Kapitel 1

Software

Wie bereits in den letzten Abschnitten erläutert wird der Regler als diskretes System implementiert. Deshalb spielt die verwendete Hardware und darauf ausgeführte Software eine zentrale Rolle. Dieser Umstand wird dadurch verstärkt, dass sämtliche Versuche, wie z.B. die Justierung der Sensoren, Systemidentifikation und Erprobung verschiedener Reglerkonzepte, in Form eines Programms durchgeführt werden. Aus diesem Grund widmet sich dieser Abschnitt dem Aufbau einer Software-Infrastruktur, welche die effiziente Entwicklung von mechatronischen Anwendungen ermöglicht.

1.1 Zielplattform, Sensorik und Aktorik

In diesem Projekt wird ein BeagleBone Black[1] ([?]) in Kombination mit einer Linux-Distribution verwendet um die digitale Regelung zu realisieren. Die Plattform basiert auf einem AM335x Sitara Prozessor ([?]), der mit einer Taktrate von 1GHz betrieben wird. Des weiteren steht eine single precision NEON FPU, für die Berechnung von Gleitkommaoperationen, zur Verfügung. Diese Plattform reduziert die nötige Rechenzeit für gewöhnliche Filter- und Regelungsalgorithmen auf wenige Mikrosekunden. Somit kann die durch die Berechnung resultierende Totzeitbei dem Reglerentwurf vernachlässigt werden. Das Linux-Betriebssystem bringt weitere Vorteile für die Entwicklung des Gesamtsystems mit sich. Zunächst existiert eine Vielzahl von Werkzeugen für die Entwicklung von Embbeded-Linux-Anwendungen. Dadurch wird der nötige Zeitaufwand für die Implementierung des Reglerprogramms reduziert. Des weiteren kann bei der Entwicklung auf Pakete und Bibliotheken der Linux-Gemeinde zurückgegriffen werden. Somit können auch komplexe Subsysteme, wie z.B. der hier verwendete TCP/IP-Server, in das Gesamtsystem eingebettet werden. Zuletzt kann das Dateisystem genutzt werden um Konfigurationen für Filter- und Regleralgorithmen auszutauschen, wodurch die Erprobung von verschiedenen Reglerkonzepten vereinfacht wird. Allerdings muss der Einfluss des Betriebssystem auf das Zeitverhalten der Regelung kritisch betrachtet werden. Einerseits kann die Abtastung an äquidistanten Stützstellen nicht mehr garantiert werden. Anderseits entsteht durch die Verwendung von Linux-Treibern Verzögerungen, die zu weiteren Totzeiten führen.

Aus diesem Grund wird zunächst die verwendete Peripherie und deren softwareseite Auswertung vorgestellt. Auf dem Würfelgehäuse sind insgesamt sechs MPU9250-Module ([?]) montiert. Die Module besitzen jeweils einen Beschleunigungs- und Drehratensensor, welche genutzt werden um einen Teil des Zustandvektors zu bestimmen. Die Kommunikation zwischen den Sensormodulen und dem BeagleBone Black erfolgt über einen SPI-Bus. Das das SPI-Modul des BeagleBone Black nur einen CS-Pin besitzt, wird dieser über einen Analogschalter ([?]) mit den Sensoren verbunden. Drei digitale Ausgänge des BealgeBone sind

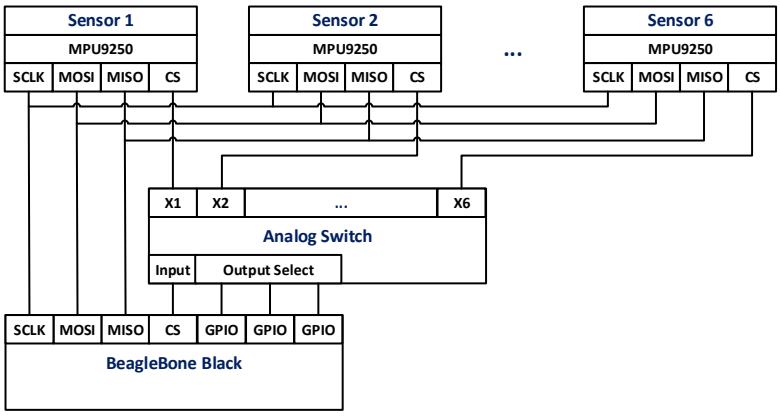

Abbildung 1.1: Blockschaltbild Sensorkommunikation

mit den Steuereingängen des Schalters verbunden um die CS-Leitung auf den Ausgang des

[1] Im weiteren wird die Kurzform BeagleBone bzw. die Abkürzung BBB verwendet.

gewünschten Sensors zu legen. Im Quellcode werden die Peripheriegeräte durch Klassen repräsentiert. Für die Steuerung der Digitalausgänge wird die Klasse *CGPIO* implementiert, welche als Konstruktorargument die Nummer des zu konfigurierenden Pins entgegennimmt. Des weiteren bietet sie Methoden zum Setzen bzw. Rücksetzen des Ausgangs. Hierfür wird die Schnittstelle des Treibers im Dateisystem verwendet. Zur Steuerung des Schalters wird die Klasse *CSwitch* aus drei Instanzen des Typs *CGPIO* komponiert und implementiert die Methoden *selectXi()* um die Schalterausgänge auszuwählen. Analog wird für die Konfiguration und Auswertung der Sensoren die Klasse *CMPU9250* implementiert. Zur Interaktion mit dem SPI-Treiber wird die Posix-Funktion *ioctl()* verwendet, welche zu einer kürzeren Ausführungszeit der Treiberaufrufe führt und im Vergleich zu der Dateisystemschnittstelle detaillierte Konfigurationsoptionen bietet. Die Klasse nutzt die SPI-Schnittstelle um die Sensoren in der *init()*-Methode zu konfigurieren. Anschießend können über die Methode *fetchData()* die aktuellen Beschleunigungs- und Winkelgeschwindigkeitswerte ausgelesen werden.

Die letztendliche Anwenderschnittstelle bietet die Klasse *CSensorSystem*, welche aus einer Instanz des Typ *CSwitch* und *CMPU9250* komponiert wird. Im Konstruktur werden die sechs Sensoren initialisiert und anschließend über die Methode *fetchSensorData()* ausgelesen. Die Daten werden in der Struktur *SSensorData* gespeichert, welche Membervariablen für die Messwerte besitzt.

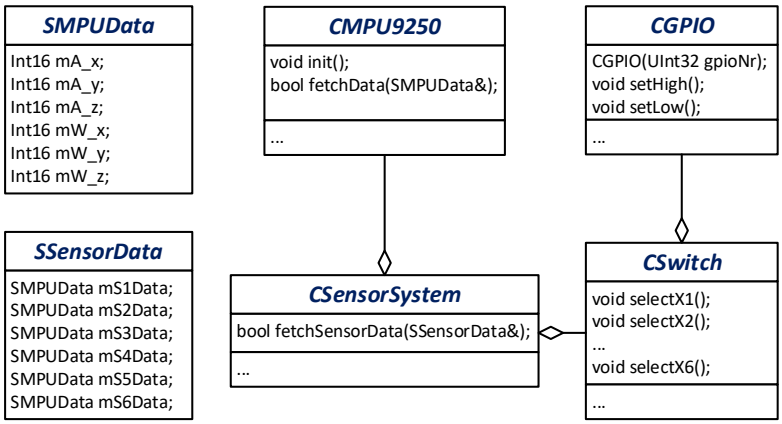

Abbildung 1.2: Klassendiagramm der Sensorschnittstelle

Die Stellgrößen der Regelung werden durch drei Motoren generiert, die jeweils über einen Treiberbaustein kontrolliert werden. Jeder Motortreiber ist mit zwei digitalen Ausgängen verbunden. Diese steuern die Freigabe und Drehrichtung des Motors. Die Vorgabe des Drehmoments erfolgt über ein PWM-Signal. Zusätzlich erfassen die Motortreiber die Drehzahl und geben diese in Form eines Analogsignals an das BeagleBone zurück. Die Klasse *CP-*

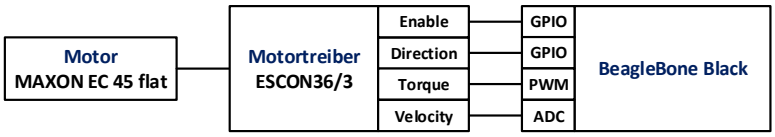

Abbildung 1.3: Blockschaltbild der Motoransteuerung

WM ermöglicht das Setzen eines Duty-Cycles, wobei die Treiberschnittstelle im Dateisystem verwendet wird. Eine Instanz dieser Klasse wird in *CMotor* genutzt um das gewünschte Drehmoment einzustellen. Des weiteren werden zwei Instanz von *CGPIO* genutzt um die Freigabe und Drehrichtung des Motors einzustellen. Für das Auslesen der ADC-Werte wird ebenfalls eine Klasse *CADC* angelegt. Da der Linux-Treiber für die Nutzung der AD-Wandler teilweise fehlerhaft ist wird ein alternative Vorgehensweise zur Nutzung der Peripherie genutzt. Hierbei wird mittels der Posix-Funktion *mmap()* der Adressbereich der ADC-Peripherie in den Userspace gelegt. Dadurch kann in der Anwendung direkt auf die ADC-Register zugegriffen werden. Durch diesen Ansatz ergeben sich zwei Vorteile. Einerseits werden dem Nutzer keine Einschränkungen durch die Treiberschnittstelle aufgezwungen. Andererseits wird die nötige Zeit der AD-Wandlung durch den direkten Zugriff auf die Register reduziert. Allerdings ist diese Vorgehensweise mit einem größeren Implementierungsaufwand verbunden und wird deshalb nur genutzt wenn die Einschränkungen des Treibers nicht annehmbar sind. die Klasse *CSensorSystem* wird um eine Instanz von *CADC* erweitert und umfasst somit die vollständige Sensorik des Systems. Um die Peripherie vollständig zu kapseln wird die Klasse *CHardware* aus einer Instanz von *CMotor* und *CSensorSystem* komponiert.

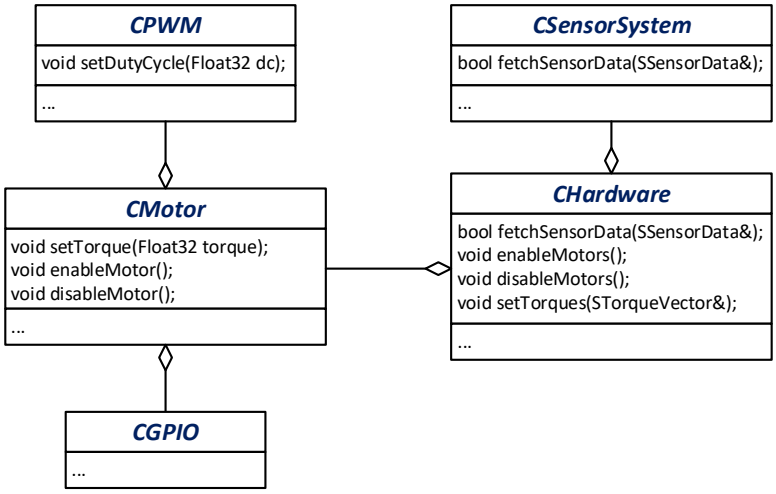

Abbildung 1.4: Klassendiagramm der Hardwareansteuerung

1.2 Implementierung des Signalflusses

Im nächsten Schritt muss der Signalfluss implementiert werden. Hierfür wird ein Ansatz der Template-Metaprogrammierung verwendet, um ein einheitliches Konzept für die Umsetzung von Singalverarbeitungsalgorithmen zu schaffen. Ebenso soll das Konzept Änderung im Singalfluss ermöglichen, ohne dabei größere Eingriffe im Quellcode vornehmen zu müssen. Als Beispiel wird das folgende Blockschaltbild verwendet. Zunächst wird eine, auf den Sensorwerten basierende, Zustandsschätzung durchgeführt. Dieser Vektor wird anschließend gefiltert und zur Berechnung des Reglers genutzt.

Abbildung 1.5: Blockschaltbild des Signalflusses

Für die Implementierung werden die Signale als Datenobjekte implementiert. Das heißt es werden Strukturen oder Klassen entworfen, welche die Daten enthalten und ggf. Methoden für den Zugriff oder Bearbeitung bieten. Für dieses Beispiel repräsentieren die Strukturen *SSensorData* und *SStateVector* den Sensor- bzw. Zustandsvektor. Für die Modellierung der Stellgröße genügt eine *float*-Variable.

```
 1  struct SSensorData
 2  {
 3    Float32 mS1;
 4    Float32 mS2;
 5    Float32 mS3;
 6  };
 7  struct SStateVector
 8  {
 9    Float32 mX1;
10    Float32 mX2;
11  };
12  using UType = Float32;
```

Listing 1.1: Beispielhafte Implementierung eines Datenobjektes

Die Systeme im Signalfluss werden als Klassen implementiert, die als Aktionsobjekte bezeichnet werden. Die Klassen werden nach dem folgenden Schema entworfen, wobei als Beispiel ein Objekt zur Zustandsschätzung aufgezeigt wird.

```
 1  class CStateEstimate
 2  {
 3  public:
 4    using InputType  = SSensorData;
 5    using OutputType = SStateVector;
 6  public:
 7    const OutputType& calcOutput(const InputType& input);
 8    const OutputType& getValue() const;
 9    ...
10  private:
11    OutputType mOutput;
12  };
```

Listing 1.2: Beispielhafte Implementierung eines Aktionsobjektes

Die Aktionsobjekte definieren zunächst ihren Ein- und Ausgangstyp. Die Berechnung des Systems erfolgt über die Methode *calcOutput()*. Nun müssen die Aktionsobjekte zu dem vorgegebenen Signalfluss zusammengefasst werden. Um diesen Schritt im Entwicklungsprozess zu erleichtern, soll ein Konzept implementiert werden, dem eine Typenliste der Aktionsobjekte übergeben wird und daraus das Blockschaltbild erzeugt. Für die Implementierung der Typenliste wird der Ansatz nach ([?], S. 40 ff.) verwendet. Zunächst wird die leere Struktur *CNullType* definiert, welche als Terminierungssymbol in der Typenliste fungiert. Die Liste wird mit Hilfe der Templatestruktur *TTypeList* implementiert, welche als Parameter einen Anfangs- und Endtypen entgegennimmt.

```
1  struct CNullType{};

3  template<class HeadType, class TailType>
4  struct TTypeList
5  {
6    using Head = HeadType;
7    using Tail = TailType;
8  };
```

Listing 1.3: Implementierung der Typenlist

Die obige Implementierung unterstützt lediglich Listen der Länge zwei. Deshalb werden Makros verwendet ([?], S. 45), die rekursive Intanszierungen des Templates nutzen um die Listen beliebiger Länge zu erhalten. Mittels dieser Makros kann dann auch die Typenliste der Aktionsobjekte erzeugt werden.

```
1  #define TYPELIST_1(T1)          TTypeList<T1, CNullType>
2  #define TYPELIST_2(T1, T2)      TTypeList<T1, TYPELIST_1(T2)>
3  #define TYPELIST_3(T1, T2, T3) TTypeList<T1, TYPELIST_2(T2, T3)>
4  ...
5  using ActionObjList = TYPELIST_3(CStateEstimate, CFilter, CController);
```

Listing 1.4: Definition und Aufruf der Makors für verlängerte Typenlisten

Um nun die Aktionstypen zu einem Signalfluss-Objekt zusammenzufügen, wird das Muster der linearen Typenhierachie nach ([?], S. 62 ff.) angewandt. Dessen Aufbau erinnert an eine verkettete Liste, wobei die Typen durch Vererbung verknüpft werden. Die Templateklasse *TActionHolder* wird als Träger für die Aktionsobjekte entworfen. Im allgemeinen Fall wird dem Template der Typ eines Aktionobjektes *ActionObj* und eine beliebige Elternklasse *Base* übergeben, die beide an *TActionHolder* vererben. In der *calcOutput()*-Methode wird zunächst die Berechnung des Aktionsobjektes durchgeführt und anschließend *calcOutput()* der zweiten Elternklasse aufgerufen.

```
1  template<class ActionObj, class Base>
2  class TActionHolder : public Base, public ActionObj
3  {
4  public:
5    void calcOutput(const ActionObj::InputType& input)
6    {
7      ActionObj::calcOutput(input);
8      Base::calcOutput(ActionObj::getValue());
9    }
10 };
```

Listing 1.5: Templateklasse des Trägerobjektes

Des weiteren besteht eine Templatespezialisierung für den Fall, dass ein Aktionstyp und *CNullType*, der das Ende der Typenliste signalisiert, übergeben werden. Nun wird in der

calcOutput()-Methode lediglich das Aktionsobjekt berechnet, da das Ende der Typenliste und somit des Signalflusses erreicht ist.

```
 1 template<class ActionObj>
 2 class TActionHodler<ActionObj, CNullType> :
 3   public CNullType, public ActionObj
 4 {
 5 public:
 6   void calcOutput(const ActionObj::InputType& input)
 7   {
 8     ActionObj::calcOutput(input);
 9   }
10 };
```

Listing 1.6: Templatespezialisierung des Trägerobjektes für das Ende der Typenliste

Die zweite Templateklasse ist *TLinHierachy*, mit dem folgenden Prototyp.

```
1 template<class TList,
2          template<AtomicType, class Base> class Unit,
3          class Root = CNullType>
4 class TLinHierachy;
```

Listing 1.7: Deklaration der Templateklasse für lineare Hierarchien ([?], S. 63)

Der erste Templateparameter ist die Typenliste, welche die zu generierende Typenhierachie vorgibt. Der zweite Parameter ist eine Templateklasse, die als Träger der Objekte aus der Typenliste agiert. In diesem Anwendungsfall wird die zuvor definierte Templateklasse *TActionHolder* verwendet. Der letzte Parameter ist der Terminierungstyp der Hierarchie, welcher als Standardargument *CNullType* übergeben wird. Analog zu *TActionHolder* werden durch Spezialisierungen zwei Fälle unterschieden. Zunächst sei der Fall betrachtet, dass der Parameter *TList* eine Typenliste aus zwei beliebigen Typen ist.

```
1 template<class T1,
2          class T2,
3          template<class, class> class Unit,
4          class Root>
5 class TLinHierachy<TTypeList<T1, T2>, Unit, Root>
6   : public Unit<T1, TLinHierachy<T2, Unit, Root> >
7 {};
```

Listing 1.8: Erste Templatespezialisierung der linearen Hierarchie ([?], S. 64)

Diese Instanziierung wird solange genutzt bist das Ende der ursprünglichen Typenliste erreicht ist. Die instantiierte Klasse von *TLinHierachy*, ist eine leere Klasse die von *Unit* erbt. *Unit* wird wiederum mit *TLinHierachy* instantiiert, wobei lediglich der zweite Parameter *T2* übergeben wird. Dadurch wird die ursprüngliche Typenliste schrittweise abgearbeitet. Die zweite Templatespezialisierung wird genutzt um die Generation am Ende der Typenliste zu terminieren. Dies ist der Fall wenn *TLinHierachy* mit einer Typeliste der Länge eins instantiiert wird.

```
1 template<class T, template<class, class> class Unit, class Root>
2 class TLinHierachy<TYPELIST_1(T), Unit, Root>
3   : public Unit<T, Root>
4 {};
```

Listing 1.9: Zweite Templatespezialisierung der linearen Hierarchie ([?], S. 64)

Für das hier aufgeführte Beispiel ergibt sich die folgende Vererbungshierarchie. Der Vorteil

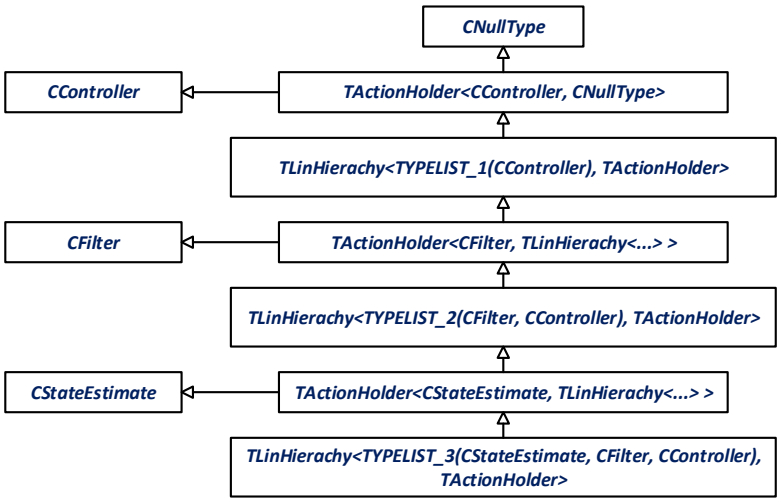

Abbildung 1.6: Klassendiagramm der generierten Hierarchie

dieses Konzept, welches mit einem nicht zu vernachlässigenden Programmieraufwand verbunden ist, zeigt sich bei der letztendlichen Nutzung. Sind die Aktionsobjekte definiert können sie in wenigen Zeilen zu dem Signalfluss zusammengesetzt werden.

```
1  using ActionList = TYPELIST_3(CStateEstimate, CFilter, CController);
2  using SignalFlow = TLinHierachy<ActionList, TActionHandler>;
3  SignalFlow mySF;
```

Listing 1.10: Anwendungsbeispiel der linearen Hierarchie

Sollen nun im Projektverlauf einzelne Elemente ausgetauscht oder erweitert werden muss lediglich die Typendefinition von *ActionList* geändert werden. Da die Hierarchie mittels Vererbung realisiert wird, können durch die Angabe des Namensraum die Methoden aller Elternklassen verwendet werden. Beispielsweise zeigt der folgende Ausschnitt die Abfrage aller berechneten Signale.

```
1  SStateVector x_estimate = mySF.CStateEstimate::getValue();
2  SStateVector x_filtered = mySF.CFilter::getValue();
3  UType        u          = mySF.CController::getValue();
```

Listing 1.11: Beispiel für den Zugriff auf Aktionsobjekte in der Hierarchie

Ebenso können zur Laufzeit konfigurierbare Elemente oder Verzweigungen implementiert werden. Angenommen zur Filterung sollen entweder ein Komplementär-Filter (*CCompFilter*) oder ein Tiefpass erster Ordnung (*CPT1*) genutzt werden. Dann kann eine Klasse *CFilterSystem* aus diesen beiden komponiert werden, die zusätzliche Methoden zur Filterauswahl bietet.

```
1  class CFilterSystem
2  {
3  public:
4    using InputType  = SStateVector;
5    using OutputType = SStateVector;
6  public:
7    const OutputType& calcOutput(const InputType& input)
8    {
9      mCompFilter.calcOutput(input);
10     mPT1Filter.calcOutput(input);
```

```
12      return this->getValue();
13    }
14    const OutputType& getValue()
15    {
16      switch(mActiveFilter)
17      {
18        case EFilter::CompFilter:
19          return mCompFilter.getValue();
20        case EFilter::PT1Filter:
21          return mPT1Filter.getValue();
22        default:
23          return input;
24      }
25    }
26    void setFilter(EFilter filter)
27    {
28      mActiveFilter = filter;
29    }
30  private:
31    CCompFilter mCompFilter;
32    CPT1        mPT1Filter;
33    EFilter     mActiveFilter;
34  };
```

Listing 1.12: Beispiel für die komponierte Aktionsobjekte

Analog können auch komplexere Verzweigungen realisiert werden, wobei *TLinHierachy* zur Generation von Teilzweigen des Blockschaltbildes verwendet werden kann.

1.3 Aufbau der Komponentenarchitektur

In dem letzten Abschnitt wurden die elementaren Funktionen des Regelkreises, wie die Peripherieinteraktion und der Signalfluss, implementiert. Um eine effiziente Versuchsdurchführung zu ermöglichen müssen allerdings weitere Funktionalitäten bereitstehen. Einerseits müssen einzelne Elemente des Regelkreises während der Ausführung konfigurierbar sein. Beispielsweise soll zwischen unterschiedlichen Regler umgeschaltet werden und einzelne Parameter geändert werden können. Des weiteren müssen die gemessen und berechneten Werte des Signalflusses an einen Entwicklungsrechner übertragen werden. Dort werden die Daten visualisiert und für eine spätere Analyse gespeichert. Folglich muss ein Kommunikationskonzept implementiert werden um Daten zwischen der Ziel- und Entwicklungsplattform auszutauschen. Um die Berechnung des Regelkreises und die Kommunikationsaufgaben voneinander zu trennen wird eine Komponentenarchitektur eingeführt ([?], S. 279 ff.). Die erste Komponente, welche als Regelungskomponente bezeichnet wird, führt die Berechnung des Signalflusses und die Interaktion mit den Peripheriegeräten durch. Des weiteren übernimmt sie die logische Steuerung der Versuchsabläufe. Die Kommunikationskomponente ist für die Verbindung mit dem Entwicklungsrechner verantwortlich und ermöglicht den Datenaustausch zwischen den beiden Plattform. Hierfür wird ein TCP/IP-Server verwendet. Durch die Verwendung

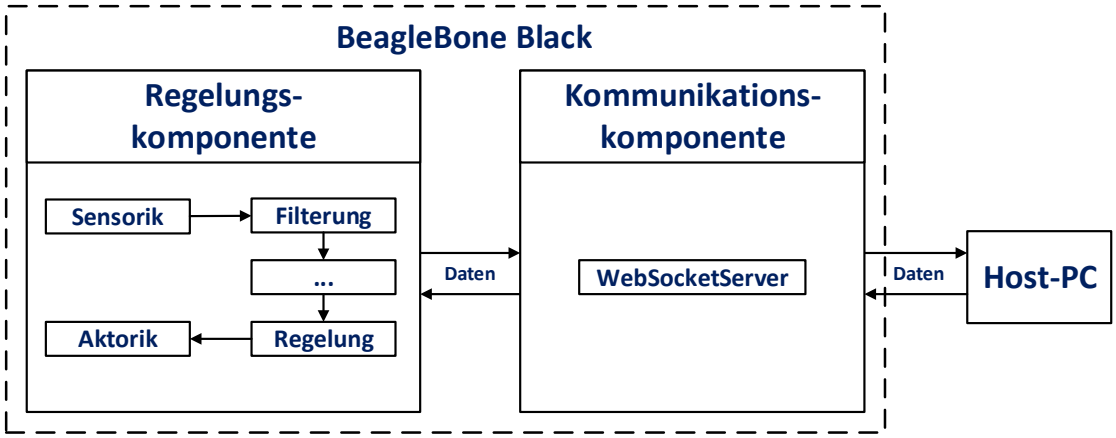

Abbildung 1.7: Blockschaltbild Gesamtsystem

der Komponentenarchitektur entstehen einige Vorteile. Zunächst können die beiden Komponenten in unterschiedlichen Threads ausgeführt werden. Dadurch wird die Bearbeitung der beiden Aufgabenbereiche zum Großteil entkoppelt. Der Datenaustausch zwischen den Komponenten reduziert sich auf eine kontrollierte Schnittstelle, wodurch die Fehleranfälligkeit des Gesamtsystems minimiert wird. Des weiteren können die Aufgaben priorisiert werden, da der Scheduler eine preemptive Round-Robin-Strategie verfolgt ([?], S. 19). Somit kann die Ausführung der Regelungskomponente, welche für die Bearbeitung der zeit- und sicherheitsrelevanten Aufgaben zuständig ist, gegenüber der Kommunikationseinheit priorisiert werden. Hieraus resultiert, dass das Zeitverhalten der Regelung als deterministisch angenommen werden kann.

Für die Implementierung wird das Interface *IRunnable* definiert, welches virtuelle Methoden zur Initialisierung und Ausführung der Komponenten vorschreibt. Diese Schnittstelle

wird von der abstrakten Klasse *AComponentBase* geerbt, welche Membervariablen zum Datenaustausch besitzt. Die letzendlichen Komponenten werden in Form der beiden Klassen *CControlComp* und *CCommComp* realisiert. Die Erzeugung der Threads erfolgt mit Hilfe der Klasse *CThread*, deren Instanz als Trägerobjekte der Threads agieren ([?], S. 108 ff.).

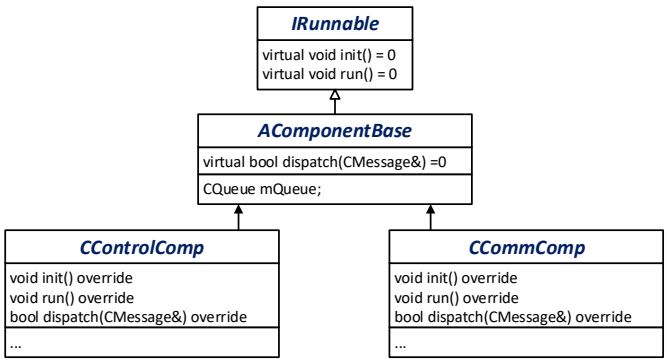

Abbildung 1.8: Klassendiagramm der Komponenten

Im nächsten Schritt muss ein Weg zum Datenaustausch zwischen den Komponenten etabliert werden. Einerseits müssen Messdaten und Signale aus dem Regelkreis von der Regelungs- and die Kommunikationseinheit gesendet werden, welche diese anschließend an den Host-PC weiterleitet. Andererseits werden die Steuerbefehle des Host-PCs von der Kommunikationskomponente empfangen und an die Regelungskomponente gereicht. Da in diesem Anwendungsfall lediglich kleine Datenmengen versendet werden, werden Nachrichten für deren Austausch verwendet ([?], S. 196). Die Nachrichten werden in Form der Klasse *CMessage* implementiert, welche aus einem Datenfeld, einem Ereignis und einem Zeitstempel komponiert werden. Das Ereignis wird als Enumeration realisiert und gibt den Inhalt der Daten bzw. den jeweiligen Befehl wieder. Der Zeitstempel gibt den Abtastzeitpunkt der Signale aus dem Regelkreis wieder.

```
 1  enum class EEvent
 2  {
 3    DEFAULT_IGNORE = 0,
 4    TIMER_TICK     = 1,
 5    STATE_DATA     = 2,
 6    ...
 7  }
 8  class CMessage
 9  {
10  public:
11    EEvent getEvent() const;
12    UInt8* getDataPtr();
13    ...
14  public:

16  private:
17    static constexpr UInt32 sDataSize = 32U;

19    EEvent  mEvent;
20    Float32 mTime;
21    UInt8   mData[sDataSize];
22  };
```

Listing 1.13: Beispielhafte Implementierung der Events und Nachrichten

Des weiteren besitzt die Klasse *CMessage*Methoden und Konstruktoren um die Datenfelder entsprechend zu füllen und auszulesen. Um die Nachrichten zu empfangen besitzen die Komponenten Eingangspuffer die als Queues implementiert werden. Die Erzeugung der Nachrichten wird von einem Proxy übernommen ([**?**], S. 285 ff.), der Methoden für die unterschiedlichen Ereignisse bereitstellt. Des weiteren kennt der Proxy die Queues der Komponenten und legt neue Nachrichten, in Abhängigkeit von dem jeweiligen Event, in die Eingangspuffers des zugehörigen Empfängers.

```
1  class CProxy
2  {
3    public:
4      bool transmitStateVector(const StateVector& x);
5      bool onTimerTick();
6      bool onClientConnect();
7      ...
8  };
```

Listing 1.14: Aufbau der Proxy-Klasse

Mit Hilfe der Nachrichtenkommunikation kann auch das Ablaufschema der Komponenten verallgemeinert werden. Beim Starten der Threads werden die Komponenten zunächst über den Aufruf ihrer *init()*-Methoden initialisiert, anschließend wird die *run()*-Methode ausgeführt, welche in diesem Fall eine Endlosschleife darstellt. In einem Durchlauf wird geprüft ob neue Nachrichten vorhanden sind, trifft dies zu wird die Nachricht über die *dispatch()*-Methode verarbeitet. Andernfalls legt sich der Thread schlafen bis neue Nachrichten zur Verfügung stehen. Hieraus ergeben sich zwei Vorteile. Einerseits werden die Komponenten nach einem einheitlichen Konzept entworfen, lediglich die Implementierung der *init()*- und *dispatch()*-Methode unterscheiden sich, andererseits wird durch die Synchronisation der Komponenten deren Laufzeit reduziert.

1.4 Entwurf der Regelungskomponente

Die Aufgabe der Regelungskomponente besteht darin den Kontroll- und Signalfluss der verschiedenen Versuche zu steuern. Diese Umsetzung erfolgt mit Hilfe eines Zustandautomats, welcher die Trennung der Kontrolllogik und der auszuführenden Aktionen ermöglicht. Des weiteren kann die Applikation bei diesem Ansatz problemlos durch weitere Versuche und Anwendungsfälle erweitert werden. Prinzipiell lässt sich der logische Ablauf der Komponente mit einem einfachen Zustandsdiagramm modellieren. Auf der obersten Ebene existiert ein *Standby*-Zustand, der die Inaktivität der Komponente widerspiegelt. Des weiteren enthält diese Ebene Zustände für die verschiedenen Versuche. Diese werden betreten, wenn die Komponente ein Event mit dem entsprechenden Befehl zur Ausführung des Versuchs erhält.

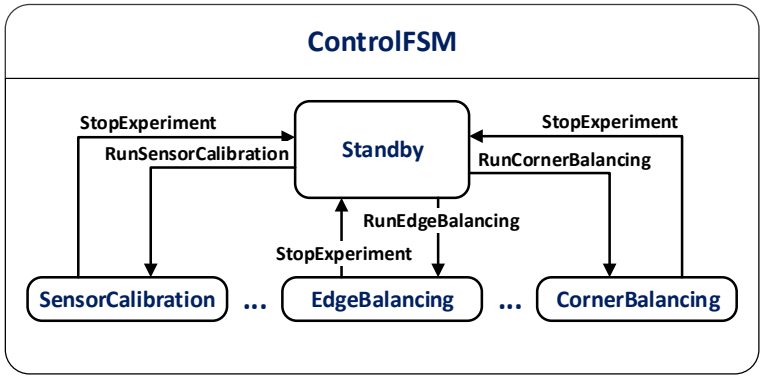

Abbildung 1.9: Zustandsdiagramm der Kontrol-Komponente

Ein solches Zustandsdiagramm kann z.B. mit einer objektorientierten Adaption der Methode von Samek ([**?**], S. 246 ff.) implementiert werden. Hierbei werden Zustände als Methoden der FSM realisiert, wobei für Oberzustände eigene Klassen entworfen werden, die wiederum Methoden für die jeweiligen Unterzustände besitzen. Die Referenzierung des aktuellen Zustandes erfolgt über einen Funktionenzeiger, der auf die Methode des entsprechenden Zustandes gerichtet ist.

Als Basis der Typenhierachie dient die abstrakte Klasse *AState*, welche den Zustandstypen als Methodenzeiger definiert, eine Standardimplementierung der *dispatch()*-Methode vorgibt und statische Membervariablen deklariert um die Verarbeitung interner Events zu ermöglichen.

```
 1  class AState
 2  {
 3  protected:
 4    using StatePtr = bool (AState::*)(CMessage&);
 5  public:
 6    virtual bool onInitial(CMessage& msg) = 0;
 7    virtual bool dispatch(CMessage& msg);
 8    ...
 9  protected:
10    StatePtr mStatePtr;
11    static constexpr StatePtr sInitial =
12        static_cast<StatePtr>(AState::onInitial);
13    static CMessage sInternalQueue;
14    static UInt32   sQueueSize;
15  };
```

Listing 1.15: Abstrakte Basisklasse für die Zustände

Die Methode *onInitial()* wird verwendet um die FSM und Oberzustände über ein *Init*-Event zu initialisieren. Die *dispatch()*-Methode beschränkt sich auf den Aufruf des aktuellen Zustands.

```
1  bool AState::dispatch(CMessage& msg)
2  {
3    return *(this->mStatePtr)(msg);
4  }
```

Listing 1.16: Defintion der *dispatch()*-Methode

Die auszuführenden Aktionen, wie z.B. die Berechnung des Regelkreises, werden in der Klasse *CActionHandler* gekapselt ([?], S. 225). Dadurch erfolgt eine klare Trennung des Kontroll- und Signalflusses. Prinzipiell besitzt *CActionHandler* Methoden, die jeweils beim Betreten und Verlassen der Zustände aufgerufen wird. Zusätzlich kann er um nötige Hilfsmethoden erweitert werden.

```
 1  class CActionHandler
 2  {
 3  public:
 4    void enterStandby();
 5    void exitStandby();
 6    void enterSensorCalibration();
 7    void exitSensorCalibration();
 8    ...
 9    void sampleSensorCalibration();
10    void sampleEdgeBalancing();
11    ...
12  private:
13    CThread    mTimerThread;
14    CTimerTask mTimerTask;

16    CHardware         mHardware;
17    EdgeBalancingSF   mEdgeBalancingSF;
18    ConrnerBalacingSF mCornerBalacingSF;
19  };
```

Listing 1.17: Beispielhafte Implementierung des Actionhandlers

Für die Zeitgebung wird einer separater Timer-Task *CTimerTask* verwendet, der von *CActionHandler* verwaltet wird. Die Timerklasse kann mittels der Methoden *pause()* und *resume()* pausiert bzw. gestartet werden. Während der Ausführung schläft der Timer für eine konfigurierbare Abtastzeit und erzeugt anschließend über den Proxy ein *TimerTick*-Event, welches an die Regelungskomponente weitergeleitet wird.

```
 1  class CTimerTask : public IRunnable
 2  {
 3  public:
 4    void run() override
 5    {
 6      while(true)
 7      {
 8        mRunningSem.take(true);
 9        mRunningSem.give();
10        usleep(mPeriod);
11        mProxyPtr->timerTick();
12      }
13    }
```

```
14    bool pause(bool waitForever){return mRunningSem.take(waitForever);}
15    bool resume(){mRunningSem.give();}
16    void setPeriod(Int32 period){mPeriod = period;};
17    ...
18  private:
19    CBinarySemaphore mRunningSem;
20    Int32            mPeriod;
21    CProxy*          mProxyPtr;
22  };
```

Listing 1.18: Aufbau des Timer-Tasks

Der Ansatz nach Samek bringt bei diesen Anwendungsfall einen Nachteil mit sich. Für die meisten Versuche genügt eine simple Kontrolllogik, weshalb diese als einfacher Unterzustand realisiert werden. Das hat eine flache Zustandshierarchie zur Folge, die der Anzahl von Versuchen entsprechend breit ist. In der Implementierung resultiert hieraus eine umfangreiche Klasse *CFSM*, da diese für jeden Unterzustand um eine Methode erweitert wird. Ebenso nimmt der Umfang der Klasse *CActionHandler* mit der Anzahl der Versuche kontinuierlich zu. Dieses Problem kann zwar durch die Aufteilung in mehrere Actionhandler vermieden werden, allerdings wird dadurch die Komplexität von *CFSM* weiter erhöht. Diese Problematik wird dadurch verschärft, dass es sich bei der Zustandsmaschine um den Kern der Anwendung und somit kritischen Abschnitt der Anwendung handelt. Die zu Grunde liegende Komponentenarchitektur wird zu Projektbeginn erstellt und danach kaum manipuliert. Im Gegensatz dazu wird die Zustandmaschine während des Projektsverlaufes ständig verändert und erweitert, weshalb eine unübersichtliche Implementierung besonders negativ auffällt. Aus diesem Grund wird im nächsten Schritt eine alternative Vorgehensweise vorgestellt, die sich die spezielle Struktur des Zustandsdiagrammes zu Nutze macht um eine effiziente Implementierung zu schaffen.

Zunächst sei angemerkt, dass es nicht möglich ist direkt zwischen zwei Versuchszuständen zu wechseln. Ein Versuchszustand kann nur aus dem Zustand *Standby* betreten werden. Des weiteren wird nach dem Verlassen eines Versuchszustandes immer in *Standby* gewechselt. Dadurch kann die Kontrolllogik der Zustandsmaschine verallgemeinert werden. Ist der momentane Unterzustand nicht *Standby* und es trifft ein *StopExperiment*-Event ein, so wird der Zustand verlassen und in *Standby* gewechselt. Befindet sich die FSM in *Standby* wird bei Eintreffen eines Events geprüft ob ein Zustandswechsel erfolgen muss. Diese Prüfung kann an die Zustände abgegeben werden. Somit stellt die Zustandsmaschine in diesem Fall lediglich eine Anfrage an alle Versuchszustände ob diese betreten werden möchten.

Die Implementierung der Zustandsmaschine setzt sich folglich aus einem *Standby*-Zustand und einer Liste von Versuchszuständen zusammen. Um eine übersichtliche Codestruktur zu erhalten werden diese als Klassen entworfen, die von der abstrakten Basisklasse *AState* erben.

```
 1  class AState
 2  {
 3  public:
 4    virtual bool dispatch(CMessage&) = 0;
 5    virtual bool tryEntry(CMessage&, AState*&) = 0;
 6    virtual void onEntry() = 0;
 7    virtual void onExit()  = 0;
 8  private:
 9    static CMessage sInternalQueue;
10    static UInt32   sQueueSize;
11  };
```

Listing 1.19: Angepasste Implementierung der abstrakten Zustandsklasse

Die Methode *dispatch()* dient zur Verteilung von eintreffenden Nachrichten. Mit Hilfe von *tryEntry()* kann die Zustandmaschine prüfen ob der Zustand, in Abhängigkeit des Events, betreten werden möchte. Der folgende Ausschnitt zeigt eine mögliche Implementierung für den Zustand *SensorCalibration*.

```
 1  bool CSensorCalibration::tryEntry(CMessage& msg, AState*& statePtr)
 2  {
 3    EEvent event = msg.getEvent();
 4    if(EEvent::RUN_SENSORCALIBRATION == event)
 5    {
 6      statePtr = this;
 7      return true;
 8    }
 9    return false;
10  }
```

Listing 1.20: Beispielhafte Definition der Methode *tryEntry()*

Das zweite Argument ist eine Zeigerreferenz auf den Zustandszeiger der FSM. Falls ein Zustand betreten werden soll überschreibt er die Referenz mit seinem this-Zeiger und gibt *true* zurück um den Konsum des Events zu signalisieren. Die Methoden *onEntry()* und *onExit()* werden zum Betreten bzw. Verlassen des Zustandes verwendet. Um auch die auszuführenden Aktionen zu trennen, wird für jede Zustandsklasse ein Actionhandler implementiert. Diese erben von der Klasse *CActionBase*, die gemeinsame Ressourcen als statische Membervariablen deklariert. Ein Beispiel wären hierfür Instanzen der Klassen *CHardware* oder *CTimerTask*.

Um die Zustandsklassen zu einer Liste zusammenzufassen wird wieder eine lineare Typenhierarchie verwendet. Hierfür muss zunächst ein Trägerobjekt *TStateHolder* entworfen werden.

```
 1  template<class State, class Base>
 2  class TStateHolder : public Base
 3  {
 4    bool tryEntry(CMessage& msg, AState*& statePtr)
 5    {
 6      bool consumed = mState.tryEntry(msg, statePtr);
 7      if(consumed == false)
 8      {
 9        return Base::tryEntry(msg, statePtr);
10      }
11      return consumed;
12    }
13  private:
14    State mState;
15  };
16  template<class State>
17  class TStateHolder<State, CNullType> : public CNullType
18  {
19  public:
20    bool tryEntry(CMessage& msg, AState*& statePtr)
21    {
22      return mState.tryEntry(msg, statePtr);
23    }
24  private:
25    State mState;
26  };
```

Listing 1.21: Implementierung der Trägerklasse für Zustände

Analog zu *TActionHolder* wird mittels einer Templatespezialisierung unterschieden ob das Ende der Typenliste erreicht ist. Ist dies nicht der Fall wird zunächst geprüft ob der getragene Zustand betreten werden soll. Trifft dies nicht zu wird die *tryEntry()*-Methode des nächsten Elements in der Hierarchie aufgerufen. Ein Unterschied zu *TActionHolder* ist, dass eine Komposition aus dem Zustandsobjekt verwendet wird. Dadurch werden die Methoden des Zustandes geschützt. Die FSM kann lediglich auf den, über ihren Zustandszeiger referenzierten, Zustand zugreifen. Die Implementierung der Zustandsmaschine basiert ebenfalls auf einem Template, welchem die Typenliste der Versuchszustände übergeben wird. Des weiteren erbt die Templateklasse von *AState* um Zugriff auf die interne Queue zu erhalten.

```
 1 template<class StateList>
 2 class TFSM : public AState
 3 {
 4 public:
 5   bool dispatch(CMessage& msg) override;
 6   bool tryEntry(CMessage& msg, AState*& statePtr) override;
 7   void onEntry() override;
 8   void onExit() override;
 9   bool onStandby(CMessage& msg);
10   void handleUnconsumedEvent(CMessage& msg);
11 private:
12   AState*                mStatePtr;
13   TLinHierach<StateList> mStateList;
14   CAction                mAction;
15 };
```

Listing 1.22: Implementierung der Templateklasse für die Zustandsmaschine

Neben dem Interface von *AState* besitzt die Klasse zwei weitere Methoden. Wobei die erste den Zustand *Standby* repräsentiert. Die zweite Methode wird genutzt um nicht konsumierte Events, was für gewöhnlich einem Fehlverhalten der FSM entspricht, abfängt. Zunächst wird die *dispatch()*-Methode betrachtet. Zu Beginn wird der aktuelle Unterzustand aufgerufen. Falls das Ereignis nicht konsumiert wird und es sich um *StopExperiment* handelt, wird der aktuelle Unterzustand verlassen und in *Standby gewechselt*. Zuletzt wird die interne Queue abgearbeitet.

```
 1 template<class StateList>
 2 bool TFSM<StateList>::dispatch(CMessage& msg)
 3 {
 4   bool consumed = false;
 5   if(mStatePtr == nullptr)
 6   {
 7     consumed = this->onStandby(msg);
 8   }
 9   else
10   {
11     consumed = mStatePtr->dispatch(msg);
12   }

14   if(consumed == false)
15   {
16     EEvent event = msg.getEvent();
17     if(EEvent::StopExperiment == event)
18     {
19       mStatePtr->onExit();
20       mStatePtr = nullptr;
21       mAction.entryStandby();
22     }
23   }
```

```
25    while(squeueSize > 0U)
26    {
27      CMessage internalMsg(sInternalQueue);
28      sQueueSize = 0U;
29      consumed = mStatePtr->dispatch(internalMsg);
30    }
31    return consumed;
32  }
```

Listing 1.23: Definition der Methode *dispatch()*

Im Zustand *Standby* wird das Ereignis an alle Unterzustände übergeben um zu prüfen, ob diese betreten werden sollen.

```
 1  template<class StateList>
 2  bool TFSM<StateList>::onStandby(CMessage& msg)
 3  {
 4    bool consumed = this->tryEntry(msg);
 5    if(consumed == true)
 6    {
 7      mAction.exitStandby();
 8      mStatePtr->onEntry();
 9    }
10    return consumed;
11  }
```

Listing 1.24: Implementierung der Methode *onStandby()*

Die Methode *tryEntry()* stößt lediglich die Abfrage der Unterzustände an.

```
1  template<class StateList>
2  bool TFSM<StateList>::tryEntry(CMessage& msg, AState*& statePtr)
3  {
4    return mStateList.tryEntry(mg, statePtr);
5  }
```

Listing 1.25: Implementierung der Methode *tryEntry()*

Die Vorteile dieses Konzept verdeutlichen sich wieder bei der Anwendung. Für jeden Versuch wird eine Zustandsklasse und Actionhandler entworfen, wodurch eine übersichtliche Projektstruktur entsteht. Um die letztendliche Zustandsmachine zu erhalten wird lediglich *TFSM* mit der gewünschten Liste von Zustandstypen instantiiert.

```
1  using StateList  = TYPELIST_4(CSensorCalib, CADCCalib,
2                                CEdgeBalance, CCornerBalance);
3  using ControlFSM = TFSM<StateList>;
4  ControlFSM myFSM;
```

Listing 1.26: Beispielhafte Instantiierung der Zustandsmaschine

Sollen weitere Zustände hinzugefügt oder entfernt werden muss lediglich die Typdefinition von *StateList* angepasst werden. Um Coderedundanzen zu vermeiden kann für die Unterzustände auch eine Templateklasse entworfen werden, die das Eintrittsereignis und den Actionhandler als Parameter entgegennimmt.

```
1  /* TSubState.h */
2  template<const EEvent entryEvent, class Action>
3  class TSubState : public AState
4  {
5  public:
6    bool tryEntry(CMessage& msg, AState*& statePtr) override
7    {
8      if(entryEvent == msg.getEvent())
9      {
```

```
10        statePtr = this;
11        return true;
12      }
13      return false;
14    };
15    bool dispatch(CMessage& msg) override;
16    void onEntry() override;
17    void onExit() override;
18  private:
19    Action mAction;
20  };
```

Listing 1.27: Templateklasse für einfache Versuchszustände

Die Unterzustände spezialisieren dann lediglich die Methoden *dispatch()*, *onEntry()* und *onExit()*, wie das folgende Beispiel zeigt.

```
 1  /* CADCCalib.cpp */
 2  using CADCCalib = TSubState<EEvent::RunADCCalib, CADCCalibAction>;
 3  template<>
 4  bool CADCCalib::dispatch(CMessage& msg)
 5  {
 6    EEvent event = msg.getEvent();
 7    if(EEvent::TIMERTICK == event)
 8    {
 9      mAction.sampleADCCalib();
10      return true;
11    }
12    ...
13    return false;
14  }
15  template<>
16  void CADCCalib::onEntry()
17  {
18    cout << "Entering ADC-Calibration . . . " << endl;
19    mAction.resumeTimer();
20  }
21  template<>
22  void CADCCalib::onExit()
23  {
24    cout << "Exiting ADC-Calibration . . . " << endl;
25    mAction.pauseTimer();
26  }
```

Listing 1.28: Beispielhafte Instantiierung des Versuchstemplate

1.5 Entwurf der Kommunikationskomponente

Die zweite Komponente des Systems ist für die Kommunikation mit einem Desktop-PC verantwortlich. Unter die Kommunikationsaufgaben fällt einerseits die Übermittlung relevanter Daten des Regelkreises, wie z.B. der aktuelle Wert des Zustandvektors. Andererseits nimmt die Kommunikationskomponente Steuerbefehle entgegen und leitet diese an die Regelungskomponente weiter. Dadurch können während der Laufzeit Konfigurationen der Versuche durchgeführt werden. Beispiele hierfür sind die Auswahl des zu verwendeten Regler- oder Filterkonzeptes. Für die Verbindung zwischen der Ziel- und Entwicklungsplattform wird das TCP/IP-Protokoll verwendet, wobei die BeagleBone-Anwendung als Server fungiert.

Der Kontrollfluss der Kommunikationskomponente wird ebenfalls als Zustandsmaschine modelliert. Diese besitzt die beiden Zustände *Standby* und *Running*. Im Zustand *Standby* wartet der TCP/IP-Server auf die Verbindungsanfrage eines Clients. Ein Verbindungsaufbau wird durch ein entsprechendes Event signalisiert, woraufhin die Zustandsmaschine in *Running* wechselt. In diesem Zustand werden Nachrichten der Regelungskomponente an den Client und umgekehrt von dem Client an die Regelungskomponente weitergeleitet. Der Verbindungsabbruch durch den Client wird wiederum durch ein Event signalisiert. Daraufhin kehr die Zustandsmaschine in *Standby* zurück und wartet auf eine erneute Verbindungsanfrage. Für die Implementierung der Zustandsmaschine wird die objektorientierte Methode nach

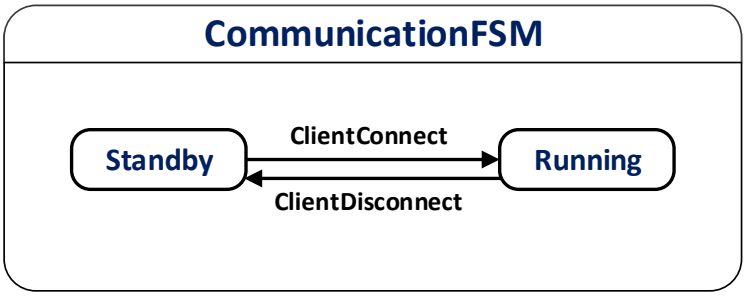

Abbildung 1.10: Zustandsdiagramm der Kommunikations-Komponente

Samek verwendet (Abschnitt...). An dieser Stelle werden keine Templates verwendet, da die Maschine lediglich zwei Unterstände besitzt. Des weiteren ist nicht zu erwarten, dass sich das Zustandsdiagramm im Entwicklungsprozess verändert.

Der Actionhandler der Zustandsmaschine besitzt eine Instanz der Klasse *CServer*, welche die nötigen Serverfunktionalitäten bereitstellt.

```
1  class CServer
2  {
3  public:
4    void init();
5    bool waitForClient();
6    bool transmitMessage(CMessage& msg);
7    bool receiveMessage(CMessage& msg);
8    ...
9  };
```

Listing 1.29: Implementierung der Server-Klasse

In der *waitForClient()*-Methode wird der aufrufende Thread blockiert bis ein Client eine Verbindung aufbaut. Anschließend können mittels der Methoden *transmitMessage()* und *receiveMessage()* Nachrichten versendet bzw. empfangen werden. Da es sich bei einer TCP/IP-Verbindung um einen Bytestream handelt wird der Typ *CMessage* als Datenpaket in der Übertragung verwendet. Das heißt das lediglich Instanzen von *CMessage* übermittelt werden, wobei die Objekte byteweise versendet und empfangen werden. Im Anschluss werden die einzelnen Bytes wieder zu einer Instanz von *CMessage* zusammengesetzt.

Da das Versenden und Empfangen von Nachrichten parallel erfolgen soll, wird die Kommunikationskomponente um einen zweiten Thread erweitert. Dieser wird in der Klasse *CReceiveTask* implementiert, welche von *IRunnable* erbt. Dieser Thread ist sowohl für das Empfangen von Nachrichten als auch die Annahme von Verbindungsanfragen zuständig.

```
 1  void CReceiveTask::run()
 2  {
 3    while(true)
 4    {
 5      if(mServer.waitForClient() == true)
 6      {
 7        sProxy.clientConnected();
 8      }

10      CMessage receivedMsg;
11      while(mServer.receiveMessage(receivedMsg)
12      {
13        sProxy.routeClientMessage(receivedMsg);
14      }
15      sProxy.clientDisconnect();
16    }
17  }
```

Listing 1.30: Implementierung des Receive-Tasks

Der Thread ruft zunächst die *waitForClient()*-Methode der *CServer*-Instanz auf, welche den Thread solange blockiert bis eine Verbindung aufgebaut wurde. Daraufhin erzeugt der Task mittels des Proxy ein Ereignis um die Komponente zu benachrichtigen. Im nächsten Schritt ruft der Task die *receiveMessage()*-Methode des Servers in einer Schleife auf. Diese blockiert den Thread bis entweder eine Nachricht eintrifft oder die Verbindung abgebrochen wird. Die Fallunterscheidung erfolgt mit Hilfe des Rückgabewertes von *receiveMessage()*, wobei im Fall einer empfangenen Nachricht diese über den Proxy an die Regelungskomponente weitergeleitet wird. Wird die Verbindung von dem Client abgebrochen verlässt der Task die Empfangsschleife. Anschließend wird der Verbindungsabbruch über den Proxy signalisiert und wieder auf eine erneute Anfrage gewartet.

Der Hauptthread der Komponente wird über deren Eventqueue synchronisiert. Hierbei sind lediglich drei Fälle zu unterscheiden. Die Events *ClientConnect* und *ClientDisconnect* führen jeweils zu einem Zustandswechsel der Zustandsmaschine. Befindet diese sich im Zustand *Running* wird zusätzlich geprüft ob es sich um eine Nachricht handelt, welche über den Server an den Desktop-Client weitergeleitet werden muss.

1.6 Aufbau der Desktop-Anwendung

Der letzte Teil der Software-Infrastruktur besteht in einer Anwendung, welche in einer gewöhnlichen Desktopumgebung ausgeführt wird und die Interaktion mit der Zielplattform ermöglicht. In diesem Fall wird als Desktopbetriebssystem Ubuntu 16.04 verwendet. Die Anwendung stellt eine graphische Benutzeroberfläche zur Verfügung, welche einerseits relevante Daten der Versuche visualisiert und andererseits Bedienelemente bietet um Versuchskonfigurationen vorzunehmen.

Für die Implementierung der Benutzeroberfläche wird die Bibliothek Qt verwendet, welche übliche Bedien- und Grafikelemente zur Verfügung stellt. Der Aufbau der Anwendung ist in zwei Threads unterteilt. Der erste verwaltet eine Instanz der Klasse *CClient*. Dieses Objekt stellt das Gegenstück zu dem Server dar, der auf dem BeagleBone Black ausgeführt wird. Die Klasse *CClient* bietet ebenfalls die Methoden *receiveMessage()* und *transmitMessage()* um Nachrichten zu Empfangen bzw. zu Versenden. Des weiteren kann die Methode *connectToServer()* genutzt werden um eine Verbindung mit dem BeagleBone Black herzustellen.

Der zweite Thread verwaltet die Benutzeroberfläche. Diese setzt sich aus Plots zur Darstellung der Versuchsdaten und den nötigen Bedienelementen für die Versuchskonfiguration zusammen. Für die Kommunikation zwischen den beiden Threads wird Qts Signal/Slot-Prinzip verwendet. Der Qt-Meta-Compiler ermöglicht es Methoden als Signale bzw. Slots zu deklarieren. In der Anwendung können die Signale eines Objekts mit den Slots eines weiteren Verbunden werden. Anschließend kann ein Objekt seine Signale emittieren woraufhin der verbundene Slot aufgerufen wird. Als Beispiel sei eine Gruppe von Schalter zur Auswahl des Reglerkonzeptes genannt. Wird einer der Schalter gedrückt löst dieser ein Signal aus, welches mit einem Slot des Grafik-Threads verbunden ist. In diesem wird die Auswahl ausgewertet und ein entsprechendes Signal ausgelöst. Das Signal des Grafik-Threads ist wiederum an einen Slot des Kommunikations-Threads gekoppelt. In dem Slot wird eine Instanz von *CMessage* mit dem passenden Event angelegt und über den Client an das BeagleBoneBlack gesendet. Umgekehrt wird bei dem Eintreffen von Versuchsdaten ein entsprechendes Signal des Kommunikations-Thread emittiert. In dem zugehörigen Slot des Grafik-Threads werden die Daten verarbeitet und in den Plots angezeigt.