLPF\_10HZ;

  imuInit.baroCoeff = IMU\_BARO\_FILTER\_COEFF\_16;

  imuInit.baroTempOS = BARO\_TEMP\_OS\_2X;

  imuInit.baroPressOS = BARO\_PRESS\_OS\_16X;

  imuInit.baroSBT = BARO\_STANDBY\_0P5MS;

  imuInit.htim = &htim17;

  errorCode = IMU\_Init(&imuInit);

  if(errorCode != IMU\_OK)

    Sensor\_ErrorHandler(IMU, errorCode);

  Terminal\_Print("IMU OK\n\r");

  // initialize receiver reception with DMA

  Terminal\_Print("Receiver start ... ");

  errorCode = Receiver\_Init(SBUS, &huart1);

  if(errorCode != RECEIVER\_OK)

    Sensor\_ErrorHandler(RECEIVER, errorCode);

  Terminal\_Print("Receiver OK\n\r");

  Terminal\_Print("Initilisation finished -> Start Real Time System ...\n\r");

  ...

}

Bei jeder Initialisierungsfunktion werden bestimmte Parameter übergeben. Diese bestimmen zum Beispiel, welche Peripherie für die einzelnen Funktionen verwendet werden soll. In der Funktion *Receiver\_Init()* wird zusätzlich die Real Time System Interrupt Service Routine (MAIN\_ISR) gestartet, die alle 8ms das komplette System aktualisiert und die Steuerung der Drohne ermöglicht.

Am Terminal wird vor jeder Initialisierung der Text „<Fehlerquelle> start …“ und nach der erfolgreichen Initialisierung der Text „<Fehlerquelle> OK“. Im Fall eines Initialisierungsfehler wird die Funktion *Sensor\_ErrorHandler()* aufgerufen *(siehe:* [*Kapitel 5.1.1*](#_Umgang_mit_Initialisierungsfehler_1)*)*.

Als letzten Schritt wird in einer Endlosschleife die Akkuspannung eingelesen *(siehe:* [*Kapitel 5.2.4*](#_Spannungsüberwachung)*)*. Diese Funktion befindet sich nicht in der MAIN\_ISR, weil der Einlesevorgang mittels dem One-Wire-Protokolls ungefähr 26ms dauert, was zu lange für die Updaterate von 8ms der ISR ist.

Dateiname: main.c | Ausschnitt aus main() Funktion

int main(void)

{

  ...

/\* Infinite loop \*/

  while(1)

  {

    errorCode = DS2438\_ReadVoltage();

    if(errorCode == DS2438\_VOLTAGE\_ERROR)

      Receiver\_FailsafeHandler();

  }

}

Wenn die Akkuspannung einen kritischen Wert unterschritten hat, wird die Funktion *Receiver\_FailsafeHandler()* aufgerufen. Diese bewirkt, dass alle Motoren ausgeschalten werden und die Drohne eine Warnung an die Groundstation schickt.

### Umgang mit Initialisierungsfehler

Dateiname: status\_handling.c

 /\*\*

  \* @brief This function completely stops the program

  \* @param sens what sensor has the error

  \* @param errorCode

  \* @retval None

  \*/

void Sensor\_ErrorHandler(Sensors sens, int8\_t errorCode)

{

  char txt[100];

  // choose error source

  switch(sens)

  {

    case DATA\_TRANSMIT:

      sprintf(txt, "DATA TRANSMIT ERROR | Code: %d\n\r", errorCode);

      break;

    case DS2438:

      sprintf(txt, "DS2438 ERROR | Code: %d\n\r", errorCode);

      break;

    case IMU:

      sprintf(txt, "IMU ERROR | Code: %d\n\r", errorCode);

      break;

    case RECEIVER:

      sprintf(txt, "RECEIVER ERROR | Code: %d\n\r", errorCode);

      break;

    case DSHOT:

      sprintf(txt, "DSHOT ERROR | Code: %d\n\r", errorCode);

      break;

    case PID:

      sprintf(txt, "PID ERROR | Code: %d\n\r", errorCode);

      break;

    default:

      sprintf(txt, "wrong sensor ERROR | Code: %d\n\r", errorCode);

      break;

  }

  // output error message

  Terminal\_Print(txt);

  // turn red LED on and the blue LED off

  \_\_HAL\_TIM\_SET\_COMPARE(LED\_TIM, LED\_RED\_CHANNEL, 10000);

  \_\_HAL\_TIM\_SET\_COMPARE(LED\_TIM, LED\_BLUE\_CHANNEL, 0);

  // disable all interrupts

  \_\_disable\_irq();

  // infinite loop

  while(1);

}

Diese Funktion gibt über das Terminal die Errorquelle mit dem Errorcode aus.

Beispiel: RECEIVER Error | Code 13

Für eine Fehlercodeerklärung – *siehe:* [*Kapitel 5.1.1.1*](#_Übersicht_Fehlercodes_1)

Weiters wird die rote LED durchgehend eingeschalten, und die blaue LED ausgeschalten. Alle Interrupts werden deaktiviert, und das Programm wird durchgehend mit einer Endlosschleife pausiert. Um den Initialisierungsprozess neu zu starten, muss der Mikrocontroller zurückgesetzt werden.

#### Übersicht Fehlercodes

Für jede Verbindung gibt es eigene Fehlercodes, die sich in enum-Objekte befinden. Diese werden in den Header-Dateien der einzelnen Komponente als Datentyp mit den Namen *<Komponente>\_Status* definiert. Zusätzlich hat jedes Objekt einen Wert, der angibt, dass kein Fehler aufgetreten ist: diese sind mit *<Komponente>\_OK* definiert.

**IMU-Fehlercodes:**

Dateiname: IMU\_10DOF.h

typedef enum IMU\_Status

{

    IMU\_OK = 0,

    IMU\_ADDRESS\_ERROR = 1,      // wrong I2C slave address

    IMU\_I2C\_ERROR = 2,          // no I2C typedef set

    IMU\_TIM\_ERROR = 3,          // no TIM typedef set

    IMU\_MPU\_WHOAMI\_ERROR = 10,  // MPU9250 who am i value wrong

    IMU\_MAG\_WHOAMI\_ERROR = 11,  // AK8963 who am i value wrong

    IMU\_BARO\_CHIPID\_ERROR = 12, // BMP280 who am i value wrong

    IMU\_BARO\_INIT\_ERROR = 13    // BMP280 init timeout

} IMU\_Status;

**DS2438 Fehlercodes:**

Dateiname: DS2438.h

typedef enum DS2438\_Status

{

    DS2438\_OK = 0,

    DS2438\_ERROR = 1,           // sensor not found or initialisation error

    DS2438\_VOLTAGE\_ERROR = 2    // battery voltage too low

} DS2438\_Status;

**DShot Fehlercodes:**

Dateiname: dshot\_own.h

typedef enum DShot\_Status

{

    DSHOT\_OK = 0,

    DSHOT\_TIM\_ERROR = 100 // no TIM typedef set

} DShot\_Status;

**Receiver Fehlercodes:**

Dateiname: receiver.h

typedef enum Receiver\_Status

{

    RECEIVER\_OK = 0,                // receiver ok

    RECEIVER\_UART\_ERROR = 1,        // uart configuration doesnt match selected protocol

    RECEIVER\_PWM\_ERROR = 2,         // pwm timer not set

    RECEIVER\_PPM\_ERROR = 3,         // IBUS selected: PPM not configured correctly

    RECEIVER\_TIMEOUT = 4,           // no data signal found

    PROTOCOL\_ERROR = 5,             // selected protocol wrong

    IBUS\_ERROR = 6,                 // IBUS UART DMA not starting

    IBUS\_HEADER\_ERROR = 7,          // IBUS header is wrong

    IBUS\_CHECKSUM\_ERROR = 8,        // IBUS checksum is wrong

    IBUS\_SIGNAL\_LOST\_ERROR = 9,     // IBUS signal lost

    SBUS\_ERROR = 10,                // SBUS UART DMA not starting

    SBUS\_HEADER\_ERROR = 11,         // SBUS header is wrong

    SBUS\_FOOTER\_ERROR = 12,         // SBUS footer is wrong

    SBUS\_SIGNAL\_LOST = 13,          // SBUS signal lost flag is set

    SBUS\_SIGNAL\_FAILSAFE = 14       // SBUS signal failsafe flag is set

} Receiver\_Status;

## Bestimmen der Akkuspannung - DS2438

Für die Diplomarbeit wird der Smart Battery Monitor DS2438 für die Spannungsüberwachung des Akkus verwendet, um zu verhindert, dass die Akkuspannung unter einen kritischen Grenzwert fällt und die Drohne abstürzt. Die Kommunikation zwischen dem Flight-Controller und dem DS2438 findet über das One-Wire Protokoll statt.

Als Vorlage für die Programmierung wurde ein in der 4. Klasse erstelltes Projekt verwendet. Nach Absprache mit den Autoren (Lukas Lindmayr und Marcel Bieder) wurde die Erlaubnis für die Verwendung und Anpassung an die FPV-Drohne Diplomarbeit erteilt.

### One-Wire Protokoll

Der Flight Controller kommuniziert mit dem DS2438 über das One-Wire Protokoll. Dieses besteht aus einer einzelnen halbduplexen, bidirektionalen Leitung. Der Cortex-M7 hat keine One-Wire Peripherie, deswegen muss das Protokoll selbständig implementiert werden. Die Kommunikation mit dem DS2438 beginnt immer aus einer Initialisierungssequenz. Als nächstes wird ein ROM-Funktionsbefehl ausgeführt. Danach wird ein Memory-Funktions-Befehl gesendet, und am Ende werden die Daten ausgetauscht.

**GPIO-Einstellungen in STM32CubeMX**:

Ein Bild, das Text, Screenshot, Software, Zahl enthält.

Automatisch generierte Beschreibung

Abbildung 11: STM32CubeMX Einstellungen DS2438

Damit die Kommunikation funktionieren kann, müssen in STM32CubeMX der *GPIO mode* auf *Output Open Drain* und der *GPIO Pull-up/Pull-down* auf *No pull-up and no pull-down* gesetzt werden.

Der Pin PC0 wird am Flight-Controller für DS2438\_DQ (Datenleitung) verwendet. Dieser Pin kann bei dem Funktionsaufruf von *DS2438\_Init()* bestimmt werden. *(siehe:* [*Kapitel 5.2.3*](#_Initialisierung_DS2438)*)*

#### One-Wire Schreibzyklen

Bei der Kommunikation wird das niederwertigste Bit immer zuerst gesendet.

**Um das Bit 1 zu senden, müssen folgende Zeitbedingungen eingehalten werden:**

Ein Bild, das Text, Diagramm, Reihe, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 12: One-Wire Bit 1 senden

**Um das Bit 0 zu senden, müssen andere Zeitbedingungen eingehalten werden**:

Ein Bild, das Text, Reihe, Diagramm, Quittung enthält.

Automatisch generierte Beschreibung

Abbildung 13: One-Wire Bit 0 senden

**Ein Byte zum DS2438 senden:**

Dateiname: DS2438.c

/\*\*

 \* @brief This function writes one byte to the DS2438

 \* @param byte byte to write

 \* @retval None

 \*/

void DS2438\_WriteByte(uint8\_t byte)

{

  for(int8\_t i = 0; i < 8; i++)

  {

    DS2438\_WriteBit(byte & 0x01);

    byte >>= 1;

  }

}

/\*\*

 \* @brief This function writes one bit to the DS2438

 \* @param bit bit to write

 \* @retval None

 \*/

void DS2438\_WriteBit(int8\_t bit)

{

  if(bit == 1)

  {

    HAL\_GPIO\_WritePin(ds2438\_GPIOPort, ds2438\_GPIOPin, GPIO\_PIN\_RESET);

    DS2438\_DelayUs(10);

    HAL\_GPIO\_WritePin(ds2438\_GPIOPort, ds2438\_GPIOPin, GPIO\_PIN\_SET);

    DS2438\_DelayUs(70);

  }

  else

  {

    HAL\_GPIO\_WritePin(ds2438\_GPIOPort, ds2438\_GPIOPin, GPIO\_PIN\_RESET);

    DS2438\_DelayUs(60);

    HAL\_GPIO\_WritePin(ds2438\_GPIOPort, ds2438\_GPIOPin, GPIO\_PIN\_SET);

    DS2438\_DelayUs(10);

  }

}

Um auf eine bestimmte Page Daten schreiben zu können, müssen diese zuerst auf das Scratchpad geschrieben werden. Danach muss das Scratchpad in den ROM oder RAM kopiert werden.

Zuerst wird die Initialisierungssequenz ausgeführt, und danach wird der Skip-ROM- (0xCC) und der Write-Scratchpad-Befehl (0x4E) ausgeführt. Anschließend muss die gewünschte Page-Nummer übertragen werden. Darauf folgen die Daten und eine zweite Initialisierungssequenz. Damit die Daten vom Scratchpad auch in den Speicher des Sensors übertragen werden, muss noch ein Skip-ROM- (0xCC) und ein Copy-Scratchpad-Befehl (0x48), gefolgt von der Page-Nummer, ausgeführt werden.

**Zu einer Page Daten senden:**

Dateiname: DS2438.c

/\*\*

 \* @brief This function writes the data to one page of the DS2438

 \* @param page page number (0 - 7)

 \* @param pageData data of page

 \* @return DS2438\_Status

 \*/

DS2438\_Status DS2438\_WritePage(uint8\_t page, int16\_t \*pageData)

{

  // reset + presence pulse

  if(DS2438\_Reset() == DS2438\_ERROR)

    return DS2438\_ERROR;

// copy current data to scratchpad

  DS2438\_WriteByte(DS2438\_SKIP\_ROM);

  DS2438\_WriteByte(DS2438\_WRITE\_SP);

  DS2438\_WriteByte(page);

  for(uint8\_t i = 0; i < 9; i++)

    DS2438\_WriteByte(pageData[i]);

// reset + presence pulse

  if(DS2438\_Reset() == DS2438\_ERROR)

    return DS2438\_ERROR;

  DS2438\_WriteByte(DS2438\_SKIP\_ROM);

  DS2438\_WriteByte(DS2438\_COPY\_SP);

  DS2438\_WriteByte(page);

  return DS2438\_OK;

}

#### One-Wire Lesezyklen

Genau wie bei den Schreibzyklen gibt es auch spezielle Zeitkriterien, die bei den Lesezyklen eingehalten werden müssen. Es wird auch das niederwertigste Byte zuerst empfangen.

Ein Bild, das Text, Diagramm, Reihe, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 14: One-Wire Bit empfangen

**Ein Byte vom DS2438 auslesen:**

Dateiname: DS2438.c

/\*\*

 \* @brief This function reads one byte from the DS2438 (LSB first)

 \* @return uint8\_t

 \*/

uint8\_t DS2438\_ReadByte(void)

{

  uint8\_t byte = 0;

  for(int8\_t i = 0; i < 8; i++)

    byte |= (DS2438\_ReadBit() << i);

  return byte;

}

/\*\*

 \* @brief This function reads one bit from the DS2438

 \* @return int8\_t

 \*/

int8\_t DS2438\_ReadBit(void)

{

  int8\_t bit = 0;

  HAL\_GPIO\_WritePin(ds2438\_GPIOPort, ds2438\_GPIOPin, GPIO\_PIN\_RESET);

  DS2438\_DelayUs(10);

  HAL\_GPIO\_WritePin(ds2438\_GPIOPort, ds2438\_GPIOPin, GPIO\_PIN\_SET);

  DS2438\_DelayUs(10);

  // read current pin level

  bit = HAL\_GPIO\_ReadPin(ds2438\_GPIOPort, ds2438\_GPIOPin) == GPIO\_PIN\_SET;

  DS2438\_DelayUs(60);

  return bit;

}

Um von einer bestimmen Page Daten lesen zu können, müssen diese zuerst vom Sensor auf das Scratchpad geschrieben werden. Danach müssen die Daten vom Scratchpad eingelesen werden.

Zuerst wird die Initialisierungssequenz ausgeführt und danach wird der Skip-ROM- (0xCC) und der Recall-Memory-Befehl (0xB8) ausgeführt. Anschließend muss die gewünschte Page-Nummer übertragen werden. Darauf folgt eine zweite Initialisierungssequenz. Um jetzt die Daten vom Scratchpad zu lesen, muss ein Skip-ROM- (0xCC) und ein Read-Scratchpad-Befehl (0xBE), gefolgt von der Page-Nummer, ausgeführt werden. Danach können die Daten auf der One-Wire-Leitung empfangen werden.

**Von einer Page Daten auslesen:**

Dateiname: DS2438.c

/\*\*

 \* @brief This function reads the data from one page of the DS2438

 \* @param page page number (0 - 7)

 \* @param pageData data of page

 \* @return DS2438\_Status

 \*/

DS2438\_Status DS2438\_ReadPage(uint8\_t page, int16\_t \*pageData)

{

  // reset + presence pulse

  if(DS2438\_Reset() == DS2438\_ERROR)

    return DS2438\_ERROR;

// copy current data to scratchpad

  DS2438\_WriteByte(DS2438\_SKIP\_ROM);

  DS2438\_WriteByte(DS2438\_RECALL\_MEM);

  DS2438\_WriteByte(page);

  // reset + presence pulse

  if(DS2438\_Reset() == DS2438\_ERROR)

    return DS2438\_ERROR;

// read scratchpad data

  DS2438\_WriteByte(DS2438\_SKIP\_ROM);

  DS2438\_WriteByte(DS2438\_READ\_SP);

  DS2438\_WriteByte(page);

  for(int8\_t i = 0; i < 9; i++)

    pageData[i] = DS2438\_ReadByte();

  return DS2438\_OK;

}

#### Initialisierungssequenz

Die Initialisierungssequenz besteht immer aus einen Reset- und einen Presence-Puls, die die folgenden zeitlichen Anforderungen einhalten müssen:

Ein Bild, das Text, Diagramm, Reihe, parallel enthält.

Automatisch generierte Beschreibung

Abbildung 15: One-Wire Initialisierungssequenz

Wenn der One-Wire Slave bei dem Presence-Puls nicht nach 60µs bis 240µs die Leitung in den high-Zustand setzt, besteht ein Problem bei der Kommunikation zwischen Master und Slave.

**Initialisierungssequenz DS2438:**

Dateiname: DS2438.c

/\*\*

 \* @brief This function resets / checks device presence

 \* @return DS2438\_Status

 \*/

DS2438\_Status DS2438\_Reset(void)

{

  // reset DS2438

  // send reset pulse (min 480us)

  HAL\_GPIO\_WritePin(ds2438\_GPIOPort, ds2438\_GPIOPin, GPIO\_PIN\_RESET);

  DS2438\_DelayUs(480);

  // release line -> change to receive mode

  HAL\_GPIO\_WritePin(ds2438\_GPIOPort, ds2438\_GPIOPin, GPIO\_PIN\_SET);

  // wait until slave sends presence pulse

  DS2438\_DelayUs(70);

  // read current pin state

  int8\_t pin = HAL\_GPIO\_ReadPin(ds2438\_GPIOPort, ds2438\_GPIOPin);

  DS2438\_DelayUs(410);

  // check pin state (0 -> found, 1 -> not found)

  if(pin == GPIO\_PIN\_SET)

    return DS2438\_ERROR;

  return DS2438\_OK;

}

#### ROM-Funktionsbefehl

Es gibt viele verschiedene ROM-Funktionsbefehle. Für das Spannungsmonitoring ist nur der SKIP-ROM-Befehl (0xCC) wichtig. Dieser Befehl erlaubt es dem Master, direkt einen Memory-Funktionsbefehl zu senden, ohne einen ROM-Code senden zu müssen. Dadurch wird Zeit gespart.

#### Memory-Funktionsbefehl

Der einzige Memory-Funktionsbefehl, der wichtig für das Spannungsmonitoring ist, ist der CONVERT-V (0xB4) Befehl. Diese Anweisung startet die Analog-Digital-Spannungswandlung des Sensors. Wenn die Umwandlung beendet ist, wird das ADB-Flag (A/D Conversion Busy Flag) vom Sensor auf 0 gesetzt.

### Registerübersicht DS2438

Der DS2438 ist unterschiedliche Register (Pages) unterteilt, die alle notwendigen Einstellungen und Messdaten des Sensors beinhalten.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Abbildung 16: Registerübersicht DS2438

Für die Spannungsüberwachung ist nur die Page 0 wichtig, da hier alle Einstellungen und Messwerte gesetzt beziehungsweise ausgelesen werden.

|  |  |
| --- | --- |
| **Byte** | **Verwendungszweck** |
| 0 | Einstellungen setzen |
| 3 und 4 | Nieder- und hochwertiges Byte des Spannung-Messwertes |

### Initialisierung DS2438

Dateiname: DS2438.c

/\*\*

 \* @brief This function initializes the DS2438

 \* @param htim pointer to TIM\_HandleTypeDef (timer for us delay)

 \* @param gpio\_Port GPIOx (port of DQ Pin)

 \* @param gpio\_Pin GPIO\_PIN\_x (pin of DQ Pin)

 \* @return DS2438\_Status

 \*/

DS2438\_Status DS2438\_Init(TIM\_HandleTypeDef \*htim, GPIO\_TypeDef \*gpio\_Port, uint16\_t gpio\_Pin)

{

  if(htim == NULL)

    return DS2438\_ERROR;

  DS2438\_DelayTimer = htim;

  HAL\_TIM\_Base\_Start(DS2438\_DelayTimer); // start timer for DS2438\_DelayUs

  ds2438\_GPIOPort = gpio\_Port;

  ds2438\_GPIOPin = gpio\_Pin;

  if(DS2438\_Reset() == DS2438\_ERROR)

    return DS2438\_ERROR;

// set Vad as A/D converter input

  int16\_t pageData[9] = {0x00};

  if(DS2438\_ReadPage(0x00, pageData) == DS2438\_ERROR)

    return DS2438\_ERROR;

// pageData[0] |= 0x08; // supply voltage

  pageData[0] &= 0xF7;    // external input

  if(DS2438\_WritePage(0x00, pageData) == DS2438\_ERROR)

    return DS2438\_ERROR;

// check current voltage

  int8\_t errorCode = DS2438\_ReadVoltage();

  if(errorCode != DS2438\_OK)

    return errorCode;

  return DS2438\_OK;

}

Der/Die BenutzerIn kann sich aussuchen, ob die Versorgungsspannung oder die Spannung am Eingang vom A/D-Wandler gemessen werden soll. Dies funktioniert mit dem AD-Bit in der Page 0 im Byte[0]. Wenn das AD-Bit auf 1 gesetzt wird, wird die Versorgungsspannung gemessen. Wenn das Bit auf 0 gesetzt wird, misst der Sensor die Eingangsspannung vom A/D-Wandler.

### Spannungsüberwachung

Die eingelesen Spannungswerte werden in folgendem Format gespeichert:

Ein Bild, das Text, Schrift, Zahl, Reihe enthält.

Automatisch generierte Beschreibung

Abbildung 17: DS2438 Spannungsregister Format

Um mit den eingelesenen Werten einen Spannungswert zu bilden, müssen MSB und LSB zusammengefügt werden. Da die Werte im Spannungsregisters eine Auflösung von 10mV pro LSB haben, muss der Registerwert mit 10mV multipliziert oder durch 100 gerechnet werden, um den eigentlichen Spannungswert zu bestimmen.

**Spannung einlesen:**

Dateiname: DS2438.c

/\*\*

 \* @brief This function reads the current voltage value of the DS2438

 \* @attention the voltage gets stored in the global variable 'ds2438\_Voltage'

 \* @return DS2438\_Status

 \*/

DS2438\_Status DS2438\_ReadVoltage(void)

{

  // start measurement (send CONVERT T command)

  if(DS2438\_StartVoltageMeasurement() == DS2438\_ERROR)

    return DS2438\_ERROR;

// wait for measurement to be complete (ADB flag: 1 = busy, 0 = ready)

  while(DS2438\_ReadControlVoltageFlag());

  int16\_t pageData[9] = {0x00};

  // read data

  if(DS2438\_ReadPage(0x00, pageData) == DS2438\_ERROR)

    return DS2438\_ERROR;

// extracting voltage bytes

  int16\_t voltageLSB = pageData[3];

  int16\_t voltageMSB = pageData[4];

  ds2438\_Voltage = (((voltageMSB & 0x3) << 8) | (voltageLSB)) / 100.0;

  ds2438\_Voltage \*= 3; // times 3 because of resistor voltage divider

  if(ds2438\_Voltage <= DS2438\_MIN\_VOLTAGE)

      return DS2438\_VOLTAGE\_ERROR;

  return DS2438\_OK;

}

Da der A/D-Wandler des DS2438 nur einen Spannungsbereich von 0-10V hat, muss die Eingangsspannung vor dem Sensor geteilt werden. Auf der Platine des Flight-Controllers ist vor dem Eingang des A/D-Wandlers des DS2438 ein 3:1 Spannungsteiler eingebaut. Daher wird der Spannungswert mit einem Faktor von drei multipliziert, um den eigentlichen Messwert zu bestimmen.

Der endgültige Spannungswert wird in der globalen Variable *ds2438\_Voltage* gespeichert. Wenn die Spannung einen kritischen Wert unterschreitet (standardmäßig 19V) liefert die Funktionen einen DS2438\_VOLTAGE\_ERROR.

Wenn der Error geschickt wird, sollte das Fliegen der Drohne beendet werden, damit der Akku nicht beschädigt wird.

## Real Time System Interrupt (MAIN\_ISR)

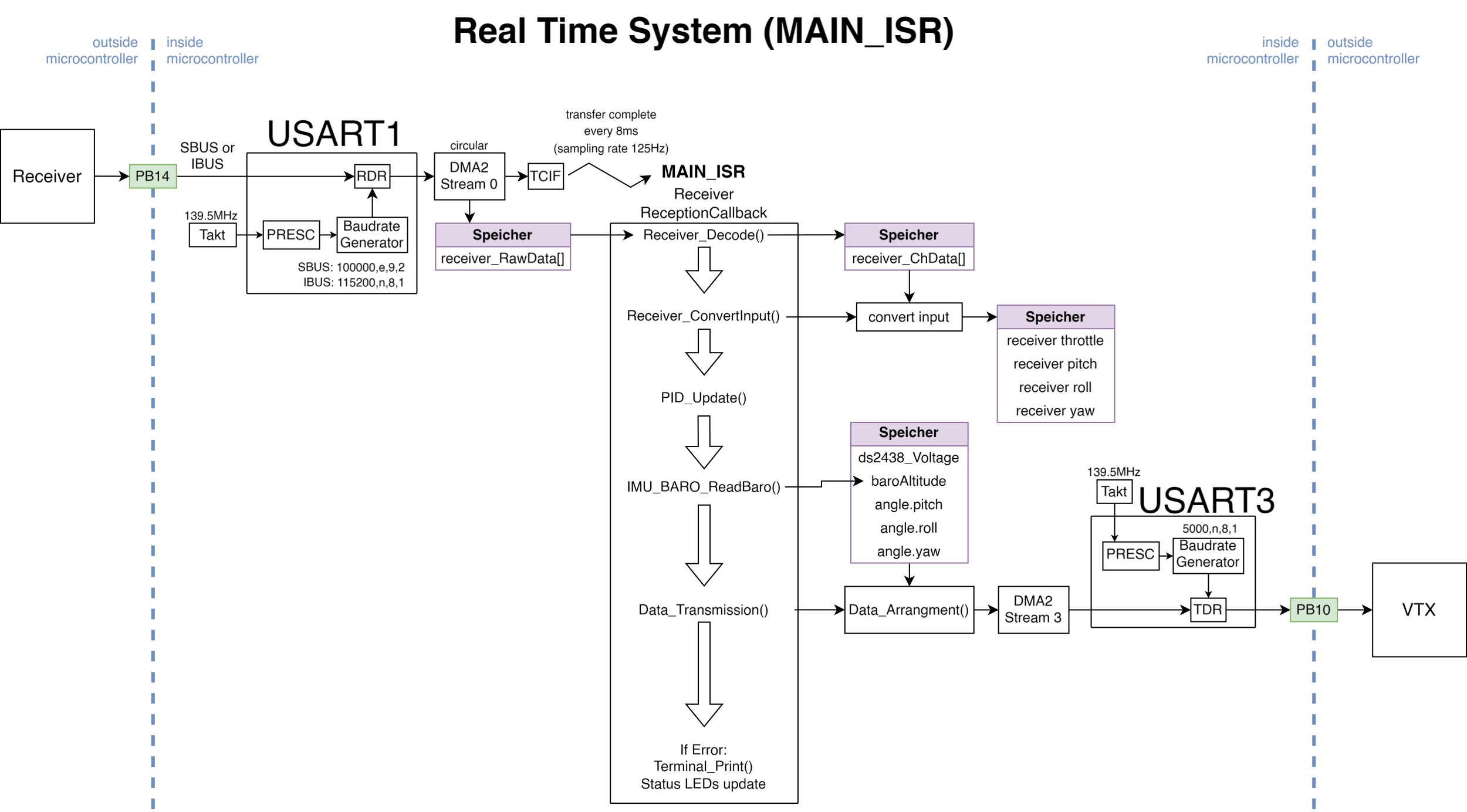


Abbildung 18: Real Time System Interrupt Architektur

Die MAIN\_ISR wird alle 8ms vom Receiver ausgelöst. Dieser sendet auch bei Verbindungsverlust zu der Fernsteuerung alle 8ms ein Datenpaket. Diese Datenpakete werden mit der USART1-Peripherie eingelesen und mit dem DMA-Controller abgespeichert.

Die empfangenen Rohdaten werden mit der Funktion *Receiver\_Decode()* in die einzelnen Kanäle dekodiert *(siehe:* [*Kapitel 5.4.4 Teil #1*](#_Empfangssoftware)*)*.

Mit der Funktion *Receiver\_ConvertInput()* werden die Kanalwerte in Throttle-, Pitch-, Roll- und Yaw-Werte umgewandelt *(siehe:* [*Kapitel 5.4.4 Teil #2*](#_Empfangssoftware)*)*. Dabei ist der Throttle-Wert von 0% bis 100% die Motorgeschwindigkeit für alle Motoren und die anderen alle ein Wert in Grad für die Drehung, um die jeweilige Achse.

Diese Daten werden dann mit der Funktion *PID\_Update()* zum PID-Regler als Sollwerte geschickt *(siehe:* [*Kapitel 5.8.2*](#_PID-Algorithmus)*)*.

Danach wird vom Barometer über I2C1-Peripherie der Luftdruck eingelesen, in die Höhenlage umgewandelt und im Speicher abgelegt *(siehe: Kapitel* [*5.5.4.3*](#_Einlesen_der_Barometer-Daten)*,* [*5.5.4.4*](#_Berechnung_der_Höhe)*)*.

Als letzten Schritt werden die Sensordaten (Spannung, Höhe, Lagewinkel) zur Groundstation mit Hilfe der USART3-Peripherie und dem DMA-Controller geschickt *(siehe: Kapitel Ben daten senden)*.

Wenn ein Fehler während der MAIN\_ISR auftritt, kann der aktuelle Status über das Terminal und eingebaute Status-LEDs angezeigt werden *(siehe: Kapitel* [*5.9*](#_Terminal_Übertragung_und)*,* [*5.10*](#_Terminal_Ausgabe_1)*,* [*5.11*](#_Status_–_LEDs)*)*.

## Einlesen der Daten von Fernsteuerung

Um eine FPV-Drohne steuern zu können, muss mindestens ein 4-Kanal Sender und Empfänger verwendet werden, um die gewünschten Steuerungsdaten, Throttle, Pitch, Roll und Yaw, zu senden. Mit weiteren Kanälen können zusätzliche Funktionen, wie zum Beispiel ON/OFF-Switch, realisiert werden.

Für das Projekt wurde der Turnigy 9X 9Ch Mode 2 Transmitter mit dem TGY-iA6C Receiver ausgewählt. Diese Kombination bietet eine große Auswahl von programmierbaren Schaltern, deren Stellungen mittels eines schnellen, digitalen seriellen Protokolls übertragen wird.



Abbildung 19: Fernsteuerung



Abbildung 20: Receiver

In der Abbildung 3 wird die verwendete Tastenbelegung auf der Fernsteuerung dargestellt:

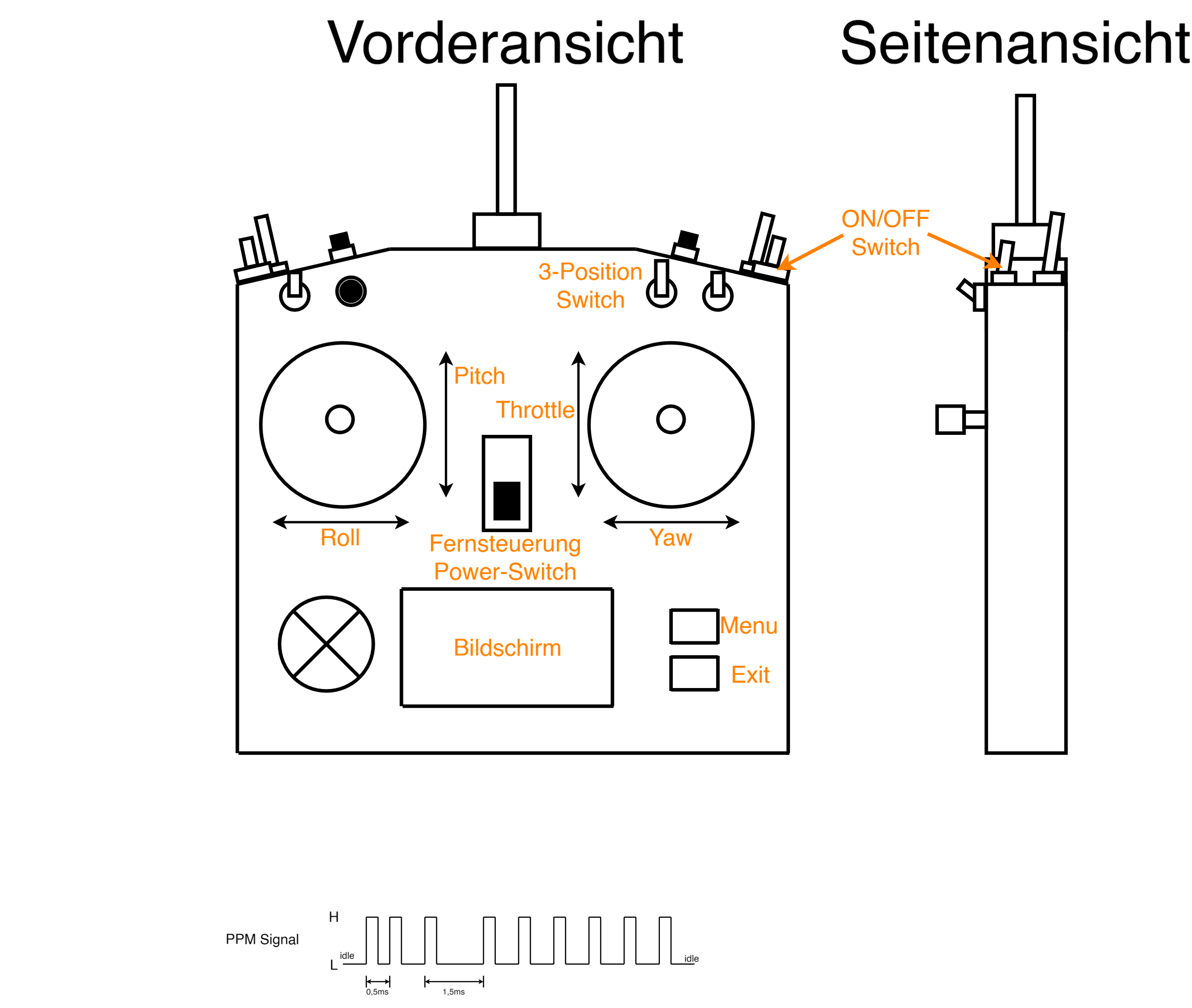


Abbildung 21: Fernsteuerung Tastenbelegung

Die zwei großen Joysticks reichen aus, um die Drohne steuern zu können. Für eine bessere Bedienung werden zwei zusätzlich Schalter verwendet.

Der ON/OFF-Switch ermöglicht das Ein- und Ausschalten der Fernsteuerungseingaben.

Mit dem 3-Position-Switch kann zwischen mehreren Flugmodi unterschieden werden.

Um die richtige Funktionalität der Schalter zur gewährleisten - *siehe:* [*Kapitel 5.4.1*](#_Konfiguration)

Wenn der ON/OFF-Switch nach unten zeigt, schalten sich die Motoren aus und alle Fernsteuerungsbefehle werden übernommen. Wenn der Schalter nach oben zeigt, wirkt der 3-Position-Switch als Modus Auswahl.

In der oberen Stellung befindet sich die Drohne im „Safe-Mode“. Das bedeutet, dass der maximale Throttle-Wert, der zur ESC geschickt werden kann, begrenzt wird.

In der mittleren Stellung schaltet wird der „Normal-Mode“ eingeschaltet. Dieser Modus verhält sich gleich, wie der „Safe-Mode“, nur erlaubt er, höhere Motordrehzahl.

Die untere Position stellt einen Hover-Mode ein. Die Drohne versucht jetzt, sich selbst gerade in der Ebene mit Sensoren- und Throttlewerten zu halten.

Wichtig: Wenn die Versorgungsspannung der Fernsteuerung weniger als 8,5V beträgt, beginnt ein Buzzer alle 5 Sekunden den Benutzer zu alarmieren. Die aktuelle Spannung kann auf dem Bildschirm der Fernsteuerung überprüft werden.

**Switch Error:**

Beim Starten der Fernsteuerung kann es vorkommen, dass am Bildschirm „switch-error!“ angezeigt wird. Dieser Fehler wird angegeben, wenn die Schalter nicht auf High-Position sind.

Das bedeutet: Beim Starten der Fernsteuerung müssen alle Schalter nach oben zeigen beziehungsweise die Schalter, die auf der Oberseite montiert worden sind, müssen nach hinten zeigen, wie in Abbildung 3 dargestellt ist.

### Konfiguration Fernsteuerung

Damit die ausgewählte Fernsteuerung mit unserem System funktioniert, müssen grundlegende Vorbereitungen vor der Inbetriebnahme getroffen und kontrolliert werden.

**Verbindung mit Empfänger:**

Um die Verbindung zu testen, schaltet man die Fernsteuerung und den Receiver ein. Die eingebaute Status-LED des Receivers fängt schnell zum Blinken an. Wenn die LED nach ein paar Sekunden anfängt durchgehend zu leuchten, ist die Fernsteuerung mit dem Receiver verbunden.

Wenn die LED anfängt langsamer zu blinken, wurde die Verbindung nicht gefunden und die Fernsteuerung muss mit dem Receiver neu gekoppelt werden:

1. Schalte die Fernsteuerung und den Receiver aus
2. Halte den Bind-Knopf auf der Rückseite der Fernsteuerung, während diese angeschaltet wird
3. Schalte den Receiver ein
4. Warte, bis die Status-LED auf dem Receiver durchgehend leuchtet
5. Lasse den Bind-Knopf los und starte die Fernsteuerung neu, um die Koppelung zu testen

**Einstellungen in Fernsteuerung:**

Damit die Signale richtig von der Flugsoftware interpretiert werden können, müssen folgende Einstellungen in der Fernsteuerung getroffen werden:

Beim längeren gedrückt halten der MENU-Taste wird eine Auswahl zwischen „System Setting“ und „Function Setting“ angezeigt.

Die folgenden Tabellen zeigen die Einstellung, die für die richtige Funktionsweise notwendig sind:

|  |  |
| --- | --- |
| **System Setting** | |
| Type Select | ACRO |
| Modeuat | PPM |
| Stick Set | MODEL 1 |
| Output Select | PPM s-BUS |

|  |  |  |
| --- | --- | --- |
| **Function Setting** | | |
| AUX-CH  (für ON/OFF Switch) | CH5 Gear | |
| PROG. MIX  (für 3-Position-Switch) | MIX1 | STATE: ACT  MASTER: GYR  SLAVE: FLP  OFFSET: 000  UPRATE: -100  DNRATE: 100  SW: NOR |
| MIX2 | STATE: ACT  MASTER: GYR  SLAVE: FLP  OFFSET: 000  UPRATE: 000  DNRATE: 000  SW: ID1 |
| MIX3 | STATE: ACT  MASTER: GYR  SLAVE: FLP  OFFSET: 000  UPRATE: 100  DNRATE: -100  SW: ID2 |
| DISPLAY | Kontrolle der Kanal-Ausgänge | |

Diese Einstellungen sind notwendig, damit der ON/OFF Switch und der 3-Position-Switch richtig dekodiert werden, und eine richtige Funktionalität nachweisen können.

Zur Kontrolle werden in Balkendiagrammen die einzelnen Kanalwerte unter DISPLAY angezeigt.

### Unterstützte Protokolle

Der Receiver unterstützt eine gleichzeitige Signalausgabe von einem analogen und digitalen Signal. Der Pin PPM gibt immer ein PPM-Signal aus, während der zweite Pin S.BUS entweder ein S.Bus oder I.Bus Signal ausgibt. Die Ausgabe kann in den Fernsteuerungseinstellungen festgelegt werden *(siehe:* [*Kapitel 5.4.1*](#_Konfiguration)*)*.

#### PPM (Pulse Position Modulation)

Das PPM-Signal ist das einzige analoge Signal, das der Receiver ausgibt. Die Daten können mit der Timer-Peripherie eingelesen werden, indem die Länge der Periodendauer bestimmt wird. Die Daten kommen in 9 Kanälen mit einem 8,1ms low-aktiven Ruhezustand. Die Kanalwerte befinden sich in den Bereich 0,5ms (0% Throttle) bis 1,5ms (100% Throttle).

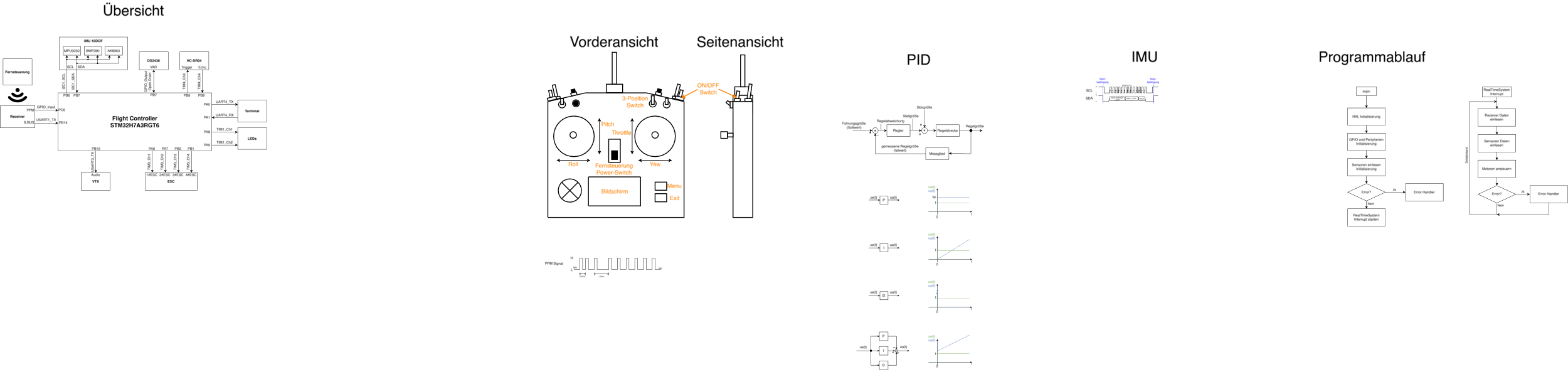


Abbildung 22: Beispiel PPM-Signal

Dadurch, dass die digitalen Protokolle um ein Vielfaches schneller und nicht sehr fehleranfällig sind, unterstützt die Steuerungssoftware das PPM-Protokoll nicht. Das PPM-Signal wird nur bei der Initialisierung für einen Verbindungstest mit GPIO-Input und nicht für das Fliegen der Drohne verwendet.

#### S.Bus

Das S.Bus-Protokoll ist das empfohlene Protokoll für die Benutzung der Drohne. Es handelt sich um ein digitales Protokoll, das mittels der UART-Peripherie eingelesen wird. Das Protokoll ist schneller als das analoge Signal PPM und weniger fehleranfällig als das I.Bus-Protokoll, da das Signal mit invertierten Pegeln übertragen wird. Die Datenpakete werden kontinuierlich gesendet, sind ungefähr 3ms lang und besitzen einen ungefähr 4,7ms langen low-aktiven Ruhezustand. Der digitale Wertebereich befindet sich zwischen 350 und 1680.

**UART-Konfiguration in STM32CubeMX**:

Ein Bild, das Text, Screenshot, Software, Webseite enthält.

Automatisch generierte Beschreibung

Abbildung 23: STM32CubeMX Einstellungen S.Bus

Die wichtigsten UART-Einstellungen sind in der folgenden Tabelle dargestellt:

|  |  |
| --- | --- |
| **Einstellung** | **Wert** |
| Mode | Single Wire (Half-Duplex) |
| Baud Rate | 100000 Bits/s |
| Word Length | 9 Bits (including Parity) |
| Parity | Even |
| Stop Bits | 2 |
| RX Pin Active Level Inversion | Enable |

**Protokolldaten:**

Das S.Bus Protokoll besteht aus 25 Bytes:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Byte[0] | S.Bus header, 0x0F | | | |
| Byte[1…22] | Daten, 16 Kanäle | | | |
| Byte[23] | Bit[0] (0x01)  Kanal 17 | Bit[1] (0x02)  Kanal 18 | Bit[2] (0x04)  Frame Lost Flag | Bit[3] (0x08)  Failsafe Flag |
| Byte[24] | S.Bus footer, 0x00 | | | |

Die Bytes[1…22] beinhalten die Daten der einzelnen Kanäle. Jeder Kanal besteht aus 11 Bits, die nacheinander gesendet werden, wobei die ersten Bits des Kanals den niedrigsten Stellenwert haben.

Beispiel für eine Dekodierung:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Byte** | 0 | 1 | 2 | 3 | … | 23 | 24 |
| **Daten**  header  0b001 0011 0001  Kanal 1  0b001 1001 0001  Kanal 2  flags  footer | 0x0F | 0b00110001 | 0b00110001 | 0b00110001 | … | 0b00000000 | 0x00 |

#### I.Bus

Das I.Bus-Protokoll ist das zweite, digitale Protokoll, welches der Receiver ausgeben kann. Die Daten werden über die UART-Peripherie eingelesen. Die Datenpakete werden kontinuierlich gesendet, sind ungefähr 3ms lang und besitzen einen ungefähr 4,7ms langen high-aktiven Ruhezustand. Im Gegensatz zum S.Bus Protokoll kann mit I.Bus kein Verbindungsverlust festgestellt werden. Der digitale Werteberiech befindet zwischen 1070 und 1920.

**UART-Konfiguration in STM32CubeMX**:

Ein Bild, das Text, Screenshot, Software, Computersymbol enthält.

Automatisch generierte Beschreibung

Abbildung 24: STM32CubeMX Einstellungen I.Bus

Die wichtigsten UART-Einstellungen sind in der folgenden Tabelle dargestellt:

|  |  |
| --- | --- |
| **Einstellung** | **Wert** |
| Mode | Single Wire (Half-Duplex) |
| Baud Rate | 115200 Bits/s |
| Word Length | 8 Bits (including Parity) |
| Parity | None |
| Stop Bits | 1 |
| RX Pin Active Level Inversion | Disable |

**Protokolldaten:**

Das I.Bus Protokoll besteht aus 32 Bytes:

|  |  |
| --- | --- |
| Byte[0] | Protokolllänge, 0x20 |
| Byte[1] | Command Code, 0x40 |
| Byte[2…29] | Daten, 14 Kanäle |
| Byte[30…31] | Checksumme |

Die Bytes[2…29] beinhalten die Daten der einzelnen Kanäle. Jeder Kanal besteht aus 2 Bytes, welche in der little-endian-byte-order geschickt werden. Die Checksumme besteht auch aus 2 Byte, welche in der little-endian-byte-order gesendet werden. Die Checksumme berechnet sich aus 0xFFFF weniger der Summe der ersten 30 Bytes.

Beispiel für die Dekodierung:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Byte** | 0 | 1 | 2 | 3 | … | 30 | 31 |
| **Daten**  Protokolllänge  Command Code  0x05DC  Kanal 1  0x3412  Checksumme | 0x20 | 0x40 | 0xDC | 0x05 | … | 0x12 | 0x34 |

Beispiel Checksumme:

### Initialisierung Empfangssoftware

Dadurch, dass der Receiver die Daten kontinuierlich sendet, wird ein DMA (direct memory access) - Controller zum Einlesen verwendet, der durchgehend auf die Werte wartet und diese einliest.

**DMA-Konfiguration in STM32CubeMX:**

Ein Bild, das Text, Screenshot, Software, Zahl enthält.

Automatisch generierte Beschreibung

Abbildung 25: Einstellungen DMA für Receiver in STM32CubeMX

Die wichtigste Einstellung ist, dass der Modus als *Circular* festgelegt wird. Dadurch wird der DMA-Stream automatisch neu gestartet, wenn dieser den Einlesevorgang abgeschlossen hat. Dies ermöglicht ein durchgehendes Einlesen.

Dateiname: receiver.c

/\*\*

 \* @brief This function calibrates and starts uart receive dma with selected protocol

 \* @param proto protocol to use (SBUS / IBUS)

 \* @param huart pointer to a UART\_HandleTypeDef structure (input u(s)art)

 \* @return Receiver\_Status

 \*/

Receiver\_Status Receiver\_Init(Receiver\_Protocol proto, UART\_HandleTypeDef \*huart)

{

  receiver\_InputUART = huart;         // set input uart

  receiver\_SelectedProtocol = proto;  // set serial protocol

  // set custom reception complete ISR

  HAL\_UART\_RegisterCallback(receiver\_InputUART, HAL\_UART\_RX\_COMPLETE\_CB\_ID, Receiver\_ReceptionCallback);

  switch(receiver\_SelectedProtocol)

  {

    /\*\*

     \* 115200 baud

     \* 8 data bits, 1 stop bit, no parity

     \* LSB first, not inverted

     \* 32 Bytes:

     \*      Byte[0]: protocol length, 0x20

     \*      Byte[1]: command code, 0x40

     \*      Byte[2-29]: channel data, 14 channels, 2 byte each, little endian

     \*      Byte[30-31]: checksum, little endian, 0xFFFF - sum of other 30 bytes = checksum

     \*/

    case IBUS:

    {

      // check if uart is configured via baudrate

      if(receiver\_InputUART->Init.BaudRate != 115200)

        return RECEIVER\_UART\_ERROR;

      // check if transmitter is connected (ppm signal reception)

      uint8\_t timeout = 0;

      int8\_t tmp\_PinState = HAL\_GPIO\_ReadPin(RECEIVER\_PPM\_GPIO\_Port, RECEIVER\_PPM\_Pin);

      while(tmp\_PinState == HAL\_GPIO\_ReadPin(RECEIVER\_PPM\_GPIO\_Port, RECEIVER\_PPM\_Pin))

      {

        // if the ppm signal doesn't change in 10ms -> error

        if(timeout++ > 10)

          return RECEIVER\_PPM\_ERROR;

        HAL\_Delay(1);

      }

      uint8\_t tmp[2] = {0};

      timeout = 0;

      // calibrate reception to begin of protocol

      while(!(tmp[0] == 0x20 && tmp[1] == 0x40))

      {

        // if the header is wrong 100x -> error

        if(timeout++ > 100)

          return RECEIVER\_TIMEOUT;

        HAL\_UART\_Receive(receiver\_InputUART, tmp, 2, 3);

      }

      HAL\_Delay(4); // wait to sync to next data packet

      // start DMA read i.bus signal

      if(HAL\_UART\_Receive\_DMA(receiver\_InputUART, receiver\_RawData, 32) != HAL\_OK)

        return IBUS\_ERROR;

      // set min/max values of receiver input data

      receiver\_InputLimits.min = 1070;

      receiver\_InputLimits.max = 1920;

      break;

    }

    /\*\*

     \* 100000 baud

     \* 9 data bits, 2 stop bit, even parity

     \* LSB first, inverted

     \* 25 Bytes:

     \*      Byte[0]: protocol header, 0x0F

     \*      Byte[1-22]: channel data, 16 channels, 11 bits each

     \*      Byte[23]:

     \*          bit[4]: signal failsafe flag

     \*          bit[5]: signal lost flag

     \*          bit[6]: digital channel 18

     \*          bit[7]: digital channel 17

     \*      Byte[24]: protocol footer, 0x00

     \*/

    case SBUS:

    {

      // check if uart is configured via baudrate

      if(receiver\_InputUART->Init.BaudRate != 100000)

        return RECEIVER\_UART\_ERROR;

      // check if transmitter is connected (ppm signal reception)

      uint8\_t timeout = 0;

      int8\_t tmp\_PinState = HAL\_GPIO\_ReadPin(RECEIVER\_PPM\_GPIO\_Port, RECEIVER\_PPM\_Pin);

      while(tmp\_PinState == HAL\_GPIO\_ReadPin(RECEIVER\_PPM\_GPIO\_Port, RECEIVER\_PPM\_Pin))

      {

        // if the ppm signal doesn't change in 10ms -> error

        if(timeout++ > 10)

          return RECEIVER\_PPM\_ERROR;

        HAL\_Delay(1);

      }

      uint8\_t tmp = 0;

      timeout = 0;

      // calibrate reception to begin of protocol

      while(tmp != 0x0F)

      {

        // if the header is wrong 100x -> error

        if(timeout++ > 100)

          return RECEIVER\_TIMEOUT;

        HAL\_UART\_Receive(receiver\_InputUART, &tmp, 1, 4);

      }

      HAL\_Delay(4); // wait to sync to next data packet

      // start DMA read s.bus signal

      if(HAL\_UART\_Receive\_DMA(receiver\_InputUART, receiver\_RawData, 25) != HAL\_OK)

        return SBUS\_ERROR;

      // set min/max values of receiver input data

      receiver\_InputLimits.min = 350;

      receiver\_InputLimits.max = 1680;

      break;

    }

    // wrong or no protocol selected

    case NO\_PROTO:

    default:

      return PROTOCOL\_ERROR;

      break;

  }

  // set value range and half value

  receiver\_InputLimits.delta = receiver\_InputLimits.max - receiver\_InputLimits.min;

  receiver\_InputLimits.half = (receiver\_InputLimits.max + receiver\_InputLimits.min) / 2;

  return RECEIVER\_OK;

}

In den Programm *Receiver\_Init()* wird je nach ausgewählten Empfangsprotokoll (SBUS oder IBUS) die UART-Peripherie mit den Datenpaketen synchronisiert und dann mit dem DMA-Controller durchgehend eingelesen.

Der Receiver schickt auch bei Verbindungsverlust alle 8ms ein Datenpaket. Bei Empfangsabschluss wird ein Interrupt ausgelöst. Die Interrupt Service Routine ist mit der Funktion *Receiver\_ReceptionCallback()* realisiert. In dieser Funktion erfolgt die gesamte Echtzeitregelung der Drohne *(siehe:* [*Kapitel 5.4.4*](#_Empfangssoftware)*)*.

### Empfangssoftware

Wenn der DMA-Controller ein Packet fertig eingelesen hat, wird ein Interrupt mit der Funktion *Receiver\_ReceptionCallback()* aufgerufen:

Dateiname: receiver.c

/\*\*

 \* @brief This function is the ISR for DMA receiver reception complete (called every 8ms)

 \* @details all data gets decoded, PID updated and data send to groundstation

 \* @param huart

 \*/

void Receiver\_ReceptionCallback(UART\_HandleTypeDef \*huart)

{

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  --------------------- part #1: receiver input ---------------------

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  uint8\_t errorCode;

  float throttle = 0, pitch = 0, roll = 0, yaw = 0;

  errorCode = Receiver\_Decode(); // decode raw data to channel data

  // check for decode errors

  if(errorCode != RECEIVER\_OK)

  {

    // check connection lost

    if(errorCode == IBUS\_SIGNAL\_LOST\_ERROR || errorCode == SBUS\_SIGNAL\_LOST || errorCode == SBUS\_SIGNAL\_FAILSAFE)

      Receiver\_FailsafeHandler();

    // output error code

    sprintf(txt, "Receiver Error %d\n\r", errorCode);

    Terminal\_Print(txt);

  }

  else

  {

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    ---------------------- part #2: motor control ---------------------

    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    // convert to throttle speed and stick positions to angles

    Receiver\_ConvertInput(throttle, pitch, roll, yaw);

    // update target values

    PID\_Update(throttle, pitch, roll, yaw);

  }

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  ---------------------- part #3: get IMU data ----------------------

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  IMU\_GetAngles();      // get pitch, roll, yaw

  IMU\_BARO\_ReadBaro();  // get altitude

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  ----------- part #4: data transmission to groundstation -----------

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  static int8\_t dataTransmitDelay = 0;

  // insert time delay between packets

  if(dataTransmitDelay++ >= 60)

  {

    static int8\_t packetSelect = 0;

    // check what packet to send

    if(packetSelect == 0)

      DATA\_TRANSMISSION\_1(ds2438\_Voltage, baroAltitude, 0x00);

    else

      DATA\_TRANSMISSION\_2(angle.pitch, angle.roll, angle.yaw);

    packetSelect = packetSelect == 0;

    dataTransmitDelay = 0;

  }

}

**Teil #1 – receiver input, Receiver\_Decode():**

Zu Beginn werden die rohen empfangenen Daten (Joystick- und Schalterwerte) mit der Funktion *Receiver\_Decode()* in Kanalwerte dekodiert und in den globalen Array *receiver\_ChData[]* gespeichert.

Dateiname: receiver.c

/\*\*

 \* @brief This function decodes the receiver raw data depending on the protocol

 \* @details

 \* i.bus channel values from 1070 - 1920

 \* s.bus channel values from 350 - 1680

 \*

 \* what channel does what depends on the defines found in receiver.h:

 \*  - RECEIVER\_YAW\_CHANNEL

 \*  - RECEIVER\_PITCH\_CHANNEL

 \*  - RECEIVER\_THROTTLE\_CHANNEL

 \*  - RECEIVER\_ROLL\_CHANNEL

 \*  - RECEIVER\_ONOFF\_SWITCH\_CHANNEL

 \*  - RECEIVER\_MODESEL\_SWTICH\_CHANNEL

 \* @return Receiver\_Status

 \*/

Receiver\_Status Receiver\_Decode(void)

{

  switch(receiver\_SelectedProtocol)

  {

    case IBUS:

    {

    // if reception input start at the last byte -> reorder for correct order (rotate left)

      if(receiver\_RawData[1] == 0x20 && receiver\_RawData[2] == 0x40)

      {

        uint8\_t tmp = receiver\_RawData[0];

        for(int8\_t i = 1; i < 32; i++)

          receiver\_RawData[i - 1] = receiver\_RawData[i];

        receiver\_RawData[31] = tmp;

      }

      // check if protocol header is correct

      if(receiver\_RawData[0] != 0x20 || receiver\_RawData[1] != 0x40)

        return IBUS\_HEADER\_ERROR;

    // check if checksum is correct (0xFFFF - sum of other 30 bytes = checksum)

      uint16\_t sum = 0;

      for(int8\_t i = 0; i < 30; i++)

        sum += receiver\_RawData[i];

      uint16\_t checksum = (receiver\_RawData[31] << 8) | receiver\_RawData[30];

      if((0xFFFF - sum) != checksum)

        return IBUS\_CHECKSUM\_ERROR;

    // decode channel data (14 channels, 2 bytes each, little endian byte order)

      for(int8\_t i = 0, j = 0; i < 14; i++, j += 2)

        receiver\_ChData[i] = (receiver\_RawData[j + 3] << 8) | receiver\_RawData[j + 2];

    // check disconnection

      Receiver\_IBusFailsafeCheck();

      if(receiver\_SameDataCounter > 250)

        return IBUS\_SIGNAL\_LOST\_ERROR;

      break;

    }

    case SBUS:

    {

    // if reception input start at the last byte -> reorder for correct order (rotate left)

      if(receiver\_RawData[0] == 0x00 && receiver\_RawData[1] == 0x0F)

      {

        uint8\_t tmp = receiver\_RawData[0];

        for(int8\_t i = 1; i < 25; i++)

          receiver\_RawData[i - 1] = receiver\_RawData[i];

        receiver\_RawData[24] = tmp;

      }

      // check if protocol header is correct

      if(receiver\_RawData[0] != 0x0F)

        return SBUS\_HEADER\_ERROR;

    // check if protocol footer is correct

      if(receiver\_RawData[24] != 0x00)

        return SBUS\_FOOTER\_ERROR;

    // check signal lost flags

      if(receiver\_RawData[23] & 0x04)

        return SBUS\_SIGNAL\_LOST;

    // check signal failsafe flag

      if(receiver\_RawData[23] & 0x08)

        return SBUS\_SIGNAL\_FAILSAFE;

    // decode channel data (16 channels, 11 bits each, lsb first)

      for(int8\_t i = 0, j = 0; i < 16; i += 8, j += 11)

      {

        receiver\_ChData[i + 0] = ((receiver\_RawData[j + 1] >> 0) | (receiver\_RawData[j + 2] << 8)) & 0x7FF;

        receiver\_ChData[i + 1] = ((receiver\_RawData[j + 2] >> 3) | (receiver\_RawData[j + 3] << 5)) & 0x7FF;

        receiver\_ChData[i + 2] = ((receiver\_RawData[j + 3] >> 6) | (receiver\_RawData[j + 4] << 2) | (receiver\_RawData[5] << 10)) & 0x7FF;

        receiver\_ChData[i + 3] = ((receiver\_RawData[j + 5] >> 1) | (receiver\_RawData[j + 6] << 7)) & 0x7FF;

        receiver\_ChData[i + 4] = ((receiver\_RawData[j + 6] >> 4) | (receiver\_RawData[j + 7] << 4)) & 0x7FF;

        receiver\_ChData[i + 5] = ((receiver\_RawData[j + 7] >> 7) | (receiver\_RawData[j + 8] << 1) | (receiver\_RawData[9] << 9)) & 0x7FF;

        receiver\_ChData[i + 6] = ((receiver\_RawData[j + 9] >> 2) | (receiver\_RawData[j + 10] << 6)) & 0x7FF;

        receiver\_ChData[i + 7] = ((receiver\_RawData[j + 10] >> 5) | (receiver\_RawData[j + 11] << 3)) & 0x7FF;

      }

      break;

    }

    // wrong or no protocol selected

    case NO\_PROTO:

    default:

      return PROTOCOL\_ERROR;

      break;

  }

  return RECEIVER\_OK;

}

Diese Funktion wandelt die empfangenen Daten in S.Bus/I.Bus Kanalwerte um.

Es wird kontrolliert, ob keine Protokollfehler, wie zum Beispiel Header- oder Footer-Error, vorkommen. Im Fall eines Fehlers wird der dazugehörige Fehlercode von der Funktion zurückgegeben – *siehe:* [*Kapitel 5.1.1.1*](#_Übersicht_Fehlercodes_1)

Da das I.Bus-Protokoll keine Failsafe Flags besitzt, wird der Verbindungsstatus mit der Funktion *Receiver\_IBusFailsafeCheck()* überprüft.

Dateiname: receiver.c

/\*\*

 \* @brief This function saves the current channel data and check if its the same as before

 \* @param huart pointer to UART\_HandleTypeDef

 \* @retval None

 \*/

void Receiver\_IBusFailsafeCheck(void)

{

  // only used for IBUS because SBUS does have a signal lost / failsafe flag

  if(receiver\_SelectedProtocol != IBUS)

    return;

  static uint16\_t receiver\_OldChData[16] = {0};   // previous channel data

  // when off -> don't check

  if(receiver\_ChData[RECEIVER\_ONOFF\_SWITCH\_CHANNEL] < receiver\_InputLimits.half)

  {

    receiver\_SameDataCounter = 0; // reset counter

  }

  else

  {

    int8\_t same = 1; // data is same flag

    for(int8\_t i = 0; i < 14 && same == 1; i++)

    {

      // check if the old channel data is not the same as the current

      if(receiver\_OldChData[i] != receiver\_ChData[i])

      {

        receiver\_SameDataCounter = 0; // reset channel data check

        same = 0;

      }

      // increment receiver\_SameDataCounter when data is the same

      else if(i == 14 - 1)

        (receiver\_SameDataCounter == UINT16\_MAX - 10) ? receiver\_SameDataCounter = 260 : receiver\_SameDataCounter++;

    }

  }

  // save current channel data

  for(int8\_t i = 0; i < 14; i++)

    receiver\_OldChData[i] = receiver\_ChData[i];

}

Im Falle des Verbindungsverlust sendet der Receiver durchgehend die exakt selben Kanalwerte. Daher zählt diese Funktion, wie oft das exakt selbe Paket mit denselben Werten gesendet worden ist. Dafür wird der Zähler *receiver\_SameDataCounter* verwendet. Wenn der Wert über 250 ist, liefert das Dekodierungsprogramm *Receiver\_Decode()* einen Failsafe-Error.

**Teil #2 – motor control, Receiver\_ConvertInput():**

Die dekodierten Joystickpositionen (Kanalwerte) werden dann mit der Funktion *Receiver\_ConvertInput()* in Prozentwerte für Throttle und in Gradwerte für Pitch, Roll und Yaw umgewandelt. Diese sind in weiteren Schritte für den PID-Regler wichtig.

Dateiname: receiver.c

/\*\*

 \* @brief This function converts the input from the receiver to thorttle and angles

 \* @details

 \* The max throttle values per mode can be changed in receiver.h with:

 \*  - ESC\_SAFEMODE\_THR\_MAX

 \*  - ESC\_NORMALMODE\_THR\_MAX

 \*  - ESC\_OFFMODE\_THR

 \*  - ESC\_TURN\_OFFSET\_MAX

 \* @param throttle percent of throttle speed

 \* @param pitch stick position in degrees

 \* @param roll stick position in degrees

 \* @param yaw stick position in degrees

 \* @return None

 \*/

void Receiver\_ConvertInput(float throttle, float pitch, float roll, float yaw)

{

// turn red LED off

  \_\_HAL\_TIM\_SET\_COMPARE(LED\_TIM, LED\_RED\_CHANNEL, 0);

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  ---------------------- check ON / OFF switch ----------------------

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  static uint8\_t droneOffModeFlag = 1;

  // top position (< half) = turn drone off

  if(receiver\_ChData[RECEIVER\_ONOFF\_SWITCH\_CHANNEL] < receiver\_InputLimits.half)

  {

    droneOffModeFlag = 1; // set off flag

    throttle = 0;     // turn motors off

  }

  // check if throttle stick in the lowest postiion after turning on/off switch to on

  else if(droneOffModeFlag == 1 && receiver\_ChData[RECEIVER\_THROTTLE\_CHANNEL] > receiver\_InputLimits.min + 10)

  {

    throttle = 0; // turn motors off

  }

  // normal control

  else

  {

    droneOffModeFlag = 0;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    -------------- check 3 position switch (mode select) --------------

    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    uint16\_t esc\_MaxThr = ESC\_SAFEMODE\_THR\_MAX;

    uint8\_t hoverModeFlag = 0;

    // top position (< half) = safemode

    if(receiver\_ChData[RECEIVER\_MODESEL\_SWTICH\_CHANNEL] < receiver\_InputLimits.half - 10)

      esc\_MaxThr = ESC\_SAFEMODE\_THR\_MAX;    // set max throttle value

    // middle position (half +- 10) = normalmode

    else if(receiver\_ChData[RECEIVER\_MODESEL\_SWTICH\_CHANNEL] >= receiver\_InputLimits.half - 10 && receiver\_ChData[RECEIVER\_MODESEL\_SWTICH\_CHANNEL] <= receiver\_InputLimits.half + 10)

      esc\_MaxThr = ESC\_NORMALMODE\_THR\_MAX;  // set max throttle value

    // down position = extra mode hover mode

    else

      hoverModeFlag = 1;

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    --------------- calculate throttle input (up / down) --------------

    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    // get joystick position

    throttle = (float)(receiver\_ChData[RECEIVER\_THROTTLE\_CHANNEL] - receiver\_InputLimits.min) / receiver\_InputLimits.delta;

    throttle \*= esc\_MaxThr; // get real thorttle value

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    ----------- calculate pitch input (forwards / backwards) ----------

    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    // get joystick position

    pitch = (float)(receiver\_ChData[RECEIVER\_PITCH\_CHANNEL] - receiver\_InputLimits.min) / receiver\_InputLimits.delta;

    pitch = (pitch - 0.5f) \* 2; // check forwards or backwards position

    pitch \*= ESC\_TURN\_OFFSET\_MAX;  // get stick position in degrees

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    --------------- calculate roll input (left / right) ---------------

    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    // get joystick position

    roll = (float)(receiver\_ChData[RECEIVER\_ROLL\_CHANNEL] - receiver\_InputLimits.min) / receiver\_InputLimits.delta;

    roll = (roll - 0.5f) \* 2; // check left or right position

    roll \*= ESC\_TURN\_OFFSET\_MAX; // get stick position in degrees

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    --------- calculate yaw input (rotate left / rotate right) --------

    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    // get joystick position

    yaw = (float)(receiver\_ChData[RECEIVER\_YAW\_CHANNEL] - receiver\_InputLimits.min) / receiver\_InputLimits.delta;

    yaw = (yaw - 0.5f) \* 2; // check left or right position

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    -------------------------- check values ---------------------------

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    // check if hovermode

    if(hoverModeFlag == 1)

    {

      pitch = 0;

      roll = 0;

      yaw = 0;

    }

  }

}

Nach dem Konvertieren werden die Receiver-Daten (Throttle, Pitch, Roll, Yaw) als Sollwerte zum PID-Regler mit der Funktion *PID\_Update()* gesendet *(siehe:* [*Kapitel 5.8.2*](#_PID-Algorithmus)*)*.

**Teil #3 – get IMU data, IMU\_GetAngles() + IMU\_BARO\_GetBaro():**

Für die Erklärung und Funktionsweise der Befehle

– *siehe: Kapitel* [*5.5.3.4*](#_Berechnen_der_Lagewinkel)*,* [*5.5.4.3*](#_Einlesen_der_Barometer-Daten)*,* [*5.5.4.4*](#_Berechnung_der_Höhe)

**Teil #4 – data transmission to groundstation, DATA\_TRANSMISSION\_1/2():**

Für die Erklärung und Funktionsweise der Befehle

– *siehe: Kapitel Ben Data Transmission*

## Inertial Measurement Unit (IMU)

Inertial Measurement Unit (IMU) ist ein Sammelbegriff für alle Sensoren, die die Kräfte auf einen Körper messen. Typischerweise besteht ein IMU aus einem Gyroskop und einen Beschleunigungsmesser (Accelerometer). In komplexeren Systemen sind weitere Sensoren, wie ein Kompass (Magnetometer) und ein Luftdrucksensor (Barometer) zusätzlich vorhanden.

Der Sensor wird meistens mit dem Begriff „Degrees Of Freedom / DOF“ beschrieben. Dieser Parameter gibt an, wie viele Achsen der Sensor messen kann.

Das heißt, dass zum Beispiel ein 6DOF-IMU, der aus einem Gyroskop und einen Accelerometer besteht, um sechs Achsen messen kann – pro Sensor die x-, y- und z-Achse.

Für die FPV-Drohne wird ein 10DOF-IMU mit einem Gyroskop, Accelerometer, Barometer und Magnetometer verwendet, die auf dem IMU-Breakout vorhanden sind. Mit diesen Sensoren werden die Lagewinkel, die Flughöhe, Temperatur und Regelgrößen für PID-Regler gemessen.

### I²C Protokoll

Das IMU-Breakout auf der FPV-Drohne kann nur mit dem I²C-Protokoll angesprochen werden. Dieses Protokoll besteht aus drei Leitungen: SDA (Datenleitung), SCL (Taktleitung) und GND (Masseleitung) und basiert auf dem Master-Slave-Prinzip.

Um die Datenübertragung zu starten, muss eine Startbedingung erfüllt werden: während die SCL-Leitung im high-Ruhezustand ist, wechselt die SDA-Leitung auf einen low-Zustand.

Nach der Startbedingung schickt der Master im System (Flight Controller) ein 400kHz Rechtecksignal mit einen Duty Cycle von 50% auf die SCL-Leitung. Zuerst wird auf der SDA-Leitung wird die I²C-Addresse des I²C-Slaves geschickt. Danach werden gleichzeitig die Daten gesendet beziehungsweise empfangen und bei einer steigenden Flanke der SCL-Leitung interpretiert. Nach jedem Datenbyte schickt der Empfänger der Daten ein Acknowledge-Bit (ACK / 0), um den Empfang der Daten zu bestätigen. Wenn das Acknowledge-Bit einen high-Pegel hat (not Acknowledge, NACK), trat bei der Übertragung ein Fehler auf.

Am Ende jeder Übertragung wird ein not-Acknowledge-Bit (NACK / 1) empfangen. Darauf folgt eine Stoppbedingung, die eine zeitlich invertierte Startbedingung ist. Während die SCL-Leitung sich in den high-Ruhezustand befindet, wechselt die SDA-Leitung von einen low-Zustand auch in den high-Ruhezustand.



Abbildung 26: I²C Datentransfer

**I²C Einstellung in STM32CubeMX:**

Ein Bild, das Text, Screenshot, Software, Webseite enthält.

Automatisch generierte Beschreibung

Abbildung 27: STM32CubeMX Einstellungen IMU

Um die schnellste Kommunikation mit dem IMU-Breakout festzulegen, muss die Einstellung *I2C Speed Mode* auf *Fast Mode* und *I2C Speed Frequency (KHz)* auf *400* gesetzt werden.

Um zwischen mehreren I²C Geräten zu unterscheiden, hat jedes Gerät eine eigene I²C-Adresse. Die Adresse besteht aus einer 7-Bit-Zahl. Während der Kommunikation muss nach der Sensoradresse ein achtes Bit (R/W Bit) zur Adresse hinzugefügt werden, welches bei Schreibzyklen auf 0 gesetzt und bei Lesezyklen auf 1 gesetzt werden muss.

|  |  |  |
| --- | --- | --- |
| **Sensor** | **Slave Adressen + R/W Bit** | |
| **Schreiben** | **Lesen** |
| MPU9250 | 0xD0 | 0xD1 |
| BMP280 | 0xEE | 0xEF |
| AK8963 | 0x18 | 0x19 |

Damit mit den Sensoren des IMUs kommuniziert werden kann, müssen bestimmte Lese- und Schreibzyklen eingehalten werden:

#### Schreibzyklus IMU

Ein Bild, das Text, Screenshot, Reihe, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 28: I²C Schreibzyklus

|  |  |
| --- | --- |
| **Signal** | **Beschreibung** |
| S | Startbedingung |
| AD+W | Slave-Adresse + Write Bit |
| ACK | Acknowledge Bit |
| RA | Register Adresse |
| P | Stoppbedingung |

Es können mehrere Byte direkt hintereinander geschickt werden. Dabei können beliebig viele Datenbyte mit Acknowledge-Bit vor der Stoppbedingung gesendet werden. Ein interner Zeiger erhöht die Register Adresse um eine Stelle pro Datenbyte.

**Schreiben zu IMU-Register**:

Dateiname: IMU\_10DOF.c

/\*\*

 \* @brief This function writes an amount of bytes to registers from the IMU

 \* @param sensor MPU9250, AK8963 (MAG), BMP280 (BARO)

 \* @param regAddr register address

 \* @param data data to write

 \* @return IMU\_Status

 \*/

IMU\_Status IMU\_WriteRegister(IMU\_Sensor sensor, uint8\_t regAddr, uint8\_t data)

{

  // determine the I2C device address

  uint16\_t devAddress;

  switch(sensor)

  {

    case MPU9250:

      devAddress = IMU\_MPU\_I2C\_ADDR;

      break;

    case AK8963:

      devAddress = IMU\_MAG\_I2C\_ADDR;

      break;

    case BMP280:

      devAddress = IMU\_BARO\_I2C\_ADDR;

      break;

    default:

      return IMU\_ADDRESS\_ERROR;

  }

  // read register(s)

  HAL\_I2C\_Mem\_Write(imu\_ComI2C, devAddress, regAddr, I2C\_MEMADD\_SIZE\_8BIT, &data, 1, 1000);

  return IMU\_OK;

}

Die Funktion *HAL\_I2C\_Mem\_Write()* sendet automatisch die angegeben Daten und Befehle an den I²C-Peripheriebaustein, die für die Kommunikation notwendig sind. Wenn nach einer Sekunde die Kommunikation noch nicht abgeschlossen ist, liefert die Funktion einen Timeout-Error.

#### Lesezyklus IMU

Ein Bild, das Text, Reihe, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 29: I²C Lesezyklus

|  |  |
| --- | --- |
| **Signal** | **Beschreibung** |
| S | Startbedingung |
| AD+W | Slave-Adresse + Write Bit |
| ACK | Acknowledge Bit |
| RA | Register Adresse |
| AD+R | Slave-Adresse + Read Bit |
| NACK | not Acknowledge Bit |
| P | Stoppbedingung |

Wie beim Schreibablauf können auch mehrere Byte direkt hintereinander eingelesen werden. Dabei wird die Kommunikation mit NACK vom Master im System beendet. Ein interner Zeiger erhöht die Register Adresse um eine Stelle pro Datenbyte.

**Lesen von IMU-Register/n:**

Dateiname: IMU\_10DOF.c

/\*\*

 \* @brief This function reads an amount of bytes from registers from the IMU

 \* @param sensor MPU9250, AK8963 (MAG), BMP280 (BARO)

 \* @param regAddr register address

 \* @param data data pointer

 \* @param rxBytes amount of bytes to read

 \* @return IMU\_Status

 \*/

IMU\_Status IMU\_ReadRegister(IMU\_Sensor sensor, uint8\_t regAddr, uint8\_t \*data, uint8\_t rxBytes)

{

  // determine the I2C device address

  uint16\_t devAddress;

  switch(sensor)

  {

    case MPU9250:

      devAddress = IMU\_MPU\_I2C\_ADDR;

      break;

    case AK8963:

      devAddress = IMU\_MAG\_I2C\_ADDR;

      break;

    case BMP280:

      devAddress = IMU\_BARO\_I2C\_ADDR;

      break;

    default:

      return IMU\_ADDRESS\_ERROR;

  }

  // read register(s)

  HAL\_I2C\_Mem\_Read(imu\_ComI2C, devAddress, regAddr, I2C\_MEMADD\_SIZE\_8BIT, data, rxBytes, 1000);

  return IMU\_OK;

}

Die Funktion *HAL\_I2C\_Mem\_Read()* sendet automatisch die Befehle an den I²C-Peripheriebaustein, die für die Kommunikation notwendig sind, und speichert die Daten in der angegeben Variable. Wenn nach einer Sekunde die Kommunikation noch nicht abgeschlossen ist, liefert die Funktion einen Timeout-Fehler.

### IMU-Verbindungstest

Alle drei Sensoren haben ein Who-Am-I-Register. Diese Register haben festgelegte Werte, die nicht geändert werden können.

Es werden alle Sensoren überprüft, obwohl der Magnetometer nicht verwendet wird, damit eine vollständige Verbindungsüberprüfung mit dem gesamten IMU-Breakout durchgeführt werden kann.

|  |  |  |
| --- | --- | --- |
| **Sensor** | **Who Am I Register Adresse** | **Registerinhalt** |
| MPU9250 | 0x75 | 0x71 |
| BMP280 | 0xD0 | 0x58 |
| AK8963 | 0x00 | 0x48 |

Wenn in den angegeben Registern nicht die angegebenen Werte stehen besteht ein Problem bei dem Datenaustausch mit dem I²C-Bus.

**Verbindungstest aller IMU-Sensoren:**

Dateiname: IMU\_10DOF.c

/\*\*

 \* @brief This function checks the connection of all sensors on the IMU

 \* @attention This function enables the bypass mode in the MPU9250

 \* @return IMU\_Status

 \*/

IMU\_Status IMU\_CheckConnection(void)

{

  uint8\_t regVal[3] = {0x71, 0x48, 0x58};

  uint8\_t regAddr[3] = {IMU\_MPU\_WHOAMI\_ADDR, IMU\_MAG\_WHOAMI\_ADDR, IMU\_BARO\_CHIPID\_ADDR};

  uint8\_t sensor[3] = {MPU9250, MAG, BARO};

  uint8\_t timeout;

  uint8\_t data = 0x00;

  for(uint8\_t i = 0; i < 3; i++)

  {

    timeout = 0;

    while(data != regVal[i])

    {

      if(IMU\_ReadRegister(sensor[i], regAddr[i], &data, 1) != IMU\_OK)

        return IMU\_ADDRESS\_ERROR;

      if(timeout++ > 100)

        return IMU\_MPU\_WHOAMI\_ERROR + i;

    }

    if(sensor[i] == MPU9250)

    {

      // enable bypass mode

      if(IMU\_WriteRegister(MPU9250, IMU\_MPU\_INT\_PIN\_CFG\_ADDR, 0x02) != IMU\_OK)

        return IMU\_ADDRESS\_ERROR;

      IMU\_DelayUs(10000);

    }

  }

  return IMU\_OK;

}

Damit auf das Magnetometer zugegriffen werden kann, muss im MPU9250 im bypass enable Register das BYPASS\_EN – Bit gesetzt werden.

Dadurch werden die I²C-Leitung durch den Sensor durchgeführt, und es kann direkt auf das Magnetometer zugegriffen werden.

Wenn das Bit nicht gesetzt wird, müssen im MPU9250 komplexe Einstellungen gesetzt werden, damit er als I²C-Master zu externen Sensoren fungieren kann.

Wenn nach 100 Lesezyklen der Registerwert immer noch falsch ist, liefert die Funktion einen *IMU\_<Sensor>\_WHOAMI\_ERROR*.

### Bestimmen der Lagewinkel - MPU9250

#### Registerübersicht MPU9250

In der folgenden Abbildung wird die Registerübersicht, der verwendeten Registern des MPU9250 dargestellt:

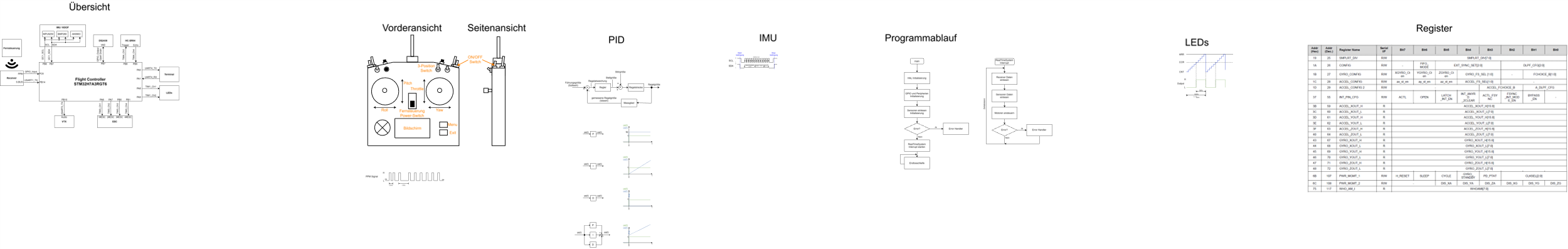


Abbildung 30: Registerübersicht MPU9250

|  |  |
| --- | --- |
| **Register** | **Verwendungszweck** |
| SMPLRT\_DIV | dividiert die interne Abtastrate |
| CONFIG | digitaler Tiefpassfilter für Gyroskop |
| GYRO\_CONFIG | Messbereich für Gyroskop |
| ACCEL\_CONFIG | Messbereich für Accelerometer |
| ACCEL\_CONFIG 2 | digitaler Tiefpassfilter für Accelerometer |
| INT\_PIN\_CFG | direkter I²C-Zugriff auf Magnetometer |
| ACCEL\_<Achse>OUT\_H/L | Messwerte von Accelerometer  <Achse>: X/Y/Z - Messachse  H/L: hoch- oder niederwertiges Byte |
| GYRO\_<Achse>OUT\_H/L | Messwerte von Gyroskop  <Achse>: X/Y/Z - Messachse  H/L: hoch- oder niederwertiges Byte |
| PWR\_MGMT\_1 | Sensor reset und Taktquelle |
| PWR\_MGMT\_2 | Sensoren aktivieren |
| WHO\_AM\_I | Überprüfung der Verbindung |

#### Initialisierung Accelerometer und Gyroskop

Dateiname: IMU\_10DOF.c | Ausschnitt aus IMU\_Init() Funktion

/\*\*

 \* @brief This function initialzes the 10DOF IMU (accel, gyro, mag, baro)

 \* @param imuInit pointer to IMU\_InitTypeDef

 \* @return IMU\_Status

 \*/

IMU\_Status IMU\_Init(IMU\_InitTypeDef \*imuInit)

{

...

// reset MPU

  IMU\_WriteRegister(MPU9250, IMU\_MPU\_PWR\_MGMT\_1\_ADDR, 0x00);

  IMU\_DelayUs(10000);

  // auto select best clk source

  IMU\_WriteRegister(MPU9250, IMU\_MPU\_PWR\_MGMT\_1\_ADDR, 0x01);

  // enable gyro and accel

  IMU\_WriteRegister(MPU9250, IMU\_MPU\_PWR\_MGMT\_2\_ADDR, 0x00);

  // select full scale range for gyro and accel

  IMU\_WriteRegister(MPU9250, IMU\_MPU\_ACCEL\_CONFIG\_ADDR, imuInit->accelFS << 3);

  IMU\_WriteRegister(MPU9250, IMU\_MPU\_GYRO\_CONFIG\_ADDR, imuInit->gyroFS << 3);

  // select digital low pass filter for gyro and accel

  IMU\_WriteRegister(MPU9250, IMU\_MPU\_ACCEL\_CONFIG\_2\_ADDR, imuInit->accelDLPF);

  IMU\_WriteRegister(MPU9250, IMU\_MPU\_CONFIG\_ADDR, imuInit->gyroDLPF);

  // select fastest sample rate

  IMU\_WriteRegister(MPU9250, IMU\_MPU\_SMPLRT\_DIV\_ADDR, 0x00);

  // calculate sensitivity scale factor (LSB/g and LSB/(°/s))

  accelSens = IMU\_ACCEL\_RES\_MAX / (1 << imuInit->accelFS);

  gyroSens = IMU\_GYRO\_RES\_MAX / (1 << imuInit->gyroFS);

  // calibrate gyro

  uint16\_t amount = 2000;

  IMU\_RegCoordinates tempGyro = {0};

  int32\_t tempX = 0, tempY = 0, tempZ = 0;

  for(uint16\_t i = 0; i < amount; i++)

  {

    tempGyro = IMU\_MPU\_ReadGyro();

    tempX += tempGyro.x;

    tempY += tempGyro.y;

    tempZ += tempGyro.z;

    IMU\_DelayUs(3000);

  }

  gyroOffset.x = (float)tempX / (float)amount;

  gyroOffset.y = (float)tempY / (float)amount;

  gyroOffset.z = (float)tempZ / (float)amount;

...

  return IMU\_OK;

}

Die Messwerte eines Gyroskops driften bei Bewegung in eine Richtung *(siehe: Kapitel Gyroskop)*. Um dem dagegen zu wirken, wird ein Gyroskop Offset bestimmt. Es wird nach 2000 Messung der durchschnittliche Messwert bestimmt, der als Offsetwert verwendet wird. Im Programm wird das Ergebnis in der globalen Variable *gyroOffset* gespeichert.

#### Einlesen der Accelerometer- und Gyroskop-Daten

Die Messdaten des Accelerometers werden in den Registern 59 ACCEL\_XOUT\_H bis zum Register 64 ACCEL\_ZOUT\_L gespeichert. XOUT, YOUT und ZOUT bestimmt die einzelne Messachse und \_H und \_L bestimmt das high- und low-Byte des Messwertes.

**Auslesen der Accelerometer-Daten**:

Dateiname: IMU\_10DOF.c

/\*\*

 \* @brief This function reads accelerometer register data (x,y,z)

 \* @return IMU\_RegCoordinates

 \*/

IMU\_RegCoordinates IMU\_MPU\_ReadAccel(void)

{

  uint8\_t buffer[6] = {0};

  IMU\_ReadRegister(MPU9250, IMU\_MPU\_ACCEL\_XOUT\_H\_ADDR, buffer, 6);

  IMU\_RegCoordinates accelData = {0};

  accelData.x = ((int16\_t)buffer[0] << 8) | buffer[1];

  accelData.y = ((int16\_t)buffer[2] << 8) | buffer[3];

  accelData.z = ((int16\_t)buffer[4] << 8) | buffer[5];

  return accelData;

}

In dem Programm werden die Daten beginnend mit dem Register 59 nacheinander bis zum Register 64 eingelesen und danach zusammengefügt.

Die Messung des Gyroskops funktioniert gleich wie beim Accelerometer. Die Messdaten werden in den Registern 67 GYRO\_XOUT\_H bis zum Register 72 GYRO\_ZOUT\_L gespeichert. XOUT, YOUT und ZOUT bestimmen die einzelne Messachse und \_H und \_L bestimmen das high- und low-Byte des Messwertes.

**Auslesen der Gyroskop-Daten**:

Dateiname: IMU\_10DOF.c

/\*\*

 \* @brief This function reads gyroscope register data (x,y,z)

 \* @return IMU\_RegCoordinates

 \*/

IMU\_RegCoordinates IMU\_MPU\_ReadGyro(void)

{

    uint8\_t buffer[6] = {0};

    IMU\_ReadRegister(MPU9250, IMU\_MPU\_GYRO\_XOUT\_H\_ADDR, buffer, 6);

    IMU\_RegCoordinates gyroData = {0};

    gyroData.x = ((int16\_t)buffer[0] << 8) | (int16\_t)buffer[1];

    gyroData.y = ((int16\_t)buffer[2] << 8) | (int16\_t)buffer[3];

    gyroData.z = ((int16\_t)buffer[4] << 8) | (int16\_t)buffer[5];

    return gyroData;

}

Bemerkung:

Bei beide Programmen werden nur die Registerwerte zurückgeliefert. Um die eigentlichen Messwerte zu bekommen, müssen die Werte mit dem sensitivity scale factoren des Accelerometers und des Gyroskops gerechnet werden. Diese Faktoren werden im Initialisierungsprogramm berechnet und in den globalen Variablen *accelSens* und *gyroSens* gespeichert *(siehe:* [*Kapitel 5.5.3.2*](#_Initialisierung_Accelerometer_und)*)*.

Um die Drift vom Gyroskop entgegenzuwirken, wird der Registerwert noch mit dem Gyroskop-Offset gerechnet. Diese Werte werden auch im Initialisierungsprogramm bestimmt und in der globalen Variable *gyroOffset* gespeichert *(siehe:* [*Kapitel 5.5.3.2*](#_Initialisierung_Accelerometer_und)*).*

**Berechnung der eigentlichen Messwerte**:

IMU\_RegCoordinates gyroData = IMU\_MPU\_ReadGyro();

IMU\_RegCoordinates accelData = IMU\_MPU\_ReadAccel();

gyro.x = (gyroData.x - gyroOffset.x) / gyroSens;

gyro.y = (gyroData.y - gyroOffset.y) / gyroSens;

gyro.z = (gyroData.z - gyroOffset.z) / gyroSens;

accel.x = (accelData.x / accelSens) - 0.01f;

accel.y = (accelData.y / accelSens) - 0.02f;

accel.z = (accelData.z / accelSens) - 0.1f;

Die Offsetwerte bei der Beschleunigungsberechnung (0,01; 0,02 und 0,1) müssen händisch bestimmt werden. Wenn der Sensor auf einer ebenen Fläche gerade liegt, muss der Accelerometer eine Beschleunigung von 1g auf der z-Achse messen.

Der Offset ergibt sich aus der eigentlichen Messung auf einer ebenen Fläche von 1,01g. Dieser Vorgang muss mit 90° Drehungen für allen Achsen wiederholt werden, um die restlichen Offsetwerte zu bestimmen.

#### Berechnen der Lagewinkel

Um die Lagewinkel Pitch, Roll und Yaw zu bestimmen, wird ein Komplementärfilter auf die Messwerte (Erd-, Winkelbeschleunigung) des MPU9250 angewendet *(siehe:* [*Kapitel 2.2.5*](#_Komplementärfilter_1)*)*.

**Anwendung des Komplementärfilters:**

Dateiname: IMU\_10DOF.c

/\*\*

 \* @brief This function calculates pitch,roll and yaw

 \* @details data gets stored in the global variable 'angle'

 \* @retval None

 \*/

void IMU\_GetAngles(void)

{

  // read counter value since last function call

  uint16\_t tmpTime = \_\_HAL\_TIM\_GET\_COUNTER(imu\_DelayTIM);

  // reset timer

  \_\_HAL\_TIM\_SET\_COUNTER(imu\_DelayTIM, 0);

  // start first time call

  static int8\_t firstTimeFlag = 0;

  if(firstTimeFlag == 0)

  {

    firstTimeFlag = 1;

    return;

  }

  // calculate delta time

  imu\_DeltaTime = (float)tmpTime / 1E6f;

  // read sensors

  IMU\_RegCoordinates gyroData = IMU\_MPU\_ReadGyro();

  IMU\_RegCoordinates accelData = IMU\_MPU\_ReadAccel();

  // calculate actual values

  gyro.x = (gyroData.x - gyroOffset.x) / gyroSens;

  gyro.y = (gyroData.y - gyroOffset.y) / gyroSens;

  gyro.z = (gyroData.z - gyroOffset.z) / gyroSens;

  accel.x = (accelData.x / accelSens) - 0.01f;

  accel.y = (accelData.y / accelSens) - 0.02f;

  accel.z = (accelData.z / accelSens) - 0.1f;

  // invert axis because the sensor is upside down

  accel.z = -accel.z;

  // apply complementary filter to calc angles

  float accelPitch = atan2(accel.y, accel.z) \* RAD2DEG;

  float accelRoll = atan2(accel.x, accel.z) \* RAD2DEG;

  angle.roll = 0.98f \* (angle.roll + gyro.y \* imu\_DeltaTime) + 0.02f \* accelRoll;

  angle.pitch = 0.98f \* (angle.pitch - gyro.x \* imu\_DeltaTime) + 0.02f \* accelPitch;

  angle.yaw += gyro.z \* imu\_DeltaTime;

}

Die Funktion liest die Registerwerte vom Accelerometer und Gyroskop ein. Danach werden die eigentlichen Messwerte ausgerechnet. Mittels Komplementärfilter werden die Lagewinkel ausgerechnet, die in der Variable *angle* gespeichert werden.

Dadurch, dass der Sensor verkehrt auf der Drohne montiert ist, muss der Messwert um die z-Achse des Accelerometer invertiert werden.

### Bestimmen der Höhe - BMP280

#### Registerübersicht BMP280

In der folgenden Abbildung wird die Registerübersicht, der verwendeten Registern des BMP280 dargestellt:

Ein Bild, das Text, Screenshot, Reihe, Zahl enthält.

Automatisch generierte Beschreibung

Abbildung 31: Registerübersicht BMP280

|  |  |
| --- | --- |
| **Register** | **Verwendungszweck** |
| temp\_xlsb | Temperatur Messwerte |
| temp\_lsb/msb |
| press\_xlsb | Luftdruck Messwerte |
| press\_lsb/msb |
| config | Zeitkonstanten |
| ctrl\_meas | Oversampling- und Modus-Auswahl |
| status | Status überprüfen |
| reset | Sensor zurücksetzen |
| id | Überprüfung der Verbindung |
| calib25…calib00 | Kompensationsparameter auslesen |

#### Initialisierung Barometer

Dateiname: IMU\_10DOF.c | Ausschnitt aus IMU\_Init() Funktion

/\*\*

 \* @brief This function initialzes the 10DOF IMU (accel, gyro, mag, baro)

 \* @param imuInit pointer to IMU\_InitTypeDef

 \* @return IMU\_Status

 \*/

IMU\_Status IMU\_Init(IMU\_InitTypeDef \*imuInit)

{

...

  IMU\_WriteRegister(BMP280, IMU\_BARO\_RESET\_ADDR, 0xB6); // reset barometer

  // check if status bit for device = ready

  uint8\_t timeout = 0, status = 1;

  while(status != 0x00)

  {

    IMU\_ReadRegister(BMP280, IMU\_BARO\_STATUS\_ADDR, &status, 1);

    // check 100 times max

    if(timeout++ > 100)

      return IMU\_BARO\_INIT\_ERROR;

  }

  IMU\_BARO\_ReadCompensationValues();

  // set standby time and time constant of IIR filter

  uint8\_t config = ((imuInit->baroSBT << 5) | (imuInit->baroCoeff << 2));

  IMU\_WriteRegister(BMP280, IMU\_BARO\_CONFIG\_ADDR, config);

  // set oversampling settings for temperature and pressure measurement and set normal mode

  uint8\_t ctrl = (imuInit->baroTempOS << 5) | (imuInit->baroPressOS << 2) | 0x03;

  IMU\_WriteRegister(BMP280, IMU\_BARO\_CTRL\_MEAS\_ADDR, ctrl);

  // get current altitude level

  IMU\_DelayUs(UINT16\_MAX - 1);

  float baroSum = 0;

  for(uint16\_t i = 0; i < amount; i++)

  {

    IMU\_BARO\_ReadBaro();

    baroSum += baroAltitude;

    IMU\_DelayUs(1000);

  }

  // calc offset with the average

  baroAltitudeOffset = baroSum / amount;

  return IMU\_OK;

}

Das Barometer wird immer im Normal-Mode betrieben.

Für die richtige Luftdruck- und Höhenbestimmung müssen die Registerwerte kompensiert werden. Diese Kompensationswerte werden vom Sensor zur Verfügung gestellt und müssen vor Beginn der Werteberechnung eingelesen werden. Das Einlesen wird mit der Funktion *IMU\_Baro\_ReadCompensationValues()* realisiert.

**Kompensationswerte für Barometermessung auslesen:**

Dateiname: IMU\_10DOF.c

/\*\*

 \* @brief This function reads the temperature and pressure compensation values

 \* @details values get stored in variable "baroCompensation"

 \* @retval None

 \*/

void IMU\_BARO\_ReadCompensationValues(void)

{

  uint8\_t buffer[24];

  IMU\_ReadRegister(BMP280, IMU\_BARO\_DIG\_T1\_L\_ADDR, buffer, 24);

  baroCompensation.T1 = (buffer[1] << 8) | buffer[0];

  baroCompensation.T2 = (buffer[3] << 8) | buffer[2];

  baroCompensation.T3 = (buffer[5] << 8) | buffer[4];

  baroCompensation.P1 = (buffer[7] << 8) | buffer[6];

  baroCompensation.P2 = (buffer[9] << 8) | buffer[8];

  baroCompensation.P3 = (buffer[11] << 8) | buffer[10];

  baroCompensation.P4 = (buffer[13] << 8) | buffer[12];

  baroCompensation.P5 = (buffer[15] << 8) | buffer[14];

  baroCompensation.P6 = (buffer[17] << 8) | buffer[16];

  baroCompensation.P7 = (buffer[19] << 8) | buffer[18];

  baroCompensation.P8 = (buffer[21] << 8) | buffer[20];

  baroCompensation.P9 = (buffer[23] << 8) | buffer[22];

}

Die Daten werden vom Hersteller angegeben und sind konstant in den Registern gespeichert. Die zusammengefügten Werte werden in der globalen Variable *baroCompensation* gespeichert.

#### Einlesen der Barometer-Daten

Die Messdaten vom BMP280 werden in den Registern 0xF7 press\_msb bis 0xFC temp\_xlsb gespeichert.

**Einlesen der Barometerwerte:**

Dateiname: IMU\_10DOF.c | Ausschnitt aus IMU\_BARO\_ReadBaro() Funktion

/\*\*

 \* @brief This function reads the barometer values and calculates temperature, pressure and altitude

 \* @details values gets stored in global variables 'baroTemperature', 'baroPressure' and 'baroAltitude'

 \* @retval None

 \*/

void IMU\_BARO\_ReadBaro(void)

{

  uint8\_t buffer[6] = {0};

  IMU\_ReadRegister(BMP280, IMU\_BARO\_PRESS\_ADDR, buffer, 6);

  int32\_t adcPress = ((int32\_t)buffer[0] << 12) | ((int32\_t)buffer[1] << 4) | ((int32\_t)buffer[2] >> 4);

  int32\_t adcTemp = ((int32\_t)buffer[3] << 12) | ((int32\_t)buffer[4] << 4) | ((int32\_t)buffer[5] >> 4);

  int32\_t fineTemp;

  int32\_t temp = IMU\_BARO\_CompensateTemp(adcTemp, &fineTemp);

  uint32\_t press = IMU\_BARO\_CompensatePress(adcPress, fineTemp);

  // convert register values to real temp and pressure values

  baroTemperature = (float)temp / 100.0;

  baroPressure = (float)press / 256.0;

...

}

Für die Berechnung müssen die Temperatur und der Luftdruck kompensiert werden. Dafür werden die Funktionen *IMU\_BARO\_CompensateTemp()* und *IMU\_BARO\_CompensatePress()* verwendet. Diese Funktionen sind im Datenblatt des Sensors vom Hersteller vorgegeben und wurden für die Nutzung in der Steuerungssoftware angepasst und übernommen.

**Barometer Temperatur- und Luftdruckwerte kompensieren:**

Dateiname: IMU\_10DOF.c

/\*\*

 \* @brief This function compensates the temperature according to the datasheet

 \* @details

 \* Returns temperature in DegC, resolution is 0.01 DegC. Output value of “5123” equals 51.23 DegC.

 \* @param adcTemp measured temperature

 \* @param fineTemp

 \* @return int32\_t (temperature)

 \*/

int32\_t IMU\_BARO\_CompensateTemp(int32\_t adcTemp, int32\_t \*fineTemp)

{

  int32\_t var1, var2, T;

  var1 = ((((adcTemp >> 3) - ((int32\_t)baroCompensation.T1 << 1))) \* ((int32\_t)baroCompensation.T2)) >> 11;

  var2 = (((((adcTemp >> 4) - ((int32\_t)baroCompensation.T1)) \* ((adcTemp >> 4) - ((int32\_t)baroCompensation.T1))) >> 12) \* ((int32\_t)baroCompensation.T3)) >> 14;

  \*fineTemp = var1 + var2;

  T = (\*fineTemp \* 5 + 128) >> 8;

  return T;

}

/\*\*

 \* @brief This function compensates the pressure according to the datasheet

 \* @details

 \* Returns pressure in Pa as unsigned 32 bit integer in Q24.8 format (24 integer bits and 8 fractional bits).

 \* Output value of “24674867” represents 24674867/256 = 96386.2 Pa = 963.862 hPa

 \* @param adcPress measured pressure

 \* @param fineTemp

 \* @return uint32\_t (pressure)

 \*/

uint32\_t IMU\_BARO\_CompensatePress(int32\_t adcPress, int32\_t fineTemp)

{

  int64\_t var1, var2, p;

  var1 = ((int64\_t)fineTemp) - 128000;

  var2 = var1 \* var1 \* (int64\_t)baroCompensation.P6;

  var2 = var2 + ((var1 \* (int64\_t)baroCompensation.P5) << 17);

  var2 = var2 + (((int64\_t)baroCompensation.P4) << 35);

  var1 = ((var1 \* var1 \* (int64\_t)baroCompensation.P3) >> 8) + ((var1 \* (int64\_t)baroCompensation.P2) << 12);

  var1 = (((((int64\_t)1) << 47) + var1)) \* ((int64\_t)baroCompensation.P1) >> 33;

  if(var1 == 0)

  {

    return 0; // avoid exception caused by division by zero

  }

  p = 1048576 - adcPress;

  p = (((p << 31) - var2) \* 3125) / var1;

  var1 = (((int64\_t)baroCompensation.P9) \* (p >> 13) \* (p >> 13)) >> 25;

  var2 = (((int64\_t)baroCompensation.P8) \* p) >> 19;

  p = ((p + var1 + var2) >> 8) + (((int64\_t)baroCompensation.P7) << 4);

  return (uint32\_t)p;

}

#### Berechnung der Höhe

Mit dem gemessenen Luftdruck kann die aktuelle Höhe über dem Meeresspiegel ausgerechnet werden. Dafür wird eine Höhenberechnungsformel aus dem Datenblatt übernommen:

Ein Bild, das Text, Schrift, Diagramm, Reihe enthält.

Automatisch generierte Beschreibung

Abbildung 32: Formel Berechnung Höhe über Meeresspiegel

altitude … Höhe über den Meeresspiegel in Meter (m)

p … gemessener Luftdruck in Hektopascal (hPa)

p0 … typischer Luftdruck am Meeresspiegel, normalerweise 1013,25hPa

**Höhe über Meeresspiegel bestimmen:**

IMU\_10DOF.c | Ausschnitt aus IMU\_BARO\_ReadBaro() Funktion

void IMU\_BARO\_ReadBaro(void)

{

...

  // convert pressure to altitude according to datasheet

  float presshPa = baroPressure / 100;

  baroAltitude = (44330.0 \* (1.0 - pow(presshPa / 1013.25, 1.0 / 5.255))) - baroAltitudeOffset;

}

Dadurch, dass in der Berechnung die Variable *baroAltitudeOffset* verwendet wird, berechnet sich die relative Höhe der Drohne zum Boden. Wenn dieser Teil der Berechnung entfernt wird, kann die absolute Höhe bestimmt werden.

## Motorregelalgorithmus

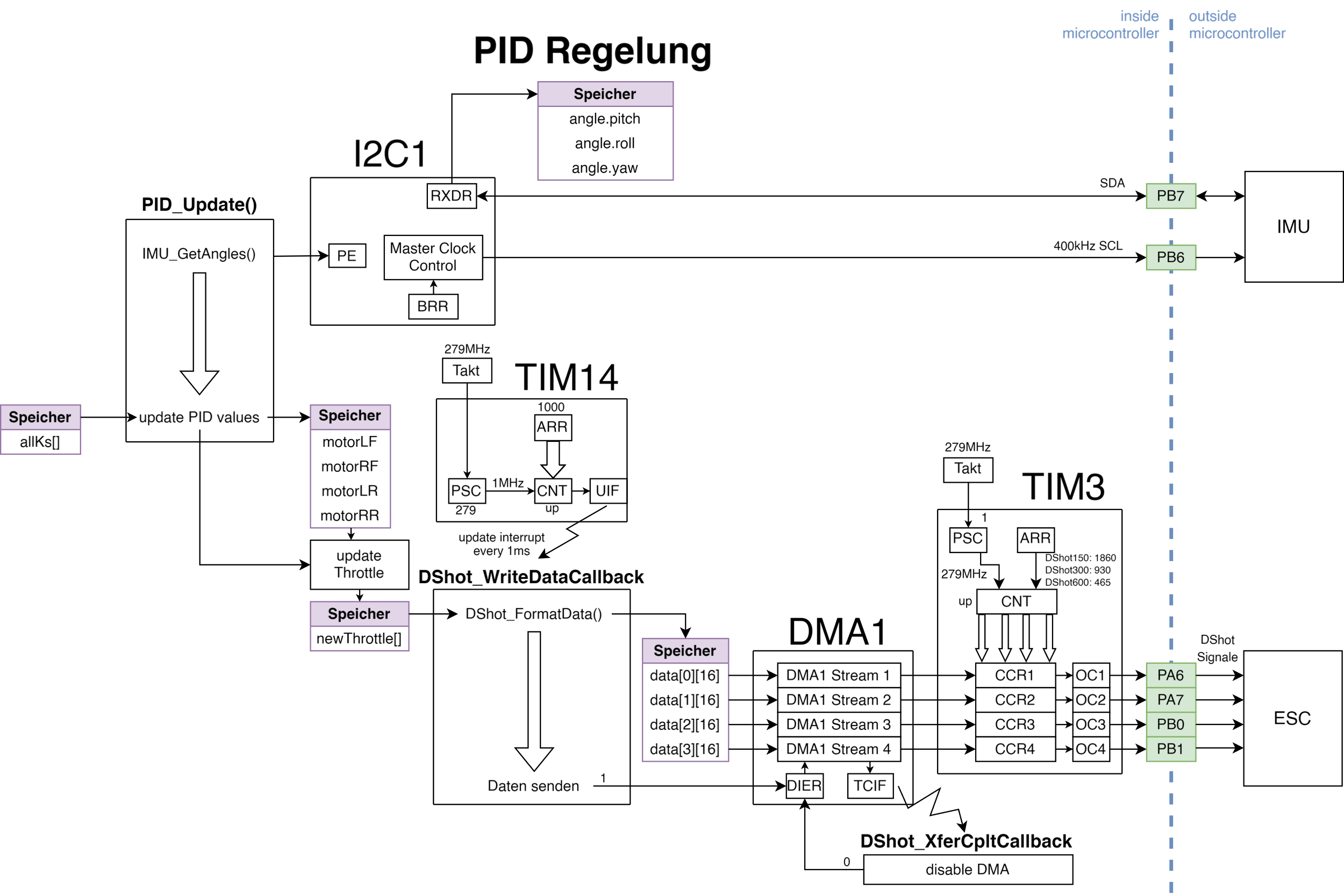


Abbildung 33: Motorregelalgorithmus Architektur

Am Beginn des PID-Regelalgorithmus wird der Regelgröße (Istwert) mit Hilfe des IMU bestimmt. Dafür werden die MPU9250-Daten vom Accelerometer und Gyroskop mit der I2C1-Peripherie eingelesen und diese in Lagewinkel umgewandelt und in der Variable globalen Variable *angle* gespeichert *(siehe: Kapitel* [*5.5.3.3*](#_Einlesen_der_Accelerometer-)*,* [*5.5.3.4*](#_Berechnen_der_Lagewinkel)*)*.

Als nächsten Schritt bestimmt der Regler die neuen Ausgänge, die ein Prozentwert (0-100%) der Geschwindigkeit der Motoren sind. Diese werden dann mit der Funktion *DShot\_SendThrottle()* an den Ausgangsalgorithmus übergeben *(siehe:* [*Kapitel 5.7.3*](#_Motoransteuerung_Software)*)*.

Damit der ESC nicht in einen Standby-Modus übergeht müssen durchgehend Werte geschickt werden. Die TIM14-Peripherie löst alle 1ms ein Update Interrupt *DShot\_WriteDataCallback()* aus. In der ISR wird ein DShot-Paket aus den Throttle-Werten gebildet und dann mit Hilfe des DMA-Controllers zu den PWM-Ausgängen der TIM3-Peripherie geschickt *(siehe:* [*Kapitel 5.7.3*](#_Motoransteuerung_Software)*).*

Damit alle vier Timer-Kanäle gleichzeitig das Ausgangssignal schicken, wird bei der Beendung des Datentransfers ein Interrupt ausgelöst, indem der DMA-Controller deaktiviert wird. Damit jetzt alle vier Kanäle gleichzeitig senden können, wird in *DShot\_WriteDataCallback()* zuerst alle Einstellungen getätigt und danach erst der Transfer vom DMA-Controller gestartet *(siehe:* [*Kapitel 5.7.3*](#_Motoransteuerung_Software)*).*

Dadurch herrscht zu jeder Zeit die kleinstmögliche Zeitdifferenz zwischen den Ausgangskanälen.

## Motoransteuerung

### DShot Protokoll

Zum Ansteuern der Motoren müssen Befehle an dem Electronic Speed Controller (ESC) geschickt werden. Dafür wird das DShot-Protokoll verwendet. Im Vergleich zur analogen PWM-Ansteuerung der ESC, werden die DShot-Signale um ein Vielfaches schneller gesendet und dadurch, dass es sich um ein digitales Protokoll handelt, wird keine Kalibrierungssequenz der Throttle Minimal- und Maximalwerte benötigt.

Das Protokoll besteht aus 16 Bits. Um zwischen 0 und 1 zu unterscheiden, wird der PWM Duty-Cycle des Signals verändert. Ein Duty-Cycle von 75% entspricht einer ‚1‘ und ein Duty-Cycle von 37,5% entspricht einer ‚0‘. Das Protokoll gibt es in mehreren Geschwindigkeiten (DShot150, DShot300, DShot600, DShot1200), wobei die Zahl angibt, in welcher Frequenz (fDShot in Kilohertz) ein Bit gesendet wird. Der/Die BenutzerIn kann sich aussuchen mit welcher Geschwindigkeit die Bits gesendet werden soll – *siehe:* [*Kapitel 5.7.2*](#_Initialisierung_Motoransteuerung)

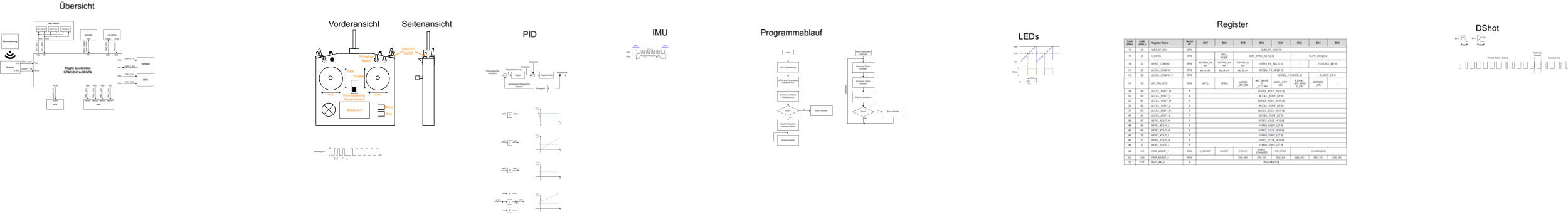


Abbildung 34: DShot Bit 0/1 Duty Cycle + Geschwindigkeit

Die ersten 11 Bits beinhalten den Throttle-Wert in Prozent, wobei 48 = 0% Throttle und 2048 = 100% Throttle. Die Zahlenwerte 0 bis 47 sind für spezielle Befehle reserviert.

Mit dem nächsten Bit kann eine Telemetrie-Anfrage gesendet werden, die bei einer erweiterten Version des Protokolls (bidirektionalen DShot) verwendet werden kann. Die ESC, die für die Diplomarbeit verwendet wird, unterstützt diese Variante nicht.

Die letzten 4 Bits bilden eine Checksumme, die aus der XOR-Verknüpfung der ersten 12 Bits in jeweils Viererblöcken gebildet wird. Die Ruhezeit zwischen Paketen ist im low-Zustand.

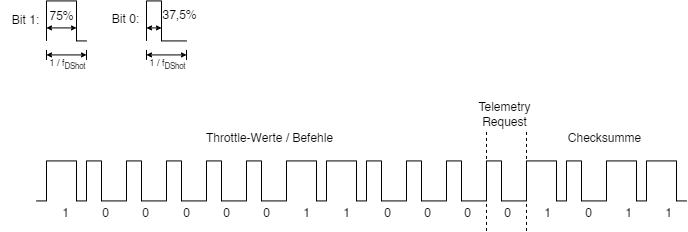


Abbildung 35: DShot Beispiel Übertragung

In der Beispielübertragung wird der Zahlenwert 0b10000011000 (1048) geschickt. Dieser entspricht einen Throttle-Wert von 50%. Die Telemetry Request ist ausgeschaltet, und die Berechnung der Checksumme ergibt dem Wert 0b1011, der am Ende des Pakets ausgegeben wird.

### Initialisierung Motoransteuerung

Dateiname: dshot\_own.c

/\*\*

 \* @brief This function initializes the output ESC DShot signal

 \* @param htim pointer to TIM\_HandleTypeDef (output timer)

 \* @param protocol DSHOT150, DSHOT300, DSHOT600

 \* @param updateTim pointer to TIM\_HandleTypeDef (executes 1ms interrupt)

 \* @return DShot\_Status

 \*/

DShot\_Status DShot\_Init(TIM\_HandleTypeDef \*htim, ESC\_OutputProtocol protocol, TIM\_HandleTypeDef \*updateTim)

{

  // set and check timer pointer

  DShot\_OutputTim = htim;

  if(DShot\_OutputTim == NULL)

    return DSHOT\_TIM\_ERROR;

  // set the right timer frequency, 279MHz / (prescaler \* autoreload)

  \_\_HAL\_TIM\_SET\_PRESCALER(DShot\_OutputTim, 1 - 1);

  \_\_HAL\_TIM\_SET\_AUTORELOAD(DShot\_OutputTim, protocol - 1);

  oneDC = ceil((float)protocol \* 0.75);   // calculate duty cycle for sending bit 1

  zeroDC = ceil((float)protocol \* 0.375); // calculate duty cycle for sending bit 0

  // define custom transfer complete ISR

  DShot\_OutputTim->hdma[ESC\_LF\_DMA\_ID]->XferCpltCallback = DShot\_DMA\_XferCpltCallback;

  DShot\_OutputTim->hdma[ESC\_RF\_DMA\_ID]->XferCpltCallback = DShot\_DMA\_XferCpltCallback;

  DShot\_OutputTim->hdma[ESC\_LR\_DMA\_ID]->XferCpltCallback = DShot\_DMA\_XferCpltCallback;

  DShot\_OutputTim->hdma[ESC\_RR\_DMA\_ID]->XferCpltCallback = DShot\_DMA\_XferCpltCallback;

  // set output low

  \_\_HAL\_TIM\_SET\_COMPARE(DShot\_OutputTim, ESC\_LF\_TIM\_CH, 0);

  \_\_HAL\_TIM\_SET\_COMPARE(DShot\_OutputTim, ESC\_RF\_TIM\_CH, 0);

  \_\_HAL\_TIM\_SET\_COMPARE(DShot\_OutputTim, ESC\_LR\_TIM\_CH, 0);

  \_\_HAL\_TIM\_SET\_COMPARE(DShot\_OutputTim, ESC\_RR\_TIM\_CH, 0);

  // start all timers in pwm output mode

  HAL\_TIM\_PWM\_Start(DShot\_OutputTim, ESC\_LF\_TIM\_CH);

  HAL\_TIM\_PWM\_Start(DShot\_OutputTim, ESC\_RF\_TIM\_CH);

  HAL\_TIM\_PWM\_Start(DShot\_OutputTim, ESC\_LR\_TIM\_CH);

  HAL\_TIM\_PWM\_Start(DShot\_OutputTim, ESC\_RR\_TIM\_CH);

  // set custom ISR for 1ms interrupt + start timer

  HAL\_TIM\_RegisterCallback(updateTim, HAL\_TIM\_PERIOD\_ELAPSED\_CB\_ID, DShot\_WriteDataCallback);

  HAL\_TIM\_Base\_Start\_IT(updateTim);

  // at beginning send 0% throttle to every motor

  if(DShot\_SendThrottle(0, 0, 0, 0) != DSHOT\_OK)

    return DSHOT\_TIM\_ERROR;

  HAL\_Delay(5000);

  return DSHOT\_OK;

}

Das Protokoll wird mit der Timer-Peripherie gesendet, welche ein PWM-Signal ausgibt. Damit das DShot-Protokoll erstellt wird, schickt der DMA-Controller die richtigen Werte für die Änderung des Duty-Cycle der Signale.

### Motoransteuerung Software

Dadurch, dass sich Motoren nicht automatisch ausschalten, wird ein DShot-Packet jede Millisekunde gesendet. Dafür wird der *updateTim* verwendet. Wenn dieser Timer einen Überlauf ergibt, wird ein Interrupt ausgelöst.

Dateiname: dshot.c

/\*\*

 \* @brief This function is the call for a 1ms interrupt, to send every 1ms

 \* @retval None

 \*/

static void DShot\_WriteDataCallback(TIM\_HandleTypeDef \*htim)

{

  static uint16\_t prevThrottle[4] = {1,1,1,1};  // stores previous throttle values

  static uint16\_t data[4][18] = {0};  // stores each bit (duty cycle value) for every motor

  if(newThrottle[0] != prevThrottle[0] || newThrottle[1] != prevThrottle[1] || newThrottle[2] != prevThrottle[2] || newThrottle[3] != prevThrottle[3])

  {

    DShot\_FormatData(newThrottle, 0, data);

    for(int8\_t i = 0; i < 4; i++)

      prevThrottle[i] = newThrottle[i];

  }

  // start dma transfer to the capture compare register

  HAL\_DMA\_Start\_IT(DShot\_OutputTim->hdma[ESC\_LF\_DMA\_ID], (uint32\_t)&data[0][0], ESC\_TIM\_GET\_CCR\_ADDR(ESC\_LF\_TIM\_CH), 18);

  HAL\_DMA\_Start\_IT(DShot\_OutputTim->hdma[ESC\_RF\_DMA\_ID], (uint32\_t)&data[1][0], ESC\_TIM\_GET\_CCR\_ADDR(ESC\_RF\_TIM\_CH), 18);

  HAL\_DMA\_Start\_IT(DShot\_OutputTim->hdma[ESC\_LR\_DMA\_ID], (uint32\_t)&data[2][0], ESC\_TIM\_GET\_CCR\_ADDR(ESC\_LR\_TIM\_CH), 18);

  HAL\_DMA\_Start\_IT(DShot\_OutputTim->hdma[ESC\_RR\_DMA\_ID], (uint32\_t)&data[3][0], ESC\_TIM\_GET\_CCR\_ADDR(ESC\_RR\_TIM\_CH), 18);

  // reset counter to get rid of delay between channels

  \_\_HAL\_TIM\_SET\_COUNTER(DShot\_OutputTim, 0);

  // enable dma / start sending

  \_\_HAL\_TIM\_ENABLE\_DMA(DShot\_OutputTim, TIM\_DMA\_CC1);

  \_\_HAL\_TIM\_ENABLE\_DMA(DShot\_OutputTim, TIM\_DMA\_CC2);

  \_\_HAL\_TIM\_ENABLE\_DMA(DShot\_OutputTim, TIM\_DMA\_CC3);

  \_\_HAL\_TIM\_ENABLE\_DMA(DShot\_OutputTim, TIM\_DMA\_CC4);

}

Diese Funktion bereitet die Daten für die Übertragung vor und startet den DMA-Controller.

Wenn die Datenübertragung des DMA-Controllers abgeschlossen hat, wird ein Interrupt ausgelöst.

Dateiname: dshot\_own.c

/\*\*

 \* @brief This function is the ISR for DMA transmit complete

 \* @param hdma

 \* @retval None

 \*/

static void DShot\_DMA\_XferCpltCallback(DMA\_HandleTypeDef \*hdma)

{

  // disable DMA to get rid of the delay between channels

  if(hdma == DShot\_OutputTim->hdma[TIM\_DMA\_ID\_CC1])

  {

    \_\_HAL\_TIM\_DISABLE\_DMA(DShot\_OutputTim, TIM\_DMA\_CC1);

  }

  else if(hdma == DShot\_OutputTim->hdma[TIM\_DMA\_ID\_CC2])

  {

    \_\_HAL\_TIM\_DISABLE\_DMA(DShot\_OutputTim, TIM\_DMA\_CC2);

  }

  else if(hdma == DShot\_OutputTim->hdma[TIM\_DMA\_ID\_CC3])

  {

    \_\_HAL\_TIM\_DISABLE\_DMA(DShot\_OutputTim, TIM\_DMA\_CC3);

  }

  else if(hdma == DShot\_OutputTim->hdma[TIM\_DMA\_ID\_CC4])

  {

    \_\_HAL\_TIM\_DISABLE\_DMA(DShot\_OutputTim, TIM\_DMA\_CC4);

  }

}

Diese Funktion schaltet den DMA-Controller für alle Kanäle ab. Dies ermöglicht, dass alle Pakete zur selbe Zeit gesendet werden.

Um den Throttlewert der Motoren zu ändern, muss folgende Funktion ausgeführt werden:

Dateiname: dshot\_own.c

/\*\*

 \* @brief This function formats the motor data for the DShot protocol

 \* @param motorLF percent of throttle value of left front motor (0-100)

 \* @param motorRF percent of throttle value of right front motor (0-100)

 \* @param motorLR percent of throttle value of left rear motor (0-100)

 \* @param motorRR percent of throttle value of right rear motor (0-100)

 \* @retval DShot\_Status

 \*/

DShot\_Status DShot\_SendThrottle(double motorLF, double motorRF, double motorLR, double motorRR)

{

  if(DShot\_OutputTim == NULL)

    return DSHOT\_TIM\_ERROR;

  /\*\*

   \* timer channel 1 = left front motor

   \* timer channel 2 = right front motor

   \* timer channel 3 = left rear motor

   \* timer channel 4 = right rear motor

   \*/

   // convert to dshot throttle format (48 = 0% throttle, 2047 = 100% throttle)

  newThrottle[0] = 48 + 20 \* motorLF;

  newThrottle[1] = 48 + 20 \* motorRF;

  newThrottle[2] = 48 + 20 \* motorLR;

  newThrottle[3] = 48 + 20 \* motorRR;

  return DSHOT\_OK;

}

Diese Funktion speichert die neuen Throttlewerte in der Variable *newThrottle*, die in dem *DShot\_WriteDataCallback()* auf Änderung überprüft wird.

Um diese Daten in das richtige Format zu ändern, wird die Funktion *DShot\_FormatData()* ausgeführt.

Dateiname: dshot\_own.c

/\*\*

 \* @brief This function converts the throttle value to duty cycle bits

 \* @param throttle Throttle values (0-2047)

 \* @param telemetry telemetry request bit

 \* @param data formatted data by the function, array of 75% or 37.5%

 \* @retval None

 \*/

void DShot\_FormatData(uint16\_t \*throttle, int8\_t telemetry, uint16\_t data[4][18])

{

  uint16\_t withoutCS, complete, div;

  // format the data to packets

  for(int8\_t i = 0; i < 4; i++)

  {

    // first 12 bits (without Checksum)

    withoutCS = (throttle[i] << 1) | telemetry;

    // format whole data frame

    complete = withoutCS << 4 | ((withoutCS ^ (withoutCS >> 4) ^ (withoutCS >> 8)) & 0x0F);

    // convert each bit to the specific duty cycle length

    div = 0x8000;

    for(int8\_t j = 0; j < 16; j++)

    {

      data[i][j] = (complete & div) ? oneDC : zeroDC;

      div >>= 1;

    }

  }

}

Die Datenpakete werden zusammengestellt und danach in den richtigen Duty-Cycle-Wert umgewandelt.

## PID-Regler

### Initialisierung PID-Regler

Damit die Filterkoeffizienten während der Programmlaufzeit geändert werden können, wird der DMA-Controller verwendet. Dafür wird die UART4-Peripherie, die auch für die Terminal Ausgabe *(siehe:* [*Kapitel 5.10*](#_Terminal_Ausgabe_1)*)* verwendet wird.

Ein Bild, das Text, Screenshot, Software, Zahl enthält.

Automatisch generierte Beschreibung

Abbildung 36: STM32CubeMX Einstellung PID DMA Empfang

Zum Initialisieren wird die Funktion *PID\_Init()* ausgeführt.

Dateiname: PID.c

/\*\*

 \* @brief This function initializes the change Ks system

 \* @param huart pointer to UART\_HandleTypeDef (input uart)

 \* @retval None

 \*/

void PID\_Init(UART\_HandleTypeDef \*huart)

{

    // start reception with uart

    HAL\_UART\_RegisterCallback(huart, HAL\_UART\_RX\_COMPLETE\_CB\_ID, PID\_ChangeKs);

    HAL\_UART\_Receive\_DMA(huart, (uint8\_t \*)&receiveData, 10);

}

Diese Funktion startet den DMA-Controller, der auf 10 Bytes von der UART-Peripherie wartet und in dem Array *receiveData[]* speichert. Dadurch können Werte im Format „0 0.123456“ gesendet werden. Die erste Zahl gibt den Array-Index von dem globalen Array *PID\_AllKs[9]* und die restliche Zahl den Wert der Stelle an.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **Wert** | KP Roll | KI Roll | KD Roll | KP Pitch | KI Pitch | KD Pitch | KP Yaw | KI Yaw | KD Yaw |

Nachdem ein Datensatz eingelesen worden ist, wird ein Interrupt mit der Funktion *PID\_ChangeKs()* ausgeführt.

Dateiname: PID.c

/\*\*

 \* @brief This function changes a PID controller coefficients via uart

 \* @param huart

 \*/

void PID\_ChangeKs(UART\_HandleTypeDef \*huart)

{

  // get index

  int index = atoi(&receiveData[0]);

  // check if index is in range

  if(index < 0 || index > 8)

  {

    Terminal\_Print("index out of range\n\r");

    return;

  }

  // get value

  char value[9];

  sprintf(value, "%c%c%c%c%c%c%c%c", receiveData[2], receiveData[3], receiveData[4], receiveData[5], receiveData[6], receiveData[7], receiveData[8], receiveData[9]);

  // check if value is in range

  double tmp = atof(value);

  if(tmp > 5 || tmp < 0)

  {

    Terminal\_Print("value out of range\n\r");

    return;

  }

  // write value to array

  PID\_AllKs[index] = tmp;

  // write confirmation to terminal (eg: 0 = 0.250000)

  sprintf(txt, "[%d] = %f\n\r", index, PID\_AllKs[index]);

  Terminal\_Print(txt);

}

Diese Funktion wandelt die Daten in Zahlen um und schreibt den Wert in den richtigen Index im Array *PID\_AllKs[]*.

Danach wird ein Bestätigungstext mit dem Arrayindex und Zahlenwert ausgegeben. Wenn dieser Text nicht zurückgesendet wird, hat es ein Problem bei der Datenübertragung gegeben.

Wenn der Index oder die Zahl außerhalb des verwendetet Zahlenbereichs ist, wird der Text „out of range“ ausgegeben.

### PID-Algorithmus

Dateiname: PID.c

/\*\*

 \* @brief This function controls the flight PID controller

 \* @details

 \* The max throttle value can be altered with the define PID\_MAX\_TURN

 \* @param inputThrottle throttle value from joysticks

 \* @param inputPitch pitch value from joysticks

 \* @param inputRoll roll value from joysticks

 \* @param inputYaw yaw value from joysticks

 \* @return PID\_Status

 \*/

PID\_Status PID\_Update(float inputThrottle, float inputPitch, float inputRoll, float inputYaw)

{

  static double I\_Roll = 0, errorRollPrev = 0;

  static double I\_Pitch = 0, errorPitchPrev = 0;

  static double I\_Yaw = 0, errorYawPrev = 0;

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  --------------- get current angles + dt ---------------

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  IMU\_GetAngles();

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  -------------------- check off mode -------------------

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  if(inputThrottle < 5)

  {

    I\_Roll = 0;

    errorRollPrev = 0;

    I\_Pitch = 0;

    errorPitchPrev = 0;

    I\_Yaw = 0;

    errorYawPrev = 0;

    DShot\_SendThrottle(0, 0, 0, 0);

    return PID\_OK;

  }

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  ------------------ calc roll output -------------------

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  double errorRoll = -inputRoll - angle.roll;

  I\_Roll += (errorRoll \* imu\_DeltaTime) \* PID\_AllKs[1];

  // limit I value to +/- max throttle addition

  if(I\_Roll > PID\_MAX\_TURN) I\_Roll = PID\_MAX\_TURN;

  else if(I\_Roll < -PID\_MAX\_TURN) I\_Roll = -PID\_MAX\_TURN;

  double rollOutput = PID\_AllKs[0] \* errorRoll + I\_Roll + PID\_AllKs[2] \* ((errorRoll - errorRollPrev) / imu\_DeltaTime);

  // limit output value to +/- max throttle addition

  if(rollOutput > PID\_MAX\_TURN) rollOutput = PID\_MAX\_TURN;

  else if(rollOutput < -PID\_MAX\_TURN) rollOutput = -PID\_MAX\_TURN;

  errorRollPrev = errorRoll;

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  ------------------ calc pitch output ------------------

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  double errorPitch = inputPitch - angle.pitch;

  I\_Pitch += (errorPitch \* imu\_DeltaTime) \* PID\_AllKs[4];

  // limit I value to +/- max throttle addition

  if(I\_Pitch > PID\_MAX\_TURN) I\_Pitch = PID\_MAX\_TURN;

  else if(I\_Pitch < -PID\_MAX\_TURN) I\_Pitch = -PID\_MAX\_TURN;

  double pitchOutput = PID\_AllKs[3] \* errorPitch + I\_Pitch + PID\_AllKs[5] \* ((errorPitch - errorPitchPrev) / imu\_DeltaTime);

  // limit output value to +/- max throttle addition

  if(pitchOutput > PID\_MAX\_TURN) pitchOutput = PID\_MAX\_TURN;

  else if(pitchOutput < -PID\_MAX\_TURN) pitchOutput = -PID\_MAX\_TURN;

  errorPitchPrev = errorPitch;

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  -------------------- calc yaw output ------------------

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  static double controlYaw = 0;

  static int8\_t changeYawFlag = 1;

  double yawOutput = inputYaw \* PID\_MAX\_TURN;

  if(yawOutput >= -1 && yawOutput <= 1)

  {

    if(changeYawFlag == 1)

      controlYaw = angle.yaw;

    changeYawFlag = 0;

    double errorYaw = controlYaw - angle.yaw;

    I\_Yaw += (errorYaw \* imu\_DeltaTime) \* PID\_AllKs[7];

    // limit I value to +/- max throttle addition

    if(I\_Yaw > PID\_MAX\_TURN) I\_Yaw = PID\_MAX\_TURN;

    else if(I\_Yaw < -PID\_MAX\_TURN) I\_Yaw = -PID\_MAX\_TURN;

    yawOutput = PID\_AllKs[6] \* errorYaw + I\_Yaw + PID\_AllKs[8] \* ((errorYaw - errorYawPrev) / imu\_DeltaTime);

    // limit output value to +/- max throttle addition

    if(yawOutput > PID\_MAX\_TURN) yawOutput = PID\_MAX\_TURN;

    else if(yawOutput < -PID\_MAX\_TURN) yawOutput = -PID\_MAX\_TURN;

    errorYawPrev = errorYaw;

  }

  else

  {

    changeYawFlag = 1;

  }

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  ----------- check min/max values + output -------------

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  float motorLF = (inputThrottle<5) ? 0 : inputThrottle-pitchOutput-rollOutput-yawOutput;

  float motorRF = (inputThrottle<5) ? 0 : inputThrottle-pitchOutput+rollOutput+yawOutput;

  float motorLR = (inputThrottle<5) ? 0 : inputThrottle+pitchOutput-rollOutput+yawOutput;

  float motorRR = (inputThrottle<5) ? 0 : inputThrottle+pitchOutput+rollOutput-yawOutput;

  if(motorLF < 5) motorLF = 0;

  if(motorRF < 5) motorRF = 0;

  if(motorLR < 5) motorLR = 0;

  if(motorRR < 5) motorRR = 0;

  if(motorLF > inputThrottle + PID\_MAX\_TURN) motorLF = inputThrottle + PID\_MAX\_TURN;

  if(motorRF > inputThrottle + PID\_MAX\_TURN) motorRF = inputThrottle + PID\_MAX\_TURN;

  if(motorLR > inputThrottle + PID\_MAX\_TURN) motorLR = inputThrottle + PID\_MAX\_TURN;

  if(motorRR > inputThrottle + PID\_MAX\_TURN) motorRR = inputThrottle + PID\_MAX\_TURN;

  DShot\_SendThrottle(motorLF, motorRF, motorLR, motorRR);

  return PID\_OK;

}

Der gesamte PID-Regler besteht aus drei PID-Reglern. Jeweils einer für die Pitch-, Roll- und Yaw-Achse. Es wird mit dem IMU-Lagewinkel und den übergebenen Variablen die Regelabweichungen gebildet, die dann weiter zu Throttlewerten umgerechnet werden.

## Terminal Übertragung und Status LEDs

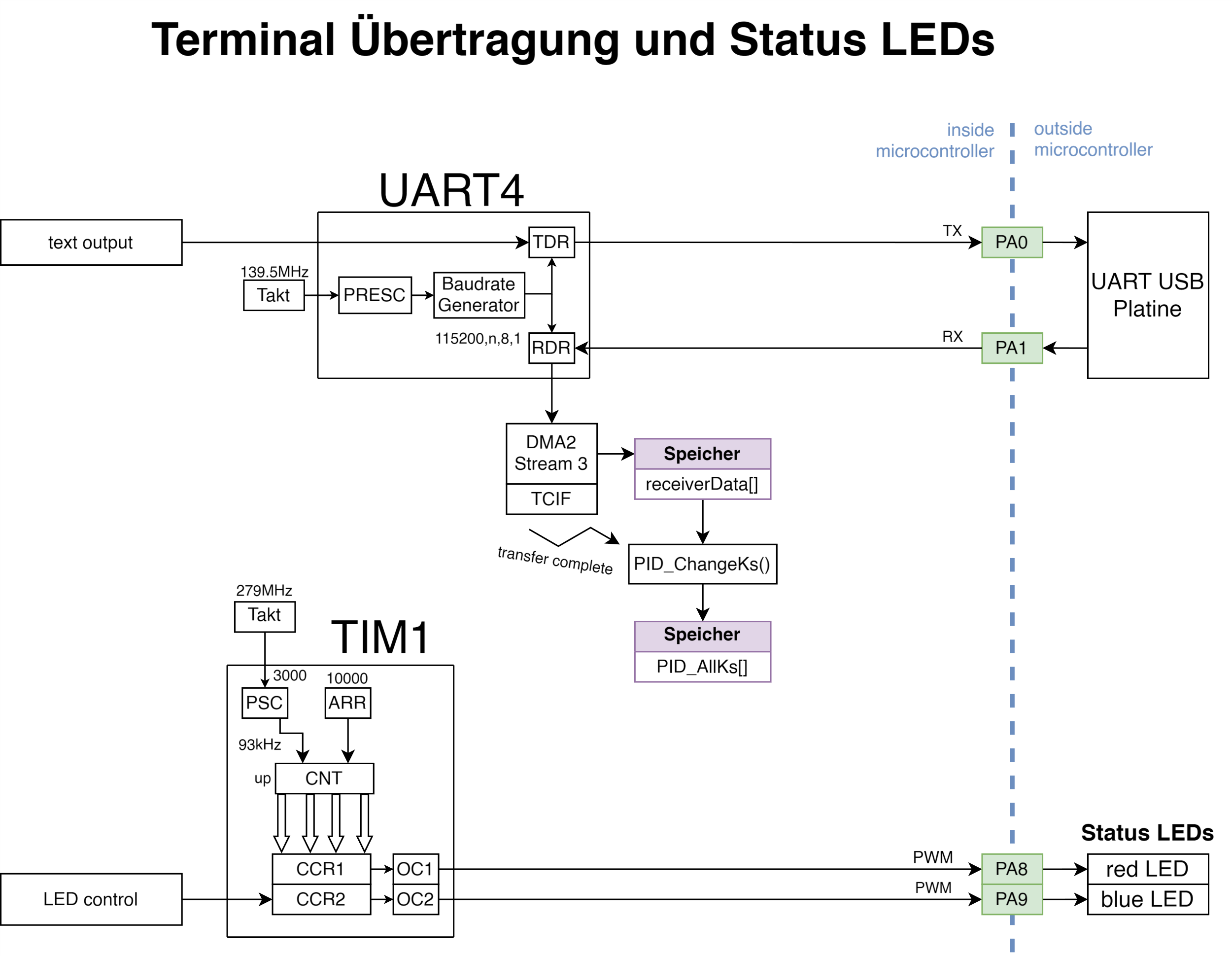


Abbildung 37: Terminal Übertragung und Status LEDs Architektur

Während des Initialisierungsprozesses und im Falle eines Errors wird ein Text am Terminal ausgegeben. Die Funktion Terminal\_Print() formatiert die Daten und schickt diese an die UART4-Peripherie. Diese können dann über eine USB-Platine an einen Terminal angezeigt werden.

Während der Initialisierungssequenz wird die Frequenz und der Duty-Cycle der TIM1-Peripherie geändert. Dadurch können die Status-LEDs in unterschiedliche Frequenz blinken oder Helligkeit leuchten. Während der Initialisierung blinkt die blaue LED in unterschiedlicher Geschwindigkeit. Im Fall eines Errors schaltet sich die blaue LED aus und die rote LED beginnt zu leuchten *(siehe:* [*Kapitel 5.11*](#_Status_–_LEDs)*)*.

Mit der UART4-Peripherie wird eine weitere wichtige Funktion ausgeführt. Es kann über die USB-Platine auch Daten zum Mikrocontroller gesendet werden. Mit diesen Daten können die PID-Regelkoeffizienten während der Laufzeit verändert werden. Dabei wartet durchgehend der DMA-Controller auf Daten. Nach dem Empfang wird eine Empfangs-ISR ausgeführt, die die Daten zu richtigen Werten umformt und in dem Array *PID\_AllKs[]* speichert

*(siehe:* [*Kapitel 5.10*](#_Terminal_Ausgabe_1)*)*.

## Terminal Kommunikation

Die Terminal Ausgabe ist essenziell für die Programmkontrolle, da der aktuelle Inhalt von Variablen direkt ausgegeben werden kann. Im Vergleich zum Debuggen wird das Programm dabei nur kurz unterbrochen, was bei Interrupt gesteuerten System zu Problemen führen kann.

Die Kommunikation findet mithilfe der UART-Peripherie statt. Diese ist mit einer Zusatzplatine verbunden, um den Datenaustausch mit einem externen System zu ermöglichen *(siehe Kapitel Marcel)*.

**UART-Einstellungen in STM32CubeMX:**

Ein Bild, das Text, Screenshot, Software, Website enthält.

Automatisch generierte Beschreibung

Abbildung 38: STM32CubeMX Einstellungen Terminal

Für den Empfang der Daten wird die Erweiterung „Serial Monitor“ [SERM] für Visual Studio Code verwendet.

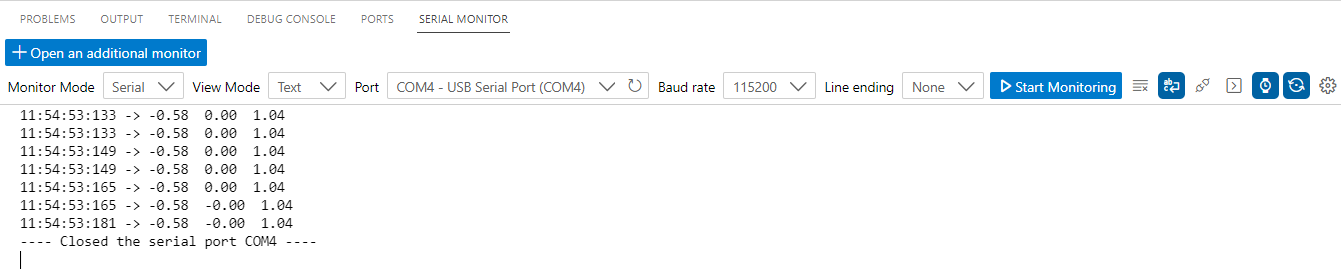


Abbildung 39: Terminal Bild

Wichtig:

Die Baudrate, die für die Datenübertragung verwendet wird, ist 115200 Bits/s.

**Text zum Terminal senden:**

Dateiname: status\_handling.c

/\*\*

 \* @brief This function prints a string to the terminal

 \* @param string

 \* @retval none

 \*/

void Terminal\_Print(char \*string)

{

  HAL\_UART\_Transmit(&huart4, (uint8\_t \*)string, strlen(string), HAL\_MAX\_DELAY);

}

**Beispiel Programmaufruf:**

// with only text

Terminal\_Print("wichtige Daten\n\r");

// with variable

char text[] = "wichtige Daten\n\r";

Terminal\_Print(text);

## Status – LEDs

Auf der Flight-Controller Platine befinden sich eine rote und eine blaue LED. Die blaue wird zum Initialisierungsstatus, und die rote LED wird für zur Anzeige von Errors verwendet.

|  |  |  |
| --- | --- | --- |
| **Blaue LED** | leuchtet nicht | Programm nicht gestartet |
| blinkt alle 0,5 Sekunden | DS2438 initialisiert |
| blinkt alle 0,4 Sekunden | IMU initialisiert |
| blinkt alle 0,3 Sekunden | PID initialisiert |
| blinkt alle 0,2 Sekunden | DShot initialisiert |
| blinkt alle 0,1 Sekunden | Receiver initialisiert |
| leuchtet durchgehend | Initialisierungen abgeschlossen |
| **Rote LED** | leuchtet nicht | kein Error |
| leuchtet durchgehend | Error während Initialisierungen |
| blinkt alle 0,1 Sekunden | Failsafe von Receiver oder DS2438 ausgelöst |

**Einstellungen STM32CubeMX:**

**Ein Bild, das Text, Screenshot, Software, Computersymbol enthält.

Automatisch generierte Beschreibung**

Abbildung 40: STM32CubeMX Einstellungen LEDs

Für die Änderung der Frequenz des PWM-Signals wird folgende Formel verwendet:

fBlink … Ausgangsfrequenz

fTimer … Timer Eingangsfrequenz, für Diplomarbeit 279MHz

PSC … prescaler Registerwert

ARR … auto-reload Registerwert, für Diplomarbeit fixiert auf 10000-1

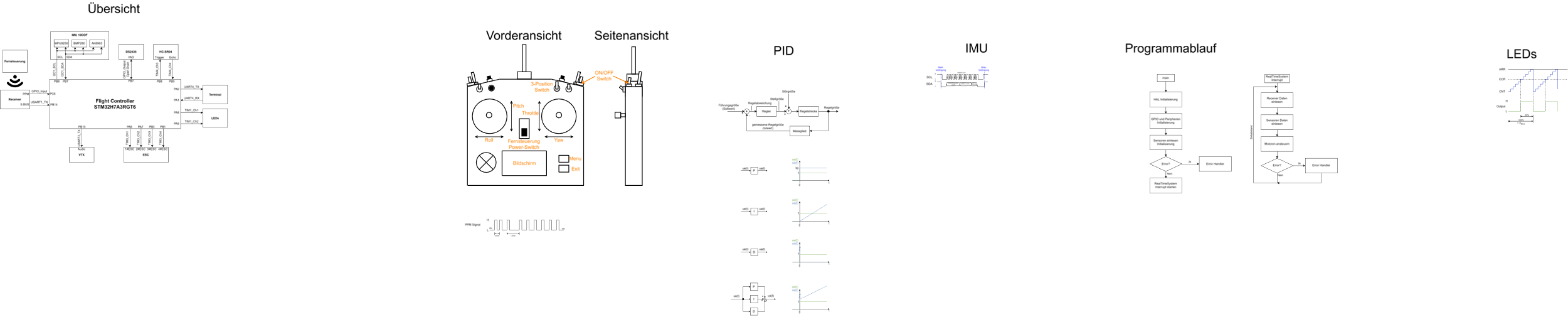


Abbildung 41: Beispiel PWM-Signalverlauf

Wenn der Wert im Counter-Register (CNT) unter dem Wert im capture/compare Register (CCR) ist, wird ein low-Zustand ausgegeben. Wenn der CNT-Wert größer ist als der CCR-Wert ist, dann wird ein high-Zustand ausgegeben. Durch die Änderung des CCR-Wertes kann der Duty Cycle verändert werden.

Für die Ansteuerung der LEDs wird ein Duty-Cycle von 50% verwendet. Dafür muss der Wert im CCR-Register die Hälfte des Wertes im ARR-Register gesetzt werden – *siehe Abbildung 46*

Um die Werte während der Programmlaufzeit ändern zu können, verwendet man folgende Funktionen:

// set duty cycle for LEDs

\_\_HAL\_TIM\_SET\_COMPARE(&htim1, TIM\_CHANNEL\_1, 0);

\_\_HAL\_TIM\_SET\_COMPARE(&htim1, TIM\_CHANNEL\_2, 5000);

// set frequency ~2Hz

\_\_HAL\_TIM\_SET\_PRESCALER(&htim1, 14000 - 1);

// start timer for LEDs

HAL\_TIM\_PWM\_Start(&htim1, TIM\_CHANNEL\_1);

HAL\_TIM\_PWM\_Start(&htim1, TIM\_CHANNEL\_2);

Hinweis:

Die Funktionen erlauben es, auch die Registerwerte zu ändern, ohne die Signal-Ausgabe unterbrechen zu müssen.

# Datenübertragung der Mess- und Videodaten (Ben)

# Visualisierungs-App

# Videostreaming

# Videoserver-Port Forwarding

# Ergebnisse

Entweder Gesamt System irgendwelche Bilder

Oder wenn nicht, dann einzelne Teile die funktionieren zeigen / beschreiben

Das Kapitel ist, um die Ergebnisse zu verkaufen, basically alles was geht

Name könnte sich noch ändern

# Anhang

## Einführung CAD – Software (Fusion 360) (Ben)

### UI und Projekterstellung

### Skizze anfertigen

### Körper erstellen

### Schrift und Bilder einfügen

## 3D – Druck (Ben)

### 3D – Drucker

### Filamente

### 3D – Drucker Software (Ultimaker Cura)

## Inbetriebnahme Anleitung

## Projektplan

Ein Bild, das Text, Screenshot, Reihe, Zahl enthält.

Automatisch generierte Beschreibung

## Projektkosten

## Projekttagebuch

## Einführung Entwicklungsumgebung - Steuerungssoftware

Um den Mikroprozessor STM32H7A3RGT6 auf der Flight-Controller Platine zu programmieren, wird die Entwicklungsumgebung Keil µVision5 in der Version V5.38.0.0 in Kombination mit STM32CubeMX in der Version 6.10.0 und Vision Studio Code verwendet. Programmiert wird dieser mithilfe eines DAPLink (CMSIS-DAP) – Interface über die SWD-Schnittstelle.

Mit STM32CubeMX können die Grundeinstellungen des Mikrocontrollers, wie die Peripherie- und Taktversorgungseinstellungen, mit einer grafischen Oberfläche einfach getätigt und automatisch eine Initialisierungssoftware in der Programmiersprache C generiert werden. Diese Software wird mithilfe von HAL (hardware abstraction layer) erstellt. Dieses System bietet eine Menge APIs, die es ermöglichen mit einfachen Funktionen komplexe Befehle und Einstellungen in einen STM32 Mikrocontroller zu tätigen.

Keil µVision5 bietet eine integrierte Entwicklungsumgebung mit eingebautem Assembler, Compiler und Debugger. Es gibt eine beschränkte kostenlose Version im Internet, die aber nicht für die Programmgröße der Diplomarbeit ausreicht. Daher muss eine kostenlose Community-Lizenz von Keil aktiviert werden [KLIZ].

Für die Programmierung des C-Sourcecodes wird Visual Studio Code verwendet. Mit der Erweiterung „Keil Assistent“ [KASS] kann Keil µVision5 direkt von einer Benutzeroberfläche gesteuert werden. Visual Studio Code bietet eine Vielfalt von Funktionen, die den Arbeitsablauf produktiver und einfacher gestalten.

**STM32CubeMX Projekt erstellen:**

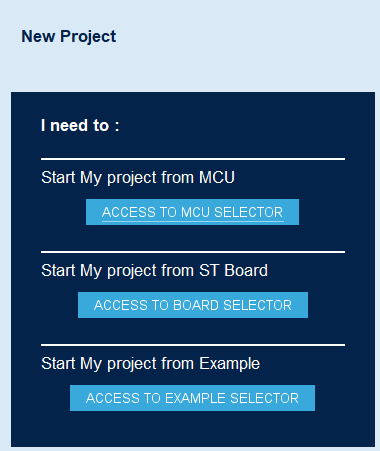


Abbildung 42: STM32CubeMX access to MCU selector

Nach dem Starten von der Software muss der zu programmierende Mikrocontroller ausgewählt werden.

Bei vorgefertigten Entwicklungsplatinen, kann „access to board selector“ oder „access to example selector“ verwendet werden.

Für die Diplomarbeit wird eine eigenerstellte Platine mit Mikrocontroller verwendet, daher wird die Option „access to MCU selector“ verwendet.

Nach der Auswahl werden die aktuellen Informationen der Mikrocontroller und Entwicklungsplatinen von der STM32 Datenbank runtergeladen.

Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 43: STM32CubeMX Liste von Mikrocontroller

Mit Hilfe von Filtern und Suchoptionen kann der gewünschte Mikrocontroller ausgewählt werden. Bevor das Projekt erstellt wird, können die Eigenschaften der Auswahl kontrolliert werden. Um das Projekt zu erstellen, muss der Knopf „Start Project“ gedrückt werden.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Abbildung 44: STM32CubeMX Auswahl Mikrocontroller

Mit der grafischen Oberfläche können die gewünschten Peripherie-, Pin-, Takt- und Projekteinstellungen getätigt werden.

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

Abbildung 45: STM32CubeMX grafische Oberfläche

Um die effizienteste Programmstruktur für die Diplomarbeit zu erstellen, müssen wichtige Eistellungen getätigt werden:

Als Erstes muss die Taktquelle ausgewählt werden. Da externe Quarzoszillatoren verwendet werden, muss in der Kategorie RCC für „High Speed Clock (HSE)“ und „Low Speed Clock (LSE)“ die Einstellung „Crystal/Ceramic Resonator“ ausgewählt werden.

Ein Bild, das Text, Software, Computersymbol, Webseite enthält.

Automatisch generierte Beschreibung

Abbildung 46: STM32CubeMX HSE/LSE Einstellung

Mit der „Clock Configuration“ kann die gesamte Taktstruktur nach Anforderungen verändert werden:

Ein Bild, das Diagramm, Text, Reihe, Plan enthält.

Automatisch generierte Beschreibung

Abbildung 47: STM32CubeMX clock configuration

Wichtig: Wenn externe Oszillatoren verwendet werden, muss auf der linken Seite, die Einstellung „Input frequency“ auf die vorhandene Frequenz gesetzt werden.

Für die Diplomarbeit wird ein 8MHz Quarz Oszillator, der auf eine Systemfrequenz von 279MHz erhöht wird, verwendet.

Um den genierten Code mit der Keil µVision5 Entwicklungsumgebung verwenden zu können, müssen unter Project Manager gewisse Einstellungen getätigt werden:

Ein Bild, das Text, Screenshot, Reihe, Zahl enthält.

Automatisch generierte Beschreibung

Abbildung 48: STM32CubeMX project settings

Eine für die Drohne notwendige Einstellung ist unter Project Manager 🡪 Advanced Settings 🡪 Register Callback. In diesen Bereich können für einzelne Peripherien eigenerstellte Callback-Funktionen aktiviert werden.

Für die FPV-Drohne ist die Einstellung für die TIM-, UART- und USART-Peripherie aktiviert werden.



Abbildung 49: STM32CubeMX register callback

Eine Callback-Funktion wird beim Auslösen eines Interrupts von der ISR (Interrupt Service Routine) ausgelöst.

Zum Beispiel: *void HAL\_TIM\_PeriodElapsedCallback (TIM\_HandleTypeDef \* htim)* wird jedes Mal ausgeführt, wenn eine Timer-Peripherie einen Überlauf-Interrupt auslöst. Diese Funktion ist in der Datei *stm32h7xx\_hal\_tim.c* mit dem Präfix *\_\_weak* definiert. Das bedeutet, wenn die Funktion in einer anderen Datei ausprogrammiert wird, wird diese mit der neuen Version ersetzt.

Die Einstellung von Register Callback erlaubt es, die Funktion mit einer eigenen zu ersetzen. Dafür muss die Funktion *HAL\_TIM\_RegisterCallback()* aufgerufen werden.

Zum Beispiel:

|  |
| --- |
| HAL\_TIM\_RegisterCallback(&htim15, HAL\_TIM\_PERIOD\_ELAPSED\_CB\_ID, RealTimeSystemCallback); |

Von der Timer15-Peripherie wird das Überlauf-Interrupt-Callback mit der Funktion *RealTimeSystemCallback()* ersetzt.

Nach der fertigen Einstellung des Projekts, können die Programmdateien mit dem Knopf „Generate Code“ erstellt werden. Daraufhin wird folgende Ordnerstruktur erstellt:

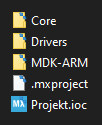


Abbildung 50: STM32CubeMX Ordnerstruktur

* Core: alle Programm- und Headerdateien
* Drivers: alle HAL-Dateien, die für das Projekt notwendig sind
* MDK-ARM: Keil µVision5 Projektdateien

**Keil µVision5 Einstellungen:**

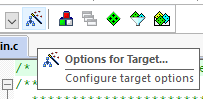


Abbildung 51: µVision Zauberstab

Die Einstellungen des Projekts können mit dem Zauberstab „Options for Target…“ eingestellt werden.

Die wichtigsten Einstellungen sind die Target- und Debug-Einstellungen:

Ein Bild, das Text, Screenshot, Display, Zahl enthält.

Automatisch generierte Beschreibung

Abbildung 52: µVision Target Einstellungen

Ein Bild, das Text, Screenshot, Zahl, Display enthält.

Automatisch generierte Beschreibung

Abbildung 53: µVision Debug Einstellungen

Für die Diplomarbeit wird der default compiler version 6 und der CMSIS-Dap Debugger verwendet.

Das von STM32CubeMX erstellte Programm muss nicht geändert werden, und eigener Code soll zwischen den Kommentaren „USER CODE BEGIN“ und „USER CODE END“ eingefügt werden, damit, wenn ein neuer Code generiert wird, der eigene nicht überschrieben wird.

Um das Programm auf den Mikrocontroller zu spielen, müssen folgende Knöpfe verwendet werden:



Abbildung 54: µVision Build/Flash Knöpfe

Auf der linken Seite kann das Projekt mit virtuellen Ordnern strukturiert werden.

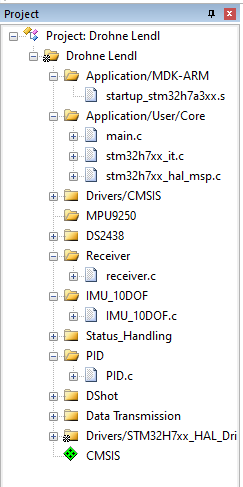


Abbildung 55: µVision Ordnerstruktur

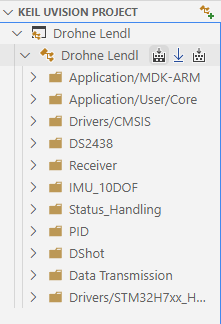


Abbildung 56: Visual Studio Code Ordnerstruktur

**Visual Studio Code:**

Mit Visual Studio Code kann dieselbe Ordnerstruktur des Keil µVision5 Projekts geöffnet und dessen Funktionen gesteuert werden *(siehe: Abbildung 55 und 56)*.

# Quellen

## Gedruckte Medien

Für die Quellenangabe bei Fachbüchern ist, wie nachfolgend dargestellt, vorzugehen (löschen):

Vorname Nachname: Titel. Untertitel. - Verlagsort: Verlag, Jahr.

Vorname Nachname: Titel. Untertitel. Auflage - Verlagsort: Verlag, Jahr.

z.B.:

[REI13] Jürgen Reichhardt: Lehrbuch Digitaltechnik - Eine Einführung mit VHDL  
 3. Auflage, München: Oldenbourg Verlag, 2013  
 ISBN: 978-3-486-72765-4

## Online

|  |  |
| --- | --- |
| [KLIZ] | ARM Keil: Community Lizenz Anleitung  <https://www.keil.arm.com/mdk-community/>  (letzter Aufruf: 28.02.2024) |
| [KASS] | Visual Studio Marketplace: Keil Assistant Erweiterung  <https://marketplace.visualstudio.com/items?itemName=CL.keil-assistant>  (letzter Aufruf: 28.02.2024) |
| [SERM] | Visual Studio Marketplace: Serial Monitor Erweiterung  <https://marketplace.visualstudio.com/items?itemName=ms-vscode.vscode-serial-monitor>  (letzter Aufruf: 28.02.2024) |
| [SBDW] | digitalwire: Futaba S-Bus Protokoll  <https://digitalwire.ch/de/projekte/futaba-sbus/>  (letzter Aufruf: 28.02.2024) |
| [IBDSP] | The FlySky iBus protocol  <https://blog.dsp.id.au/posts/2017/10/22/flysky-ibus-protocol/>  (letzter Aufruf: 28.02.2024) |
| [IBGH] | STM32 HAL iBUS  <https://github.com/mokhwasomssi/stm32_hal_ibus>  (letzter Aufruf: 28.02.2024) |
| [SBGH] | STM32 RadioLink SBUS DMA  <https://github.com/osos11-Git/STM32_RadioLink_SBUS_DMA/tree/main>  (letzter Aufruf: 02.03.2024) |
| [DSBW] | Brushless Whoop: DSHOT – the missing handbook  <https://brushlesswhoop.com/dshot-and-bidirectional-dshot/>  (letzter Aufruf: 28.02.2024) |
| [DSBF] | Betaflight: DSHOT  <https://betaflight.com/docs/development/Dshot>  (letzter Aufruf: 02.03.2024) |
| [DSGH] | stm32 hal dshot  <https://github.com/mokhwasomssi/stm32_hal_dshot/tree/main>  (letzter Aufruf: 02.03.2024) |
| [IMUWS] | Waveshare: 10 DOF IMU Sensor  <https://www.waveshare.com/wiki/10_DOF_IMU_Sensor_(C)>  (letzter Aufruf: 28.02.2024) |
| [M7] | STM32H7A3RGT6 Datenblätter  <https://www.st.com/en/microcontrollers-microprocessors/stm32h7a3rg.html>  (letzter Aufruf: 28.02.2024) |
| [M7HAL] | STM32H7 HAL Datenblätter  <https://www.st.com/en/embedded-software/stm32cubeh7.html#documentation>  (letzter Aufruf: 28.02.2024) |
| [DS2438] | DS2438 Datenblatt  <https://www.analog.com/media/en/technical-documentation/data-sheets/DS2438.pdf>  (letzter Aufruf: 28.02.2024) |
| [COFIL] | Complementary filter and relative orientation with MPU9250  <https://www.hackster.io/hibit/complementary-filter-and-relative-orientation-with-mpu9250-d4f79d>  (letzter Aufruf: 28.02.2024) |

# Verzeichnis der Abbildungen

[Abbildung 1: Regler-Blockschaltbild 21](#_Toc162121250)

[Abbildung 2: P-Glied Sprungantwort 22](#_Toc162121251)

[Abbildung 3: P-Glied Schaltsymbol 22](#_Toc162121252)

[Abbildung 4: I-Glied Schaltsymbol 22](#_Toc162121253)

[Abbildung 5: I-Glied Sprungantwort 22](#_Toc162121254)

[Abbildung 6: D-Glied Schaltsymbol 23](#_Toc162121255)

[Abbildung 7: D-Glied Sprungantwort 23](#_Toc162121256)

[Abbildung 8: PID-Regler Schaltsymbol 23](#_Toc162121257)

[Abbildung 9: PID-Regler Sprungantwort 23](#_Toc162121258)

[Abbildung 10: Flussdiagramm Programmablauf 25](#_Toc162121259)

[Abbildung 11: STM32CubeMX Einstellungen DS2438 32](#_Toc162121260)

[Abbildung 12: One-Wire Bit 1 senden 33](#_Toc162121261)

[Abbildung 13: One-Wire Bit 0 senden 33](#_Toc162121262)

[Abbildung 14: One-Wire Bit empfangen 36](#_Toc162121263)

[Abbildung 15: One-Wire Initialisierungssequenz 38](#_Toc162121264)

[Abbildung 16: Registerübersicht DS2438 40](#_Toc162121265)

[Abbildung 17: DS2438 Spannungsregister Format 42](#_Toc162121266)

[Abbildung 18: Real Time System Interrupt Architektur 44](#_Toc162121267)

[Abbildung 19: Fernsteuerung 46](#_Toc162121268)

[Abbildung 20: Receiver 46](#_Toc162121269)

[Abbildung 21: Fernsteuerung Tastenbelegung 47](#_Toc162121270)

[Abbildung 22: Beispiel PPM-Signal 50](#_Toc162121271)

[Abbildung 23: STM32CubeMX Einstellungen S.Bus 51](#_Toc162121272)

[Abbildung 24: STM32CubeMX Einstellungen I.Bus 52](#_Toc162121273)

[Abbildung 25: Einstellungen DMA für Receiver in STM32CubeMX 54](#_Toc162121274)

[Abbildung 26: I²C Datentransfer 67](#_Toc162121275)

[Abbildung 27: STM32CubeMX Einstellungen IMU 67](#_Toc162121276)

[Abbildung 28: I²C Schreibzyklus 68](#_Toc162121277)

[Abbildung 29: I²C Lesezyklus 69](#_Toc162121278)

[Abbildung 30: Registerübersicht MPU9250 73](#_Toc162121279)

[Abbildung 31: Registerübersicht BMP280 79](#_Toc162121280)

[Abbildung 32: Formel Berechnung Höhe über Meeresspiegel 84](#_Toc162121281)

[Abbildung 33: Motorregelalgorithmus Architektur 85](#_Toc162121282)

[Abbildung 34: DShot Bit 0/1 Duty Cycle + Geschwindigkeit 87](#_Toc162121283)

[Abbildung 35: DShot Beispiel Übertragung 87](#_Toc162121284)

[Abbildung 36: STM32CubeMX Einstellung PID DMA Empfang 93](#_Toc162121285)

[Abbildung 37: Terminal Übertragung und Status LEDs Architektur 98](#_Toc162121286)

[Abbildung 38: STM32CubeMX Einstellungen Terminal 100](#_Toc162121287)

[Abbildung 39: Terminal Bild 100](#_Toc162121288)

[Abbildung 40: STM32CubeMX Einstellungen LEDs 102](#_Toc162121289)

[Abbildung 41: Beispiel PWM-Signalverlauf 103](#_Toc162121290)

[Abbildung 42: STM32CubeMX access to MCU selector 107](#_Toc162121291)

[Abbildung 43: STM32CubeMX Liste von Mikrocontroller 108](#_Toc162121292)

[Abbildung 44: STM32CubeMX Auswahl Mikrocontroller 108](#_Toc162121293)

[Abbildung 45: STM32CubeMX grafische Oberfläche 109](#_Toc162121294)

[Abbildung 46: STM32CubeMX HSE/LSE Einstellung 109](#_Toc162121295)

[Abbildung 47: STM32CubeMX clock configuration 110](#_Toc162121296)

[Abbildung 48: STM32CubeMX project settings 110](#_Toc162121297)

[Abbildung 49: STM32CubeMX register callback 111](#_Toc162121298)

[Abbildung 50: STM32CubeMX Ordnerstruktur 111](#_Toc162121299)

[Abbildung 51: µVision Zauberstab 112](#_Toc162121300)

[Abbildung 52: µVision Target Einstellungen 112](#_Toc162121301)

[Abbildung 53: µVision Debug Einstellungen 112](#_Toc162121302)

[Abbildung 54: µVision Build/Flash Knöpfe 112](#_Toc162121303)

[Abbildung 55: µVision Ordnerstruktur 113](#_Toc162121304)

[Abbildung 56: Visual Studio Code Ordnerstruktur 113](#_Toc162121305)

# Begleitprotokoll

**Schule**: 106,95 Stunden

**Freizeit**: 232,43 Stunden

**Summe**: 339,38 Stunden

# Anhang

## — Projektdokumentation (Kostendarstellung, Besprechungsprotokolle etc.)

## — Technische Dokumentation (technische Beschreibungen, Berechnungen,

## Konstruktionszeichnungen, Versuchsberichte, betriebswirtschaftliche Kalkulationen etc.)

## Schaltungen, Zeichnungssätze, sonstiges