Simulink objektorientiert mit C-Kernfunktionen - Anleitung mit Demo- / Test-Version

Dr.-Ing. Hartmut Schorrig

9. Dezember 2017

Zusammenfassung

Dieser Artikel stellt Prinzipien dar und ist eine Anleitung für Generierung von Sfunctions in Simulink unter Nutzung von Kern-Funktionalitäten in C. Die Generierung wird aus Informationen in den C-Headerfiles gesteuert. Es sind wenig zusätzliche Aufwendungen in Scripts notwendig.

Der Artikel ist gültig zusammen mit dem File ***Example_ObjO.zip***. Dieses File enthält ein Beispielmodell, die Sources der Sfunctions, die entsprechenden Generierscripts und das Inspector-Tool.

Die Zielgruppe für diesen Artikel und die hier vorgestellte Technologie sind Embedded-C-Entwickler, die entweder Simulink noch nicht kennen oder Simulink-Entwickler, die an C orientiert sind und gleichzeitig die Objektorientierte Programmierung im Blick haben.

Inhaltsverzeichnis

1 Motivation

C ist für Embedded Applikationen der Standard schlechthin. Die Gründe liegen in der Hardwarenähe und der Flexibilität für verschiedene Hardware-Anforderungen. Applikationen werden aber mehr und mehr komplex, entsprechend der steigenden Leistungsfähigkeit von Prozessoren und Boards und der gestiegenen Anforderungen an die Software-Applikationen. Die Softwaretechnologie muss schritthalten, sonst geht der Überblick über die Software verloren.

Objektorientierte Ansätze, Nutzung C++, Einsatz der UML (*Unified Modelling Language*) als grafische Unterstützung gibt es seit über 20 Jahren. Die andere Richtung sind grafisch orientierte Baustein-Verdrahtungen wie sie beispielsweise mit Simatic-CFC oder Simadyn-Struc schon seit Beginn der 90-ger Jahre bekannt sind. Simulink mit integrierter Codegenerierung gehört seit langem auch dazu.

Solche Systeme haben aber ihre Eigenheiten. Die gesamte Softwaretechnologie muss umgestellt werden. *Weg von C* bedeutet möglicherweise und intuitiv auch das *Weg von Hardwarenähe*.

Der hier gezeigte Ansatz basiert auf C, das weiterhin für Kern-Funktionen verwendet wird. Damit ist direkt die Hardware in gewohnter Weise ansprechbar. Der Neuwert mit dem Einsatz von Simulink ist der besserer Überblick über die Gesamtfunktionalität. Das wird durch die grafische Notationsform erreicht. Simulink arbeitet sehr gut mit C zusammen. Anders jedoch als in Standard-Simulink-Applikationen wird die Objektorientierung in den Vordergrund gerückt. Objektorientierung bedeutet: Stärkere Sicht auf Daten statt auf Funktionalität. Da auch die Codegenerierung der Zusammenschaltung der Simulink-Blöcke C als Implementierungssprache verwendet, steht auf Implementierungsebene in gewohnter Weise reines C zur Verfügung. Es ist also keine Umstellung von Generierumgebungen für die Zielplattform notwendig.

Eine weitere möglicherweise wichtige Motivation für diesen Ansatz besteht in Folgendem: Der aus Simulink für ein Zielsystem generierte Code (Embedded Coder) ist zwar nach der Generierung in gewohnter Weise mit den bekannten zielsystemspezifischen Compiler- und Linker-Tools weiter verarbeitbar. Der Code kann für sich gelesen und verstanden werden, jedoch ist die Zuordnung von Codeteilen zu grafischen Modellelementen nicht ganz einfach. Man denke an eine heute verwendete Modellierung mit Simulink für ein Zielprodukt, dass in 20 bis 30 Jahren noch gepflegt werden muss. Das kann sich auf eine Parameteranpassung beziehen, den Austausch eines Treibers für neue Hardware, oder die Neucompilierung für eine innovierte Hardware-Plattform. Diese Arbeiten sind im reinen C möglich. Für die Archivierung der gesamten Toolkette ausgehend vom Simulink über erneute Codegenerierung ist jedoch ein hoher Aufwand notwendig, der sich beispielsweise auch in der notwendigen Einarbeitung in ein dann schon jahrzehntealtes Tool zeigt. Wenn also heute abgeschätzt werden kann, dass der Pflegeaufwand für die Zukunft erheblich sein könnte, dann kann dies ein Grund für die Entscheidung gegen die Simulink-Technologie sein. Wenn jedoch der erzeugte Code genügend gut durchschaubar ist für die spätere Pflege, dann ist die Entscheidung *heute* für die bessere Softwaretechnologie mit Simulink deutlich einfacher. Mit dem Einsatz von C im Kern sind diese Kernfunktionen im generierten Code gut auffindbar und erleichtern wesentlich die Navigation im generiertem Code für später notwendige Anpassungen.

Die Objektorientierung in Simulink - unabhängig von C - stellt eine andere, möglicherweise bessere Übersicht über die Funktionalität in den Simulink-Modellen dar. Letzlich ist damit eine bessere Organisation der Abarbeitung möglich, was den Rechenzeitbedarf reduziert. Gemeinsam mit C ist damit die schnelle Abarbeitung im Fokus.

Dieser Artikel zeigt die Technologie in der konkreten Anwendung am Beispiel eines Modells einschließlich Tool zur S-Funktion-Generierung für Simulink, eines generierten Codes daraus und den Möglichkeiten der Dateninspektion mit dem Tool *Inspector*. Der Artikel ist gültig mit / für den zip-File www.vishia.org/Smlk/Example_ObjO.zip, der anhand eines Beispiels alle Elemente enthält. Mit dem Inhalt dieses zip.Files sind eigene Arbeiten als Trial-Version möglich. Die Vollversion wird mit einem Vertrag mit *vishia* mit persönlicher Beratung bereitgestellt. Notwendig ist eine Standard-Simulink-Installation (*Mathworks*), mit Embedded Coder (*Mathworks*) für die Codegenerierung aus den Simulink-Teilen für Embedded Ziel-Hardware.

2 Objektorientierte Denkweise und Simulink

Charakteristisch für Simulink ist die datenflussorientierte Darstellung: Verbindungspfeile sind immer in Datenflussrichtung gezeichnet. Für den Regelungstechniker ist das eine gewohnte Denkweise. In der Objektorientierung, veranschaulicht mit der UML, geht es dagegen um Aggregationen zwischen Klassen, Abhängigkeiten, Referenzierung eines Objektes aus einem anderen heraus um dessen Operationen (Methoden) aufzurufen. Beides scheinen gegensätzliche Ansätze zu sein. Die Gemeinsamkeit ist, dass mit grafisch dargestellten Blöcken die Anwenderprogrammierung besser gegliedert dargestellt wird.

Die Objektorientierung ist in der Softwaretechnologie stark verwurzelt und etabliert. Wechselt man nun zu Simulink, dann ist schon die Frage zu stellen, ob die Objektorientierung nun passé sei? Andererseits - Simulink Modelle objektorientiert auffassen?

Sobald man sich in Simulink nicht auf der Ebene der einfachen Verknüpfungen bewegt sondern Blöcke miteinander verbindet, kann dies ebenso objektorientiert aufgefasst werden. Blöcke stellen Objekte dar. Der entscheidende Schritt hin zur Objektorientierung ist, dass die Blöcke nicht mit Einzelsignalen verknüpft werden, auch nicht mit Bussen, sondern mit einem *Handle*, das die Speicheradresse von Interndaten repräsentiert. Dazu sind mindestens die Blöcke, die Handle bereitstellen und verarbeiten, als Sfunction zu realisieren die in C formuliert werden.

Für diesen Ansatz werden im Simulink S-Funktionsbausteine (*Sfunctions, Function blocks, FB*) unterteilt in

- *Object-FB*: Enthält die Daten. Enthält eine Operation. Empfängt Datenverbindungen von anderen Object-FB über Handle. Liefert den eigenen Handle als Output.
- *Operation-FB*: Enthält keine Daten, gehört zu einem Object-FB, dessen Handle als Input gegeben ist.

Diese objektorientierte Denkweise in Simulink kann zu übersichtlicheren Modellen führen. Dies sollten die nachfolgenden Beispiele zeigen. Wichtig ist, dass das für den Regelungstechniker transparent nachvollziehbar und verständlich ist, um die notwendige Akzeptanz zu erzeugen. Für den objektorientiert denkenden Informatiker, der eher eine *Unified Modelling Language* (UML) gewohnt ist, bietet Simulink damit eine neue nutzbare Plattform. Letztlich kann eine Objektorientierung in Simulink auch zur Rechenzeiteffizienz des generierten Codes beitragen, da referenzierte Module von mehreren Instanzen genutzt werden können und Kopieraufwand im Maschinencode eingespart wird.

Das Bild auf der Folgeseite zeigt ein Beispiel, dass in den weiteren Anleitungen benutzt wird:

Im Beispiel geht es um schwingende Messgrößen, die für die weitere Verarbeitung gefiltert, in zwei Komponenten und damit drehende Vektoren gewandelt und letzlich statische Werte abgeleitet werden. Für die Filterung und Vektorbildung wird ein *Harmonischer Oszillator* eingesetzt, der fremdbestimmt schwingend einerseits die entsprechenden Frequenzen filtert und andererseits mit seinen zwei Integratoren zwei möglichst exakt 90 Grad versetzt schwingenden Signale bildet. Damit stehen zwei Komponenten im Vektorraum zur Verfügung, die als drehender Zeiger die Schwingbewegung abbilden. Diese Anordnung wird als *Orthogonaloszillator* bezeichnet. Mittels Parallelschaltung der auf Oberschwingungen abgeglichenen Orthogonaloszillatoren wird die Grundschwingung möglichst sauber abgebildet. Für die Weiterverarbeitung werden dann die drehenden Vektoren in stehende Größen wandelt (sogenannte Park-Transformation). Damit kann die Weiterverarbeitung in einer langsameren Abtastzeit erfolgen. Das mag beispielgebend sein und für Applikationen aus elektrischen Netzen (Wechselstrom) und der Mechanik (drehende Wellen) hilfreich.

Ein Orthogonaloszillator als Grundglied kann nun mit einem rein grafischen Modell im Simulink gut beschrieben werden. In diesem Beispiel wird aber diese Grundfunktion in C realisiert und als Black-Box für den Regelungsentwickler als Sfunction angeboten. Damit wird dem Ansatz "*Im Kern C* " genüge getan. "

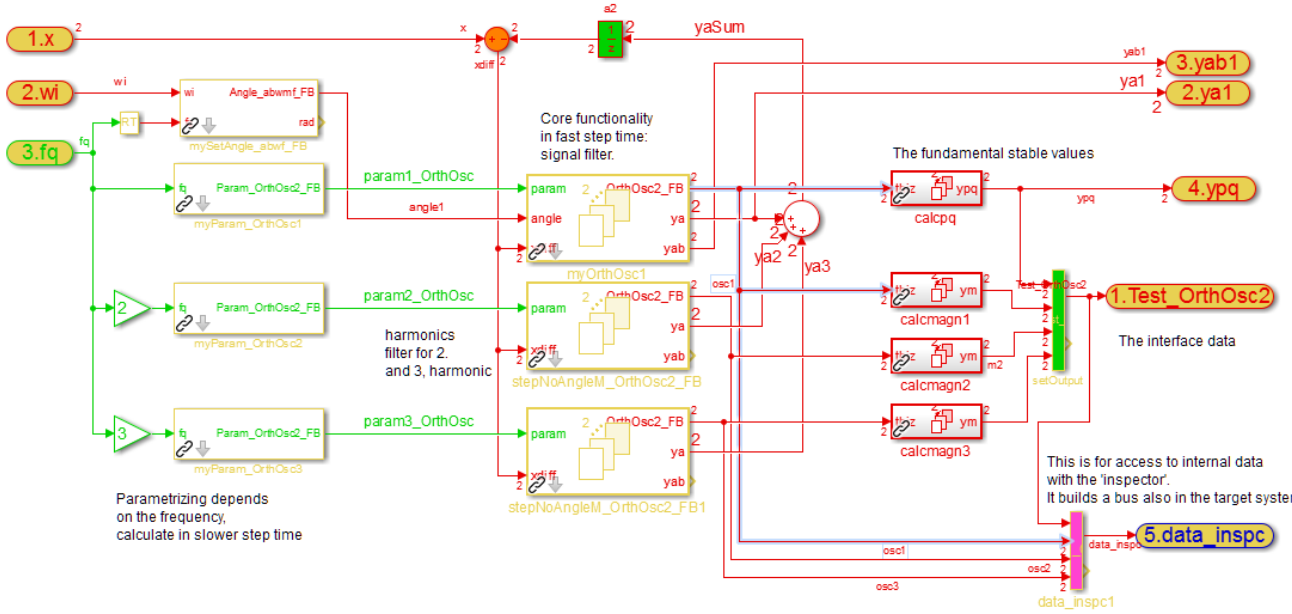

Abbildung 1: Simulink Orthogonaloszillator - Objectmodell

Die links im Bild angeordneten `Param_OrthOsc2_FB` bilden Object-FBs, die Parameter aufbereiten und über den Ausgang als Handle zur Verfügung stellen. Sie werden im Beipiel in einer langsamen Abtastzeit berechnet, da die über die Frequenz als sich langsamer veränderten Größe die Paramter angepasst werden müssen. Die Frequenz ist Eingangssignal dieser Parameter-FBs.

In Bildmitte sind die eigentlichen Orthogonaloszillatoren ebenfalls als Object-FB angeordnet. Im UML- oder objektorientierten Sinne aggregieren diese die Parameter-FBs. In Simulink ist im Vergleich zu einer UML-Darstellung lediglich die Pfeilrichtung gedreht, vom aggregierten zum nutzendem FB. Der nutzende FB erhält die Information über das Aggregat als Handle. Diese Handleverbindung erfolgt initial und fest, im Sinne der Aggregation, und braucht also keine Rechenzeit zur Runtime. Eine Assoziation im Sinne der UML - wechselnde Verbindung zu verschiedenen Instanzen, ist auf diese Weise auch möglich und nur eine Entscheidung der Steptime-Zuordnung.

Rechts im Bild werden mehrere Operation-FB gezeigt, die jeweils eine Operation des `OrthOsc2_FB` ausführen. `calcpq` führt die Wandlung der oszillierenden komplexen Größe des Orthogonaloszillators (drehender Zeiger) in eine stehende Größe. Dazu ist ein Referenzwinkel notwendig. Der Referenzwinkel wird nicht hier eingespeist, wie man es in klassischer Simulink-Denkweise erwartet. Da der Referenzwinkel eine grundsätzliche Größe des Objects oder im UML-Sinn der class `OrthOsc2_FB` ist, ist er selbstverständlich in diesem Object enthalten, bzw. genauer, wird referenziert im aggregierten `Angle_abwmf_FB`. Darunter bedinden sich Operation-FB zur Bildung der Amplituden der Grundschwingung und der Oberschwingungen.

Die Ausgangsgrößen des Moduls sind nun im grün gekennzeichneten Block `setOutput` zusammengefasst. Das ist ein Object-FB, der nur für die Zusammenfassung der Ausgangsgrößen in diesem Modell genutzt wird. Optisch sieht der FB wie ein *Bus-Creator* aus. Funktionell ist es auch so etwas wie ein Bus-Creator, realisiert allerdings als Sfunction. Siehe auch Kapitel 4 Seite 10.

Rechts unten ist nun ein weiterer Bus-Creation-ähnlicher FB angeordnet, der die Handle aller Object-FB zusammenfasst und für den Datenzugriff nach außen für Debugzwecke (Inspector) bereitstellt, siehe Kapitel 12 Seite 52.

Die meisten der hier sichtbaren Verbindungsleitungen sind Handle-Verbindungen. Diese werden nur initial verwendet, auch wenn die Abtastzeit-Farbe in diesem Bild dies anders darstellt. Die Handle-Verbindungen entspringen bei Object-FB als Sfunction immer aus dem Output Port 0, rechts oben. Diese Regelung ist auch bei den hier benutzten *For-each*-Subsystemen beibehalten. Die

Handle-Verbindung führt den konstanten Handle bzw. die Speicheradresse im generiertem Code des Zielsystems. Sie wird in der `init_`-Routine in den Interndaten vermerkt, das ist die Aggregation. Zur Laufzeit braucht dann diese Verbindung nicht mehr beachtet werden. Das erspart einiges an Datentransfer in der schnellen Abtastzeit. Die Ports sind allerdings für Simulink mit der schnellen Abtastzeit gekennzeichnet, weil Simulink daraus die Abarbeitungsreihenfolge (*Execution-order*) ableitet. Insbesondere werden alle Verbindungen zu dem Inspector-Datensammler, hier rechts unten, nur in der Initialisierung bearbeiten. Der Inspector-Zugriff belastet auch im Zielsystem nicht die schnelle Abtastzeit.

3 Grafisches Modell, Sfunctions und Codegenerierung

Ein Simulink-Modell stellt die Funktionalität übersichtlich dar, kann komplexere Einheiten als Module oder Komponenten bilden. Letzlich ist solch ein Modell geeignet um im Kundengespräch die Realisierung der Anforderungen darzustellen oder in der Softwarepflege den notwendigen Überblick zu vermitteln. Simulink ist gegebenenfalls dazu besser geeigneta als C-Codes, deren Struktur mehr oder weniger genau etwa mit UML abgebildet werden. Die Realisierung der Kern-Funktionen in C bilden eine geeignete Brücke zum Implementierungssystem für die Zielhardware.

In Simulink wird die gesamte Funktionalität in grafischer Notationsform beschrieben. Diese Denkweise liegt einem Regelungstechniker nahe. Es gibt andererseits das *Black-Box-Prinzip*: Ein Modul wird benutzt, dessen Verhalten an den Ein- und Ausgängen ist wohl definiert. Wie das Modul diese Funktion erfüllt, ist bei der Nutzung des Moduls nicht wichtig. Sehrwohl kann das Modul im inneren auch betrachtet werden, in einer anderen Betrachtungsebene. Das Black-Box-Prinzip ist dem Regelungstechniker zumeist geläufig.

Das Beispielmodell zeigt deutlich, dass die Funktionalität gut dargestellt wird mit Nutzung der Module der OrthOsc2_FBs. Das wesentliche am Modell ist die Zusammenschaltung. Der Innenaufbau der Module würde in Simulink einige Additionen, Multiplikationen und Speicherglieder zeigen, nicht sehr spektakulär. Diese Dinge lassen sich einfach auch in C programmieren. Die hier beschriebene Denkweise geht davon aus, dass im Kern - aus verschiedenen Gründen - C genutzt werden soll. In C sind auch Zuordnungen zu Abtastzeiten, Steuerungen wann Parameter neu berechnet werden (wer ruft dies auf) und dergleichen günstig zu programmieren, ebenso Synchronisationsmechanismen mit dem Betriebssystem.

Sfunction-Wrapper

Die Sfunctions sind FunctionBlocks in Simulink, die eine direkte in C geschriebene Funktionalität enthalten. Um die C-Funktionalität im Simulink abarbeiten zu können, sind einige Aufwendungen notwendig. Beispielsweise muss Simulink abfragen können, wieviel In- und Output-Ports es gibt, welche Abtastzeiten zugeordnet sind usw. usf. Der Aufruf der Funktionalität in der Abtastzeit ist dabei fast der kleinere Aufwand. Jedoch müssen auch dafür die Input-Daten aus dem Simulink-Datenhaushalt gelesen, aufbereitet, und die Output-Daten wieder geschrieben werden. Dieser Zusatzaufwand mag als Wrapper für kleine C-Funktionalitäten hoch erscheinen, ist aber rechenzeittechnisch gesehen nicht das Problem. PC-Prozessoren sind schnell. Jedoch ist es ein programmtechnischer Aufwand. Ein *Sfunction-Wrapper* um die eigentlichen C-Kernanweisungen herum ist ein C-File, das als main-File für eine spezielle *MEX*-Dynamic Library für Simulink compiliert wird. Der benutzte Compiler ist dabei meist ein Visual-Studio-Standard-Compiler, der von Matlab über das `mex`-Command aufgerufen wird. Diese Dinge sind in der Matlab-Hilfe bekannt und erläutert.

tlc-Files

Simulink kennt weiterhin den Accelerator-Mode. Dabei wird der Code nicht mit den eben vorgestellten Sfunction-Wrapper aufgerufen, sondern der Code wird compiliert und als Maschinencode abgearbeitet. Ein tlc-File (*Target Language Compiler-Controllfile*) für jede Sfunction beschreibt, wie die direkt in C geschriebenen Teile richtig einzubinden sind. Ein tlc-File enthält Steueranweisungen und Quellcodes mit *Replacement*-Elementen.

Die tlc-Files dienen letztlich auch dazu, für die Codegenerierung im Zielsystem die direkt beigestellten C-Codes richtig einzubinden.

Standard-Angebot in Matlab für Sfunctions

Matlab-Simulink bietet zur Anwenderunterstützung dieses Systems der Sfunction - eigene C-Codes einbinden - Unterstützung an. Das *Legacy code tool*, im Simulink-Standardumfang enthalten, kann verwendet werden, um aus einer Beschreibung der Einbindung der C-Codes in Simulink in einem Matlab-Script-File die entsprechenden Wrapper und tlc-Files zu generieren. Das *Legacy code tool* ist erstrangig dafür gedacht, eher komplexen C-Codes in ein Simulink-Modell aufzurufen. Für die Einbindung eher kleiner C-Module ist der Schreibaufwand etwas hoch, da für jede einzelne C-Funktion ein extra Matlab-Script geschrieben werden muss. Sonderfälle, etwa String-Parameter oder parameterabhängige Typen von Ein- und Ausgängen werden nicht unterstützt.

Schreibt man dagegen die Sfunction-Wrapper und tlc-Files manuell, dann sind die Möglichkeiten der Gestaltung sehr hoch. Beispielsweise kann der

Typ von Ein- und Ausgängen (Anzahl Vektorelemente, float oder int, Bus-Typ) und die Zeitscheibenzuordnung mit der Einbettung im Modell geregelt werden. Die angeschlossenen Verbindungen im Modell bestimmen diese Eigenschaften. Die Eigenschaften werden in den Wrapper-Funktionen der Sfunction aus der Simulink-Umgebung abgefragt und abgelegt. Bei der Codegenerierung können diese Daten dann einerseits den C-Funktionen aufbereitet übergeben werden, andererseits gibt es auch in den tlc-Files `%if`-Anweisungen, die zum Zeitpunkt der Codegenerierung entscheiden, welche Programmzeilen generiert werden. - Die Sfunction-Wrapper und tlc-Files können anhand von Mustern und der Hilfe-Doku erstellt werden. Jedoch ist das Selbstschreiben durchaus aufwändig und damit auch fehleranfällig.

Die dritte Möglichkeit ist eine grafische Bedienung zur Bestimmung der Eigenschaften von Sfunctions.

Generator für die Sfunction-Wrapper und tlc-Files direkt aus den Informationen in den Headerfiles

Wenn kleine und viele Kern-Funktionen in C in ein Simulink-Modell eingebettet werden sollen, dann ist die Arbeit der Erstellung der Sfunctions für jede Kern-C-Funktion mit dem *Legacy-code-tool* zu hoch. Es muss ausreichend sein, mit wenigen Annotations in den Headerfiles der C-Programme einen Generator so zu steuern, der die Sfunction-Wrapper und tlc-Files automatisch ohne Zusatzarbeit erstellt. Die Komplexität wird hier von der Einzelbearbeitung in das Generiertool verlagert. Hat man spezifische Anforderungen, dann wird dies im Generiertool angepasst und steht dann für alle Sfunctions zur Verfügung. Der einzelne C / Simulink-Entwickler muss damit nicht alle Details täglich bearbeiten, die Arbeit der spezifischen Anpassung lässt sich zentralisieren. Dabei ist Aufwand und Verständnis für diese Anpassungen durchaus vom einzelnen Entwickler im Mittelstand erbringbar.

Im folgenden Kapitel werden die Schreibweisen in den eigenen C-Codes in den Headerfiles gezeigt, wie sie für die Standardgenerierung verwendet werden. Die Schreibweisen sind anpassbar an die jeweils gültigen Style-Guides.

Die Generierung wird mit einem ZBNF-Parser und JZtxtcmd-Generator ausgeführt. Diese beiden Tools sind als Open-Source verfügbar. Beide werden mit textuellen Scripts gesteuert. Diese Scripts enthalten das Knowhow der Sfunction-Wrapper- und tlc-Generierung. Mit Anpassung und Weiterentwicklung der Scripts sind weitergehende Anforderungen erfüllbar.

Siehe folgenden Abschnitt und Abschnitt 9.

4 Praktische Arbeit am Beispiel: Datensammler

C-Code des Datensammlers im Beispielmodell setOutput

Dieses Kapitel soll am einfachen Beispiel zeigen, wie der Datensammler `setOutput` im Beispielmodell in C in einem Headerfile programmiert wird und wie sich diese Arbeitsweise in die Modellierung einfügt. Die detaillierte Erläuterung der Gestaltungsmöglichkeiten in einem Headerfile finden sich im Kapitel 6, Seite 14

Die Daten werden vom gezeigten Modell erarbeitet und sollen hier einem beliebig anderem Modul bereitgestellt werden. Im Beispiel muss die Datenaufbereitung in einer schnellen Abtastzeit arbeiten, beispielsweise 25 us für 240 Abtastwerte einer drehenden Welle mit 10000 U/min. Die bereitgestellten Daten sind aber statische Daten der Drehbewegung und beispielsweise in einem Takt von 10 ms weiter zu verarbeiten. Angenommen, dies passiert in 2 verschiedenen Prozessoren. Die 25 us-Abtastzeit läuft in einem Prozessor ohne Betriebssystem im Interrupt. Die Daten werden im Zielsystem in ein Dual-Port-RAM geschrieben, beispielsweise mit einem FPGA realisiert. Der zweite Prozessor hat ein Linuxkernel, Kommunikations-Schnittstellen und/oder ein User-Interface und kann den 10 ms-Interrupt passend verarbeiten. Das Datenschreiben in 25 us und das Datenlesen in 10 ms arbeiten damit aber nicht zeitlich koordiniert, asynchron. Es kann sein, dass während des Lesens neu geschrieben wird.

Die Daten sollten konsistent sein. Die Frage der Organisation der Abarbeitung hat wenig mit der eigentlichen Funktion zu tun und wäre viel zu komplex in das Simulink-Modell hineinzutragen. Daher sind diese Dinge in C besser aufgehoben, möglicherweise auch in personeller oder Verantwortungs-Trennung für deren Programmierung.

Dennoch muss die Entscheidung, welche Daten in das Dateninterface stehen sollen, aus der Simulink-Ebene einfach zugänglich sein: In Simulink wird bestimmt, um welche Daten es geht. In der Hardware- Betriebssystem- und Ablauforganisations-Ebene wird bestimmt wie die Daten transportiert werden und wie auf sie zugegriffen wird.

Zum Modell zugehörig gibt es einen Headerfile, im Unterverzeichnis `+genSfn`. Dieses ist im Simulink-Editor bearbeitbar und wird dort auch für den C-Code highlighted dargestellt (hier gekürzt):

```
typedef struct Data_Output_Test_OrthOsc2_t
{
  /**A sequence number to check the consistence of data. */
  int32 seq;

  /**The static values of fundamental osciallation. */
  float_complex pq1[2];

  /**Magnitude of the fundamental oscillation*/
  float m1[2];
  .....
} Data_Output_Test_OrthOsc2;

/**ObjectFB which contains the outputs.
* @param idenntObj for debugging
* @simulink Sfunc
*/
INLINE_Fwc void ctor_Output_Test_OrthOsc2(Output_Test_OrthOsc2* thiz, int identObj)
{ ....

/**Sets the values of the ObjectFB which contains the outputs.
```

```
* @simulink Sfunc
*/
INLINE_Fwc void set_Output_Test_OrthOsc2
 ( Output_Test_OrthOsc2* thiz, float_complex pq1[2]
 , float m1[2], float m2[2], float m3[2])
{ .....
```

Will man nun aus der Simulink-Intension heraus andere oder weitere Daten einbringen, dann ist die Editierung dieses Files auch einem Regelungsentwickler zumutbar, ähnlich wie auch Matlab-Scripts textuell sind. Die Schreibweise ist übersichtlich. Die hier mit `.....` angedeuteten Implementierungen der Operationen stehen ebenfalls direkt im Header und haben keine hohe Komplexität. Selbst die Organisation des Wechselpuffers ist verstehbar. Die set-Operation lautet hier im Volltext:

```
/**Sets the values of the ObjectFB which contains the outputs.
* @simulink Sfunc
*/
INLINE_Fwc void set_Output_Test_OrthOsc2(Output_Test_OrthOsc2* thiz
  , float_complex pq1[2],float m1[2], float m2[2], float m3[2])
{
  int32 seq = thiz->ix +1;
  Data_Output_Test_OrthOsc2* data = &thiz->data[seq & 1];
  data->seq = seq;
  memcpy(data->pq1, pq1, sizeof(data->pq1));
  memcpy(data->m1, m1, sizeof(data->m1));
  memcpy(data->m2, m2, sizeof(data->m2));
  memcpy(data->m3, m3, sizeof(data->m3));
  thiz->ix = seq;  //write back for usage after fill data.
}
```

Der C-Programmierer erkennt schnell die Funktion. Wenn weitere Daten eingebracht sind, dann wird eine weitere gleichartige Zeile '*nach Schema*' ergänzt. Kleine Flüchtigkeits-Schreibfehler werden beim Compilieren gemeldet.

Nach Editierung wird das Script `+genSfn/TestOrthOscObjO_genSfn.m` ebenfalls aus dem Matlab heraus gestartet. Damit wird die Sfunction neu erzeugt und kann sofort verwendet werden. Die Anpassung im Modell ist in Simulink beschrieben und bekannt (Eingänge verdrahten).

5 Arbeitsschritte, Tools, Codegenerierung

Der hier beschriebene Ansatz geht von folgendem aus:

- Kern-Funktionen und Datenstrukturen werden in C formuliert.
- Aus den Headerfiles wird der Code der Wrapper für Simulink-Sfunctions generiert. Diese rufen dann direkt die C-Kernfunktionen auf.
- Aus den Datentypdefinitionen in den Headerfiles werden Matlab-Busdefinitionen in `*.m`-Files generiert, die dann im Workspace des Matlab zur Verfügung stehen.
- Aus den Headerfiles werden die sogenannten `tlc`-Files generiert, die Simulink sowohl für die Codegenerierung für ein Zielsystem braucht als auch für den *Accelarator-Mode* im Simulink.
- Mit den Sfunctions und weiteren Simulink-Elementen wird ein Modell erzeugt, dass die Zusammenschaltung der Kern-Funktionalitäten in C (in den Sfunctions) beschreibt. Dabei wird nach *Object-FB* und *Operation-FB* unterschieden und eine ObjektOrientierte Denkweise unterstützt. Der Aufruf der Operation-FB zu einem zugehörigen Object-FB kann dabei in einem anderen Modell-Bestandteil erfolgen (adäquat zum Aufruf einer public-Methode einer Klasse aus einer anderen Klasse).
- Ein Simulink-Modul kann nun wiederum so aufgebaut werden, dass es als *Atomaren Subsystems* oder referenziertes Modell auch als Object-FB erscheint mit einem Handle nach außen und Handles von außen für Operation-FBs (Operation-Module). Das ObjektOrientierte Prinzip kann also auch auf höherer Ebene im Simulink verwendet werden.
- Aus einem so gebildeten Gesamt-Simulink-Subsystem wird Code für ein Zielsystem generiert.
- Die Einbindung des Codes in die Laufzeitumgebung des Zielsystems wird wie gewohnt in C realisiert. Die aus Simulink generierten Codes werden in die bestehende Generierumgebung eingebunden.
- Es wird empfohlen den Zielsystem-Code auch als Executable für den PC zu compilieren und auch auf dem PC testen zu können. Eine notwendige Umgebung kann mit Simulink-Mitteln erstellt werden und wird mit der PC-exe insbesondere über Socket-Kommunikation gekoppelt. Dazu gibt es fertige FBs von vishia.
- Im Simulink-Modell kann ein sogenannter *Inspector-Service*-FB angeordnet, über den von außen über Socketkommunikation auf Interndaten des laufenden Simulink-Modells zugegriffen wird. Es sind dies die Interndaten, die als C-Kerne definiert wurden. Das Tool für den Zugriff ist der Inspector mit eine offen gelegten Socket-Kommunikationsprotokoll.
- Der Inspector kann auch im Zielsystem eingesetzt werden.

Damit sind folgende Tools für die Arbeit notwendig:

- C/C++-Entwicklungsumgebung für PC und für das Zielsystem
- Matlab mit Simulink von Mathworks, enthält die Unterstützung für Sfunctions.
- Codegenerierung aus Simulink als Zusatz-Modul für Simulink von Mathworks.
- **Codegenerierung für Wrapper der Sfunctiones, Busse und tlc-Files von vishia**. Dies ist Nicht-Open-Source. Lizenz erforderlich.
- Einige C-Systemfunktionen von vishia, Open-Source.
- Der ZBNF-Parser und JZtxtcmd-Textgenerator von vishia, Open-Source, Java
- Der Inspector von vishia, Open-Source, Java

Der vierte Punkt erfordert eine perönliche Konsultation mit vishia. Wenn eine Erstbetreuung von 4 Arbeitstagen und mindestens ein 1-jähriger Betreuungsvertrag im Wert von insgesamt 8 Arbeitstagen für ein Entwicklungsteam. Dieses Entwicklungsteam erhält damit die Lizenz für zeitlich unbegrenzte Nutzung. Die Betreuung kann

sich insgesamt auf die Unterstützung der Simulink-Handhabung für Codegenerierung im Zielsystem oder auch auf Belange der C-Programmierung oder Modellierung beziehen. Die Scripts der Generierung sind textuell und damit auch eigenständig anpassbar beispielsweise an Simulink-Versionen. Es kann aber im Rahmen des Lizenzvertrages auch eine längerfristige Betreuung vereinbart werden. Die Lizenz ist an ein Entwicklungsteam von ca. 5..10 Personen gebunden. Wird dieses Tool in größeren Firmen abteilungsübergreifend genutzt oder soll es Partnerfirmen weitergegeben werden, so ist dazu jeweils eine eigene Lizenz erforderlich. Bei entsprechendem Umfang kann dies kostengünstiger erfolgen.

6 Gestaltung der Anwender-Header- und C-Files für die Sfunctions

Die Headerfiles in C enthalten alle Informationen die für das jeweilige Modul notwendig sind für die S-Funktionsgenerierung. Außer dem allgemeinen Generierscript laut Kapitel 9.1, Seite 37 sind keine weiteren Scripts notwendig.

6.1 Datendefinition

Das Beispiel soll wiederum als Muster gelten. Der Typ des Object-FB `OrthOsc2_FB`, sagen wir konsequent die `class OrthOsc2_FB` wird in C in einem Headerfile als struct definiert:

```
/**Internal data of a OrthogonalOscillator.
* @simulink no-bus
*/
typedef struct OrthOsc2_FB_t
{
  ObjectJc obj;  //:The base structure
  Param_OrthOsc2_FB* par;  //:Reference to parameter, maybe calculated in other time.
  Angle_abwmf_FB* anglep;  //:Reference to angle, null is admissable.

  /**Couple factors. Note: kB should be negative for same difference B-X, A-X*/
  float kA, kB;

  float_complex yab;  //:Orthogonal components of oscillation.
  float_complex ypq;  //:optional: Orthogonal components as fundamental values, ...
  float m, mr;        //:optional: Magnitude and its reciproke, if calculated
  //
  float b_;           //:internal b component
} OrthOsc2_FB;
```

Die `struct`-Definition fasst Daten zusammen und ist der erste Schritt hin zu einer Objektorientierung. Dies sind die Daten des Object-FB im Simulink.

Wenn bestehende Quellen in dieses Schema gepackt werden sollen um Simulink zu nutzen, bisher aber eher auf statische Einzelvariable in C-Files orientiert wurde, dann stellt die Zusammenfassung zueinandergehöriger Variable in einer `struct` der erste wichtige Schritt dar, der im konventionellem C begangen wird. Einzelvariable sind in Speicherplatz und Adressierung im Maschinencode identisch mit der Zusammenfassung der Variablen in einer `struct` und deren statischer Instanziierung. Es muss lediglich formell etwas umgeschrieben werden. Dies ist aber der entscheidende Schritt beispielsweise auch zu einer mehrfachen Instanziierung eines Moduls, sei es auch nur für Testzwecke.

Die `typedef struct` selbst und die Einzelelemente können und sollten im Header kommentiert werden. Hier wird entweder die sogenannte Javadoc-Notationsform verwendet, angelehnt an die sehr einheitliche Schreibweise der Kommentierung in Java, oder es erfolgt eine platzsparende Zeilenend-Kommentierung. Für die Codegenerierung für Simulink werden die Kommentare dann automatische auch in den Simulink-Einheiten erscheinen.

Die `struct` für einen Object-FB muss mit dem Element

```
ObjectJc obj;
```

beginnen. Das ist notwendig weil im Simulink die Verbindung der FBs geprüft werden muss. Ein weiterer Grund ist die Unterstützung des Zugriffs auf die internen Daten mittels des Reflection-Mechanismus, siehe Kapitel 12.3 Seite 55. Das Datenelement `ObjectJc` benötigt insgesamt 24 Byte und muss nur bei den Hauptstrukturen der Object-FB vorhanden sein.

6.2 Datendefinition als Simulink-Bus versus private Daten

Wenn eine solche `typedef struct` mit dem Kennzeichen (*tag*)

```
  * @simulink bus
```

im Kommentar versehen ist, dann wird bei der Codegenerierung für Simulink eine Simulink-Busdefinition in ein `*.m` matlab-Script generiert, dessen Abarbeitung dann die `struct`-Daten als Matlab-Busdefinition abbildet. Man kann die Daten dann als Bus in Simulink ausgegeben und mit den Simulink-Modellmitteln dessen Datenelemente abgreifen. Zu beachten ist jedoch, dass der Bus bei der Codegenerierung auf einem eigenen Datenbereich platziert wird, also Daten kopieren. Für die Rechenzeiteffektivität ist das nicht die günstigste Lösung. Wenn aber im nichtrechezeitkritischem Bereich mit Daten im Simulinkmodell gut gearbeitet werden kann, dann ist dies zu empfehlen.

`struct` die als Busse abgebildet werden sollten in der Regel keine Zeiger enthalten. Die Anordnung der Daten muss so gewählt werden, dass bestimmte Alignment-Dinge beachtet sind. Dies gilt auch für einige Zielplattformen. Siehe dazu `http://www.vishia.org/SwEng/html/int_short.html`. Die internen Simulink-Busdaten werden direkt auf die `struct` abgebildet. Die Richtigkeit dieses Mapping wird in der startup-Phase überprüft.

Die internen Daten der Function-FB brauchen in der Regel nicht als Bus im Simulink präsentiert werden. Grundsätzlich kann man hier objektorientiert denken und die Daten als *private* ansehen. Um die Daten zu beobachten, im Simulink in der Run-Phase, können die Inspector-FBs verwendet werden, siehe 12.1, Seite 52. Der FB `GetValue_Inspc` liest Daten aus intern-struct, gibt sie am Ausgang aus, anschließbar beispielsweise an einen *Scope* im Simulink, und ist textuell konfigurierbar.

6.3 C-Funktions-Prototypdefinitionen für Object-FB und Operation-FB

Im Headerfile nach der Datendefinition folgen entweder Funktionsprototypdefinitionen oder Inline - Funktionsdefinitionen. Nach dem Schema der Objektorientierten Programmierung in C tragen die Funktionen als Suffix den Namen der zugehörigen Datenstruktur und übernehmen als erster Argument `thiz` den Zeiger auf die Daten. Funktionen, aus denen Simulink-FBs gebaut werden sollen, sind im Kommentar mit `@simulink ...` bezeichnet.

```
/**Initializes thiz object with given parameter.
 * @param identObj Identifier, will be stored in the ObjectJc base data, for debugging.
 * @param Tstep The step- or sampling time for this module.
 * @simulink ctor
 */
OrthOsc2_FB* ctor_OrthOsc2_FB(ObjectJc* othiz, int32 identObj, float Tstep);
```

Dies ist der *Constructor* eines *Object-FB*. Die Kennzeichnung als Constructor erfolgt mit

```
  @simulink ctor
```

Im Gegensatz zu allen anderen objektorientierten C-Funktionen übernimmt der Constructor wenn so angegeben nicht den Typzeiger sondern einen `ObjectJc*`-Zeiger. Die Instanz muss mit 0 initialisiert sein, die Kopfdaten des `ObjectJc`, insbesondere die eigene Adresse und die Byteanzahl des gesamten Object müssen gesetzt sein. In den generierten Codes ist dies so vorgesehen.

Alle Arguemnte des Constructors sind in Simulink *non-tunable Parameter* der Sfunction. Sie werden in Simulink entweder numerisch als `double` oder als String angegeben und entsprechend konvertiert. Der Rückgabetyp des Constructor ist der Instanzzeiger selbst, mit gleichem Wert wie `othiz`. Diese Schreibweise hat eher semantischen Charakter für die direkte C-Programmierung.

```
/**Prepares the instance data.
 * @param par aggregation to the parameter.
 * @param angle aggregation to instance which contains the angle of the signal.
 * @simulink init
 */
void init_OrthOsc2_FB(OrthOsc2_FB* thiz, Param_OrthOsc2_FB* par, Angle_abwmf_FB* angle);
```

Die mit `@simulink init` gekennzeichnete C-Funktion ist die Initialisierung des Object-FB, die Versorgung mit Aggregationen zu anderen FBs. Die Aggregierung sollte nach der Theorie zwar bereits im Constructor erfolgen. Praktisch funktioniert dies aber auch in der UML oder in Java-Anwendungen nicht, wenn Aggregationen gegenseitig stattfinden sollen. Daher ist auch in anderen Anwendungen eine sogenannte *Nachinitialisierung* gebräuchlich. Alle Zeiger-Argumente dieser C-Funktion werden in Simulink auf Eingänge der `Tinit`-Zeitkonstante definiert. Alle Nicht-Zeiger-Argumente sind *non-tunable Parameter*. In Simulink bestimmt die Verdrahtung der Tinit-Eingänge der FBs die Aggregationen. Ein FB gibt seinen eigenen Datezeiger (`thiz`) an andere FBs weiter, die diesen als Aggregation speichern. Damit kann in schnellen Zeitkonstanten die Daten der aggregierten FBs direkt gelesen bzw. deren Operationen aufgerufen werden.

```
/**Step routine. It calulates the stored values of Orthogonal Oscillation.
 * @param xAdiff Difference between Input and yaz_y Signal
 * @param xBdiff same as xAdiff for only single input, or orthogonal difference
 * @param yaz_y variable to store the a-Output.
 * @param ab_Y variable to store the complex orthogonal output..
 * @simulink Object-FB, accel-tlc
 */
INLINE_Fwc void step_OrthOsc2_FB(OrthOsc2_FB* thiz, float xAdiff, float xBdiff
                                 , float* yaz_y, float_complex* ab_y)
{
  //C-Code
}
```

Dies ist die inline-Formulierung der step-C-Funktion. Das Makro `INLINE_Fwc` wird entweder mit `inline` für C++-Compilierung ersetzt, oder mit `static` für C-Compilierung. Mit dieser inline-Formulierung wird im Zielsystem die Laufzeit optimiert.

Mit der Kennzeichnung `@simulink ObjectFB` ist dies die Operation eines Object-FB, damit wird also eine Sfunction gebildet. Die in der Reihenfolge im Headerfile davor aufgefundenen `@simulink ctor` und `@simulink init` C-Funktionen gehören zu dieser S-Funktion als Operation-FB bzw. auch zu weiter folgenden Operation-FB. Will man mit den gleichen Daten verschiedene Operation-FB mit verschiedenen Constructoren haben, dann kann man auch mehrere `@simulink ctor` usw. angeben, es gilt die Reihenfolge im Headerfile. Im file `srcc_FB/OrthOsc_FB.h` zugehörig zum Beispielmodell findet sich neben diesem Object-FB der `stepNoAngle_OrthOsc2_FB` mit selben Constructor aber anderer `init`Funktion.

Mit der Kennzeichnung `@simulink ... , accel-tlc` wird erreicht, dass beim Accelerator-Mode des Simulink der Inhalt des tlc-Files verwendet wird, um optimalen Ablaufcode zu generieren. Das ist meist zweckmäßig, aber in wenigen Fällen nicht möglich. Siehe dazu Erläuterungen von Mathworks zum Flag `SS_OPTION_USE_TLC_WITH_ACCELERATOR`.

Die Argumente sind entweder Eingänge (numerisch), Ausgänge oder tunable-Parameter, siehe Folgekapitel.

Ein Object-FB ohne Sonderkennzeichnung allokiert den notwendigen Speicherplatz innen, speichert in einem sogenannten *DWork vector* den Zeiger auf den Speicherplatz und einen zugehörigen Handle als Index auf eine Zeigertabelle als uint32-Wert. Der Handle wird am Ausgang 1 rechts oben für die Tinit-Zeitkonstante ausgegeben, und

darunter für die step-Zeitkonstante der Operation. Für ein 32-bit-Zielsystem führen diese Ausgänge direkt die Speicheradresse als uint32 typisiert abgelegt. Der Tinit-Ausgang des thiz-Handle soll für die Aggregations-Verdrahtung benutzt werden, der Tstep-Ausgang für Opertion-FBs zu diesem Object-FB.

```
/**Outputs the ab vector of this.
 * @simulink Operation-FB, accel-tlc
 */
INLINE_Fwc void ab_OrthOsc2_FB(OrthOsc2_FB* thiz, float run, float_complex* ab_y)
{
 ab_y->re = thiz->yab.re;
 ab_y->im = thiz->yab.im;
}
```

Dies ist ein einfacher Operation-FB. Das `thiz`-Argument erscheint hier als erster Input, der gesamte FB hat eine Abtastzeit. Diese wird folglich vom Anschluss des Handle des zugehörigen Operation-FB bestimmt, oder von passend eingesetzten *Rate transition blocks* des Simulink und von den anderen Signalen.

6.4 Alle @simulink Notationsmöglichkeiten an struct und Routinen im Header

Syntaktisch sind die Notationsmöglichkeiten beim Parsen des Headerfiles bei der Simulink-Generierung festgelegt. Der Parser mit dem Syntaxfile `zbnfjax/zbnf/Cheader.zbnf` verarbeitet auch die Comment-Anteile. Es wird damit vermieden, dass es wegen eines Schreibfehlers Irritationen gibt (C-Funktion erscheint nicht als Operation-FB usw.).

Folgende Notationen dürfen nach `@simulink` durch Komma getrennt stehen:

- `bus`: Kennung für eine `typedef struct ...`, es wird ein Matlab-m-File für einen Simulink-Bus generiert. Der generierte Bus kann nur für die reine Simulink-Ebene verwendet werden, in diesem Fall ist die Abbildung des Busses im generierten Code über diese Definition festgelegt. Ein Bus kann auch den S-Functions als Input oder Output übergeben werden, Bezeichnung der Argumentvariable mit `..._bus`, siehe Abschnitt 6.8, Seite 25
- `no-bus`: Kennung für eine `typedef struct ...` ohne Auswirkung, Kennzeichnung dass kein Bus generiert werden soll. Das ist damit eine private-Datenstruktur eines Object-FB wenn in den C-Funktionen entsprechend verwendet.
- `ctor`: Constructor, wird in Simulink in der `mdlInitializeConditions(...)` eines Object-FB gerufen
- `dtor`: Descructor, wird in Simulink in der `mdlTerminate(...)` eines Object-FB gerufen
- `defPortTypes`: Spezialfunktion für `DYNAMICALLY_TYPED` Ein- und Ausgänge. Bei Aufruf in der Initialisierung kann der Typ kann aus den non-tunable Parametern festgelegt werden, siehe Kapitel 6.10, Seite 28. Für die Propagierung von Typeigenschaften von aus dem Modell festgelegten Ports auf nicht festgelegte Ports.

```
/** @simulink defPortTypes*/
void propagateType_OrthOsc2_FB (
  Entry_DefPortTypeJc* types,
  EDefPortTypesJc what);
```

- `init`: C-Funktion zur Festelegung der Aggregationen und weiterer Parameter eines Object-FB, wird in Simulink in der `mdlOutputs(...)` eines Object-FB gerufen, solange die Initialisierung noch nicht vollzogen ist.
- `Object-FB`: C-Funktion als step-Routine eines Object-FB. Diese Funktion wird als S-Funktion generiert mit den davor aufgefundenen ctor, dtor, init queryOutputPorts.
- `Operation-FB`: C-Funktion als step-Routine eines Operation-FB. Diese Funktion wird als S-Funktion generiert
- `accel-tlc`: Zusatzkennung für einen Object-FB oder Operation-FB: Flag `SS_OPTION_USE_TLC_WITH_ACCELERATOR` wird in der `mdlInitializeSizes(...)` gesetzt.
- `no-thizInit`: Zusatzkennung für einen Object-FB: Es wird kein Ausgang für das thiz-Handle der Tinit-Zeitkonstante, also für Aggregationen, erzeugt. Dieser Object-FB ist nicht für Aggregationen vorgesehen.
- `no-thizStep`: Zusatzkennung für einen Object-FB: Es wird kein Ausgang für das thiz-Handle der Tstep-Zeitkonstante erzeugt. Es wird damit bestimmt, dass dieser Object-FB keine weiteren Operation-FB hat. Er kann aber entweder als Aggregation verwendet werden, oder er hat nur numerische Ausgänge oder ist eine Datensenke (*sink*), beispielsweise für die interne Ablage von Daten.
- `step-in`: Zusatzkennung an Object-FB: Es wird ein Eingang angelegt, der nur für die Organisation der Abtastreihenfolge zuständig ist. Der Eingang wird nicht in der Sfunction verwendet und daher bei der Compilierung wegoptimiert. Siehe Kapitel 7.1, Seite 31.
- `step-out`: Zusatzkennung an Object-FB: Es wird ein Ausgang angelegt, der nur für die Organisation der Abtastreihenfolge zuständig ist. Der Ausgang wird nicht in der Sfunction verwendet. Er führt eine 0 und wird bei der Compilierung dann wegoptimiert, wenn er nur auf ungenutzte Eingänge geführt ist. Siehe Kapitel 7.1, Seite 31.

6.5 Schreibweise der Argumente der relevanten C-Funktionsprototypen

Für die Schreibweise der Argumente gibt es Regeln:

- Alle Funktionsdefinitionen müssen als erstes Argument den `thiz`-Zeiger erhalten. Das ist der Zeiger auf die Instanz- oder Object-Daten. `thiz` entspricht dem `this` in C++. Es wird hier mit `z` geschrieben, da zur Compilierung häufig ein C++-Compiler benutzt wird. Aus Sicht von C++ handelt es sich hier aber um eine normale Variable, `this` ist als C++-keyword reserviert.
- Ausnahme: Der `@simulink ctor` soll als erstes Argument einen `ObjectJc* othiz` erhalten. Dann muss aber der Typ der zugehörigen Daten als Zeigertyp des Rückgabewertes definiert sein. Ein Constructor darf aber auch den typrichtigen `thiz` erhalten, dies ist für die Simulink-Generierung zulässig, entspricht aber nicht den Regeln der objektorientierten C-Programmierung mit CRJ.
- Zeigertypen werden in Simulink auf der Sfunction-Ebene als Handle-Inputs oder Handle-Outputs realisiert, wenn der Name des Arguments nicht auf `_bus` endet. Handles sind 32-bit `uint32` und werden Sfunction-intern über eine Zuordnungstabelle mit den Adressen ersetzt, bevor die eigentliche C-Kernfunktion gerufen wird. Für die Codegenerierung auf 32-Bit-Zielsysteme werden an selber Stelle dann direkt die Speicheradressen weitergegeben.
- Alle Argumente des Constructor `@simulink ctor` sind non-tunable Parameter des FB in Simulink. Daher sind hier keine Zeiger zulässig sondern nur skalare numerische Werte oder Texte als `StringJc`-Typ. Die Parameter sind in Simulink nicht *tunable* sondern sind mit der startup-Phase fest.
- Das Argument `Tstep` des Constructors bestimmt als Simulink-Parameter die Abtastzeit des FB. Es muss ein `float` oder `double` sein. Der Wert der Abtastzeit kann zweckmäßig in den Interndaten (`thiz`) gespeichert werden, muss aber nicht. Der Parameterwert wird in der `mdlInitializeSizes(...)` der Sfunction eingelesen und den Ports zugeordnet. Wenn kein `Tstep`-Argument angegeben wird, dann wird die Abtastzeit aus den Input-Ports die den Aufrufargumenten der step-Funktion zugeordnet sind bestimmt. Alle Inputports der step-Funktion müssen aus der selben Abtastzeit versorgt werden.
- TODO: Das Argument `TstepOffset` des Constructors bestimmt die Scheibe (*Slice*) in die der FB in die Abtastzeit eingenordnet wird. In der Sfunction in Simulink wird `ssSetOffsetTime(...)` damit aufgerufen. Man kann FBs mit der gleichen untersetzten Abtastzeit in Simulink in verschiedene Scheiben einordnen. Für die Codegenerierung aus Simulink für das Zielsystem wird für jede Scheibe eine eigene step-Funktion generiert. Diese kann dann im Zielsystem ebenfalls untersetzt aufgerufen werden.
- String-Parameter dürfen beim Constructor und `@simulink init` angegeben werden, und zwar mit der Typkennzeichnung `StringJc`. Dieser Typ in `Fwc/fw_String.h` definiert enthält den `const char*` zeiger und die Länge des Strings. Die Weiterverarbeitung kann im normalen C erfolgen. Die Übergabe von `const char*` direkt, als 0-terminierter String wie im Standard-C üblich, ist derzeit (Stand 2017-10) nicht vorgesehen.
- Numerische Parameter werden als double im Parameterfeld der Sfunction notiert, werden aber vor der Übergabe in den Ziel-typ gecastet.
- Zeigertyp-Argumente des `@simulink init` die nicht auf `_bus` oder `_ybus` enden, sind werden als Handle-Inputs (uint32) der Tinit-Zeitkonstante des damit gebildeten Object-FB umgesetzt. Das sind die Argumente für Aggregationen.
- Alle anderen Argumente des `@simulink init` werden wie bei einem `@simulink Object-FB` behandelt, aber in der Tinit-Zeitscheibe. Insbesondere können `_param` angegeben werden, aber auch numerische Inputs und Outputs oder `_bus` Inputs und `_ybus`-Outputs. Die init-C-Funktion wird nur solange zyklisch aufgerufen, bis intern das Flag `isInitialized((ObjectFB*)thiz)` gesetzt ist. D.h. die Inputs- und Outputs müssen

entweder aus Konstanten gebildet werden, auch aus einem Bus aus Konstantwerten, für Parametrierungen, oder von anderen Object-FB stammen aber mit Lieferung der Aggregations-Zeiger auch geliefert sein. Für den generierten Code des Zielsystems wird die init-Routine nur beim Hochlauf mehrfach aufgerufen.

- Argumente für `@simulink init`, die auf `_param` enden, sind non-tunable Parameter der damit gebildeten Sfunction.
- Argumente für `@simulink Object-FB` oder `@simulink Operation-FB` die auf `_param` enden, sind tunable Parameter der damit gebildeten Sfunction.
- Werden bei `step...()` Zeiger von nicht-Skalartypen verwendet, deren Argument-Name nicht auf `_bus` endet, so sind dies für die FBs in Simulink Handle. Der Handlewert wird aber in der `Tstep`-Zeit verarbeitet und kann beispielsweise zwischen mehreren FBs umgeschaltet werden oder auch 0 sein. Im UML-Sinne sind das *Association*, keine *Aggregation*.
- Zeiger von Nicht-Skalartypen, deren Namen auf `_bus` enden, werden für Simulink als Bus-Inputs umgesetzt. Der Typ muss also als `@simulink bus` gekennzeichnet sein und im Matlab als Bus-Typ exisitieren. Insbesondere ist dies für Signale in Simulink gedacht, die über einen *Bus-Creator* im Simulink zusammengefasst werden. Eine Bus-Instanz wird im Simulink intern als Speicherbereich geführt. Der Kernfunktion sowohl in der Sfunction als auch im generierten target-Code aus Simulink wird ein Zeiger auf diesen Bereich typgerecht übergeben. Siehe Abschnitt 6.8, Seite 25
- Zeiger von Nicht-Skalartypen, deren Namen auf `_ybus` enden, werden für Simulink als Bus-Outputs umgesetzt. Der Typ muss also als `@simulink bus` gekennzeichnet sein und im Matlab als Bus-Typ exisitieren. Eine Bus-Instanz wird im Simulink intern als Speicherbereich geführt. Der Kernfunktion sowohl in der Sfunction als auch im generierten target-Code aus Simulink wird ein Zeiger auf diesen Bereich typgerecht übergeben. Der Speicherinhalt wird gefüllt. Hinweis: Das Speicherabbild des Busses in Simulink muss mit dem Speicherabbild in C übereinstimmen. Dies wird in der `mdlInitializeSizes(...)` geprüft. Zur Bildung der `typedef struct [...]` für einem Simulink-Bus müssen die Regeln des Alignment beachtet werden, siehe Hinweise in 6.1 Seite 14. Dies gilt auch für die Bus-Inputs. Siehe Abschnitt 6.8, Seite 25
- Numerische Skalartypen sind normale Input-Größen in Simulink. Die Skalartypen werden mit `float`, `double`, `float_complex`, `double_complex`, `int32`, `uint32`, `int16`, `uint16`, `int8`, `uint8`, `boolean` bezeichnet. Im C sind die nicht-Standard-Typen in einem Headerfile `compl_adaption.h` definiert, siehe Kapitel TODO. In Simulink werden die dazu passenden Simulink-Typen zugeordnet.
- Complex-Größen müssen in C mit `float_complex` und `double_complex` gekennzeichnet werden.
- Für numrische Inputs können Zeiger verwendet werden, um eindimensionale numerische Vektoren zu verarbeiten.Entsprechend werden im Simulink Vektor-Inputs der angegebenen Dimension erzeugt. Für C werden Vektoren bekannterweise als Zeiger übergeben. Der Speicherbereich der Vektoren liegt außerhalb. In C kann also auch zurückgeschrieben werden. Das Rückschreiben ist aber für Simulink so nicht vorgesehen und funktioniert deshalb nicht, weil in der Simulations-Engine des Simulink die Vektoren möglicherweise auf eine Zwischenspeicher vor der Zeigerbildung umkopiert werden. Zurückschreiben auf Input-Vektoren im Code der Funktionen darf also nicht genutzt werden. Konsquenterweise sollten die Zeiger als `float const*` oder `const float *` gekennzeichnet werden. Dies wird aber nicht extra geprüft.
- Numerische Outputs der Object-FB und Operation-FB müssen als Zeiger realisiert werden, und im Namen mit `_y` enden.
- Inputs mit dem Typ `void const*` werden als `DYNAMICALLY_TYPED` und `DYNAMICALLY_SiZED` angelegt. Entsprechend der angeschossenen Quelle in Simulink wird beim Modellstart und bei der Codegenerierung der Typ festgestellt. Der Typ und die Anzahl Elemente wird in ein Array abgelegt, dass als Input mit

dem Namen `InputTypes_` übergeben wird. Damit wird der C-Funktion mitgeteilt welche Inputs aktuell verdrahtet sind. Die Auswertung obliegt der C-Ebene. Über diesen Weg kann eine C-Funktion beispielsweise Vektoren mit verschiedenen numerischen Typen verarbeiten. Hinweis: Die back-propagation von Outputs hierfür wird nicht unterstützt. Jedoch können Outputs aus den Eigenschaften der Input-Typen festgelegt werden, siehe folgend.

- Outputs mit dem Typ `void*` und der Namens-Endung `_y` müssen geeignet in C definiert werden. Dies kann abhängig von Parametern geschehen. Dafür die C-Funktion `@simulink queryOutputPorts` vorgesehen. Dieser muss die Referenz auf ein Array vom Typ `Entry_QueryPortTypeJc` übergeben werden. Das Array wird gefüllt mit den vorgesehenen Typen der Outputs.

Die Codegenerierung für das Zielsystem aus Simulink legt die Speicherbereiche für Vektoren, Skalare Output-Werte und Busse nach einem *Bus-Creator* passend und richtig an und übergibt den C-Kernfunktionen wie erwartet die Zeiger auf diese Bereiche.

6.6 Ausprogrammierung der C-Funktionen, Reflection

Einfache C-Funktionen insbesondere für die Step-Routine (`@simulink Object-FB` oder `@simulink Operation-FB`) sollten ihre wenigen Anweisungen als `INLINE_Fwc` erhalten und werden damit inline in den Maschinencode eingebaut. Das ist effektiv.

Beim `@simulink ctor` oder `@simulink init` ist das so nicht notwendig, da es dabei nicht um schnelle Ausführung geht. Diese C-Funktionen sollten also im C-Code realisiert werden.

Constructor mit ObjectJc-Basisstruktur, Einbindung der Reflection

```
#ifndef __DONOTUSE_REFLECTION__
  #include "AngleBlocks_FB.crefl"
#endif

void ctor_Angle_abwmf_FB(Angle_abwmf_FB* thiz, int32 identObj)
{ memset(thiz, 0, sizeof(*thiz));
  initReflection_ObjectJc(&thiz->obj, thiz, sizeof(*thiz),
    &reflection_Angle_abwmf_FB, identObj);
  thiz->m = 1.0f;  //default for only-angle
  setInitialized_ObjectJc(&thiz->obj);
}
```

Der obige Anwender-Code zeigt, wie das generierte `Headername.refl`-File eingebunden wird. Das `.refl`-File enthält die `const`-Definitinen der Reflections, ist also kein Header. Das File wird aber dennoch includiert und nicht eigenständig sondern hier übersetzt.

Das Includieren ist in bedingte Compilierung geschrieben, denn dieses selbe File soll auch verwendet werden für ein Zielsystem ohne Reflection.

Der Constructor wird vom Simulink in der Sfunction gerufen mit allokiertem und vorinitialisierten Speicher. Die `memset(...0...)`-Operation ist also eigentlich nicht notwendig. Im Zielsystem sollte adäquat verfahren werden. Man kann statische mit 0 intialisierte Instanzen sehr einfach mit

```
Angle_abwmf_FB instance = {0};
```

anlegen bzw. nach dem Allokieren den Speicher sofort mit 0 initialisieren.

Der folgende Aufruf `initReflection_ObjectJc(...)` initialisiert die `ObjectJc`-Daten mit den notwendigen Informationen. In einem einfachen Zielsystem, das keine Reflection verarbeitet und eine `ObjectJc`-Basisstruktur auch nicht braucht, ist in der CRJ-Lib unter *incApplSpecific / TargetNumericSimple / applstdefJc.h* in der dort includierten *ObjectJc_simple.h* eine Definition für `initReflection_ObjectJc(...)` enthalten, die das `&reflection_...`-Argument ignoriert. Dann wird es auch nicht syntaktisch geprüft, muss also nicht vorhanden sein. Damit wird ohne Quelländerung die Kompatibilität von Quelltext für Sfunction, PC-Simulation und Zielsystem erreicht.

Scharfe Prüfung der Daten beim Constructor

```
OrthOsc2_FB* ctor_OrthOsc2_FB(ObjectJc* othiz, int32 identObj, float kA, float kB)
{ OrthOsc2_FB* thiz = (OrthOsc2_FB*) othiz;
  if(-1 != checkStrict_ObjectJc(othiz, (int32)sizeof(OrthOsc2_FB)
                                      , ident_newAllocated_ObjectJc, null, null)){
    initReflection_ObjectJc(othiz, thiz, sizeof(*thiz)
                            , &reflection_OrthOsc2_FB, identObj);
    thiz->kA = kA;
    thiz->kB = kB;
  }
  return thiz;
}
```

Wenn Software an bestimmten Stellen prüft ob Bedingungen stimmen, dann werden eingeschlichene Fehler erkannt. Wird offensichtlich geprüft dann stehen die Bedingungen an dieser Stelle fest, ohne dass ein nochmaliger Blick auf das Gesamtsoftwaresystem notwendig ist. Das erleichtert auch die Erreichung eines erforderlichen SIL-Levels der *Sicherheitsrelevanten Software*.

Dieses Beispiel zeigt die Übergabe nur des ObjectJc-Zeigers an den ctor. Der Speicherbereich wird im Sfunction-Wrapper mit der richtigen Größe allokiert und in den ObjectJc-Daten mit einer Defaultbelegung versehen. Diese umfasst:

- Die eigene Speicheradresse steht im Speicher
- Eine Size-Information wird abgelegt.
- Eine ident-Information `= ident_newAllocated_ObjectJc` wird abgelegt.

Im Zielsystem wird mit malloc-Nutzung ebenso verfahren. Werden die Daten im Zielsystem statisch angelegt, dann kann man einen Initializer benutzen:

```
OrthOsc2 myOrth = INITIALIZER_ObjectJc( myOrth, null, ident_newAllocated_ObjectJc );
```

Der `INITIALIZER_ObjectJc(...)` liefert die `{...}` mit der richtig eingetragenen Initialbelegung, siehe CRJ/source/objectBaseC.h.

Die Abfrage `checkStrict_ObjectJc(...)` testet nun, ob der übergebene Zeiger einem ObjectJc entspricht. Dazu muss die Speicheradresse mit der eigenen Adresse übereinstimmen, damit würde eine Verschiebung der Daten erkannt. Die size-Info muss größer oder gleich der verlangten sizeof(*thiz) sein, und auch das ident_newAllocated_ObjectJc muss eingetragen sein. Damit handelt es sich um eine noch nicht verwendete Instanz der passenden Größe und dies ist für die weitere Verarbeitung wichtig. Man kennt ja Programme, die ganz am Anfang Zeiger mal verwechseln, dann den falschen Speicherbereich mit richtigen Daten überschreiben, dann funktioniert in diesem Modul erstmal alles korrekt. Der Fehler fällt an ganz anderer Stelle schwer debugbar auf. Das wird durch diese Verarbeitung verhindert.

Die init-Routine

```
void init_OrthOsc2_FB(OrthOsc2_FB* thiz, Param_OrthOsc2_FB* par, Angle_abwmf_FB* angle)
{
  if( ! isInitialized_ObjectJc(&thiz->obj) && par !=null && angle !=null) {
    //yet complete:
    thiz->par = par;
    thiz->anglep = angle;
    setInitialized_ObjectJc(&thiz->obj);
  }
}
```

Die mit `@simulink init` gekennzeichnete Routine übernimmt Aggregationen von anderen Sfunctions. Das Beispiel zeigt, dass beide Aggregationen auf nicht-null geprüft werden. Im Initialisierungslauf kann ein mehrfacher Durchlauf der Initialisierung notwendig sein, wenn Aggregationen wechselseitig oder zirkular bestehen. Das ist teils notwendig. Dann werden erst im Folgedurchauf die Ausgangsports belegt, die für eine Aggregation einer Instanz davor benötigt werden. Die Ausgangsports bleiben belegt, so dass mit wenigen Durchläufen die Aggregationen bereitstehen. Dann wird die Instanz auf `isInitialized_ObjectFB()` gesetzt. Diese Bedingung wird in der step-Abtastzeit der Instanz abgefragt in der DFunction im Simulink-Lauf abgefragt. Im Zielsystem wird die init-Routine mehrmals gerufen. Eine Abfrage auf `isInitialized_ObjectFB()` erfolgt dort nicht wenn nicht in der Anwenderprogrammierung enthalten.

Gibt es keine `@simulink init`-Routine, dann muss das `setInitialized_..(...)` im Constructor gesetzt werden, wie im obigen Beispiel für `ctor_Angle...` gezeigt.

6.7 Allokierung des internen Speichers

Die Zeiger, die als Handle geführt werden, sind immer Zeiger auf Interndaten. Diese werden in den generierten Sfunction-Wrapper der FBs in Simulink über `malloc` angelegt. Sie werden auch im generierten Code für das Zielsystem über eine angegebenen Allokierungsfunktion angelegt, siehe Kapitel 9 Seite 36, oder sie werden als statischer Speicher im Zielsystem-Code pro Instanz angelegt (reentrant) und dann jeweils den C-Kernfunktionen als Zeiger übergeben. Gleiches gilt für den `thiz`-Zeiger und die Instanzdaten selbst.

6.8 Busse als Input und Output / 8-Byte-Alignment

Ein Bus im Simulink fasst Daten zusammen. Simulink kennt mehrere Möglichkeiten der Datenbündelung. Der *non-virtual Bus* wird dabei im generiertem Code immer von einer `struct` in einem Header repräsentiert. Man kann im Simulink bei der Busanlage wählen, ob die `struct`-Definition, bzw. der zugehörige Header *Imported* oder *Exported* ist. Letzteres bedeuted, dass Simulink den Header mit der Codegenerierung mit generiert. Bei *Imported* wird ein Header angegeben, der im generierten Code includiert wird und vom Anwender beigestellt werden muss. In diesem Fall muss die `struct`-Definition im Header weitgehend dem Bus entsprechen. Es ist aus Simulink-Sicht nicht notwendig, dass Anzahl und Anordnung der Datenelemente übereinstimmen. Lediglich die verwendeten Datenelemente müssen in der `struct` auffindbar sein. Es ist in diesem Fall also etwas Freiheit der C-Gestaltung gegeben.

Die Mathworks-Dokumentationen beschreiben beide Varianten gleichermaßen. An einigen Stellen wird auf den Vorteil von *Imported* verwiesen. Es wird eine nutzerfreundliche grafische Bedienung mit dem *Bus Editor* angeboten, der die Nutzung von *Exported* zwar nahelegt, aber auch *Imported* als Anwahl zulässt - hier wieder als Eigenverantwortung des Programmierers, dessen `struct`-Definition dann passen muss.

Für das hier angebotene System der Programmierung *Kerne in C, Organisation in Simulink* ist die *Imported*-Variante der Busse naheliegend. Mehr noch: Es wird die Generierung von Matlab-Script-Files für die Bus-Anlage im Simulink aus dem Header unterstützt. Die `struct` werden dazu mit `@simulink bus` gekennzeichnet siehe Abschnitt 6.4, Seite 18.

Werden in einer C-Routine für Sfunction nichtskalare Datentypen als Zeigerargumente verwendet, dann wird ohne weitere Kennzeichnung daraus ein Handle gebildet. Das ist eine Referenz, kein Bus. Wenn als Input oder Output ein Bus verwendet werden soll, dann müssen die Namen der Argumente auf `_bus` bzw. `_ybus` enden, siehe Abschnitt 6.5, Seite 19.

Alignment der Daten in einer struct - direkte Buszeigerverwendung

In den Sfunction in Simulink, die mit dem *Legacy-Code-Tool* von Mathworks generiert wurden, wird nicht vorausgesetzt, dass die `struct`-Definition im Header mit der Bus-Definition exakt übereinstimmt. Daher wird bei diesen Sfunctions der Inhalt des Busses bei Aufruf umkopiert. Simulink bietet Framework-Routinen wie `ssGetBusElementOffset(...)` an, um die Position der Elemente im Bus im Speicher bei der Simulink-Abarbeitung zu ermitteln. Bei der hier vorgestellten Sfunction-Generierung ist die Übereinstimmung von Bus und `struct` im Header jedoch gegeben, wenn das Bus-Script nicht nach der Generierung geändert wird. Damit ist das Umkopieren grundsätzlich nicht notwendig.

Allerdings gibt es bestimmte Alignment-Regeln. Diese Alignment-Regeln sind aber nicht Simulink-spezifisch sondern treten häufig sogar noch verschärft bei der Codegenerierung für verschiedene Zielprozessoren auf. Nicht jeder Prozessor unterstützt den Zugriff auf beliebige auch ungerade Adressen. Selbst in der Intel-Technologie ist der Zugriff auf beliebige Adressen mittlerweile nicht mehr immer möglich: Wird der Speicher mit einem 32-Bit oder 64-Bit-Datenbus angesprochen, dann ist mindestens für einen sogenannten *Atomic-Access* der Aufwand höher, wenn etwa ein float-Wert über 2 Speicherwort verteilt liegt.

Daher sollte vom Anwender bei der Anlage von `struct` folgende Regel immer beachtet werden: **Ein Datenelement muss in einer struct immer auf einer Adressposition liegen, die durch die Länge des Datenelements teilbar ist.** Also ein float-Wert auf einer durch vier teilbaren Adresse. Wenn man diese Regel bei jeder `struct`-Definition einzeln anwendet, dann ist sie immer erfüllt. **Zusätzlich kommt hinzu: Die Länge einer struct muss durch die Länge der Speicherbreite geteilt werden können.** Das bedeutet, bei 64-Bit-Systemen durch 8. Geht man davon aus, dass die meisten größeren Anwendungen auf einem Zielsystem laufen, dass nicht extrem speicherknapp bemessen ist, dann kann man `struct`-Defintionen gleich so anlegen, dass ihre Länge durch 8 teilbar ist. Damit ist die Kompatibilität mit einem 64-Bit-System, insbesondere der Simulation auf dem PC, gewährleistet. Die Alignment-Regeln kann man

einfach dadurch erreichen, dass die Datenelemente etwas sortiert werden.

```
//falsch:
struct .... {
  char c;
  float val;
  char c2;
}

//richtig:
struct ... {
  float val;
  char c;
  char c2;
  char spare[2];
}
```

Die beiden `spare`-Byte werden auf dem Zielsystem sowieso eingebaut, wenn das Zielsystem den Zugriff nur auf volle Adressen zulässt. Es handelt sich also nicht um Speicher-Verschwendung, sondern um Beachtung der Software-Stilregel *be explicit*. Die `struct`-Länge ist damit durch 8 teilbar. Ist die Anwendung für ein speicherknappes 16-Bit-System vorgesehen und wird die `struct` nicht als Array-Element-Typ verwendet, dann brauchen die `spare`-Elemente nicht angegeben werden und werden im Zielsystem auch nicht realisiert. Wird der `struct`-Typ als Array-Element verwendet, dann werden auf dem PC im Simulink oder auch im Zielsystem die Array-Elemente auf die Speicherbreite alignet. Die `spare` werden dann also automatisch doch wieder angelegt. Daher: *be explicit!* - im Quelltext sichtbar angeben. Dazu noch folgender Hinweis: Werden Daten als Speicherabzug aus einem Zielsystem binär übertragen und auf dem PC danach analyisiert, dann ist auch dafür die korrekte Adressierung der Daten notwendig. Eine autotische verschiedene Interpretation der Datenpositionen von verschiedenen Compilern / Zielsystemanforderungen sollte nicht provoziert werden.

Wird die Alignment-Regel, insbesondere die durch 8 teilbare struct-Länge nicht beachtet, dann kann es im Accelerator- und Rapid-Accelerator-Mode zu einer Fehlermeldung kommen:

```
Unexpected error: Internal BlockIO sizes
do not match for accelerator mex file.
```

Beim Start des Accelerator wird die Größe der gesamten Daten-struct verglichen mit der angelegten Byteanzahl in der Simulation-Engine. Diese stimmt nicht überein bei einem Alignment-Problem, was mit dieser Fehlermeldung gezeigt wird:

```
if ( ssGetSizeofGlobalBlockIO ( S) !=
  sizeof ( B_TestOrthOscObjO_T ) )
  { ssSetErrorStatus( S, "Unexpected
```

Überprüfung der Richtigkeit der Bus-Abbildung mit der struct bei Aufruf der Sfunction

Bei den mit diesem System generierten Sfunctions wird nun bei der PC-Simulation der Bus aus der Simulink-Anlage direkt als Zeiger den Sfunctions übergeben. Das Umkopieren wie bei den Sfunction aus dem *Legacy-Code-Tool* entfällt. Damit aber gesichert ist, dass das Speicherabbild stimmt, wird dieses in der Sfunction in `mdlInitializeSizes(...)` überprüft. Es werden die Namen und Positionen der Datenlemente in `struct` und Simulink-Bus-Speicherabbild verglichen. Für die Namen und Positionen in der `struct` werden dazu die Reflection-Informationen herangezogen. Für verwendete Zeigertypen als Busse müssen also die Reflections in den S-Function mit angegeben werden. Die Reflection werden autotisch generiert, siehe 9.1, Seite 37.

Für den Code auf dem Zielsystem spielt diese Problematik keine Rolle. Die Simulin-Codegenerierung verwendet die `struct`-Definition der Busse im *Imported*-Header ohne eigene zusätzliche Interpretation.

6.9 Busse mit boolean-Elemente und Bitfields in struct-Definitionen

Die Verwendung von boolean-Elemente in Bussen für Boolean-Verknüpfungen in Simulink-Modellen und die Abbildung der boolean als Bits in Bitfields ist interessant und effektiv. Diese Thematik wird auch für dieses System der S-Functions angeboten werden, mit folgender Herangehensweise:

* Keine Mischung von Bitfields in einer struct mit anderen Daten

* Der Bus enthält also nur boolean

* Mit geschachtelten Bussen (Bus im Bus, nested struct) ist dann die Mischung möglich.

* Eine Kombination mit einer Wortabbildung der Bits in einer `union` - Zugriff auch über Maskierung - ist empfohlen.

* Die Zuweisung zwischen Boolean im Bus und Bit im Bitfield wird dann in der Sfunction im Simulink per symbolischer Bezeichnung (Name des Elementes) erfolgen.

* Im Zielsystem kein Problem

Diese Lösung ist derzeit (2017-12) in Erarbeitung

6.10 Dynamische Festlegung der Porttypen - EntryQueryPortTypesJc

In Simulink-Modellen ist es auch mit Sfunctions möglich, die Datentypen und Vektor-Dimensionen der Ports aus dem Modell heraus festzulegen. Für die C-Ebene sind das `void*`-getypten Argumente oder Zeigern auf Arrays mit undefinierter Länge (`float arg[ ]` und dergleichen). Für die Simulink-Ebene der Sfunction-Programmierung gibt es für die Ports die Kennzeichnung `DYNAMICALLY_TYPED` und `DYNAMICALLY_SIZED` sowie `COMPLEX_INHERITED`. In solchen Sfunction-Wrapper müssen Routinen wie `mdlSetInputPortDataType(...port,id)` enthalten sein. Diese werden beim Modellstart von der Simulink-Engine gerufen und teilen der Sfunction den angeschlossenen Datentyp mit. Ein weiterer Routinentyp in dem Sfunction-Wrapper ist `mdlSetDefaultPortDataTypes`. In dieser Routine kann man Ports bestimmen, die nicht von der Verdrahtung im Modell berücksichtigt sind oder Ports anders bestimmen, als es die Verdrahtung ursprünglich ermittelt hat.

Im Modell kann der Typ der Ports sowohl als *forward-propagation* von den Signalen an Input-Ports bestimmt werden, als auch als *backward-propagation* von den Signalen, die am Ausgang verlangt werden. Letzteres kann allerdings erfahrungsgemäß in einem Modell eher zu Widersprüchen führen, es wird daher davon abgeraten.

Der Typ von Ausgangsports kann auch vom Typ der angeschlossenen Signale an Inputs abhängen. Dies ist eine häufige Situation. Berechnet beispielsweise eine Sfunction einen Wert aus angeschlossenen Vektoren, dann können die Vektorelemente float, double oder integer-Typen sein, auch komplexe Größen. Die Resultate richten sich nach den Typen der Eingänge.

Die Codegenerierung wird nun für ein konkretes Modell ausgeführt. Dabei stehen die Typen der Ein- und Ausgänge, nach Durchlauf der oben genannte Sfunction-Wrapper-Funktionen, fest.

Für die C-Ebene wird für diese Ports ein `void*` oder ein Array-Zeiger eines numerischen Typs vorgesehen. Bei angegebenem numerischen Typ ist nur `DYNAMICALLY_SIZED` für das betreffende Port gesetzt. Es wird damit die Adresse der Daten übergeben. In der C-Funktion muss dann für den `void*`-Fall ein entsprechender *Pointer-cast* ausgeführt werden.

Der C-Routine muss allerdings der Type der angeschlossenen Signale bekannt sein, damit ein qualifiziertes Type-Casting möglich ist und/oder die Array-Länge bekannt ist. Das geschieht mit einer `struct DefPortTypesJc`, definiert in `Fwc/objectBaseC.h` in der CRJ-Library. Ein solches Array wird bei der Sfunction-Generierung angelegt, gefüllt mit den Port-Informationen aus dem Modell. Dazu ein Beispiel:

```
/**Example of a static routine with void* arguments which uses info about port types.
 * @simulink Operation-FB.
 */
void copyArray_Example(DefPortTypesJc* portInfo, void const* input, void* output_y){
  if(portInfo->entries[0].type == 'F') {
    int ix;
    for(ix = 0; ix < portInfo-entries[0].sizeArray[0]); ++ix) {
      ((float*)output[ix] = 5.0f * ((float*)input[ix];
    }
  }
}
```

Diese Routine definiert eine Sfunction mit zwei Ports. Diese werden mit `DYNAMICALLY_TYPED` angelegt. Wie folgend noch gezeigt wird, richtet sich aber der Typ und die Elementanzahl des Output nach dem Input. Der Input hängt von der Verdrahtung im Modell ab. Die Routine überprüft nun, als Beispiel, ob ein `float`-Wert (`single` im Simulink) angeschlossen ist und kopiert die Werte, als Beispiel mit 5.0 mulitpliziert.

Sowohl für die Bestimmung der Porteigenschaften aus Parametern, als auch für die Bestimmung einzelner Porteigenschaften aus anderen vom Modell bestimmten Porteigenschaften kann der Anwender eine Routine bereitstellen:

```
/**Routine determines port properties
 * @param defPortTypes contains array reference and length of array.
 * @simulink defPortTypes.
 */
INLINE_Fwc char const* definePorts(DefPortTypesJc* defPortTypes, EDefPortTypesJc cause
StringJc arg1, StringJc arg2_param)
{
  Entry_DefPortTypeJc* portTypes = defPortTypes.ref;
  portTypes[defPortTypes->ixOutputStep] = portTypes[0];   //Set output like input.
  portTypes[defPortTypes->ixOutputStep].newDefined = 1;
  return null;  //no error.
}
```

Die C-Routine ist mit der Kennzeichnung `@simulink defPortTypes` als solche markiert. Als Argument wird jedenfalls `DefPortTypesJc* ...` erwartet. Diese `struct` enthält ein Array vom Typ `Entry_DefPortTypeJc`.

Die Routine kann weitere Argumente entgegennehmen. Das sind in diesem Beispiel Text-Parameter mit `'StringJc'` als Zeiger auf `char const*` und die Länge übergeben. Auch numerische Parameter sind zulässig. Die Parameter müssen allerdings gleichnamig im `@simulink ctor` oder in der `@simulink init`-Routine verwendet werden. Das ist zweckmäßig, damit die selben Werte auch dort verarbeitet werden können.

Diese Routine wird nun an mehreren Stellen benutzt:

=> Vor der Initialisierung der Ports wird in der Sfunction-Wrapper-Routine
`mdlInitializeSizes(...)`
diese Routine mit einem leeren Array `Entry_DefPortTypesJc` gerufen. Das Argument von Typ `EDefPortTypesJc` wird mit

- `kSetFromArg_EPropagatePortTypesJc`

besetzt. Aufgrund der weiteren Parameter kann nun der Typ der Ports aus den Parametern ermittelt werden. Die Ports werden dann erst gar nicht `DYNAMICALY_...` angelegt sondern so wie angegeben.

=> Diese gleiche Routine wird mit

- `kSetType_EPropagatePortTypesJc`
- `kSetSize_EPropagatePortTypesJc`
- `kSetComplex_EPropagatePortTypesJc`

gerufen, wenn nach dem Aufruf der Sfunction-Wrapper-Routinen `mdlSetInput-` und `mdlSetOutputPortDataType(...)` usw. die Routinen

- `mdlSetDefaultPortDataTypes(...)`
- `mdlSetDefaultPortDimensionInfo()`
- `mdlSetDefaultPortComplexSignals()`

gerufen werden. Die entsprechenden Eigenschaften der Ports aus dem Modell sind dann bereits bekannt und den betreffenden Ports zugeordnet. Das Array vom Typ `Entry_DefPortTypeJc` wird vorbelegt mit den bisher definierten Port-Typ-Informationen. Es können dann alle Ports oder bestimmte Ports neu definiert werden, also typischerweise Ausgänge in Abhängigkeit von den Eingangstypen. Es werden dann nach der Routine die Ports gesetzt, für die die Datenzelle `newDefined` auf 1 gesetzt ist. Das müssen also alle bisher `DYNAMICALLY_...` definierte sein, und können auch weitere Ports sein. Nach Durchlauf der Routine werden dann die erwünschten Ports gesetzt. Gibt es dann noch Widersprüche, dann meldet die Simulink-Engine einen Fehler.

Mitteilung der Port-Typinformation für die C-Routinen

Wenn der `@simulink ctor`, die `init`-Routine oder auch die Step-Routine ein Argument vom Typ `DefPortTypesJc*` verlangt, dann wird diese `struct` im Stack angelegt, mit den Informationen der Port-Indices gefüllt und das gefüllte Array `Entry_DefPortTypeJc` bereitgestellt mit den Port-Informationen. Die Routine `@simulink defPortTypes` wird nicht noch einmal gerufen, da die Ports bereits alle definiert sind. Nach der Codegenerierten für das Target sind die Port-Informationen nur noch Konstante, die mit geringem Rechenzeitaufwand dargebracht werden können. Insbesondere im ctor oder in der `init`-Routine können aber die Portinformationen in den Instanzdaten (thiz) geeignet gespeichert werden. Das Beispiel in diesem Abschnitt `copyArray_Example` ist eine statische Operation ohne Instanzdaten und braucht daher diese Informationen direkt.

Ein Eintrag des Typ-Arrays `EntryDefPortTypeJc` hat folgende Elemente:

- `char type`: Es werden für die numerischen Typen die Buchstaben `F D I 4 S 2 B 1` für `F=float D=double, I=int32 4=uint32, S=int16, 2= uint16, B=int8, 1= uint8` notiert. Die Buchstaben `F D I S B` entsprechen den Java-Conventionen der Kurzbezeichnung der *primitive-Types*.

  Für die komplexen Typen werden entsprechende Kleinbuchstaben verwendet, also `f=float_complex, d=double_complex i=int32_complex,` und `s=int16_complex`. Komplexe Zahlen als unsigned oder Byte werden nicht unterstützt.

  Eine `'\0'` an dieser Stelle bedeudet: *Type nicht definiert*, also `DYNAMICALLY_TYPED`.

- `uint8 sizeType`: Diese Information wird mit dem Typ zugehörig abgelegt, damit die Anwenderprogrammierung ohne Aufwand bei `memcpy` und dergleichen gleich die notwendige Byteanzahl kennt ohne die `char type`-Information auszuwerten.

- `uint8 newDefined` ist ein Eintragsplatz als Aufforderung, das Port zu definieren.

- `uint8 nrofDimensions` Enthält die Anzahl der Dimensionen. `0` definiert einen Skalar.

- `uint32 sizeArray[5]`: Ein Array kann im Standardfall bis zu 5 Dimensionen mit max. 4000000000 Elementen enthalten. Das wird in der Praxis sehr viel weniger sein. Wenn die Anzahl der Dimensionen nicht ausreicht, dann wird ein extra Speicher für `sizeArray[...]` vorgesehen und das Datenelement `sizeArray[0]` enthält den Zeiger auf diesen Bereich. Es ist also mit cast zuzugreifen: `(*(uint32**)sizeArray)[ix]`.

Es werden an dieser Stelle nicht die internen Simulink-Kennzeichnungen der Typen verwendet. Das erfolgt um die C-Programmierung eigentlich unabhängig von Simulink zu halten. Es müsste sonst das System der Simulink-Header includiert werden und diese C-Programme hätten außerhalb der Simulink-Anwendung keinen Bestand.

7 Aspekte der Modell-Gestaltung

7.1 Abarbeitungsreihenfolge

In der manuellen C-Programmierung ist die Reihenfolge durch die Sequenz der Zeilen bestimmt. Wenn man ungünstig programmiert, dann werden neu zu berechnete Werte benutzt, die noch nicht berechnet wurden und damit aus der vorigen Abtastzeit stammen. Die Folge sind nicht erwünschte fehlerhafte Totzeiten.

In einem Simulink-Modell wird die Abarbeitungsreihenfolge aus dem Signalfluss bestimmt. Damit können die oben genannten fehlerhaften Totzeiten nicht entstehen. Gibt es einen Widerspruch - neuer Wert wird schon eingeplant für seine eigene Berechnung, dann ist dies eine arithmetische Schleife. Simulink weist auf diesen Fehler hin, die Lösung ist die Nutzung des Altwertes, typischerweise mit einem unified delay-block gespeichert.

Die Objektorientierung bricht allerdings den Datenfluss dadurch, dass die Objekte sich kennen und über die in Simulink in der Tinit dargestellten Aggregationsverbindung. Diese bestimmt aber nicht die Abarbeitungsreihenfolge in einer anderen Abstastzeit.

Zur Illustration soll folgendes Modell dienen. In diesem Modell werden in C programmierte Datenspeicher und Mittelwertbildner zusammengeschaltet.

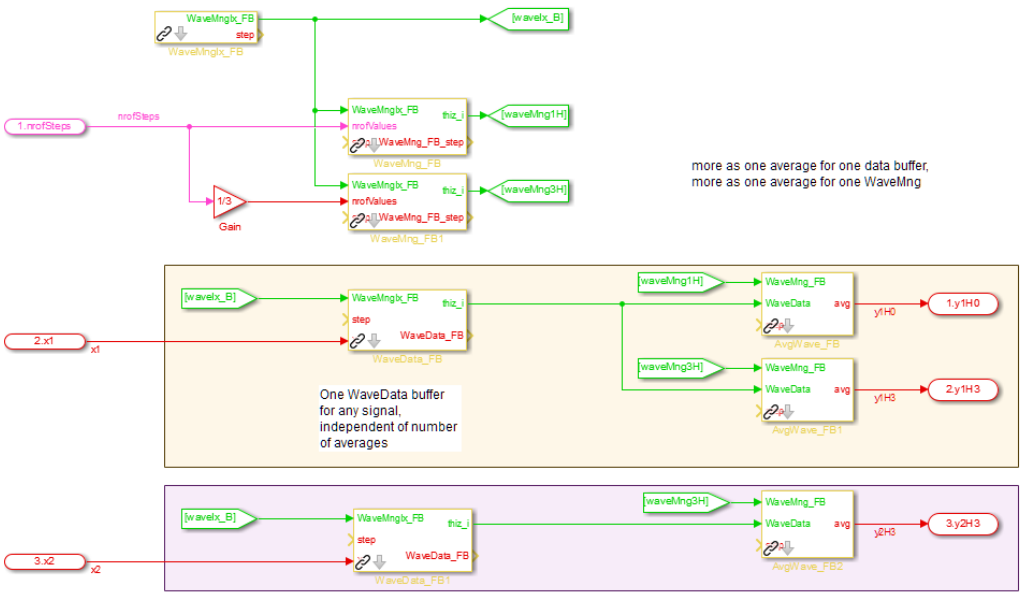

Abbildung 2: Simulink Objectmodell WaveMng ohne Abarbeitungsreihenfolge

Das Bild zeigt das eigentliche Modell für den Datenfluss. Die FBs müssen in einer Abtastzeit passend nacheinander aufgerufen werden.

Für die Rechenzeiteffizienz ist eine Indexverwaltung zentralisiert. Der FB `WaveMngIx_FB` (links oben) incrementiert eigentlich nur einen Index für die WaveBuffer. Die Incrementierung wird in der step-Abtastzeit vorgenommen, vermittelt über Parameter. Der FB erscheint als Source-Block. Die FB `WaveMngIx_FB` berechnen mit der dynamisch sich ändernden Mittelwertzeit Indizees und Faktoren für den Zugriff auf einen 'Altwert'. Das ist unabhängig von den Daten selbst. Wird beispielsweise in einer Anwendung 9 Signale mit dem gleichen Mittwelwert berechnet, beispielsweise 3-phasige Spannungen und Ströme, dann braucht es diesen FB also nur einmal. Für eine andere Mittelwertzeit gibt es den zweiten hier dargestellten `WaveMng_Ix_FB`.

Die Daten werden nun in einem `WaveData_FB` gespeichert, und zwar verwaltet von dem Index-Geber `WaveMng_Ix_FB`, aber unabhängig davon, ob

davon welche Mittelwerte gebildet werden oder etwa nur verzögert alte Daten abgegriffen werden.

Letzlich gibt es die Mittelwertbildner `AvgWave_FB`, die die Werte aus den Datenbausteinen und der Indexberechnung nutzen und damit schnell einen Mittelwert ermitteln können. Dazu folgender Hinweis: Diese FBs verarbeiten korrekt nichtganzzahlige Avg-Schrittweiten und korrigieren den möglichen numrischen Fehler der Mittelwert-Drift aufgrund floating-Rundungsprobleme.

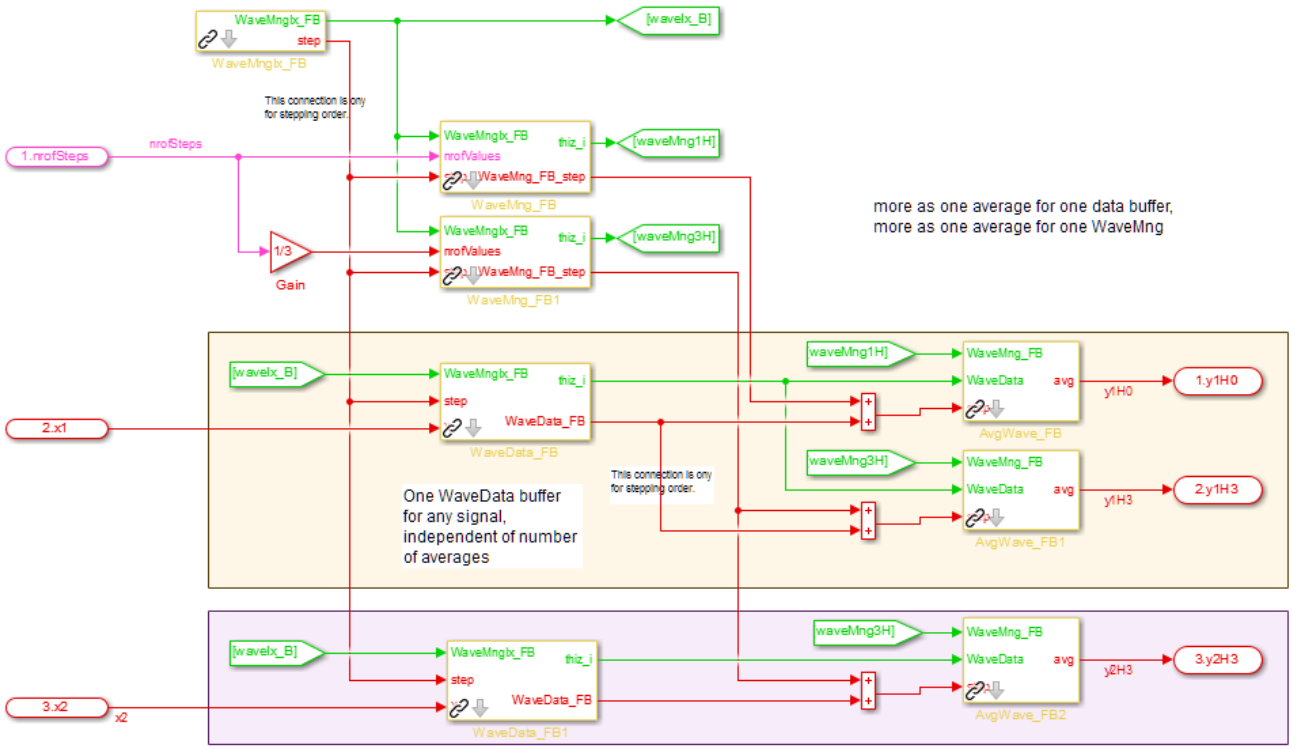

Abbildung 3: Simulink Objectmodell WaveMng mit Abarbeitungsreihenfolge

In diesem Modell ist nun die Abarbeitungszeit mit Verbindungen gekennzeichnet. Die FBs haben Ein- und Ausgänge, die zwar im Modell verdrahtet sind aber nicht auf der C-Ebene verwendet werden. Bei der Codegenerierung aus dem Modell werden zwar diese Verküpfungen mit generiert. Sie entfallen aber letzlich beim Compilieren. Ein Compiler optimiert in der Regel nicht genutzte Variable bzw. lässt sich entsprechend einstellen.

Mit der Abarbeitungsreihenfolge-Verdrahtung ist nun gekennzeichnet, dass zuerst der `WaveMngIx_FB` (links oben) abgearbeitet werden muss, da sein Ausgang als Eingang bei den `WaveMng_FB` und `WaveData_FB` benutzt wird. Damit ist gewährleistet, dass alle diese FB den gleichen Index benutzten.

Für die Bildung der Mittelwerte (rechts unten) müssen sowohl die Daten gespeichert sein, als auch die `WaveMng_FB` müssen gerechnet worden sein. Dann passen die Daten und Indizees für die Mittelwertbildung zueinander, sind konsistent.

Die in C ungenutzten Ein- und Ausgänge werden dann generiert, wenn die

```
/**
 * @simulink Object-FB, step-in, step-out.
 */ ....
```

Operation-Definition noch mit `step-in` und/oder `step-out` gekennzeichnet ist. Der step-Eingang und -Ausgang wird dabei als letzter Ein- bzw. Ausgang angeführt. Ein Eingang ist `DYNAMICALLY_TYPED`. Man kann also alle passenden Signale anschließen. Ein Ausgang ist mit `uint32` typisiert und kann damit problemlos auch mit einem `thiz`-Ausgang der Hauptabtastzeit zusammengeschaltet werden, hier sichtbar mit einer Addition. Damit passt der Ausgang auch wieder an einen Eingang. Addierte Werte werden nicht benutzt.

7.2 Objektorientierte Simulink-Module, Verbindung über Handle

Das objektorientierte Prinzip soll nicht nur für die direkte Verbindung von C-Modulen verwendet werden, sondern auch größere Simulink-Modul-Einheiten sollen in dieser Weise zusammengeschaltet werden. Der Schlüssel für diese Herangehensweise ist die Anordnung der relevanten Moduldaten in einem Object-FB. Dieser ist einerseits so gestaltbar, dass er als Datenspeicher dienen kann: Die Ausgänge führen die Alt-Daten der letzten Abarbeitung, beim Aufruf werden die Neu-Daten übernommen. Der Object-FB wirkt damit wie ein *Unit-Delay*. Andererseits ist für den Datenaustausch in C eine Datenkonsistenz programmierbar. Zur Datenkonsistenz siehe TODO.

Dieser moduleigene Object-FB stellt die Moduldaten nach außen bereit. Im Modul können sich weitere modulinterne Daten befinden, die typischerweise mit Unit-Delay und anderen Simulink-Elementen gebildet werden.

7.3 Hinweise für Modellgestaltung

- Nutze package-paths: name-clash-Verhinderung.
- Gebe bei callback im Modell nur einen File an.

8 Basisfunktionen aus der CRJ-Library

Die CRJ Library steht im Begleitbeispiel im Unterverzeichnis CRJ. Es handelt sich dabei um das Subset, dass für allgemeine Anwender-Sfunctions notwendig ist.

Die CRJ-Library wurde als *CRuntimeJavalike*-Library in C mit der Entwicklung eines Java2C-Translators (Java to C) entwickelt und enthält diejenigen Routinen, die direkt in C bereitgestellt werden. Die CRJ-Library hat sich im laufe der Zeit als hilfreich auch für die allgemeine C-Programmierung erwiesen. Der Name *CRuntimeJavalike* ist berechtigt, da einige Ansätze ähnlich wie in Java ausgeführt sind, darunter die `struct ObjectJc`, die an `java.lang.Object` als Konzept angelehnt ist. Eine allgemeine Beschreibung der CRJ-Library kann in `www.vishia.org/Jc` gefunden werden.

Das Subset der CRJ für allgemeine Anwender-S-Funtionen besteht aus folgenden wesentlichen Files und Ordnern, die vollständige Funktionalität ist in den Files selbst dokumentiert:

- `CRJ/source/OSAL`: Enthält Headerfiles für die Definition einer *Operation System Adaption Layer* - Schicht. Insbesondere ist hier die `os_sharedmem.h` mit der `os_AtomicAccess.h` notwendig, da in den Sfunctions in Windows SharedMemory verwendet wird. Für die Implementierungsebene im Zielsystem wird die OSAL-Schicht nicht vorausgesetzt.
- `CRJ/sourceSpecials/os_Windows_64/ os_sharedmem.c` Dies ist die Implementierung des Shared Memory Zugriffs für Windows.
- `CRJ/source/Fwc` enthält Basis-Framework-Files außerhalb eines Java-Konzepts. Hierbei wichtig ist:
- `CRJ/source/Fwc/fw_handle_ptr64` Funktionen für das Abbild von Adressen im Speicher über Handle. Verwendet in der Sfunction-Ebene, nicht im Zielsystem.
- `CRJ/source/Fwc/objectBaseC.h` Die Basisstruktur `ObjectJc`. Sie wird auch im Zielsystem verwendet, da in den Anwender- `struct` - Definitionen enthalten. Sie kann im Zielsystem hilfreich sein wenn dort Reflection eingesetzt werden, ansonsten nicht.
- `CRJ/source/Fwc/fw_ExcStacktrcNo.h`: Es ist möglich, mit einer in der CRJ definierten Schreibweise für Exceptionhandling zu arbeiten. Diese wird in den internen CRJ-Quellen verwendet und kann auch dem Anwender empfohlen werden. Dieses Headerfile definiert die Makros allerdings so, dass kein Exceptionhandling verwendet wird. Das ist die Voreinstellung für einfache und allgemeingültige Anwender-Sfunctions.
- `CRJ/source/Jc/ReflectionJc.h` Der Zugriff auf die Daten über Reflection is für die Sfunctions auf Simulinkebene integraler Bestandteil. Im Zielsystem können Reflection verwendet werden, müssen aber nicht. Es ist auch möglich, einen sogenannten *Target - Proxy* zu erstellen, der die Reflectioninformationen auf direkte Speicherzugriffe umsetzt und so das Zielsystem von zusätzlichem Aufwand entlastet. Siehe dazu `http://www.vishia.org/Inspc/html/proxytarget_de.html`.
- `CRJ/incApplSpecific/stdef_SmlkSfunc/applstdefJc.h` Das Headerfile `applstdefJc.h` wird in den Sources der CRJ includiert und soll im Raum der Applikation definiert werden. Es wird über den Include-Path gefunden. Dieses Verzeichnis enthält die Ausführung für Sfunctions unter MS-Windows mit einem Visual-Studio-Compiler.
- `CRJ/incApplSpecific/TagetNumericSimple` Dieses Verzeichnis im Includepath des Target enthält eine Version von `applstdefJc.h`, in der alle textuellen und Reflection - Bezüge nicht enthalten sind. Es ist insbesondere für ein Target mit nur numerischen Aufgaben (DSP) gedacht. Speicherallokierung in der startup-Phase ist vorgesehen.

- `CRJ/incApplSpecific/TargetReflAllocOnStatup` Dieses Verzeichnis im Includepath des Target enthält eine Version von `applstdefJc.h`, die mit Speicherallokierung in der Startup-Phase arbeitet, ansonsten aber das Reflection - Handling unterstützt. Es ist damit für Applikationen geeignet, die etwa auf einem ARM-Prozessor als Zielsystem arbeiten.

- `CRJ/sourceSpecials/FwConv_h/fw_StringJcSimple.h` Dieses Headerfile wird in `applstdefJc.h` includiert, wenn keine String-Verarbeitung und keine Reflection-Unterstützung im Zielsystem verwendet werden soll.

- `D:/vishia/Smlk/Example_ObjO/CRJ/sourceSpecials/cc_SmlkSfunc` Dieses Verzeichnis ist für die Generierung der Sfunction in den Includepath mit aufzunehmen. Es enthält insbesondere ein `compl_adaption.h` für die Definition compilerspezifischer Makros für den MS-Visual-Studio-Compiler im Zusammenhang mit der CRJ.

- `D:/vishia/Smlk/Example_ObjO/CRJ/sourceSpecials/cc_Msc6` Dieses Verzeichnis ist für die Generierung eines Zielsystems auf dem PC aufzunehmen, wenn ein MS-Visual-Studio-Compiler für 32 bit Systeme verwendet wird. Dies wird empfohlen um das Verhalten des generierten Codes in einer 32-Bit-Umgebung auf dem PC zu testen. Es enthält das dazu passende `compl_adaption.h`.

9 Generierung der Sfunctions

Die Generierung der Sfunction-Wrapper, tlc-Files und Bus-Scripts aus den Headerfiles wird mit einem m-File im Matlab aufgerufen. Dabei werden die erforderlichen Sekundär-Source-Files aus den Informationen in den Headerfiles generiert. Danach wird der mex-Compiler für die Sfunctions aufgerufen.

Nach der Vorlage werden folgende File-Typen in den folgenden Verzeichnissen generiert. Die Verzeichnisse sind dabei im Aufrufscript festgelegt, können also beliebig angepasst werden. Folgende Verzeichnisaufteilung ist daher 'nur' als bewährter Vorschlag zu sehen.

- `simulink/MODELDIR/+src2Sfn/pathto/OPERATION_SfH.c`: Sfunction-wrapper C-File und tlc-Files pro Sfunction-Block (pro Object-FB und Operation-FB im Headerfile)
- `simulink/MODELDIR/+src2Sfn/mex_NAME.m`: Matlab-Script zum Aufruf des mex-Compilers, jeweils für ein zmake-Aufruf bzw. pro Anwender-Generierscript. Mit einem Aufruf können mehrere Headerfiles bearbeitet werden. Wenn das Generierscript genau einen zmake-Aufruf enthält, dann sollte der `NAME` dem Namen des Generierscripts und damit dem Namen des zugehörigen Modells oder der Library entsprechen.
- `mex/*` Dieses Verzeichnis enthält die generierten mex64-DLL. Es wird empfohlen, dieses Verzeichnis parallel zu den Simulink-Quellen abzulegen und nicht in eine Versionsverwaltung aufzunehmen. Es sind ausschließlich generierte Files. Die Herauslösung aus den Simulink-Quellen ist günstig, damit etwa in einem Versionsmanagement-System die Quellen von Generaten getrennt erscheinen, oder bei Weitergabe von Informationen Quellen und Generate getrennt sind. Die generierten mex-Files sind relativ viele große Files (für dieses Beispiel bereits 16 MByte), so dass eine getrennte Handhabung sinnvoll ist. Man kann das mex-File als ein zip-File beispielsweise in einem Versionsmanagement speichern, um die Generate sofort ohne Neugenerierung verfügbar zu haben oder um einen älteren Stand sofort laufen lassen zu können.
- `mex/OPERATION_mex.mex64`: Sfunction-Library wird direkt von Simulink geladen und benutzt bei Simulationslauf (Ergebnis der mex-Compilierung)
- `mex/OPERATION_mex.mex64.pdb`: Zugehörige Programm Data Base - Outputfile der Compilierung von Microsoft Visual Studio zur Debugunterstützung: Die Ausführung von Sfunctions kann mit Visual Studio Einzelschritt-Debugged werden.
- `mex/tlc_c/OPERATION_SfH.tlc`: tlc-File, Steuerfile für Target Language Compiler und Accelerator-Mode für Simulink. Die tlc-Files werden hier für die Codegenerierung erwartet. Sie werden beim mex-Aufruf aus dem `.../+src2Sfn`-Verzeichnis kopiert.
- `gensrc_Refl/HEADER.crefl`: Reflection C-File. Pro Headerfile wird ein Reflection-File erzeugt. Dieses wird in den Quellfiles includiert. (`#include "gensrc_Refl/...crefl"`). Die Reflection-Files können auch im Anwendersystem (Target) einbezogen werden, wenn dort die gleiche Inspector-Unterstützung vorgesehen ist. Dies wird empfohlen. Damit kann ein einheitliches System der Datenzugriffe sowohl im Simulationslauf als auch im Anwendersystem genutzt werden. Soll im Zielsystem keine Reflections verwendet werden, dann kann das include bedingt formuliert werden.
- `simulink/MODELDIR/+src2Bus/MODEL_genBus.m` Matlab-Script zum Generieren der *Bus-Objects* im Matlab-Workspace. Dieses File ist dem Simulink-Modell bzw. der Library zugeordnet. Die Einordnung erfolgt deshalb innerhalb der Modell-Quellen, weil dieses Script immer beim Start des Modells aufgerufen und aufgefunden werden muss. Das Matlab-Script für die Busgenerierung sollte daher auch als Callback im Modell eingetragen sein.

9.1 Generierscript als Matlab-Script

Es wird empfohlen, pro Modell und pro Library ein Generierscript anzulegen, dessen Filename dem Namen des Modells oder der Library entspricht. Das Matlab-Script enthält am Anfang Matlab-Befehle, danach folgt für Matlab als Kommentar gesetzt das JZtxtcmd-Steuerscript für die Generierungssteuerung. Als Vorlage dient das Matlab-Script, für die Generierung der Sfunctions des Orthogonaloszillator. Das Script wird folgend in Teilen als Kasten dargestellt und die Teile darunter beschrieben. Das gesamte Script ist auf der beiliegenden Beispielsoftware unter
`simulink/lib/+genSfn/lib_OrthOsc_genBusSfunc.m` zu finden.

```
function lib_OrthOsc_genBusSfunc()
rootPath = evalin('base', 'rootPath');
clc;
clear mex;
currdir_ = pwd;
cd(rootPath);
display('Generate S-Fuctions for lib_OrthOsc ...');
system('java.exe -cp zbnfjax/zbnf.jar org.vishia.jztxtcmd.JZtxtcmd
   simulink/lib/+genSfn/lib_OrthOsc_genBusSfunc.m');
src2Bus.lib_OrthOsc_genBus();  %just generated script
src2Sfn.lib_OrthOsc_mex();    %just generated script.
display('successfull')
cd(currdir_);
end
```

Das ist der für Matlab wirksame Teil. Das Script berücksichtigt, dass ein anderes Verzeichnis aktuelle eingestellt ist als ein benanntes `rootPath`-Verzeichnis. Für das Script ist der `rootPath` relevant.

Die Generierung wird als Cmd-Zeile als Java-Aufruf ausgeführt, wobei das eigene File als Argument als JZtxtcmd-Script angegeben ist.

Die mit dem Script passend generierten m-Files werden danach aufgerufen: Busgenerierung und mex-Compilierung.

```
%{
==JZtxtcmd==
currdir = <:><&scriptdir>/../../..<.>;
include ../../../SmlkSfuncJZtc/simulinkSfuncBusGen.jzTc;
```

Mit dem `%{` ist der Rest für Matlab Comment. Das JZtxtcmd-Script beginnt an dem folgenden Label. Das ist eine Eigenschaft jedes JZtxtcmd-Scripts.

Das Script soll immer ein `currdir` festlegen. `<&scriptdir>` ist das absolute Verzeichnis in dem das Script steht. Das `currdir` zeigt hier auf das Basisverzeichnis der Simulink-Applikation (Basis des Arbeitsbereiches im Filesystem). Darauf beziehen sich dann die relativen Fileangaben.

Das Anwenderscript muss das eigentliche Generierscript includieren. Dieses ist anwendungsunabhängig, kann aber Anpassungen enthalten. Vor dem Start der JZtxtcmd-Abarbeitung werden alle includierten Scripts als Einheit in eine interne Abbildung übersetzt.

```
Fileset srcHeader =
( srcc_FB:AngleBlocks_FB.h
, srcc_FB:OrthOsc_FB.h
);
```

```
Fileset includepath =
( srcc_FB
, CRJ/source
, CRJ/source/OSAL
, CRJ/incApplSpecific/stdef_SmlkSfunc
, CRJ/incComplSpecific/cc_SmlkSfunc
, src2_refl
);
```

Im Script wird das Fileset für die Input-Files `src` definiert. Der `:` als Verzeichnistrenner statt `/` spaltet einen *Localpath*-Anteil ab, der hier aber nur aus dem Filenamen besteht. Es ist möglich, Files mit einem Verzeichnis im Include der Anwenderquellen zu erzeugen.

Das Fileset für den `includepath` ist für die Compilierung wichtig. Die hier stehenden relativen Pfade zählen auch für die Generierung ab dem `currdir`, der Basis des Arbeitsverzeichnisses. Es ist hier auch möglich, absolute Verzeichnisse anzugeben.

```
Fileset lib_Sfn =
( mex/CRJ_SmlkSfn
);

Fileset lib_Accel =
( mex/CRJ_SmlkAccel
);


Fileset lib_RAccel =
( mex/CRJ_SmlkRAccel
);
```

Diese Filesets benennen Libraries, die jeweils für Sfunctions, Accelerator-Mode und Rapid-Accelerator grundsätzlich benutzt werden sollen.

```
## All sources for compilation to the named header file, for target and raccel.
Fileset srcTlc_OrthOsc_FB =
( srcc_FB/OrthOsc_FB.c     ##same c-file as header
, srcc_FB/AngleBlocks_FB.c     ##same c-file as header
);
```

Diese Sources werden in das tlc-File notiert. Sie werden damit sowohl beim Rapid Accelerator als auch für das Zielsystem compiliert. In diesem Fall ist es der aufgerufene Code der Sfunction. Da diese Source sehr wahrscheinlich bei Neugenerierung der Sfunction auch geändert sein könnte, wird die Quelle direkt und nicht über eine Library eingebunden.

```
## All sources for compilation to the named header for SFnc additional to the srcTlc
Fileset src_OrthOsc_FB =
( &srcTlc_OrthOsc_FB      ##Use the sources which are named for tlc
, CRJ/source/Fwc/fw_ReflectionBaseTypes.c
, CRJ/source/Fwc/fw_Object.c
, CRJ/source/Fwc/fw_handle_ptr64.c
```

```
, CRJ/sourceSpecials/FwConv_c/fw_ExcStacktrcNo.c
, CRJ/sourceSpecials/os_Windows_64/os_sharedmem.c
);
```

Diese Filesets benennen die Quellfiles, die jeweils für einen Headerfile als Input (Fileset `srcHeader`) benutzt werden sollen, zusätzlich zum generierten Wrapper der Sfunctions. Wenn die compilierten Objects sich in den Libraries befinden, können diese Filesets klein gehalten werden. In diesem Fall sind die Files vollständig, es wird nichts aus der Library beim Linken zusätzlich benötigt.

Es wird davon ausgegangen, dass alle in einem Headerfile beschriebenen Sfunctions die gleichen Zusatzfiles benötigen. Dabei ist ein unnötiger Zusatzfile im konkreten Fall kein Fehler und kaum Aufwand, lediglich der mex64-Sfunction-dll-File wird etwas größer. Wenn hier Files fehlen und diese auch nicht in den Libraries gefunden werden, gibt es beim mex-Compilieren Linker-Fehler.

```
Subtextvar genAllocmem(String dst, String nameStruct, Num ident) =
<:>(<&nameStruct>*) malloc(sizeof(<&nameStruct>))  /*JZtc: user-specific */ <.>;
```

Dieses Codefragment wird in den tlc-File eingebracht und beschreibt, wie im Zielsystem eine Speicherallokierung auszusehen hat. In einer Sfunction wird dies nicht benutzt, dort wird, da auf dem PC ablaufend, immer malloc verwendet. Im Zielsystem kann es sein, dass mit statischem Speicher gearbeitet werden soll.

```
zmake "simulink/lib:lib_OrthOsc" := simulinkSfuncBusGen(.:&srcHeader
, fileBus = ":+src2Bus/*_genBus.m"
, dirMex = "mex"
, coptions = "-TP -EHa"
, dirMexcc=":+src2Sfn"
, dirSfunc = ":+src2Sfn/OrthOsc", dirTlc = ":+src2Sfn/OrthOsc"
, dirRefl = "src2_refl"
##, html=":+src2Sfn/OrthOsc/test.html"
);
```

Die **main-Routine** enthält nun das *zmake-Statement* zur Generierung der Ergebnisfiles. Die allgemeine Form eines zmake-Statement in einem JZtxtcmd-Script nennt links das zu erzeugende File (*Destination*) und rechts vom := die Bedingung der Erzeugung. Da hier mehrere Files erzeugt werden, sind deren Verzeichnis oder Namen allerdings als Argumente angegeben. Beim zu erzeugenden File (links) ist pragmatisch Verzeichnis und Name des zugehörigen Modells angegeben. Diese Angaben, getrennt mit ':' werden dann zur Namensbildung der Zielfiles verwendet. Folgende Angaben für Destinations gelten:

- Die *destination* des zmake links vom := gibt vor dem : das Verzeichnis an, das als Basisverzeichnis benutzt wird, wenn die Angaben der folgenden Verzeichnisse mit : beginnen. Das ist eine spezielle Eigenschaft dieses Generierscripts, keine allgemeine des zmake.
- Die *destination* des zmake links vom := gibt nach dem : einen File-Namen an, der als Filename verwendet wird, wenn von diesem Script genau 1 file erzeugt wird. Der Name sollte dem Namen des Modells oder der Library folgen, die die hier zu generierende Sfunction ursächlich enthält.
- `fileBus`: Pfad des zu erzeugenden Bus-Generierfiles. Es wird ein File für alle Busse (aus alle `struct`-Definitionen) erzeugt, die die Annotation `@simulink bus` tragen. Wenn der Pfad mit ':' beginnt, dann wird der Pfad der Destination-Angabe als Basis verwendet. Das ist auch bei den anderen Pfaden möglich. Das '*' wird durch den Namen der Destination-Angabe ersetzt. Damit folgt der Pfad des Bus-Generierfiles dem Pfad der Library aus der Destination-Angabe.

- `dirMex` Verzeichnispfad, in dem die generierten mex-Files abgelegt werden. Wird diese Angabe weggelassen dann erfolgt keine mex-Compilierung. Der Name des zu erzeugenden mex-Files in diesem Verzeichnis folgt dem Namen der Destination-Angabe.
- `coptions` Diese Optionen werden beim mex-Aufruf als `COMPFLAGS='$COMPFLAGS .....'` übergeben. Sie werden vom mex-Compiler dem Zielsystemcompiler durchgereicht. In diesem Fall wird mit `/TP` eine C++-Compilierung der `.c`-Quellen ausgeführt. Mit `/EHa` wird eine Exception auch dann mit `catch` abgefangen, wenn sie nicht mit `throw` erzeugt wurde sondern beispielsweise und insbesondere durch eine Memory-Access-Exception hervorgerufen wurde. Das lässt sich allerdings auch in der XML-Configurationsfile zum mex-Compiler eintragen, siehe 11.5, S. 50
- `dirMexcc` Verzeichnispfad, in dem das Matlab-Script zur Gegerierung der mex-Files abgelegt wird. Der Name des Scripts folgt der Angabe des *destination*-files des zmake-Aufrufes, also hier `lib_OrthOsc`.
- `dirSfunc` Verzeichnispfad, in dem die Wrapper.c der Sfunctions abgelegt werden und in dem das mex-Generierfile abgelegt wird. Wird diese Angabe weggelassen dann werden keine Sfunctions erzeugt. Damit ist es möglich mit diesem Script nur Bus-Matlab-Script-Files zu erzeugen. Der Name des zu erzeugenden mex-Files in diesem Verzeichnis folgt dem Namen der C-Funktion, die als Object-FB oder Operation-FB bezeichnet ist.
- `dirTlc`: Pfad in dem die tlc-Files abgelegt werden.
- `dirRefl`: Pfad in dem die Reflection-Files abgelegt werden. Pro Headerfile wird ein Reflectionfile erzeugt. Dessen Name wird aus dem Namen des Headers gebildet und ist damit eindeutig. Wird diese Angabe weggelassen oder mit `null` besetzt, dann erfolgt keine Reflectionfilegenerierung. Die Reflectionfiles können passend zum Header auch anderweitig generiert werden.
- `html` Pfad eines html-Files, der das Parserergebnis des jeweiligen Headerfiles in html-Form enthält. Dieses File ist nur notwendig, wenn die Generierscripts angepasst werden. Es ist dort ersichtlich, wie das Parserergebnis (als Java-intern-Daten) abgelegt wird. Im Muster ist dieses Argument kommentiert.

Die Schreibweise eine Zeile pro Argument(-Gruppe) mit Komma vorn ist empfohlen. Dann kann mit Kommentarzeichen `##` mit dem Rest der Zeile das Argument bei Bedarf wegkommentiert werden bzw. einfach wieder hinzugenommen werden, gezeigt mit dem `, html = ....`

`zmake` ist ein spezifischer Aufruf im JZtxtcmd ähnlich wie `call`. Die `simulinkSfuncBusGen`-Subroutine ist die zmake-Routine. Das erste Argument ist ein Fileset. Hinweis: Das zmake führt keine automatischen Prüfungen auf Datum und bedingte Übersetzung aus. Dies ist bei Bedarf, nicht in diesem Fall, im der jeweiligen zmake-Subroutine spezifisch zu programmieren.

Die Verzeichnisse werden angelegt, wenn sie nicht vorhanden sind. Vorhandene Files in den Verzeichnissen werden nicht gelöscht. Das ist ebenfalls spezifisch in dieser Weise in den Subroutinen so programmiert und kann auch geändert werden.

9.2 Compilierung der C-Quellen mit C++-Compiler

Es ist zweckmäßig, einen C++-Compiler zu verwenden statt dem C-Compiler, auch wenn nur C-Code compiliert wird. Defakto gibt es einen Quasi-C-Standard, der an Sprachfeatures von C++ partizipiert, einen C++-Compiler erfordert, ohne aber C++ selbst zu sein. Auch für die meisten Zielsysteme stehen C++-Compiler zur Verfügung. Die verbesserten Sprachfeautes sind im wesentlichen:

- Bessere Warnings bei Programmierfehlern insbesondere bei Pointer-Casting.
- Möglichkeit, Variable-Definitionen im Ablaufcode unterzubringen, nicht nur am Anfang.
- `inline`-Funktionen
- Ein verbessertes Exception-Handling, insbesondere mit der Möglichkeit, Speicherzugriffsfehler als Exception abzufangen. Letzeres hilft bei den Sfunctions, da dann nicht das *Simulink abstürzt* sondern eine klare Fehlermeldung generiert wird.

Die Generierung der Sfunction-Wrapper kann insbesondere das Exception-Handling nutzen, wenn bei erwartbaren Fehlern nicht der gesamte Matlab-Prozess beendet werden soll.

Man kann die C++-Compilation nur für die Sfunction vorschreiben, indem die `coptions="-TP "` angegeben wird, siehe voriges Kapitel.

Für die (Rapid-) Accelerator-Compilierung gilt jedoch, dass der generierte Modell-Code als C zu compilieren ist. Das hängt von zugebundene Libraries für die Kommunikation mit Simulink ab. Ein C++-Compiler erzeugt andere Linker-Labels. In C++ kann man mit `extern "C"` zwar auch mit C-Libraries mischen, das ist hier aber in der Codegenerierung so nicht vorgesehen.

Man kann folglich im XML-File, der in 11.5, S. 50 beschrieben ist, nicht generell das `-TP`-Flag für die C++-Compilierung setzen.

Die konkreten Aussagen gelten für den Microsoft-Compiler. Andere Compiler haben andere spezifische Optionen.

10 Internas der Sfunction

10.1 Port based oder FB-Abtastzeiten

Ein **Object-FB** hat grundsätzlich **zwei Abtastzeiten: Tinit** und eine entweder über die Input-Port-Verbindungen oder per Parameter determinierte **Operation-Abtastzeit**. Die Tinit-Zeit dient eigentlich nur zur Zuordnung dieser Teile in der Zielsystem-Codegenerierung. Dort werden alle Tinit-Operationen in einer eigenen step-Funktion zusammengefasst (im tlc OutputsForTID), die dann im Zielsystem am Anfang vor Beginn der Abarbeitung der anderen Zeitscheiben gerufen wird. Für die Simulink-Ebene ist das so nicht organisierbar, da im Simulink ein eigener Initialisierungs-Ablauf mit Datenverbindungen der FBs nicht vorgesehen ist. Die Zuordnung bestimmter Ports zu Tinit ist auf der Simulink-Ebene nur formalistisch, letzlich als Farbgebung bei der Anzeige der Abtastzeit im Diagramm. Da Verbindungen in Tinit nur für die Handle der Object-FB ausgeführt werden und also keine Operationen im Modell beinhalten, ist die Belegung der Verbindung am Output-Port in der Hauptabtastzeit ebenfalls erfolgreich.

Die Zeit Tinit kann man folglich auch mit einer langen Periode belegen, da sie in Simulink nicht verwendet wird und in der Zielsystemgenerierung einem eigenen Zeitregime unterliegt.

Wie ist statt dessen die Initialisierung geregelt? In der Basis-struct `ObjectJc`in den Daten des Object-FB gibt es ein Bit `isInitialized`. In der Opeartion-Abtastzeit wird jeweils **geprüft ob alle für Tinit notwendigen Inputwerte der Handle anstehen**. Ist dies der Fall dann wird die `init_`-Routine gerufen. In der `init_`-Routine kann individuell geprüft werden, ob die Input-FB in ihren Daten bereits `isInitialized ==1` ansagen, also ihre Initialisierung abgeschlossen haben. Abhängig davon wird schlussendlich `isInitialized = 1` gesetzt. Die `init_`-Routine kann also mehrfach aufgerufen, wenn vorgelagerte Object-FB noch nicht als initialisiert angesehen werden. Das ist notwendig bei **zyklischen (gegenseitigen) Referenzierungen** von Object-FBs. In der `init`-Routine können Werte berechnet werden, die der nachfolgende Object-FB dann benutzt. Bei zyklischen Referenzierungen muss allerdings auf Anwenderprogrammierebene geklärt sein, dass die Object-FB nicht gegenseitig auf `isInitialized ==1` warten. Ëiner muss den Anfang machen". Um einen solchen Fehler zu detektieren, wird mit einem Durchlaufzähler die Anzahl der init-Zyklen auf 100 begrenzt und danach wird eine Fehlermeldung abgesetzt. Man kann damit 99 FB in der Abtastreihenfolge falsch herum verdrahten (der letzte in der Reihe meldet `isInitialized =1`). Bei etwas komplexeren Initialisierungs-Abhängigkeiten kann es in der Praxis zu 2..3 Durchläufen kommen.

In den nachfolgenden Abtastschritten wird **dann wegen** `isInitialized ==1` **die step-Funktion des Object-FB** gerufen.

Ein Object-FB präsentiert seinen eigenen **Handle an Outport Port 0 und 1, Port 0 für Tinit und Port 1 für Tstep**. Beide Werte sind identisch. Der Output Port 0 wird beim ersten `init_`-Aufruf belegt, unabhängig von `isInitialized =1`. Der Output Port 1 ist 0, solange die Initialisierung noch nicht ausgeführt wurde.

Ein **Operation-FB hat grundsätzlich nur eine Abtastzeit**. Diese ist `INHERITED`. Wenn Eingangwerte verarbeitet werden, dann bestimmen diese die Abtastzeit. Der `this`-Input-Port 0 des Operation-FB muss vom Output Port 1 des Object-FB versorgt werden, damit die Abarbeitungsreihenfolge in der selben Abtastzeit geregelt ist: Der Object-FB wird zuerst verarbeitet. **Wenn der Operation-FB eine abweichenden Abtastzeit hat**, dann sollte der Output-Port 0 (Tinit) benutzt werden, mit einer *Rate-Transition*. Das ist konsequent im Design. Dann ist keine Reihenfolge zwischen der Object-FB-Operation und dem Operation-FB geregelt. Nur die Initialisierung `isInitialized ==1` wird festgestellt. Das ist häufig passend, wenn beispielsweise in einer langsamen Abtastzeit Einstellungen verändert werden. Die Datenkonsistenz ist gewährleistet, wenn entweder auch insbesondere im Zielsystem untersetzte Abtastzeiten verwendet werden oder eine Konsistenz-Sicherung mittels Mutex-Mechanismen auf C-Ebene erfolgt.

Werden von einem **Operation-FB keine weiteren Eingangswerte verarbeitet sondern nur interne Werte des zugehörigen Object-FB**, dann ist der `this`-Input-Port 0 mit dem Output-Port 1 des Object-FB zu verbinden. Dieser determiniert dann mit seiner Arbeits-Abtastzeit die Abtastzeit

des Operation-FB und sichert die korrekte Abar- beitungsfolge.

10.2 Abarbeitung des Modells im Normalmode und Debuggen im C-Code

Einige C-Funktionen der Wrapper der Sfunctions werden bereits bei *Simulation - Update Diagram* aufgerufen. Diese Informationen werden im Diagramm zum Check benötigt und gegebenenfalls angezeigt (*Disply - Signals & Ports - ...*) Hinweis: In diesen C-Funktionen können keine Daten in DWork-Vektoren gesetzt werden.

- `mdlInitializeSizes()`: In dieser C-Funktion werden die Anzahl und Typen der Ports festgelegt.
- `mdlSetInputPortSampleTime()` und `mdlSetOutputPortSampleTime()`: Diese C-Funktionen werden generiert und gerufen, wenn keine feste Abtastzeit über ein Argument `Tstep` im ctor als Parameter übergeben wird, für alle Ports, deren Abtastzeit aus dem Modell ermittelt werden kann.
- `mdlInitializeSampleTimes()`
- `mdlSetInputPortDimensionInfo()` und `mdlSetOutputPortDimensionInfo()`: Diese C-Funktionen werden generiert und gerufen, wenn die Anzahl der Arrayelemente nicht feststeht (DYNAMICALLY_SIZED)
- `mdlSetInputPortDataType()` und `mdlSetOutputPortDataType()`: Diese C-Funktionen werden generiert und gerufen, wenn Ports mit `void` bzw. `DYNAMICALLY_TYPED` typisiert sind. Es wird der im Modell festgestellte Typ übergeben. Adäquates gilt für
- `mdlSetDefaultPortDataTypes()`: Aufgerufen nachdem alle In- und Output Port Typen gesetzt worden sind. Man kann hier umdefinieren und muss insbesondere alle nicht angeschlossenen Ports ebenfalls typisieren.

Siehe dazu auch (TODO es gab hierzu eine Simulink-Übersichtsdoku über Simulation von Mathworks)

Im Normalmode wird das Modell aufgrund der intern gespeicherten und aufbereiteten Modelldaten interpretativ abgearbeitet. Die Sfunctions werden nach Datenaufbereitung gerufen. Diese Funktionen können nun die vorbereiteten DWork-Vektoren nutzen.

- `mdlStart()`: Diese Routine allokiert den Speicher für die internen Daten eines Object-FB.
- `mdlInitializeConditions()`: In dieser Routine wird der ctor eines Object-FB gerufen. Die Nontunable Parameter sind hier präsent.
- `mdlOutputs()`: Diese Routine wird dann zyklisch als Simulation gerufen. Es wird auch `mdlUpdate()` gerufen wenn vorhanden. Die `mdlOutputs()` bekommt eine `tid` (*Task Id*) mitgeteilt, wenn es verschiedene Abtastzeiten gibt. Insbesondere ein Object-FB kennt neben der step-Abtastzeit die `Tinit`. Diese wird allerdings ignoriert. Statt dessen wird die Initialisierungsroutine zyklisch anfänglich durchlaufen, bis sie erfolgreich war. Es wird dabei `isInitialized_ObjectJc()` getestet. Das zugehörige Flag muss in der Initialisierungsroutine gesetzt werden - oder im ctor, wenn es keine Initialisierungsroutine gibt.

Man kann mit einer *Microsoft Visual Studio Entwicklungsumgebung* (IDE) in diese C-Funktionen hineindebuggen. Das erfolgt in dem die IDE an den Matlab-Prozess im MS-Windows angehangen wird:

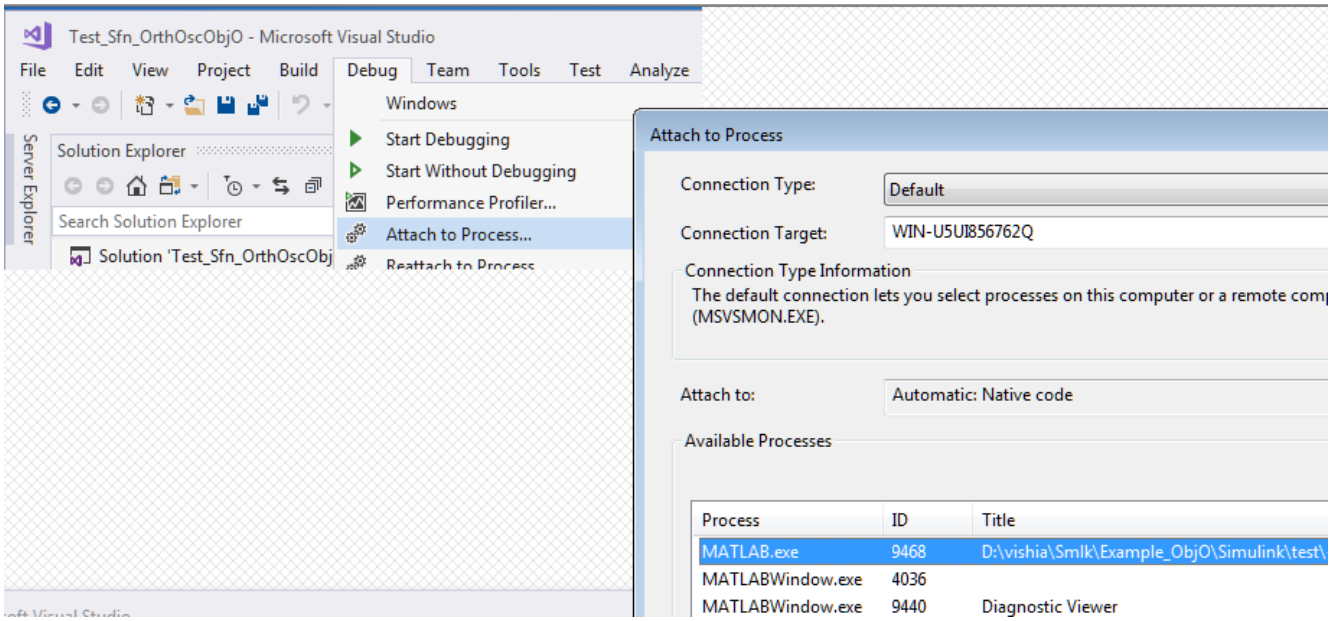

Abbildung 4: Debuggen in Sfunction - Attach to process

Das Anhängen sollte erfolgen, bevor das betreffende Modell geöffnet wird. Mit Start des *Simulation - Update Diagram* werden nun die entsprechenden Softwaremodule `my_Sfunction.mex64` im Matlab geladen. Dieses bekommt das angehängte Visual Studio mit. Wenn die Sfunction mit Debuginformationen übersetzt wurden - Option `-g` - dann existiert neben dem `my_Sfunction.mex64` die zugehörige pdb-Datei mit den Debuginformationen. In der `my_Sfunction.mex64.pdb` steht als Absolutpfad der Verweis auf die zugehörige Quellfiles. Wenn also die Sfunction auf dem gleichen Rechner - oder mit der gleichen Anordnung auf der Festplatte - übersetzt worden ist, kann Visual Studio nunmehr einen Haltepunkt in einem passenden Quellfile der richtigen Maschinencode-Speicheradresse zuordnen und beim Durchlaufen in den Debug-Einzelschritt gehen. In dieser Zeit ist allerdings Simulink nicht mehr bedienbar, da der Thread im Debug steht. Setzt man passend Haltepunkte und lässt nach Einzelschritt den Prozess jeweils weiterlaufen, dann ist ein günstiges Debuggen möglich.

11 Accelerator- und Rapid Accelerator mode

Im Normalmode wird das Modell interpretativ abgearbeitet. Die Sfunctions werden jeweils nach Vorbereitung in der mex64-dll gerufen.

Im **Accelerator-Mode** wird das gesamte Modell als große mex64 - dll übersetzt. Dazu wird aus dem Modell eine Codegenerierung gerufen, die im Unterschied zur Zielsystem-Codegenerierung direkte Aufrufe der Verbindung zum Simulink enthält. Der Code ist grundsätzlich lesbar, hilfreich bei Compilerfehlern, aber spezifisch für die Verwendung im Accelerator aufbereitet.

Abhängig von der Option `SS_OPTION_USE_TLC_WITH_ACCELERATOR` wird nun der Code der Sfunction so wie für das Zielsystem über den tlc-File in den gesamten Modellcode eingebunden - oder die Sfunction wird wie beim Normalmode aufgerufen. Letzteres braucht dann etwas mehr Rechenzeit. Es gibt Sfunction, die nicht für das Zielsystem vorgesehen sind und die Sonderbedingungen des Starts und des Terminate benötigen. Die Sfunction `step_Service_Inspc_SfH` ist ein Beispiel dafür. Diese legt einen extra Thread für die Socketkommunikation an und öffnet einen Socket. Für diese Sfunction darf `SS_OPTION_USE_TLC_WITH_ACCELERATOR` nicht gesetzt sein. Das wird gesteuert mit der Annotation

```
@simulink Object-FB, , accel-tlc.
```

Nur mit dieser vorhandenen Option wird `SS_OPTION_USE_TLC_WITH_ACCELERATOR` gesetzt.

Im **Rapid-Accelerator-Mode** wird mit dem Start der Simulation vom Modell intern C-Code generiert. Dieser wird übersetzt und zu einer Executable zusammengebunden, die dann das Modell direkt im Maschinencode in einem eigenen Prozess des Betriebssystems mit hoher Geschwindigkeit abarbeitet. Es wird eine Kommunikation mit Simulink aufgebaut, über die die Abarbeitung gesteuert werden kann und Signale beispielsweise auf einem *Scope* sichtbar sind. Hierbei werden intern keine mex64-dll mehr gerufen sondern der Code aller Sfunction wird direkt eingebunden.

Aus den mex64-Files der Sfunctions werden auch im Rapid-Accelerator die Initialisierungs-Teile benutzt. Das sind diejenigen C-Funktionen der Sfunctions, die auch beim *Simulation - Update Diagram* aufgerufen werden, siehe 10.2, seite 43.

Beim Accelerator-Mode wird zusätzlich auch die C-Funktion `mdlStart()` aufgerufen. Das führt dazu, dass Speicherplatz allokiert wird. Dieser wird allerdings mit dem ebenfalls stattfindendem Aufruf von `mdlTerminate()` wieder freigegeben.

Compiler- und Linker-Fehler

Der Ansatz der intensiven Nutzung von Sfunctions mit gegebenenfalls wesentlichen C-Anteilen muss sich in die Compilierung des Modells nahtlos einfügen. Wärend vorübersetzte und einzeln getestete Sfunctions im *Normal-Mode* der Simulation immer laufen, kann beim Start des *(Rapid-) Accelerator Mode* doch einiges schief stehen, was zur Fehlermeldung *"Unable to build a standalone executable to simulate the model 'NAME' in rapid accelerator mode."* führt. Es reicht bereits, wenn in einer zufällig offenen C-Quelle unbemerkt einige falsche Tastenanschläge hineingeraten sind. Wichtig ist die Einbindung der notwendigen zusätzlichen Quellen als C-Compilationseinheiten oder Verwendung von Libraries mit den richtigen Compileroptionen übersetzt und stimmigen Versionen von Sources und Headers, die Benennung von Include-Pathes und Beherrschung von Linker-Fehlern. Man hat hier also die gesamte Palette der Dinge, die vom Compilieren und Linken von C-Projekten bekannt sind, aber in einer anderen Umgebung. Wie bekannt gilt auch hier die Regel: *"Wenn alles funktioniert dann ist es sehr gut"*. Das sollte nun auch die Voraussetzung sein, wenn man sich eher auf die Simulation konzentrieren möchte als auf Compiler- und Linkerfehlermeldungen. Wenn es Probleme beim Compilieren und Linken gibt, die dann gelöst sind, dann verursacht eine Änderung des Modells und damit nochmalige Compilierung in der Regel nicht wieder neue Probleme. Man kann also arbeitsteilig arbeiten. Ein Regelungstechniker arbeitet mit dem Modell, die Compiler- und Linkerprobleme werden vom Informatiker zuvor gelöst.

11.1 Ablageverzeichnis für Accelerator und Rapid Accelerator

Das Modell wird codegeneriert, der generierte Code wird temporär in ein eigenes Verzeichnis geschrieben. Man kann dieses Verzeichnis in ein temporäres separates Arbeitsverzeichnis legen, möglicherweise auf einer RAM-Disk bzw. auf dem lokalen Datenträger, wenn ansonsten über Netzwerk gearbeitet wird. Die Quell- und Object-Files können so besser letztendlich gelöscht werden. Es könnte ansonsten auch Probleme geben, wenn ein Zeitstempel falsch ist, bekannt vom make-Prozess beim Compilieren. Das *cleanup* muss einfach zu bewerkstelligen sein.

Ein abweichendes Verzeichnis ist im Simulink einstellbar, im Beispiel liegt es in einer als T: eingerichteten RAM-Disk:

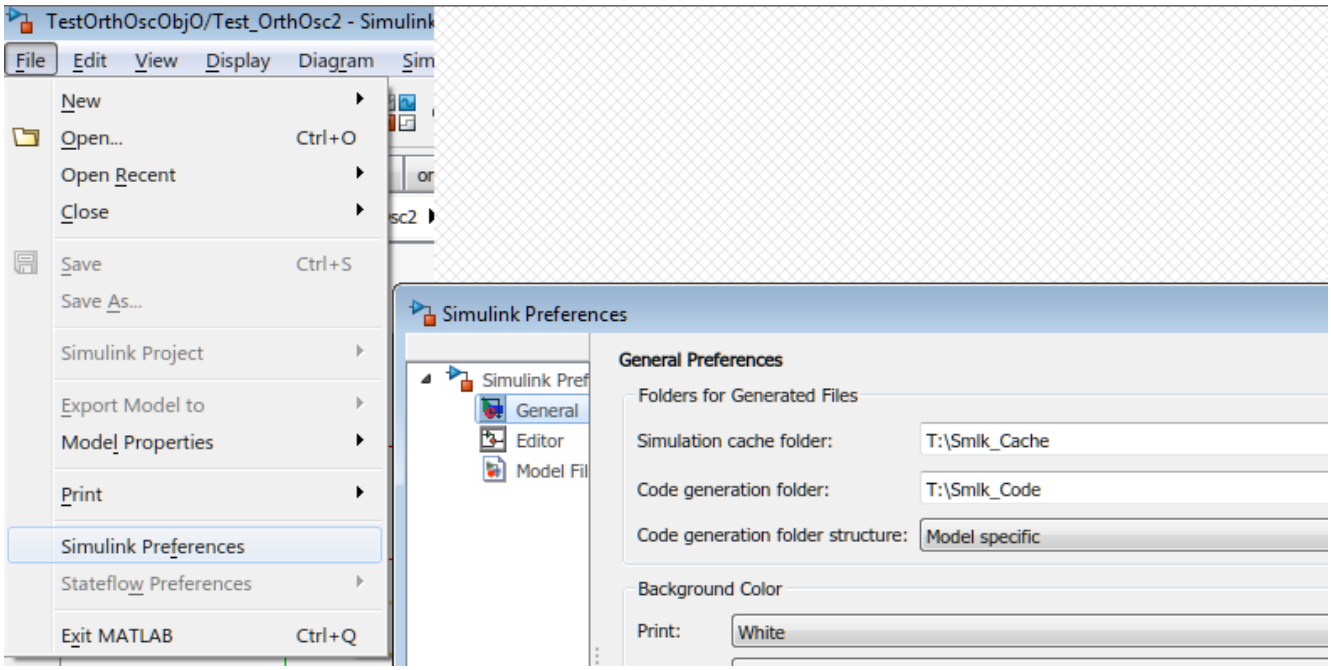

Abbildung 5: Setzen temporäres Verzeichnis für Accelerator

11.2 Welche Sources und Includepfade . rtwmakecfg.m

Das File `rtwmakecfg.m` muss vom *Rapid accelerator builder* aufgefunden werden. Dieses File wird im selben Verzeichnis gesucht, in dem die Mex-Files der Sfunctions stehen. Findet der Builder das `rtwmakecfg.m` nicht, dann gibt es bei Nutzung von Sfunctions die bekannte Fehlermeldung *"Unable to build ..."*.

Das File gehört zum Modell und wird daher - bisher manuell - dort gepflegt. Es soll dann im init-File des Modells beim Start des Modells und beim Start der Compilierung in das mex-File kopiert werden. Dazu enthält das init-File folgende Zeile:

```
system('copy "simulink\test\+TestOrthOscObj0\rtwmakecfg.m" mex\rtwmakecfg.m');
```

Das File `rtwmakecfg.m` muss den richtigen Inhalt haben, siehe folgend.

```
function makeInfo = rtwmakecfg()
disp('rtmakecfg - TestOrthOscObj0');

makeInfo.includePath = ...
{ 'D:/vishia/Smlk/Example_Obj0/simulink/lib/+genSfn'  ...
, 'D:/vishia/Smlk/Example_Obj0/srcc_FB'  ...
, 'D:/vishia/Smlk/Example_Obj0/simulink/test/+TestOrthOscObj0/+genSfn' ...
, 'D:/vishia/Jc/CRuntimeJavalike/incApplSpecific/stdef_SmlkAccelerator' ...
, 'D:/vishia/Jc/CRuntimeJavalike/incComplSpecific/cc_SmlkSfunc' ...
, 'D:/vishia/Jc/CRuntimeJavalike/source/OSAL' ...
, 'D:/vishia/Jc/CRuntimeJavalike/source' ...
};

makeInfo.sourcePath = ...
{ 'D:/vishia/Smlk/Example_Obj0/srcc_FB'  ...
, 'D:/vishia/Smlk/Example_Obj0/genSrc_refl'  ...
, 'D:/vishia/Smlk/Example_Obj0/simulink/test/+TestOrthOscObj0/+genSfn' ...
};

makeInfo.linkLibsObjs = ...
{ 'D:/vishia/Smlk/Example_Obj0/mex/CRJ_SmlkAccl.lib' ...
'D:/vishia/Smlk/Example_Obj0/mex/ws2_32.lib' ...
};

end
```

Die Display-Ausgabe der ersten Zeile ist im *Diagnostic Viewer* - Ausgabe des Ablaufes der Compilierung - wieder auffindbar. Der includePath wird benutzt für die Compilierung des Modells, adäquat wie die `mex ... -Iincludepath` - Zeilen beim Aufruf der Sfunction-Generierung.

Der `sourcePath` wird benutzt, um die zusätzlichen Sources aufzufinden, die in den tlc-Files benannt sind.

11.3 Zusätzliche Libraries beim Rapid Accelerator einbinden

Die `linkLibsObjs`-Angabe im `rtwmakecfg.m`-File benennt Libraries, die beim Linken mit beachtet werden. Man hat hier nun die Möglichkeit, vorübersetzte Quellen zu benutzen. Im Beispiel wird die Windows-Socket-Library `ws2_32-lib` extra angegeben, dazu die selbst erstellte `CRJ_SmlkAccl.lib`, die die gesamte Umgebung der CRuntimeJavalike, insbesondere die umfangreichen Quellen des Inspector in compilierter Form bereitstellt.

Diese Library wird mit einem JZcmdScript erstellt:

```
java.exe -cp ../zbnfjax/zbnf.jar org.vishia.jztxtcmd.JZtxtcmd %0 -t:_gen.bat
if errorlevel 1 goto :error
echo call make\_gen.bat
call _gen.bat
.....
exit /B

==JZtxtcmd==
include ../../../Jc/CRuntimeJavalike/make/JZtxtcmd/Msc.zmake.jztc;
String srcDir=<:><&scriptdir>/../../../Jc/CRuntimeJavalike<.>;
String objDir = "T:\libAccel.obj"; ##<:><&$OBJDIR><.>;
Fileset  includePath =
( ../../../Jc/CRuntimeJavalike/incApplSpecific/stdef_SmlkAccelerator
, ../../../Jc/CRuntimeJavalike/incComplSpecific/cc_SmlkSfunc
, ../../../Jc/CRuntimeJavalike
, ../../../Jc/CRuntimeJavalike/source
, ../../../Jc/CRuntimeJavalike/source/OSAL
);

Fileset src_make=
( source:Fwc/*.c
, source:Jc/*.c
.....
);

String compileOptions=<:>-c -DCRTAPI1=_cdecl ......<.>;

zmake "mex/CRJ_SmlkAccl.lib" := cc("../../Jc/CRuntimeJavalike" &src_make, =.....
zmake "mex/CRJ_SmlkAccl.lib" := clib("../../Jc/CRuntimeJavalike" &src_make, .....
```

Das File ist nur in Auszügen angegeben, siehe Beispiel `Example_Obj0.zip` im Pfad `makeSmlkLibs/zmake_CRJ.Accel.lib.bat`. Man kann die Library auch mit einem Standard-make-File erzeugen und hat letztlich die adäquaten Angaben zu machen. Die Compileroptionen wurden aus der Compilerausgabe kopiert, die im *Diagnostic Viewer* beim Start des Accelerator mode auch für die anderen Compileraufrufe (des Modellfiles) gelten. Somit ist sichergestellt, dass die Optionen zueinander passen. Das Make-System mit zmake ist auf http://www.vishia.org/docuZBNF/Zmake.html erläutert.

11.4 tlc-Files

Die tlc-Files (files für den *target language compiler* werden sowohl bei der Zielsystem-Codegenerierung benutzt, als auch jedenfalls für den Rapid Accelerator Mode und für den Accelerator-Mode bei gesetzter Option `SS_OPTION_USE_TLC_WITH_ACCELERATOR` in der Sfunction. Ein tlc-File enthält die Vorschrift zur Bildung des C-Codes für die einzelnen Phasen (start, init, run). Diese sind in einer eigenen tlc-Scriptsprache hinterlegt. Die C-Code-Bildungsvorschrift kann Variablennamen, C-Funktionsaufrufnamen und Comments im Klartext enthalten, die im generierten Zielsystemcode wiedergefunden werden können und so der Orientierung im generiertem Code dienen können. Bei den C-Codes für den Accelerator werden allerdings Comments ausgeblendet.

Die tlc-Files werden von der Sfunction-Generierung über das JZtxtcmd-Script generiert. Damit bestimmt der Inhalt der JZtxtcmd-Scripts den Inhalt der tlc-Files.

Im tlc-File steht auch, welche zusätzlichen Quellen und Libraries für eine S-function eingebunden werden muss, neben den Codes im tlc-File. Diese Information wird aus dem Generier-Steuerscript (siehe S. 37) aus der Abteilung

```
Fileset srcTlc_MyHeader = ( ... );
Fileset lib_MyHeader = (.... );
```

entnommen. Der Suchpfad zu diesen Quellen und Libraries steht dann im `rtwmakecfg.m`, siehe vorige Seite.

Beispiel: Das Generier-Steuerscript enthält:

```
Fileset srcTlc_AngleBlocks_FB =
( srcc_FB/AngleBlocks_FB.c
, gensrc_Refl/AngleBlocks_FB_crefl.c
);

## All sources for compilation
Fileset src_AngleBlocks_FB =
( &srcRaccl_AngleBlocks_FB
, CRJ/source/Fwc/fw_Reflect....c
......
);
```

Diese Angaben werden in das tlc-File übertragen, dabei nur die `srcTlc_...`. Die anderen Sources müssen über die Library gefunden werden.

Dazu der passende Ausschnitt aus dem generierten tlc-File:

```
%function BlockTypeSetup(block, system) void
%%
%% The Target Language must be C
%if ::GenCPP==1
%<LibReportFatalError("This S-Function must be only used with the C Target Language")>
%endif

%%For compilation: All headers which should be included:
%<LibAddToCommonIncludes("Fwc/fw_handle_ptr64.h")>  %%Always used.
%%Note: It is taken from the input header file,the local file part.
%<LibAddToCommonIncludes("AngleBlocks_FB.h")>

%%For accelerator: All compilation units (C-files) which should be add to the exec.
%%Note: It is taken from set srcRaccl_AngleBlocks_FB from the user's start script,
%%only file names, not with path:
%%
%<SLibAddToStaticSources("AngleBlocks_FB.c")>
%<SLibAddToStaticSources("AngleBlocks_FB_crefl.c")>
%%
%endfunction
```

11.5 Auswahl des Compilers und XML-Steuerdatei für die Compilierung

Die Auswahl des Compilers für Simulink erfolgt mit der Aufrufzeile im Matlab

```
mex -setup
```

Matlab sucht dann nach installierten Compilern, wählt automatisch den einen installierten Compiler aus oder bietet eine Auswahl an. Entsprechend dem Compiler wird eine XML-Datei im Windows-Userbereich angelegt und gefüllt, die Informationen über den Compiler, Include- und Librarypfade, voreingestellte Compileroptionen und dergleichen enthält.

Man kann diese XML-Datei dann anpassen, wenn der betreffende Compiler nicht direkt vorgesehen ist, andere Compileroptionen notwendig sind oder beispielsweise Linkerfehler wegen fehlender Libraries auftreten. Hierbei sollte allerdings die entsprechende Sachkenntnis über Compileraufrufe und etwas Experimentierzeit vorhanden sein. Dieses XML-File ist beispielsweise unter

```
c:\Users\hartmut\AppData\Roaming\
MathWorks\MATLAB\R2017b\mex_C_win64.xml
```

aufzufinden.

Mit einem Beispiel sollen folgend die Möglichkeiten aufgezeigt werden:

Beim Anwender wurde *Microsoft Visual Studio 2017* als sogenannt *Community-Version* installiert. Diese kostenfreie Version ist etwa für die Open-Source Entwicklung gedacht.

Bei Aufruf des Visual Studio Compilers wird ein File `VCVARSALL.BAT` aufgerufen, das sich im Filebereich des Compilers befinden und verschiedene Umgebungsvariable für den Aufruf des Compilers setzt. Dieses File ist in der XML-Datei aufgeführt als:

```
CommandLineShell="c:\Programs\Msc17\Commu
 nity\VC\Auxiliary\Build\VCVARSALL.BAT "
```

Mit der *Community-Version* hat nun dieses Batch-File gleichzeitig ein `cd` (change directory) zur Location `c:\users\hartmut\source` ausgeführt. Eine auf einem anderen PC vorhandene *Professional Version 2015* hat dies nicht ausgeführt. Ergebnis beim Accelerator-Aufruf gab es die Fehlermeldung:

```
NMAKE : fatal error U1052: file
  'TestOrthOscObjO.mk' not found
```

Das makefile war aber vorhanden. Dass die Ausführung von `VCVARSALL.BAT` das Problem verursacht, wurde erst mit einigen Experimenten erkannt.

Abhilfe ist nun, der Austausch der Zeile im XML-File

```
CommandLineShell=
 "c:\BATCH\VCVARSALL_A.BAT "
```

Dieses manuell geschriebene File ruft letztlich die `VCVARSALL.BAT` auf, umkleidet aber diesen Aufruf mit

```
set PWD1=%CD%
call c:\Programs\Msc17\Community\VC\
  Auxiliary\Build\vcvars64.bat
...
cd /D %PWD1%
echo %CD%
```

und stellt damit das richtige Verzeichnis wieder ein.

Ein anderes Problem waren fehlende Libraries in dieser Version, die mit Erweiterung des Library-Path nach Installation des Microsoft Software Developer Kit 8.1 bereitgestellt werden konnten:

```
LINKLIBS="/LIBPATH:&amp;quot;$MATLABROOT\....
 /LIBPATH:c:\Programs\WinSDKv8.1\Lib\x64
```

Es ist allerdings zu empfehlen, von Anfang an auf eine Professional Edition des Compilers zu setzen bei der keine Anpassungen notwendig sind.

Hinweis: Für die einfache Accelerator-Übersetzung bringt Simulink intern einen passenden Compiler für Windows mit (*lcc64*). Auch ein *MinGW-Compiler* ist nutzbar. Das Visual Studio ist dann notwendig, wenn in den Sfunction-Quellen gebugged werden soll.

11.6 Wie kann man Fehler erkennen und beheben?

Beim Start des Rapid Acclerator Mode wird im *Diagnostic Viewer*-Fenster sehr konkret Compiler- und Linkerfehlermeldungen angezeigt. Der Diagnostic Viewer klappt automatisch auf, wenn es zu *"Unable to build ..."* kommt. Man kann dieses Fenster unabhängig von der Fehlermeldung öffnen durch Klick auf einen Link, der während des Build-Prozess in der unteren Statusleiste des Modells erscheint. Im Diagnostic Viewer ist anwählbar, ob man Errors, Warings oder Infos sehen möchte, Icon oben. Die detailierten Compiler- und Linkerfehler kommen als *Info*.

Gibt es Compilerfehler, dann sind diese wie gewohnt in den Quellen zu korrigieren. Zu beachten ist, dass der Compiler je nach Einstellung nicht C++ sondern C compiliert. Es hängt vom verwendeten Compiler des Rapid Accelerator Mode ab, wie C-Dialekte verarbeitet werden.

Wichtig ist die richtige Wahl der Include-Pathes, enthalten in der `rtwmakecfg.m`.

Linkerfehler gibt es wenn Quellen nicht benannt worden sind, oder auch wenn Compileroptionen nicht stimmen. Das ist bekannt vom C/C++-Build-Prozess und gilt auch hier. Das Kapitel **??** Seite **??** enthält Hinweise wie eine Library zu bilden ist.

Es verbleiben aber noch die Fehler zur Laufzeit. Enthält der C-Code Laufzeitfehler und diese wurden nicht beim Vortest und beim Test der Sfunctions entdeckt, dann kann es einen Absturz geben. Hier hilft dann nur nochmals in den Vortest zu gehen. Das ist allerdings auch bekannt in der normalen C/C++-Entwicklung ohne Simulink.

Arbeitsteilig arbeiten: Der Simulink-Entwickler möchte nichts mit den Compiler/Linker-Problemen zu tun haben!

Dies ist auch zu empfehlen, wenn ein und dieselbe Person die C-Entwicklung betreibt und dann in Simulink das Gesamtprojekt testet. Wichtig ist also ein Vortest, bevor die eigentliche Kern-Funktionalität, die im Simulink-Modell auch entsprechend variiert werden kann, in den Fokus des Tests kommt.

12 Beobachten und Eingreifen mit dem Inspector

Im Simulink ist es gängige Praxis, bei Bedarf alle internen Daten darzustellen und bestimmte Daten (Parameter, Konstante) zu verändern. Dies ist schon deshalb notwendig weil Simulink ein Entwicklungswerkzeug der Regelungstechnischen Algorithmen ist. Mit dem Kapseln der Kern-Funktionen in C ist es trotz des unterstellten Blackbox-Charakters dieser Funktionsblöcke dennoch wünschenswert, deren Daten zu sehen. Dies ist auch gängige Praxis der Objektorientierung: *Sage mir Deine Daten, und ich weiß was Du bist* könnte als Leitspruch der Erkenntnis über Klassen gelten. Die Daten sind zwar für den Zugriff private gekapselt, aber die Darstellung auch der private-Attribute im Klassendiagramm der OOP gibt den notwendigen Überblick.

Dem Ansinnen, auf die private-Daten der Object-FB zuzugreifen kommt eine Technik entgegen, die vom Verfasser mit Seitenblick auf Java in der Vergangenheit entwickelt wurde und sich in der Praxis bestens bewährt hat: Zugriff über Reflection, mit dem Tool und Protokoll des Inspectors. Diese Technik auf Simulink angewendet gestattet nicht nur den Zugriff auf die private-Daten der Object-FBs von außen, sondern insgesamt:

12.1 FBs für den Inspector im Simulink

GetValue_Inspc

Dieser FB speziell für Simulink gestattet das Herauslegen von Daten aus C-FunctionBlocks beispielsweise zum Anschluss an einen Scope, als Messpunkt oder numerische Anzeige (Display). Es wäre möglich die Daten auch im Modell funktional zu verwenden. Dies kann zweckmäßig sein für Test-Auswertungen, nicht aber für die Core-Funktonalität, die damit über eine zweite Ebene neben der modellmäßigen Datenhaltung arbeiten würde. Im Zielsystem (über Codegeerierung) kann dieser FB ebenfalls für Debugzwecke eingesetzt werden, wenn beispielsweise eine Analogausgabe für Tests damit gemapped wird.

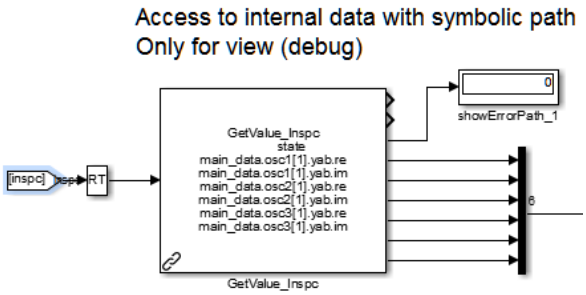

Abbildung 6: Einsatz GetValue_Inspc

Der Datenpfad wird symbolisch angegeben, wir er beim Zugriff mit dem Inspector gilt. Er wird beim Startup einmalig gelesen. Damit wird die Speicheradresse der Variablen ermittelt. Zur Runtime wird dann der Wert der Variable auf den Output gelegt.

Die derzeitige Realisierung des FB hat 6 Ausgänge festgelegt auf float bzw. single. In Vorbereitung: Typ und Anzahl von Arrayelementen wird in der Parameterangabe für den Datenpfad mit berücksichtigt, adäquat zum FB TimeSignalFloat_Inspc.

InputValuesC_Inspc

Dieser FB gestattet das Einlesen von Signalwerten aus dem Modell für den Inspector-Zugriff. Er ist gedacht nicht für die Modellteile, die über Codegenerierung im Zielsystem verwendet werden, sondern speziell für die Simulink-Modellteile der Umgebungssimulation, die meist direkt im Simulink nicht über Sfunctions realisiert werden, beispielsweise unter Nutzung von Simscape. Der FB hat 12 Eingänge belegbar mit Simulink-Signalen, auch Vektoren und komplexe Werte. Der Typ der Eingänge wird aus der Verdrahtung im Modell heraus erkannt. Es werden intern Reflection-Informationen aufbereitet, die dann für den Inspector-Zugriff über Reflection genauso wie in Sfunctions oder im Zielsystem genutzt werden. Im FB wird den Signalen ein entsprechender Name gegeben. Der FB ist als *Node* in einen Baum eingeordnet.

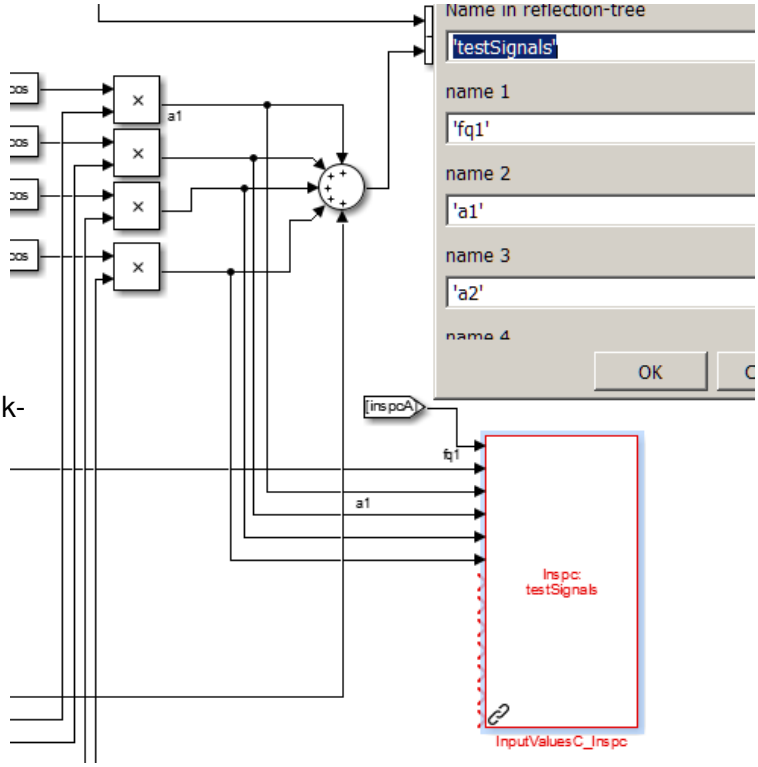

Abbildung 7: Einsatz InputValuesC_Inspc

TimeSignalsFloat_Inspc

TimeSignalsBits_Inspc

Mit diesen beiden FB werden Signal-Werte für das Modell vorgegeben. Ähnlich wie der *InputValuesC_Inspc* ist dies für die Nicht-Zielsystem-Modellteile gedacht. Der FB ist eine Kombination aus dem Inspector-Zugriff und einer intern organisierten zeitgesteuerten Vorgabe. Letztere holt sich beim Startup die Zeit-Wert-Informationen aus einem File, speziell gedacht für Testserien oder aber auch als Anfangswertvorgabe. Man kann die Werte mittels Inspector überschreiben, bis zu dem Zeitpunkt an dem sie per File neu definiert werden. Typischerweise werden oft Anfangsbelegungen vorgegeben, die dann dauerhaft überschrieben werden können da sie per Zeitsteuerung nicht wieder neu gesetzt werden. Es gibt 2 Typen von FBs: Für Float-Ausgänge und für Boolean. Die Float-Variante kann pro Ausgang für Vektoren bis max. 6 Elemente und für komplexe Größen konfiguriert werden.

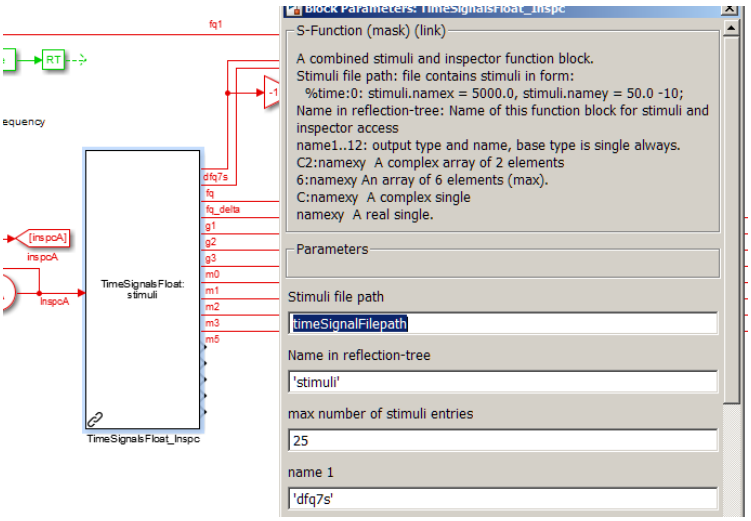

Abbildung 8: Einsatz TimeSignalsFloat_Inspc

Wait_Inspc

Dieser FB ist nur einmal im Modell anzuordnen, daher hier gleich neben dem zentralen Service _Inspc gezeigt. Der FB zählt die Zyklen ab Start und schaltet danach in einen Stop- oder Delaymodus. Stop wird mit der Angabe 0 für delay erzeugt. Dabei steht die Simulink-Abarbeitung, sie hängt im FB. Das ist dazu gedacht, nunmehr mit dem Inspector beispielsweise neue Werte für Simulis vorzugeben. Nachdem dies erfolgt ist, manuell oder per Inspector-Tool-Script, wird ebenfalls über den Inspector-Zugriff ein erneuter Stop-Punkt vorgegeben. Mittels vorgabe von
2000000000 steps (etwa der maximale Integer-Bereich) läuft die Simulation dann quasi dauerhaft.

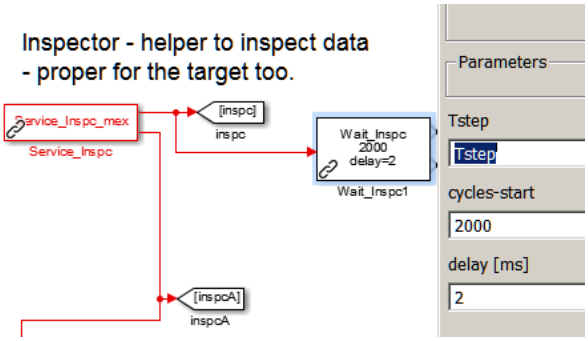

Abbildung 9: Einsatz Wait_Inspc

Die Wait-Variante hilft, eine schnelle Simulation so zu entschleunigen, dass einzelne Verläufe manuell gut beobachtbar sind. Die Scopes laufen dann auch automatisch langsamer. Es wird im FB in der angegebenen Abtastzeit ein delay über das Betriebssystem erzwungen. Auch hier hilft eine Vorgabe der cycles >0 um wieder volle Geschwindigkeit zu erhalten bis zur erneuten Stimuli-Vorgabe.

RegisterNode_Inspc_A.png

Dieser FB ist das entscheidende Bindeglied zwischen den Object-FB-Sfunctions und dem Inspector. Es wird im Modell außerhalb der Teile eingesetzt, die im Zielsystem codegeneriert erscheinen sollen, und zwar direkt an deren Ausgang. Dort soll ein Handle-Ausgang bereitstehen, der die Intern-Daten repräsentiert.

Im Zielsystem mit Einsatz der Reflection wird dieser Handle bzw. die Speicheradresse direkt in den Reflection-Tree aufgenommen, in der Umgebung zum generierten Zielsystemcode. Das geschieht für die Simulink-Ebene mit diesem FB adäquat.

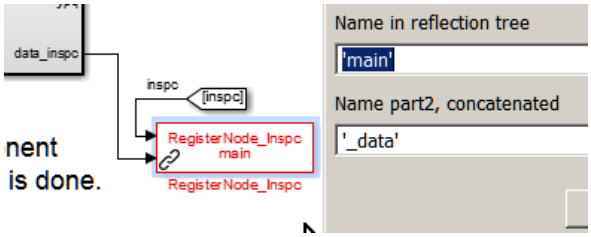

Abbildung 10: Einsatz RegisterNode_Inspc_A.png

Zum FB werden 2 Namensteile angegeben. Die Namensteile sind dann wichtig, wenn ein Namensteil über eine Variable für mehrere Instanzen geregelt wird.

Service_Inspc_A.png

Dies ist der zentrale FB für den Inspector-Service. Er wird mit der IP-Adresse parametriert.

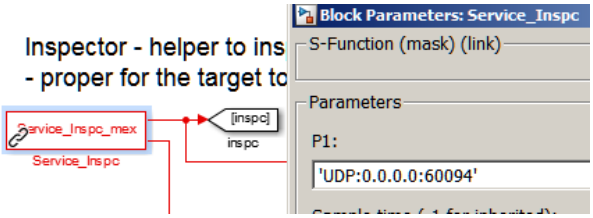

Abbildung 11: Einsatz ServiceNode_Inspc_A.png

12.2 Zugriff mit dem Insspector

Der Inspector als Tool greift über Socket-Kommunikation auf den zentralen FB ***Service_Inspc*** zu, mit der dort angegebenen IP-Adresse mit Port-Nummer (IP V4). Wenn der Inspector auf dem selben PC läuft ist die Kommunikation über local-host (loop back) sehr schnell. Über eine Netzwerkkarte mit Ethernet sind Zugriffe im 1 ms-Raster möglich. Mit dem Inspector können die Daten nicht nur visualisiert werden. Es werden folgende Funktionen bereitgestellt:

- Zugriff auf die Daten insbesondere auch wenn die Simulation gestoppt ist.
- Zugriff auf die Daten über die Baumstruktur, Zugriff auf Einzeldaten bei Bedarf, lesen und schreiben.
- Aufbau von Bildern (Datendarstellung) mit numerischen Werten und möglicher Hintergrundgrafik, gesteuert mit einem Script. Es sind sowohl verschiedene Bilder als Laschen anwählbar als auch Sub-Fenster aufklappbar. Damit ist zusätzlich zu den Möglichkeiten im Simulink-Modell eine Datenübersicht über das Gesamt-Modell möglich.
- Setzen bestimmter Daten in einer Datendarstellung, über numerische Eingabe oder Button für Einzelbits.
- Darstellung des zeitlichen Verlaufs der Daten (*Curve-View*).
- Speicherung der Daten des zeitlichen Verlaufs

Diese Möglichkeiten **sind in der gleichen Weise auch im Zielsystem unterbringbar**. Damit ergibt sich ein konformer Zugriff auf die Daten im Simulink-Modell als auch dann im Zielsystem. Wenn die Daten in gleicher Art in den Reflection-Tree eingeordnet sind, sind selbst die Datenpfade identisch. Damit brauchen die Scripts für die Datendarstellung nicht doppelt gepflegt werden bzw. können bereits in der Simulationsphase erarbeitet werden.

12.3 Das Reflection-Prinzip

Von allen `struct`-Definitionen in den Headerfiles werden Reflection-Informationen erstellt. Diese sind in C `const`-Datenstrukturen und enthalten Name, Type und Offset in der `struct`. Bei der Generierung der Sfunctions werden C-files mit dieser Konstantendefinition abgelegt, wenn als *zmake simulinkSfuncBusGen*-Aufruf-Argument `dirRefl` angegeben wird, siehe Kapitel 9, Seite 36. Die Extension dieser Files ist `.refl.c`.

Die Daten der Inspc-FunctionBlocks (InputValuesC_Inspc usw.) sind in gleicher Art organisiert, nur dass hier der Aufbau der Daten mit den angegebenen Parametern im FunctionBlock im RAM erfolgt.

Die Reflection-Infos (Typ `ClassJc`) sind in einem Baum organisiert. Das ist damit gegeben, dass Referenztypen als Referenz auf die entsprechende Reflection-Info (`ClassJc`) angegeben ist. Es gibt eine root-struct, die im FB ***Service_Inspc*** verankert ist und im Simulink zur Laufzeit mit den angehangenen Inspc-FBs gebildet wird: *InputValuesC_Inspc*, *TimeSignalBits_Inspc*, *TimeSignalFloat_Inspc* und *RegisterNode_Inspc* vorge-

geben sind. Wenn im Zielsystem eine entsprechende Root-`struct` angelegt ist, als Laufzeitumgebung, erhält man im Zielsystem den vergleichbaren Baum-Aufbau.

Es ist möglich, dass aggregierte Daten auf `struct ObjectJc` basieren und eigene Reflection-Information haben. Damit kann der Typ der Aggregation ein Basistyp sein (im einfachten Fall `ObjectJc`), das Aggregat kann ein abgeleiteter Typ sein. Das wird ausgenutzt um die Vererbung im OOP-Sinn zu unterstützen. Wenn die aggregierten Daten nicht auf `ObjectJc` basieren und damit keine eigenen Reflection-Informationen mit sich bringen, dann sind die Reflection-Informationen mit dem Typ der Referenz identisch. Für die Object-FBs als Sfunctions gilt nun, dass diese auf `ObjectJc` basieren müssen, für den Typtest des Handle. Dies hilft nun auch gleichzeitig für den Typ der Reflection.

12.4 Das Inspector-Tool

Der Inspector ist ein in Java geschriebenes Tool. Im Zipfile `Example_Obj0_Smlk.zip` ist eine jar-Library enthalten, die für Java-8 (Oracle) compiliert ist. Der Aufruf des Inspectors ist dort mit ...TODO

13 Literatur, Querverweise

Die Simulink-Funktionen sind über die Simulink-Toolhilfe in einer lokalen Installation und im Internet auffindbar und sind daher nicht extra als Literaturquelle extra aufgeführt.

Ebenfalls als bekannt und verbreitetet oder auffindbar vorausgesetzt sind die Theorien und Praktiken der UML (Unified Modelling Language) und der OOP (ObjectOriented Programming).

Die konkrete Implementierung des hier gezeigten Beispieles getestet mit Mathworks, Simulink Version 2016a sind auf der Internetseite *www.vishia.org/Smlk/Download* enthalten, Erläuterungen dazu auf *www.vishia.org/Smlk/html/OOPinSimulink.html*. Diese Seite wird vom Verfasser erstellt und gepflegt.