Softwareentwurf

Die Aufgaben der Software besteht darin, die Steuerung des Systems und die Kommunikation mit einer Bedieneinheit zu übernehmen. Als Plattform wird ein STM32F4-Discovery Board mit einem Free-RTOS Betriebssystem verwendet. In den folgenden Abschnitten werden die Aufgaben in Teilfunktion untergliedert und deren Implementation erörtert.

1 Namen- und Typkonventionen

Um einen einheitlichen Programmentwurf zu gewährleisten werden Konventionen zur Wahl von Bezeichnern festgelegt. In der Datei ”Global.h”werden die Bezeichnungen für Standarddatentypen definiert. In dieser Anwendung sind vorzugsweise vorzeichenlose Ganzzahlen zu verwenden, vorzeichenbehaftete Ganz- und Gleitpunktzahlen sind nur in zwingend erforderlichen Situation, wie z.B. Berechnung der Reglerwerte, zu verwenden. Hierbei ist besondere Sorgfalt bei der Umrechnung zwischen den Datentypen erforderlich.

- UInt8 : Vorzeichenlose Ganzzahl, 8Bit
- UInt16 : Vorzeichenlose Ganzzahl, 16Bit
- UInt32 : Vorzeichenlose Ganzzahl, 32Bit
- UInt64 : Vorzeichenlose Ganzzahl, 64Bit
- Int8 : Vorzeichenbehaftete Ganzzahl, 8Bit
- Int16 : Vorzeichenbehaftete Ganzzahl, 16Bit
- Int32 : Vorzeichenbehaftete Ganzzahl, 32Bit
- Int64 : Vorzeichenbehaftete Ganzzahl, 64Bit

Die folgenden Präfixe werden für Bezeichner verwendet.

- E : Enumeration
- C : Klasse
- A : Abstrakte Klasse
- I : Interface Klasse
- T : Template Klasse
- S : Klasse nach Singletonmuster
- m : Membervariable einer Klasse
- s : Statische Memebervariable einer Klasse

2 Interruptverarbeitung

Bei der Gestaltung von Interrupt-Service-Routinen muss Wert daraufgelegt werden kurze Funktionen zu entwerfen. D.h., dass bei Auftreten eines Interrupt die Quelle ausgewertet und daraufhin ggf. eine Nachricht erzeugt wird um das System über das Ereignis zu informieren. Alle Interruptquellen verwenden die selbe Prioritätsstufe[1].

[1] Eine Ausnahme bilden Prozessor-Faults (z.B. Memory-Bus-Fault), diese besitzen die höchste Interruptpriorität, welche für derartige Faults reserviert ist.

3 Komponentenarchitektur

Einzelne Funktionen, welche parallel zu bearbeiten sind, werden als Komponenten implementiert. In diesem Projekt wird einerseits eine Komponente zur Steuerung des Systems und andererseits eine Kommunikationskomponente eingesetzt. Beide Komponenten werden in einem separaten Task ausgeführt. Die Kommunikation der Komponenten erfolgt über ein Nachrichtensystem. Um Nachrichten entgegenzunehmen wird jede Komponente mit einer Nachrichtenschlange ausgestattet. FreeRTOS stellt hierfür threadsafe Queues zur Verfügung. Die Ausführung einer Komponente erfolgt mittels der *run*-Methode, diese wird in dem jeweiligen Task aufgerufen. Diese Grundstruktur der Komponenten wird in der abstrakten Klasse *AComponentBase* umgesetzt. Von dieser Basisklasse werden wiederum die konkreten Klassen *SControlComponent* und *SCommComponent* abgeleitet. Diese werden nach dem Singleton-Muster implementiert.

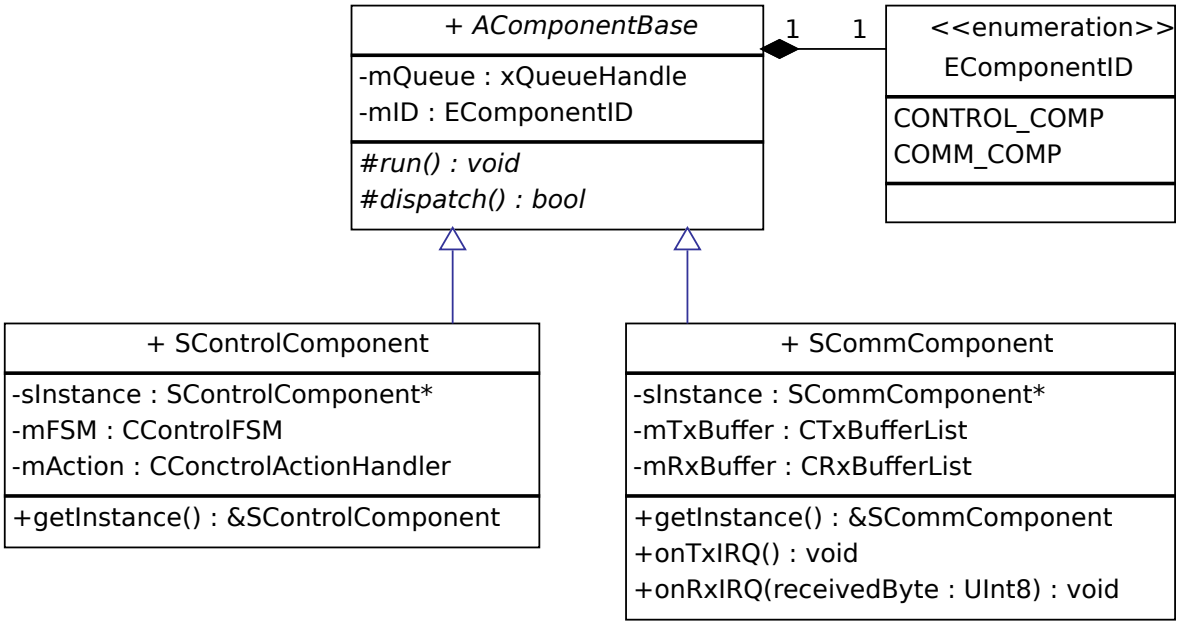

Abbildung 1: Klassendiagramm Komponentenarchitektur, Quelle: eigene Darstellung

4 Kommunikation mittels Nachrichten

Mit Hilfe von Nachrichten können die beiden Komponenten Informationen untereinander austauschen. Eine Nachricht enthält Informationen über den Sender, Empfänger und das Ereignis, welches die Nachricht veranlasst. Die Umsetzung der Nachrichten erfolgt in Form der Klasse *CMessage*. An Hand des Nachrichten Typ in *mType* kann das Event und somit der Nachrichteninhalt in *mData* interpretiert werden.

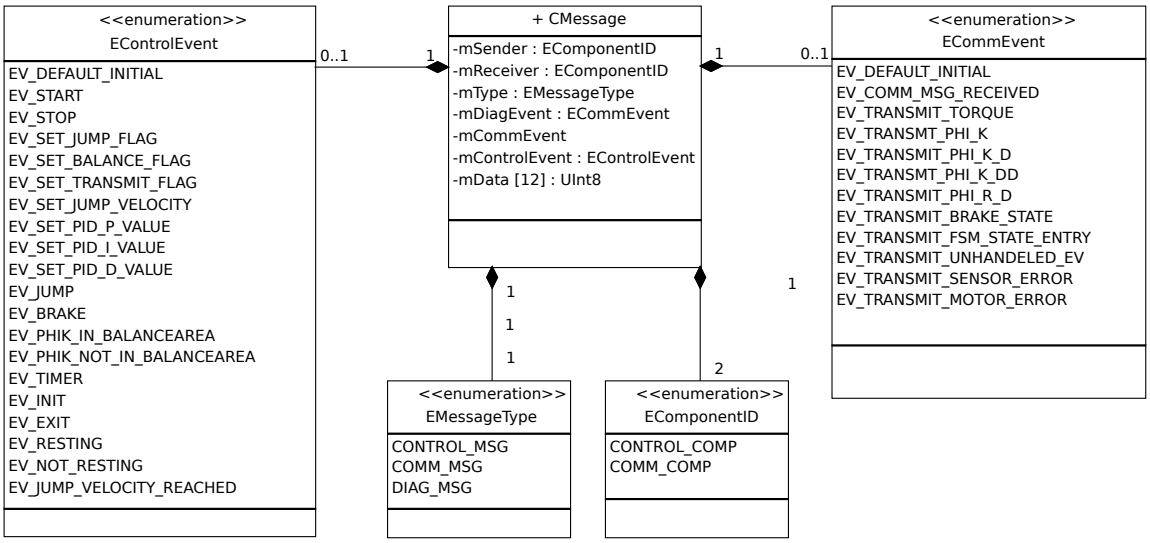

Abbildung 2: Klassendiagramm Nachrichtenstruktur, Quelle: eigene Darstellung

4.1 Erzeugen und Verteilen der Nachrichten

Die Nachrichten werden von der Klasse *SProxy* erzeugt. Diese wird nach dem Singleton-Muster implementiert und dient als globale, öffentliche Schnittstelle zu den Komponenten. Auf Grund der überschaubaren Komponentenarchitektur übernimmt diese Einheit auch die Verteilung der Nachrichten. Hierfür ist keine dynamische Registrierung der Komponenten erforderlich, da die statische Verteilungshirachie vor Laufzeit feststeht.

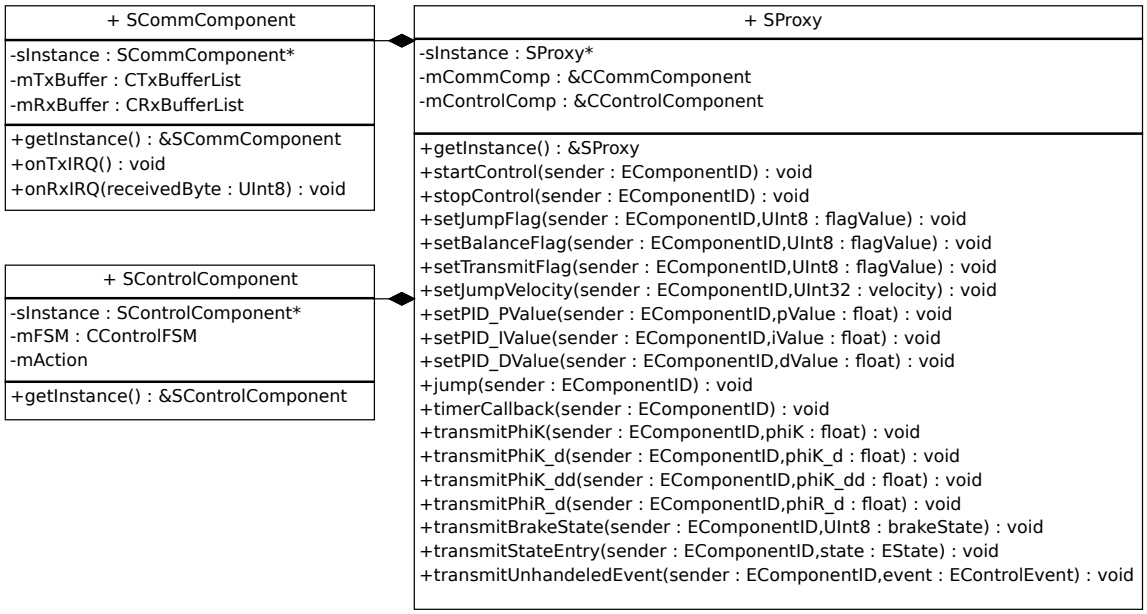

Abbildung 3: Klassendiagramm Proxy, Quelle: eigene Darstellung

5 Aufbau der Kontrollkomponente

Die Aufgabe der Kontrollkomponente besteht darin, den Würfel aufzurichten und in der gewünschten Gleichgewichtslage zu balancieren. Hierfür müssen die Sensoren ausgewertet und die Motoren entsprechend angesteuert werden. Die Regelungseinheit muss eine Schnittstelle bieten um die Parameter über die Bedieneinheit zu konfigurieren. Außerdem sollen die aktuellen Werte an die Bedieneinheit übermittelt werden. Diese Funktionen werden in Form eines Zustandsautomaten implementiert. Mit Hilfe von Events können die gewünschten Funktionen aufgerufen werden.Die Ereignisse werden als Enumeration kodiert.

Eventkodierung	**Parameter**	**Kommentar**
EV_START		Starten die Steuerung und Regelung des Würfels
EV_STOP		Beendet die Steuerung des Würfels
EV_SET_JUMP_FLAG	Flag-Value	Einstellung ob der Würfel aufspringen soll.
EV_SET_BALANCE_FLAG	Flag-Value	Einstellung ob der Würfel balancieren soll.
EV_SET_TRANSMIT_FLAG	Flag-Value	Einstellung ob Sensordaten übertragen werden sollen.
EV_SET_JUMP_VELOCITY	Velocity	Setzt die Sprungeschwindigkeit des Schwungrad.
EV_SET_PID_P_VALUE	P-Value	Setzt den P-Wert des PID-Regler.
EV_SET_PID_I_VALUE	I-Value	Setzt den I-Wert des PID-Regler.
EV_SET_PID_D_VALUE	D-Value	Setzt den D-Wert des PID-Regler.
EV_JUMP		Startet das Aufspringen des Würfel.
EV_BRAKE		Internes Event, welches das Bremsen auslöst.
EV_PHIK_IN_BALANCEAREA		Internes Event, Würfel im Reglerbereich.
EV_PHIK_NOT_IN_BALANCEAREA		Internes Event, Würfel außerhalb des Reglerbereich.
EV_TIMER		Internes Event, Softtimerüberlauf.
EV_INIT		Internes Event, für die Initialisierung eines Zustandes.
EV_EXIT		Internes Event, führt zum Verlassen eines Zustandes.
EV_RESTING		Internes Event, Würfel liegt ruhig.
EV_NOT_RESTING		Internes Event, Würfel liegt nicht ruhig.
EV_JUMP_VELOCITY_REACHED		Internes Event, Rad hat Sprunggeschwindigkeit.

Der Zustandsautomat enthält die beiden Zustände *CStateConfiguration* und *CStateRunning*. In dem Startzustand *CStateConfiguration* können die auszuführenden Funktionen der Steuerungseinheit und die Regelungsparameter festgelegt werden. Der Oberzustand *CStateRunning* enthält die drei Zustände *CStateIdle*, *CStateJump* und *CStateBalance*. Diese Zustände geben den Stand der Regelung wider. Um eine flexible Konfiguration des Würfels zu gewährleisten verfügt der Zustandsautomat über die drei Flags *mJumpFlag*, *mBalanceFlag* und *mTransmitFlag*. Diese sind als Bedingungen an die entsprechenden Zustandsübergänge gekoppelt.

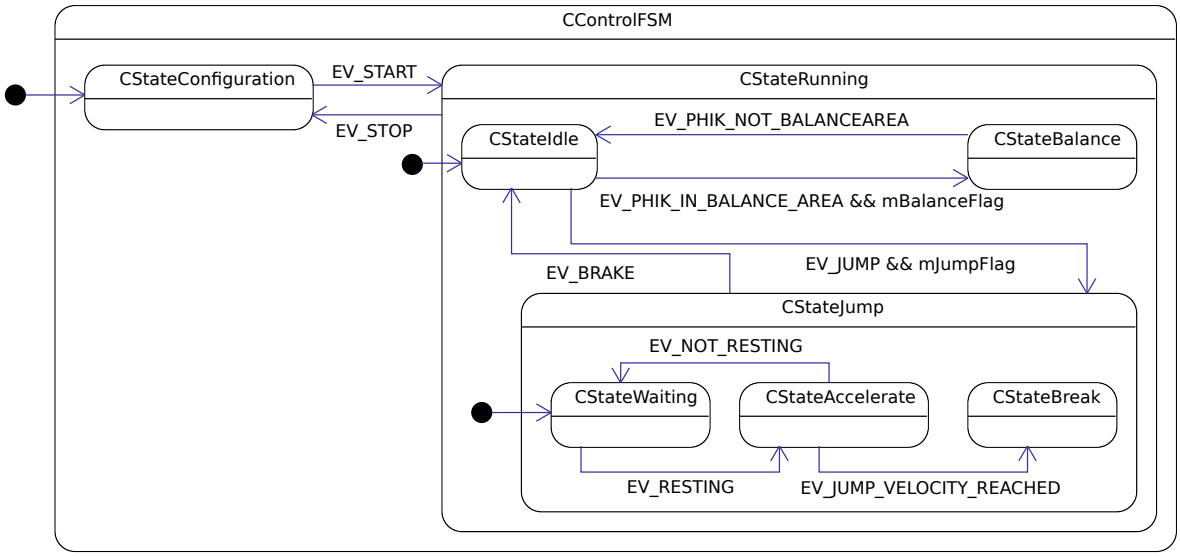

Abbildung 4: Zustandsdiagramm Kontrollkomponente, Quelle: eigene Darstellung

Der Zustandsautomat wird nach [WieTra] implementiert. Die Klasse *CFSM* dient als Basis für die Zustandsmaschine und die Oberzustände. Unterzustände hingegen werden als Methoden realisiert. Jeder Oberzustand verfügt außerdem über den Zustand *CStateDefaultInitial*. Dieser dient als Dummyzustand, der angenommen wird wenn der Oberzustand verlassen wird. Bei Betreten des Oberzustandes wird ein internes *EV_INIT* Event generiert, welches den Übergang in den eigentlichen Startzustand veranlasst.

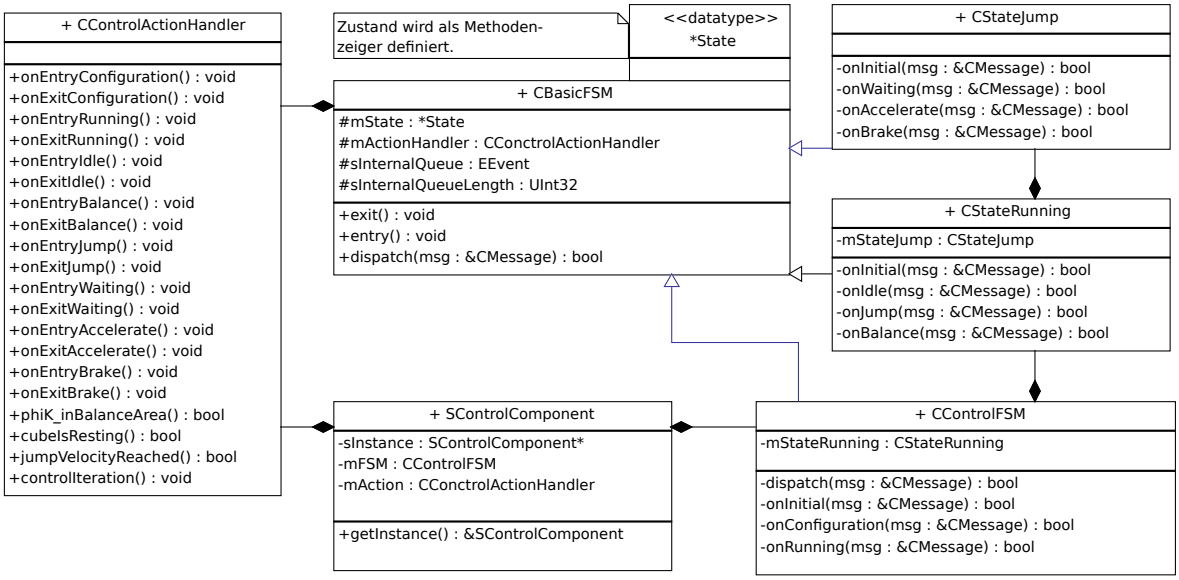

Abbildung 5: Klassendiagramm Zustandsautomat, Quelle: eigene Darstellung

Aufgaben von *CStateConfiguration*

Der Zustand *CStateConfiguration* ermöglicht die Einstellung der Steuereinheit. Einerseits kann die gewünschte Funktion durch Manipulieren der Flags *mJumpFlag*, *mBalanceFlag* und *mTransmitFlag* eingestellt werden. Diese Flags werden von der Steuereinheit gespeichert und als Bedingungen an gewisse Zustandsübergänge gekoppelt. Dadurch kann das Aufspringen, das Balancieren um den Sollwert und die Übermittlung der physikalischen Zustandswerte ein- bzw. ausgeschaltet werden. Zusätzliche können Parameter der Regelung konfiguriert werden. Einerseits kann die nötige Geschwindigkeit der Schwungmasse zum Aufspringen durch das Event *EV_SET_JUMP_VELOCITY* gesetzt werden. Andererseits können die Regelparameter konfiguriert werden. Die hierfür benötigten

Events hängen von der verwendeten Reglerstruktur ab und können somit noch nicht endgültig festgelegt werden.

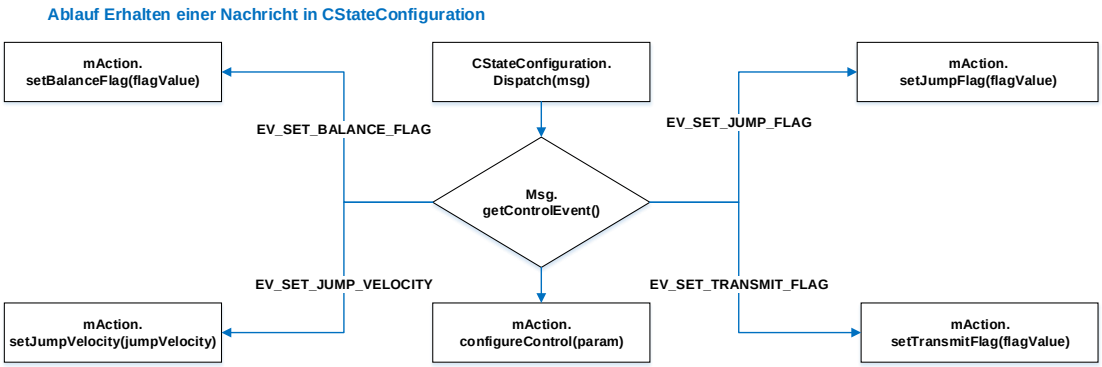

Abbildung 6: Ablaufplan CStateConfiguration, Quelle: eigene Darstellung

Aufgaben von *CStateRunning*

Die eigentliche Funktionalität der Steuerung steckt in dem Zustand *CStateRunning*. Bei Betreten dieses Zustandes wird ein Softtimer gestartet, welcher zyklisch Events an die Komponente schickt. Dadurch wird die gewünschte Abtastrate der Regelung realisiert. Erhält die FSM ein solches Timerevent werden die folgenden Aktionen ausgeführt.

In dem Zustand *CStateIdle* wird geprüft ob der Ausfallwinkel des Würfels sich im Regelungsbereich befindet. Gegebenenfalls wird das Event *EV_PHIK_IN_BALANCEAREA* in die interne Queue geschrieben.

In dem Zustand *CStateBalance* wird ebenfalls geprüft ob der Würfel sich noch im Regelungsbereich befindet. Falls dies gegeben ist wird ein Regelungszyklus durchgeführt. Hat der Würfel den vorgegebenen Bereich verlassen wird das interne Event *EV_PHIK_NOT_BALANCEAREA* ausgelöst.

In dem Zustand *CStateJump* wird nach dessen aktuellen Unterzustand entsprechend gehandelt. In dem Unterzustand *CStateWaiting* wird geprüft ob der Würfel die Ruhelage erreicht hat. Daraufhin wird das interne Event *EV_RESTING* und somit ein Wechsel in *CStateAccelerate* ausgelöst.

In dem Unterzustand *CStateAccelerate* wird ebenfalls die Ruhelage geprüft. Hat der Würfel diese verlassen so wird das interne Event *EV_NOT_RESTING* ausgelöst, woraufhin wieder in *CStateWaiting* gewechselt wird. Bleibt die Ruhelage erhalten wird geprüft ob die Schwungmasse die erforderliche Geschwindigkeit zum Aufrichten erreicht hat. Ist dies der Fall wird das interne Event *EV_JUMP_VELOCITY_REACHED* ausgelöst, welches den Übergang in *CStateBrake*.

In dem Unterzustand *CStateBrake* wird der Bremsvorgang überwacht. Bei Betreten des Zustandes wird die Bremse aktiviert und anschließend geprüft ob die Schwungmasse zur Ruhe gekommen ist. Daraufhin wird das interne Event *EV_BRAKE* ausgelöst, welches zum Verlassen des Zustandes *CStateJump* und somit zum deaktivieren der Bremse führt.

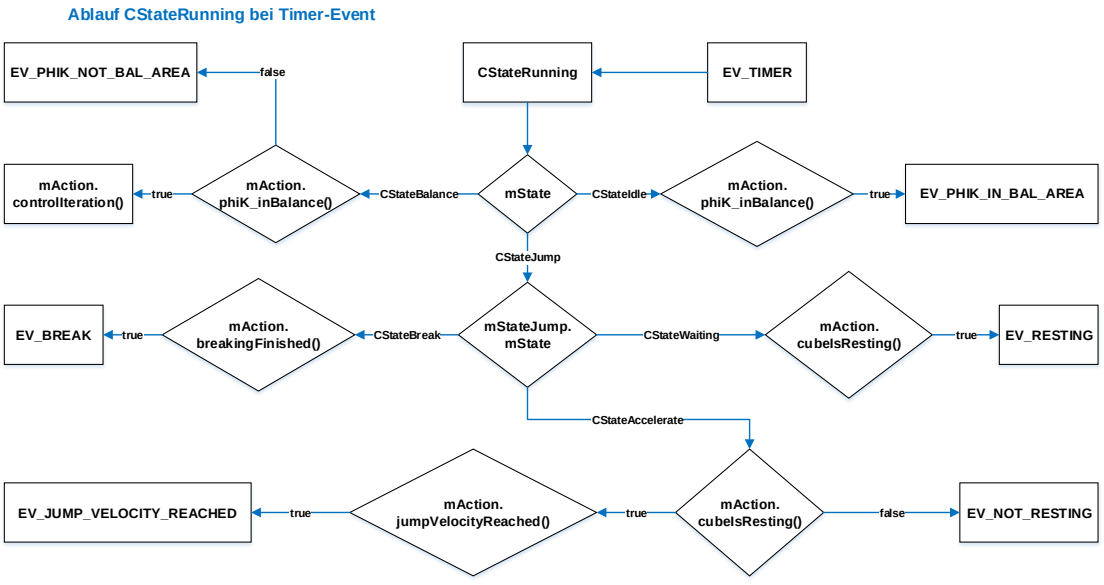

Abbildung 7: Ablaufplan CStateRunning, Quelle: eigene Darstellung

Aktionen von *CControlActionHandler*

Die auszuführenden Aktionen werden in der Klasse *CControlActionHandler* gekapselt. Es werden Methoden für Aktionen bei Betreten und Verlassen von Zuständen implementiert. Auf Übergangsfunktionen wird (vorläufig) verzichtet. Außerdem werden weitere Hilfsmethoden eingeführt, welche Aktionen auslösen die nicht an Zustandsübergänge gekoppelt sind.

Hierfür verfügt die Klasse über Instanzen des Reglers und der Klassen zur Hardwareansteuerung. Zusätzlich hält er eine Referenz auf *SSoftTimer* um diesen zu starten bzw. zu stoppen. Die oben angesprochenen Flags zur Kontrolle der Funktionalität werden ebenfalls von *CControlActionHandler* verwaltet.

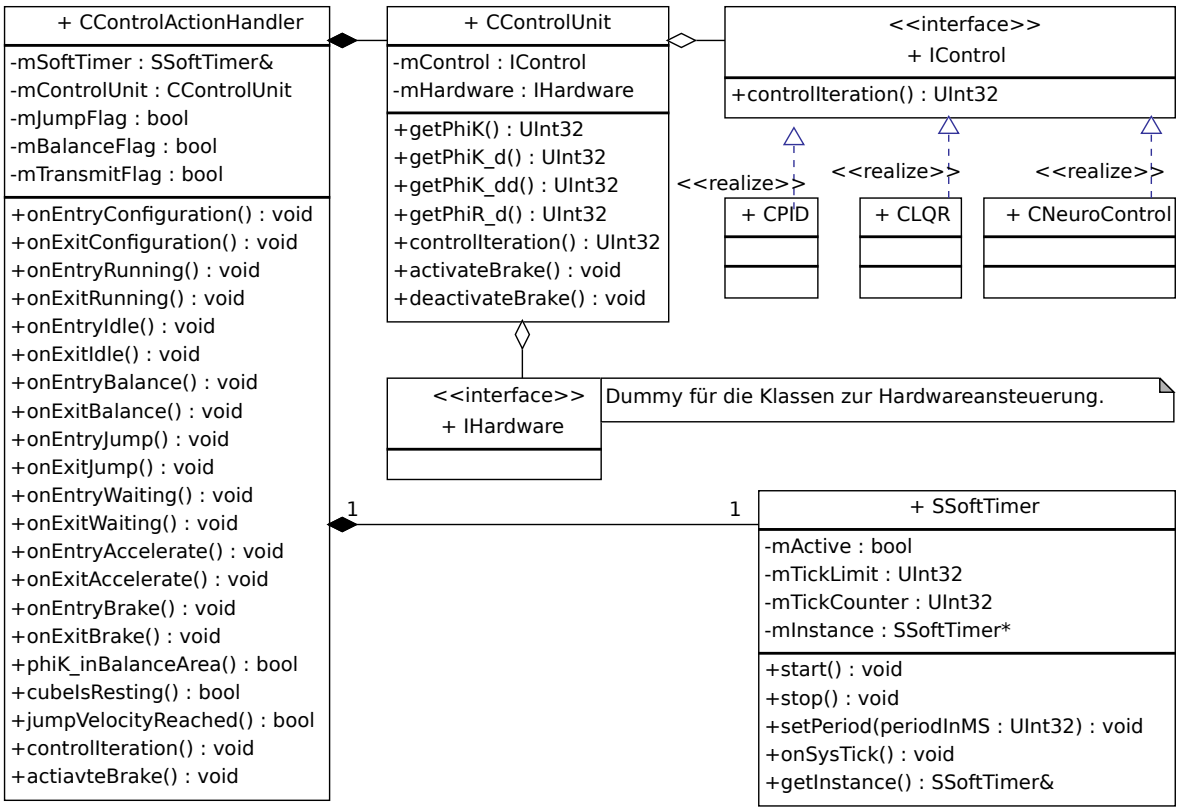

Abbildung 8: Klassendiagramm ControlActionHandler, Quelle: eigene Darstellung

Der Softtimer wird an den SysTimer gekoppelt, bei erreichen des Zählerlimit wird das Event *EV_TIMER* ausgelöst. Dadurch wird die periodische Abtastung des Regelsystem sicher gestellt. Mit Hilfe der Klassen zur Hardwareansteuerung können die Sensorwerte abgefragt und ausgewertet werden. Bei Bedarf werden entsprechende Events generiert, welche wiederum Zustandsübergänge zur Folge haben.

Die einzelnen folgende Tabelle zeigt die Methoden von *CControlActionHandler* und deren Aktionen. Aus Gründen der Übersicht und Einheitlichkeit werden auch Methoden ohne Aufgabe implementiert bzw. aufgerufen.

Methode von CControlAction	**Auszuführende Aktion**
void onEntryConfiguration()	-
void onExitConfiguration()	-
void onEntryRunning()	Softtimer starten
void onExitRunning()	Softtimer stoppen
void onEntryIdle()	Softtimerfrequenz anpassen
void onExitIdle()	-
void onEntryBalance()	Softtimerfrequenz anpassen
void onExitBalance()	-
void onEntryJump()	Softtimerfrequenz anpassen
void onExitJump()	Prüfen ob Bremse deaktiviert ist
void onEntyWaiting()	-
void onExitWaiting()	-
void onEntryAccelerate()	Motormoment setzen
void onExitAccelerate()	Motor stoppen
void onEntryBrake()	Bremse aktivieren
void onExitBrake()	Bremse deaktivieren
bool phiK_inBalanceArea()	Prüft ob der Würfel im Regelbereich
bool cubeIsResting()	Prüft ob der Würfel in Ruhelage
bool jumpVelocityReached()	Prüft ob die Sprunggeschwindigkeit erreicht
void controlIteration()	Führt einen Regelzyklus aus

6 Aufbau der Kommunikationskomponente

Die Aufgabe der Kommunikationskomponente besteht darin einerseits Befehle der Bedieneinheit anzunehmen und andererseits aktuelle Systeminformationen zu übermitteln. Die Übertragung der Daten erfolgt mittels Bluetooth und dem Serial Port Profile. Zur Ansteuerung des Bluetooth Modul wird eine serielle Schnittstelle verwendet. Folglich müssen deren Interrupts verarbeitet werden und die einzelnen Byte zu Informationspaketen zusammengesetzt werden. Die Komponente verfügt über einen Puffer, der empfangene Datenpakete enthält. Diese müssen von der Kommunikationseinheit in Nachrichten übersetzt werden. Der zweite Puffer speichert Nachrichten, welche in Form von Datenpaketen an die Bedieneinheit übermittelt werden müssen.

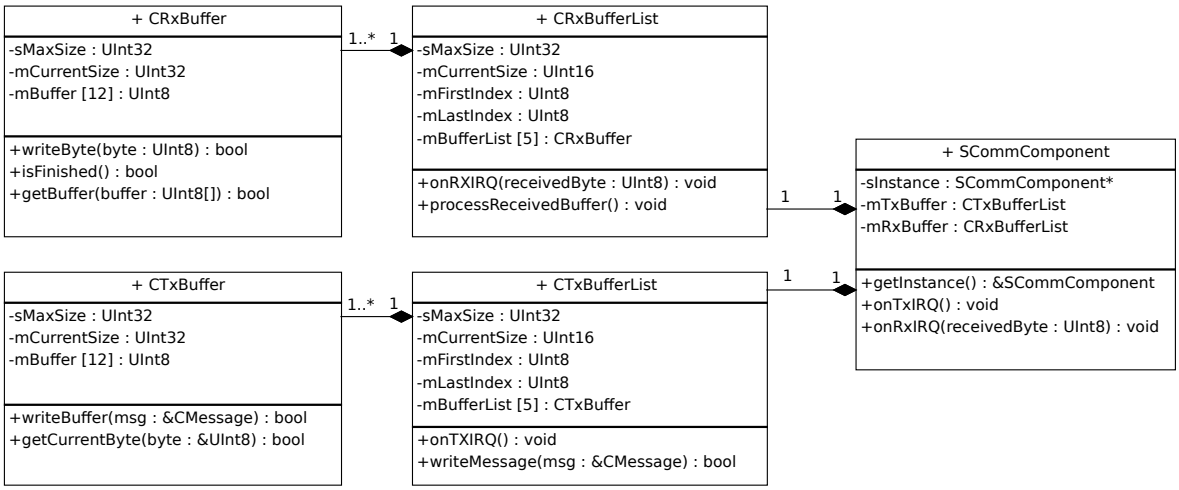

Abbildung 9: Klassendiagramm Comm-Komponente, Quelle: eigene Darstellung

Aufbau der Bluetoothpakete

Der Datenaustausch zwischen Würfel und Bedieneinheit erfolgt byteweise. Die einzelnen Bytes werden zu logischen Paketen zusammen gesetzt. Alle Pakete besitzen eine fixe, einheitliche Größe von 6 Bytes. Das erste Byte enthält die die ID des Empfängers, im zweiten Byte dient zur Speicherung des Ereignis bzw. Befehls. Die übrigen vier Bytes sind für zusätzliche Nutzdaten,wie z.B. Sensorwerte reserviert.

Der Inhalt eines empfangenen Paket wird durch entsprechenden Proxyaufruf in eine Nachricht umgesetzt. Der Aufbau der einzelnen Pakete wird in einer Tabelle dokumentiert, hierbei ist darauf zu achten bei Werten, die größer als ein Byte sind die Position des MSB und LSB zu notieren.

Tabelle 1: Aufbau der Datenpakete

Byte0	**Byte1**	**Byte2**	**Byte3**	**Byte4**	**Byte5**	**Anmerkung**
CONTROL_COMP	EV_START					
CONTROL_COMP	EV_STOP					
CONTROL_COMP	EV_SET_JUMP_FLAG	flagValue				
CONTROL_COMP	EV_SET_BALANCE_FLAG	flagValue				
CONTROL_COMP	EV_SET_TRANSMIT_FLAG	flagValue				
CONTROL_COMP	EV_JUMP					
CONTROL_COMP	EV_SET_PID_P_VALUE	P_Value(LSB)	P_Value	P_Value	P_Value(MSB)	float
CONTROL_COMP	EV_SET_PID_I_VALUE	I_Value(LSB)	I_Value	I_Value	I_Value(MSB)	float
CONTROL_COMP	EV_SET_PID_D_VALUE	D_Vaue(LSB)	D_Value	D_Value	D_Value(MSB)	float
CONTROL_COMP	EV_SET_JUMP_VELOCITY	velocity(LSB)	velocity	velocity	velocity(MSB)	SI-Einheit als float
COMM_COMP	EV_TRANSMIT_TORQUE	Torque(LSB)	Torque	Torque	Torque(MSB)	SI-Einheit als float
COMM_COMP	EV_TRANSMIT_PHI_K	PhiK(LSB)	PhiK	PhiK	PhiK(MSB)	SI-Einheit als float
COMM_COMP	EV_TRANSMIT_PHI_K_D	PhiK_d(LSB)	PhiK_d	PhiK_d	PhiK_d(MSB)	SI-Einheit als float
COMM_COMP	EV_TRANSMIT_PHI_K_DD	PhiK_dd(LSB)	PhiK_dd	PhiK_dd	PhiK_dd(MSB)	SI-Einheit als float
COMM_COMP	EV_TRANSMIT_BRAKE_STATE	brakeState				
COMM_COMP	EV_TRANSMIT_FSM_STATE_ENTRY	enteredState				
COMM_COMP	EV_TRANSMIT_UNHANDELD_EV	unhandeledControlEV				
COMM_COMP	EV_HARDWARE_ERROR	hardwareID				

Funktionsablauf des RX-Puffer

Falls ein RX-Interrupt anliegt wird zuerst geprüft ob der Puffer voll ist. Ggf. wird ein entsprechendes Fehler-Event erzeugt. Ansonsten wird das empfangene Byte in den aktuellen Puffer geschrieben. Falls dieses Datenpaket voll ist, wird das Event *EV_COM_RECEIVED* ausgelöst, welches die Verarbeitung des Datenpaket veranlasst.

Bei der Verarbeitung des Event *EV_COM_RECEIVED* wird zuerst geprüft ob ein Datenpaket vorhanden ist. Ggf. wir das erste Paket aus der FIFO-Liste entnommen und über den Proxy eine entsprechende Nachricht erzeugt.

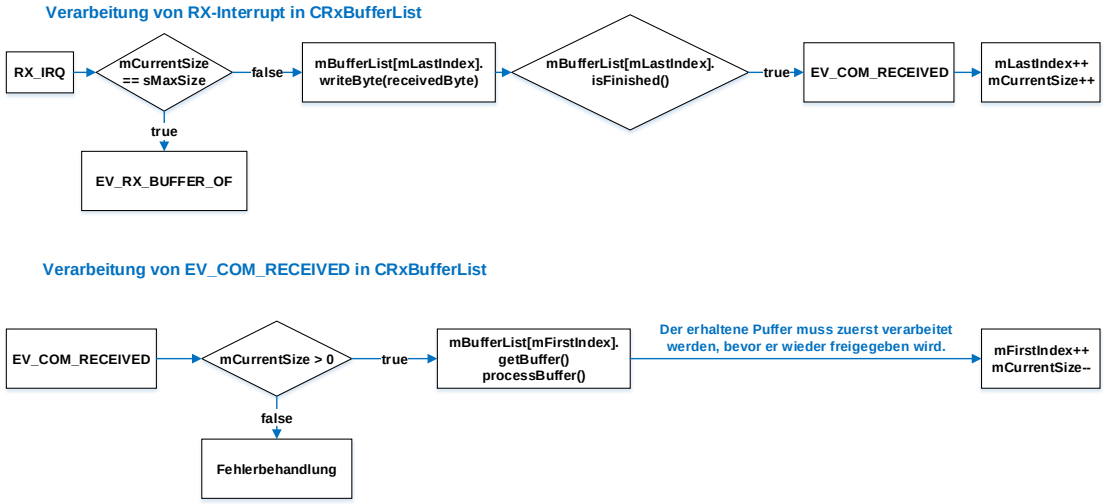

Abbildung 10: Ablaufplan RX-Puffer, Quelle: eigene Darstellung

Funktionsablauf des TX-Puffer

Die Nachrichtenvermittlung an die Bedieneinheit verläuft nach dem gleichen Prinzip. Bei dem Erhalten des Event *EV_TRANSMIT_MSG* wird geprüft ob der Puffer voll ist. Ggf. wird ein Fehlerevent produziert. Trifft dies nicht zu wird der Nachrichteninhalt in einen freien Datenpuffer geschrieben und der TX-Interrupt aktiviert.

Tritt ein TX-Interrupt auf wird ein Byte aus dem aktuellen Datenpuffer übermittelt. Anschließend wird geprüft ob der Datenpuffer vollständig übermittelt wurde. Ggf. wir dieser wieder freigegeben und falls keine weiteren Daten übermittelt werden müssen, wird der TX-Interrupt deaktiviert.

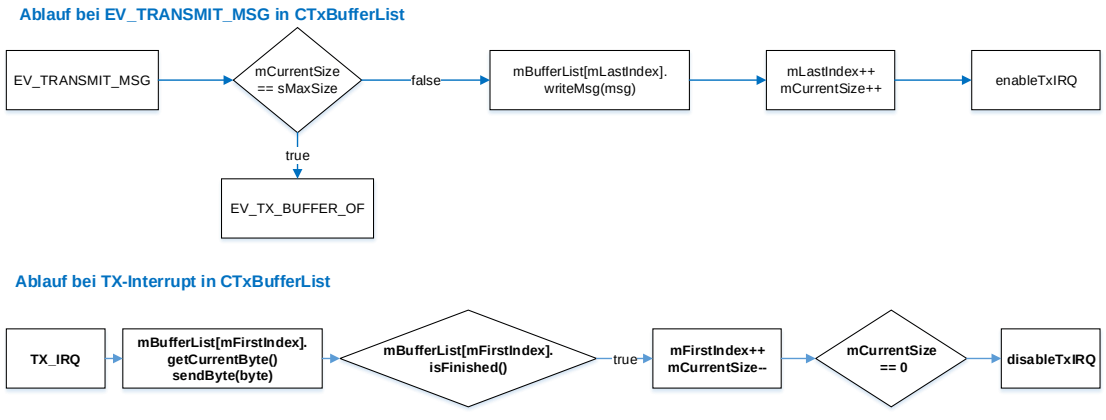

Abbildung 11: Ablaufplan TX-Puffer, Quelle: eigene Darstellung

7 Implementation des Reglers

In diesem Abschnitt wird die Umsetzung der Regler in Software erläutert. Die Regler benötigen als Eingangsgröße den Zustandsvektor x. Dieser besteht aus den Größen ϕ_K, $\dot{\phi_K}$ und $\dot{\phi_R}$. Außerdem muss die Winkelbeschleunigung $\ddot{\phi_K}$ zu Analysezwecken aufgezeichnet werden. Der Ausfallwinkel des Körpers und dessen zwei Ableitungen werden mit Hilfe von einem Magnetometer, Gyrometer und Beschleunigungssensor aufgenommen. Diese drei Sensoren sind auf einem IC verbaut und können über I2C angesteuert werden. Die aktuellen Sensorwerte werden als 16-Bit Ganzzahlen übergeben (signed unsigned?). Dem Datenblatt kann der einheitenbehaftete Faktor zur Umrechnung in die physikalische Größe entnommen werden.

7.1 Berechnung der Winkelbeschleunigung $\ddot{\phi_K}$

Der Dreiachs-Beschleunigungssensor gibt die Beschleunigung in x- und y-Richtung wieder. Daraus wird die Tangentialbeschleunigung des Sensors berechnet. Über den Abstand des Sensors zum Drehpunkt kann daraufhin die Winkelbeschleunigung des Körpers berechnet werden.

$$a_t := Tangentialbeschleunigung des Sensors$$

$$r_AS := Abstand des Sensors zum Drehpunkt$$

$$\ddot{x} := Beschleunigung in x - Richtung in \frac{m}{s}$$

$$\ddot{y} := Beschleunigung in y - Richtung in \frac{m}{s}$$

$$a_t = \sqrt{\ddot{x}^2 + \ddot{y}^2}$$

$$\ddot{\phi_K} = \frac{a_t}{r_AS}$$

7.2 Berechnung der Winkelgeschwindigkeit $\dot{\phi_K}$

Der Gyrosensor gibt die Winkelbeschleunigung in Grad pro Sekunde wieder. Dementsprechend einfach kann die Umrechnung in SI-Einheiten durchgeführt werden.

$$\dot{\phi_{K,S}} := Winkelgeschwindigkeit in Grad pro Sekunde$$

$$\dot{\phi_K} = \frac{2 * \pi}{360} \cdot \dot{\phi_{K,S}}$$

7.3 Berechnung des Ausfallwinkels ϕ_K

Der Ausfallwinkel des Körpers wird mit Hilfe eines Magnetometer bestimmt. Über die I2C versendet der Sensor die magnetische Flussdichte in x- und y-Richtung. Daraus wird der Winkel des Sensor und somit des Körpers berechnet.

ATan von x und y gibt den Winkel, wenn richtig kalibriert kann das sogar funktionieren, ausprobieren mit arduino oder so

8 Testen der Software

Die Software muss geprüft werden um den problemlosen Einsatz zu garantieren. Hierbei muss zwischen den Hardwareansteuerung, der Validierung des Reglers und der Komponentenstruktur bzw. Logik unterschieden werden. Die Softwareteile zur Ansteuerung der Hardware muss einzeln geprüft werden bevor sie in das Gesamtsystem eingehängt werden. Diese Tests erfolgen sowohl manuell als auch automatisch.

Die Validierung des Reglers erfordert einerseits die Prüfung der Implementierung. D.h. der Regler muss den Eingangsgrößen entsprechende Werte generieren. Andererseits muss der verwendete Regler mit den Anforderungen an den Regelkreis abgeglichen werden. Hierzu wird das *mTransmit-Flag* der Regelkomponente aktiviert. Daraufhin werden alle Sensor- und Reglerwerte an die Bedieneinheit übermittelt. Folglich kann die Überprüfung der oben genannten Kriterien vollautomatisch durchgeführt werden.

Die Prüfkriterien der Komponentenstruktur beinhalten die Erzeugung von Nachrichten und deren Verteilung, die Kodierung und Dekodierung von Kommunikationspaketen sowie die Zustandsübergänge und deren Folgeaktionen. Bei entsprechender Einstellung überträgt die Kontrollkomponente ihren aktuellen Zustand an die Bedieneinheit. Zusätzlich wird die Ansteuerung der Sensoren durch eine serielle Schnittstelle ersetzt. Somit können Blackbox-Tests vollautomatisch durchgeführt werden. Auftretende Fehler müssen manuell per Debugger zurückverfolgt werden.

Reglerersatz für Softwaretests

Um die Komponentenarchitektur, deren Nachrichtenstruktur und die Logik der Zustandsmaschine zu testen, wird der Regler und die Sensorik durch eine serielle Schnittstelle ersetzt. Somit ist es möglich automatisiert beliebige Prüfabläufe durchzuführen und die Softwarefunktionalität gezielt zu überprüfen.

CuBa-Anfrage			**Rx-Reaktion**	
REQUEST_PHI_K			phiK(LSB)	phiK(MSB)
REQUEST_PHI_K_D			phiK_d(LSB)	phiK_d(MSB)
REQUEST_PHI_K_DD			phiK_dd(LSB)	phiK_dd(MSB)
REQUEST_PHI_R_D			phiR_d(LSB)	phiR_d(MSB)
SET_MOTOR_TORQUE	Torque(LSB)	Torque(MSB)		
SET_BRAKE_STATE	brakeState			

Literatur

[WieTra] Joachim Wietzke, Manh Tien Tran: Automotive Embedded Systeme

[Bar] Richard Barry: Using the FreeRTOS Real Time Kernel - A Practical Guide