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 FreeRTOS Betriebssystem verwendet. In den folgenden Abschnitten werden die Aufgaben in Teilfunktion untergliedert und deren Implementation erörtert.

Namens- 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

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].

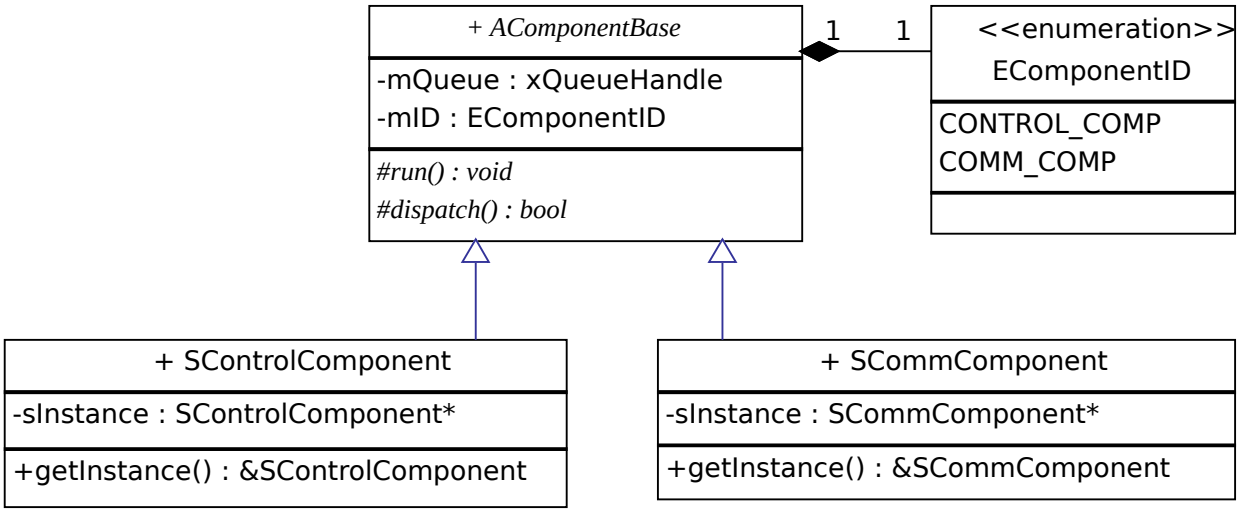
1 Eine Ausnahme bilden Prozessor-Faults (z.B. Memory-Bus-Fault), diese besitzen die höchste Interruptpriorität, welche für derartige Faults reserviert ist.

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.

Aufbau der Kontrollkomponente

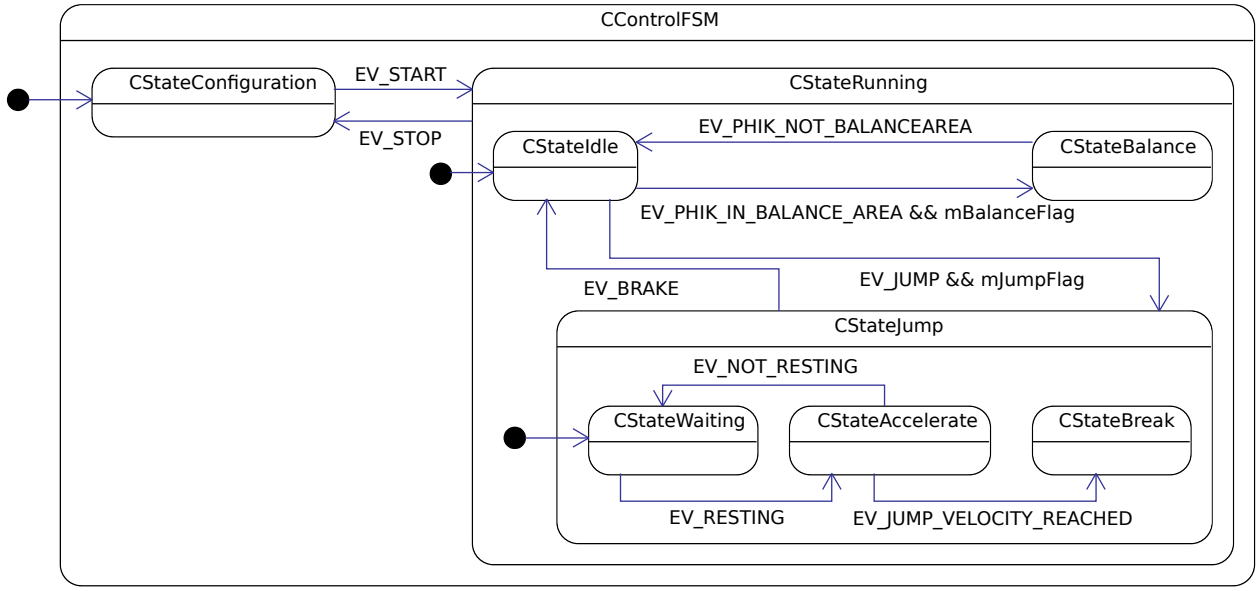
Die Aufgabe der Kontrollkomponente besteht darin, den Würfel aufzurichten und in der gewünschten Gleichgewichtslage zu balancieren. Hierfür müssen die Sensoren ausgewertet und die Motoren entsprechend angesteuert werden. Die Regelungseinheit muss eine Schnittstelle bieten um die Parameter über die Bedieneinheit 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.

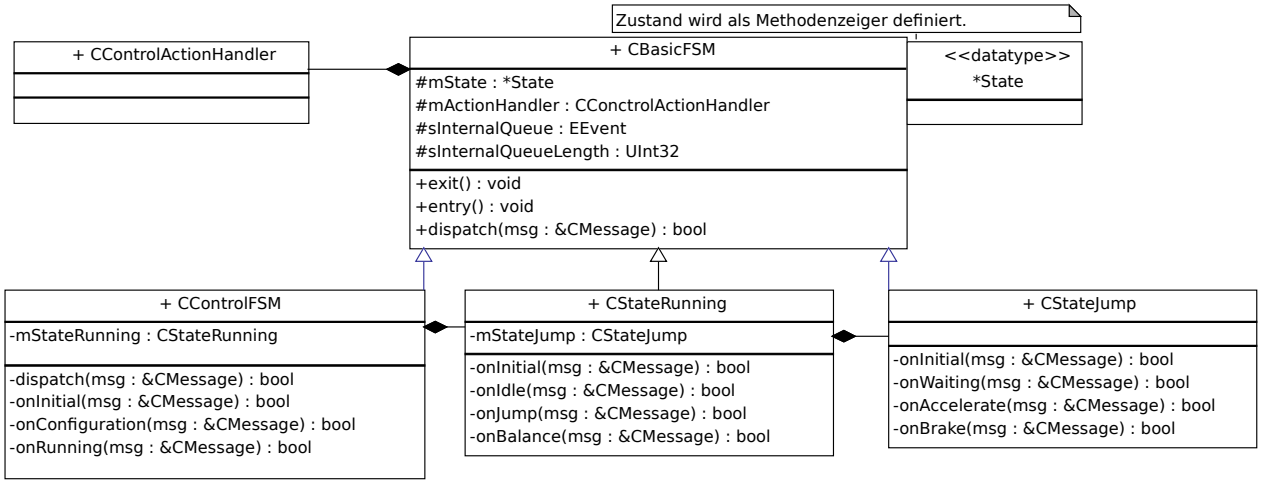
Eventkodierung	**Zusätzliche Parameter**	**Kommentar**
EV_START		Startet die Steuerung
EV_STOP		Stoppt die Steuerung
EV_SET_JUMP_FLAG	FlagValue	Setzt den Wert des JumpFlag
EV_SET_BALANCE_FLAG	FlagValue	Setzt den Wert des BalanceFlag
EV_SET_TRANSMIT_FLAG	FlagValue	Setzt den Wert des TransmitFlag
EV_SET_JUMP_VELOCITY	VelocityValue	Setzt die Geschwindigkeit der Schwungmasse zum Aufspringen

EV_SET_PID_VALUES	K_I, K_P, K_D	Setzt Verstärkungsfaktoren des Regler
EV_JUMP		Leitet das Aufspringen ein
EV_TIMER		Timerevent, realisiert Abtastzeit
EV_BRAKE		Internes Event, informiert über Betätigung der Bremse
EV_PHIK_IN_BALANCEAREA		Internes Event, informiert über betreten des Regelungsbereich
EV_PHIK_NOT_BALANCEAREA		Internes Event, informiert über verlassen des Regelungsbereich
EV_RESTING		Internes Event, Würfel in Ruheposition
EV_NOT_RESTING		Internes Event, Würfel nicht in Ruhe
EV_JUMPVELOCITY_REACHED		Internes Event, bereit zum Aufspringen

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.



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.



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.

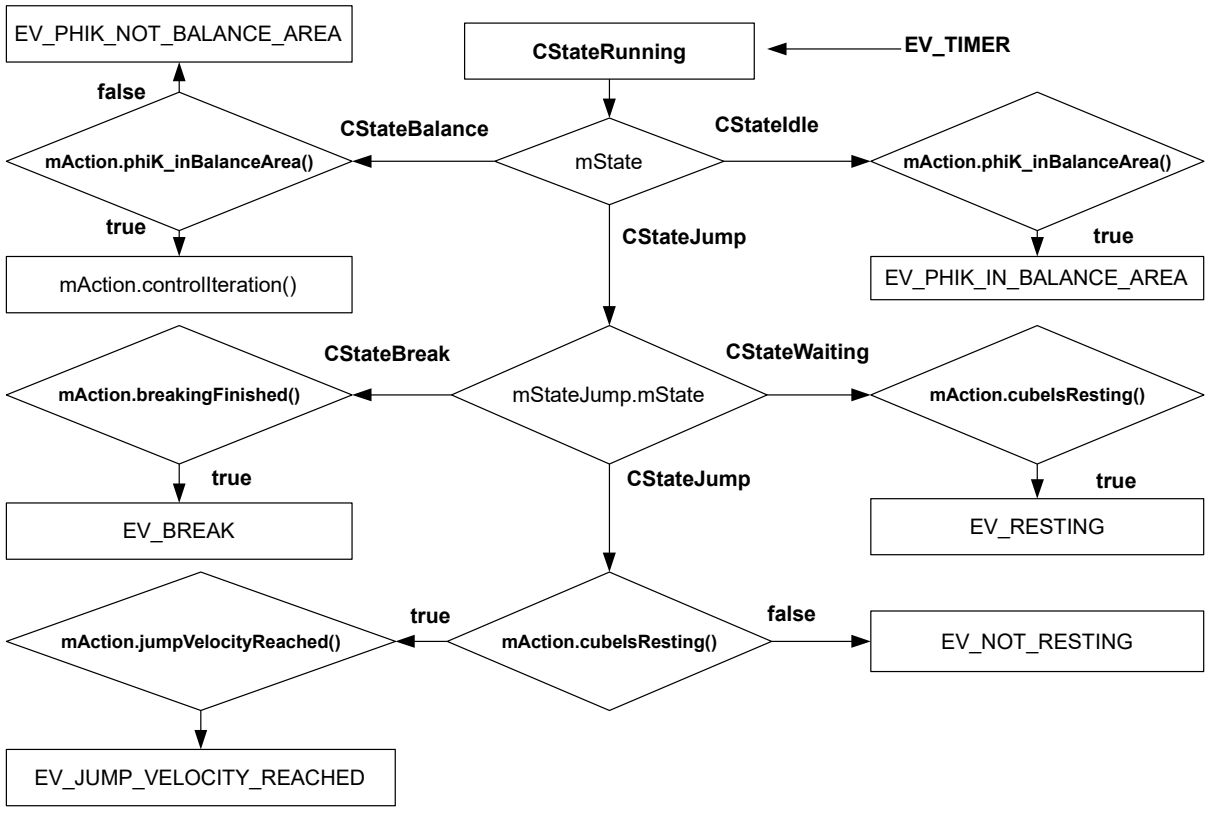
Methode von *CcontrolActionHandler*	**Auszuführende Aktion**
void onEntryConfiguration()	-
void onExitConfiguration()	-
void onEntryRunning()	Softtimer starten
void onExitRunning()	Softtimer stoppen
void onEntryIdle()	Softtimerfrequenz anpassen
void onExitIdle()	-
void onEntryBalance()	Softtimerfrequenz 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 ist
bool cubeIsResting()	Prüft ob der Würfel in Ruhelage ist
bool jumpVelocityReached()	Prüft ob die Sprunggeschwindigkeit erreicht ist
void controlIteration()	Führt einen Regelzyklus aus

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 im Regelungsbereich befindet. Gegebenenfalls wird das Event *EV_PHIK_IN_BALANCEAREA* in die interne Queue geschrieben.

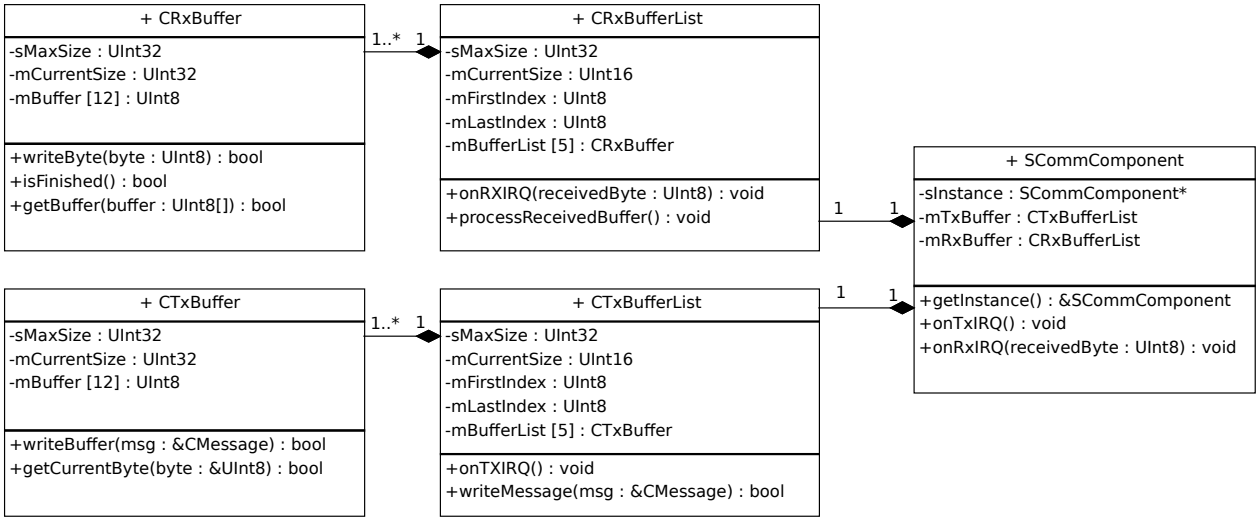
In dem Zustand *CStateBalance* wird ebenfalls geprüft ob der Würfel sich noch im Regelungsbereich befindet. Falls dies gegeben ist wird ein Regelungszyklus durchgeführt. Hat der Würfel den vorgegebenen Bereich verlassen wird das interne Event *EV_PHIK_NOT_BALANCEAREA* ausgelöst.

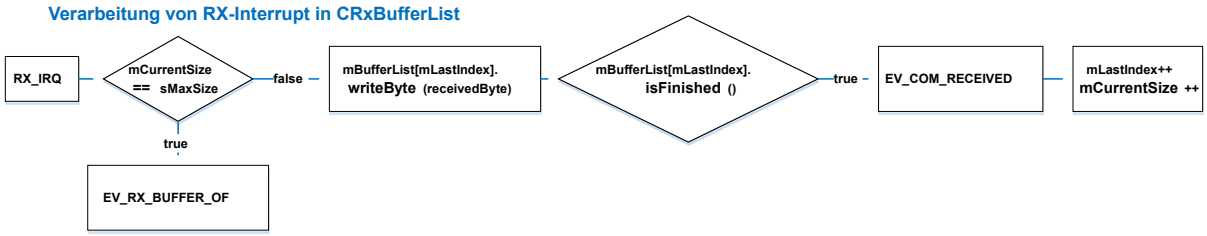
In dem Zustand *CStateJump* wird nach dessen aktuellen Unterzustand entsprechend gehandelt. In dem Unterzustand *CStateWaiting* wird geprüft ob der Würfel die Ruhelage erreicht hat. Daraufhin wird das interne Event *EV_RESTING* ausgelöst.
In dem Unterzustand *CStateAccelerate* wird ebenfalls die Ruhelage 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.
In dem Unterzustand *CStateBreak* wird der Bremsvorgang überwacht. Bei Betreten des Zustandes wird die Bremse aktiviert. In diesem wird geprüft ob die Schwungmasse zum Stillstand gekommen ist. Daraufhin wird das interne Event *EV_BRAKE* ausgelöst, welches zum Verlassen des Zustandes *CStateJump* und somit zum deaktivieren der Bremse führt.

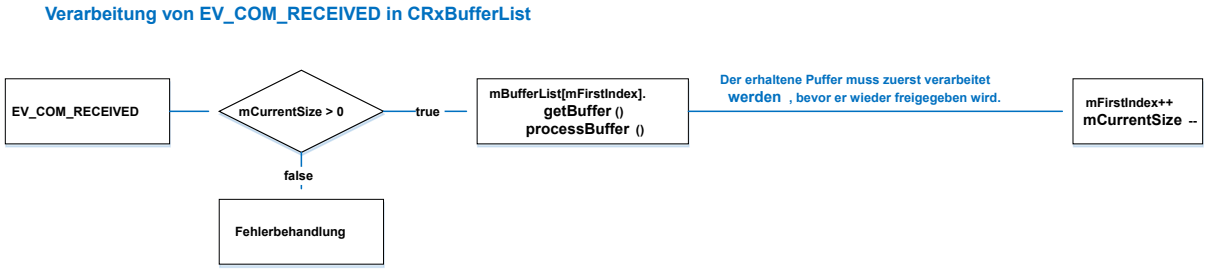


Aufbau der Kommunikationskomponente

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.



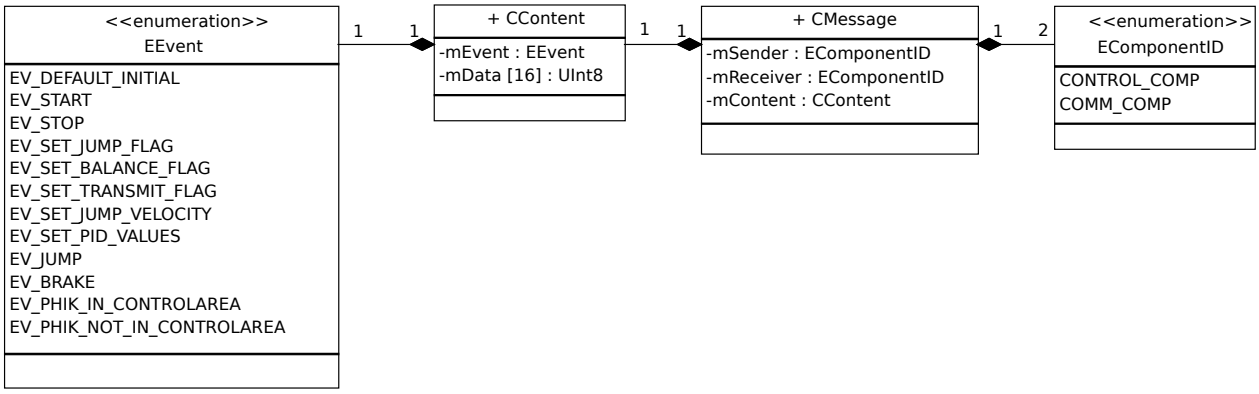




Falls ein RX-Interrupt anliegt

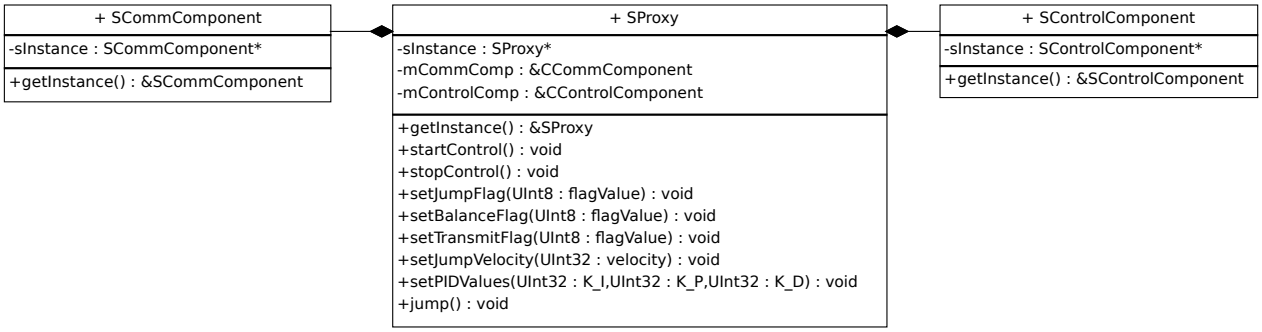
Kommunikation mittels Nachrichtenschlange

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*, welche eine Instanz der Klasse *CContent* enthält. Diese Instanz gibt die Informationen über das Ereignis wider.



Erzeugung 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.



Quellenverzeichnis:

- Automotive Embedded System; Joachim Wietzke, Manh Tien Tran
- Using the FreeRTOS Real Time Kernel – A Practical Guide; Richard Barry