Inhaltsverzeichnis

Abkürzzungsverzeichnis

TLC Target Language Compiler

STF System Target File

TMF Template Makefile

1 Einführung

Der Inhalt dieser Dokumentation besteht darin, die Umsetzung von eigenen SW-Treiber für eine Microcontroller-Plattform in Simulink zu erläutern. Hierfür wird zuerst die allgemeine Funktionsweise und insbesondere der Codegenerationsprozess mit Simulink diskutiert. Im Anschluss wird eine Übersicht über die verschiedenen Möglichkeiten dargestellt um eigene Simulink-Blöcke zu erstellen und in Quellcode zu übersetzen. MathoWorks sowohl Online- als auch PDF-Dokumentationen zur Verfügung, auf welche auch durchgehend verwiesen wird.

2 Funktionsweise von Simulink

Simulink ist eine auf Blockschaltbildern basierende Programmierplattform, welche ursprünglich dazu gedacht war Differentialgleichungen numerisch zu lösen und Simulationen durchzuführen. Im Verlauf der Zeit wurde diese Basis kontinuierlich erweitert und unterstützt mittlerweile zahlreiche Funktionen. Eine bemerkenswerte Funktionalität von Simulink ist die Generation von Quellcode für Microcontroller. Damit können sowohl in kurzer Zeit umfangreiche Applikationen entworfen werden als auch durch Kommunikationsprotokolle zwischen Host- und Target-Plattform HiL-, SiL- und PiL-Simulationen durchgeführt werden. Somit stellt es ein mächtiges Werkzeug dar um Hardwarebausteine wie Sensoren und Microcontroller auszuwerten und daraufhin Filter und Regelungssysteme zu entwerfen.

In den folgenden Abschnitten werden die verschiedenen Ausführungsmodi und der Simulationsprozess von Simulink näher erklärt.

2.1 Ablauf einer Simulink-Simulation

Der Ablauf einer Simulation läuft in verschiedenen Phasen ab. Zuerst wird ein Model initialisiert. Hierbei werden die Datentypen, Weiten und Abtastraten der Blöcke und Signale festgelegt. Anschließend werden die Blockparameter ausgewertet und die Blockreihenfolge zur Ausführung bestimmt. Im nächsten Schritt wird die Simulation in einer Schleife ausgeführt. Das einmalige durchlaufen dieser Schleife wird auch als Simulationsschritt (Simulation step) bezeichnet. Der Aufbau der Simulationsschleife hängt von den Eigenschaften des Modelles ab. Besitzt ein Simulink-Modelle Blöcke mit variablen Abtastraten wird vor jedem Simulationsschritt dessen Ausführungszeitpunkt berechnet. Bei einem Modell fixen Abtastraten wird diese Berechnung nicht benötigt. Außerdem wird zwischen Blöcken mit diskreten und kontinuierlichen Zuständen unterschieden. Ein Block mit diskreten Zuständen wird einmal pro Simulationsschritt aufgerufen um seinen, für diesen Simulationsschritt, aktuellen Zustand und Ausgangswerte zu berechnen. Bei einem Block mit kontinuierlichen Zuständen wird dieser mit einer höheren Abtastrate als das restliche Model aufgerufen. In diesen Aufrufen werden die aktuellen Ausgangswerte und Ableitungen des Blocks berechnet.

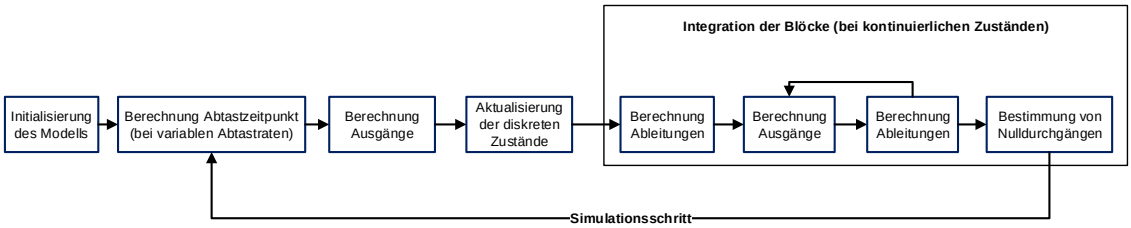

Abbildung 1: Ablauf einer Simulation, Quelle: eigene Darstellung, Inhalt aus [1]

2.2 Codegeneration mit Simulink

Simulink bietet neben der Standardausführung eines Modell auch die Generation von Quellcode, der sowohl auf der Host- als auch auf anderen Zielplattformen ausgeführt werden. Die hierfür benötigten Produkte sind *Simulink-Coder*[1] als Basis und ggf. *Embedded-Coder*, welcher die Generation von optimierten C/C++-Code für Microcontroller-Plattformen ermöglicht. Die folgenden Abschnitte stellen den Ablauf einer Codegenration für Modelle mit einfachen, fixen Abtastraten vor. Für weitere Details über die Codegeneration von komplexen Modellen, welche beispielsweise variable Abtastraten, kontinuierliche Zustände und asynchrone Scheduling-Strategien verwenden, sei auf [3] und [4].

[1] Früher als Real-Time-Workshop bekannt, deshalb wird in vielen Quellen nach wie vor von dem Real-Time-Workshop bzw. RTW gesprochen.

2.3 Ablauf einer Simulink-Codegeneration

Der Beginn einer Codegeneration ist identisch mit der üblichen Simulation. Das heißt, dass die Signale, Blockparameter und Abtastraten evaluiert werden. Anschließend legt Simulink die Ausführungsreihenfolge der Blöcke fest. Anschließend erstellt *Simulink-Coder* eine RTW-Datei (.rtw), welche diese Informationen über das Modell enthält. Im nächsten Schritt wird die RTW-Datei dem Target Language Compiler (TLC) übergeben, welcher die Generation des Quellcodes durchführt. TLC ist eine von TheMathWorks entwickelte Programmiersprache bzw. der Interpreter, welche diese auswertet. Diese Programmiersprache ermöglicht einerseits die Auswertung von RTW-Dateien, andererseits können auch Code-Datein in beliebigen Sprachen erstellt werden. Für mehr Informationen über die Funktionsweise und Syntax von TLC sei auf [2] verwiesen.

Für die Codegeneration benötigt der TLC neben der RTW-Datei des Modelles, ein System Target File (STF), ein Template Makefile (TMF), die Standard-TLC-Bibliothek und die TLC-Dateien der einzelnen Blöcke, welche die Vorschrift für die Codegeneration des Blocks enthalten.

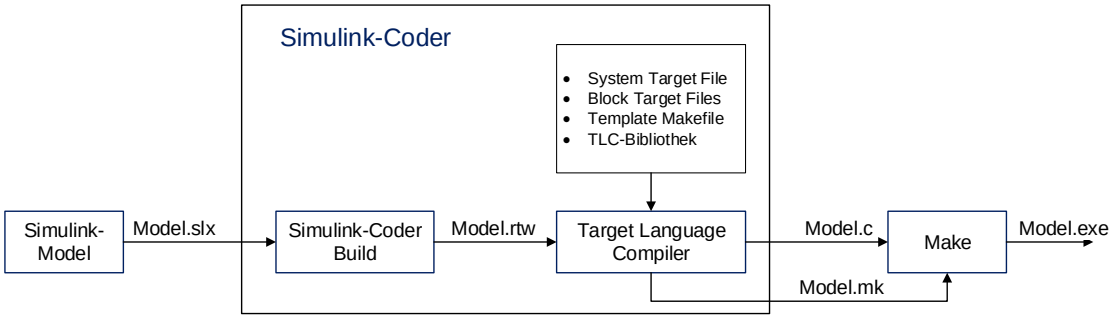

Abbildung 2: Ablauf Codegeneration, Quelle: eigene Darstellung, Inhalt aus [2]

2.3.1 System Target File

Das STF stellt den Ausgangspunkt der TLC-Codegeneration dar. In dieser Datei werden zuerst die Grundeinstellungen, wie z.B. das Codeformat und die Programmiersprache festgelegt. Anschließend werden eigenen Simulink-Blöcken TLC-Dateien zugeordnet, welche die Generationsvorschriften enthalten. Im nächsten Schritt wird die Ausführung der Simulink-Blöcke in entsprechender Reihenfolge in Quellcode übersetzt. Hierfür wird in der Regel die Datei "codegenentry.tlc"verwendet, welche von Simulink bereitgestellt wird. Diese Datei ordnet den Standard-Blöcken die entsprechenden TLC-Dateien zu, initialisiert globale Variablen und führt die Codegeneration eines einzelnen Simulationsschrittes durch. Zuletzt wird eine TLC-Datei inkludiert, welche eine main-Datei erstellt. In der main-Routine werden ggf. allgemeine Ressourcen initialisiert und die Simulationsschritte, den Abtastraten entsprechenden, aufgerufen.

2.3.2 Template Make File

Der TLC erzeugt bei der Codegeneration ein Makefile um den Quellcode in ein ausführbares Format zu übersetzen. Das Makefile wird von einem TMF abgeleitet, in welchem unter anderem die Toolchain, zusätzliche Bibliotheken und Compiler-Einstellungen festgelegt werden. Somit ermöglicht es ein TMF auch Modelle auf beliebigen Zielplattform auszuführen, insofern eine Cross-Compiler-Toolchain zur Verfügung steht.

3 Aufbau und Implementierung von Simulink-Blöcken

Ein Simulink-Block muss Funktionen bereitstellen um die einzelnen Schritte einer Simulation zu ermöglichen. Darunter fallen die Folgenden Funktionen:

- Initialisierung des Blocks
- Berechnung des nächsten Abtastzeitpunktes (falls der Block variable Abtastraten benötigt)
- Berechnung der Ausgangswerte
- Berechnung der diskreten Zustände des Blocks
- Integration, hierunter fallen die Berechnung der Ausgangswerte und Ableitungen bei Zwischenschritten (wird nur bei kontinuierlichen Zuständen benötigt)

Die Umsetzung dieser Methoden erfolgt in der Form einer sogenannten *S-Function*. Hierbei handelt es sich lediglich um eine Sammlung von Funktionen, welche die oben genannten Aktionen implementieren. Eine *S-Function* kann in unterschiedlichen Programmiersprachen implementiert werden, wobei MATLAB und C/C++ die üblichsten sind. Hier sollen lediglich *S-Functions* in C/C++ vorgestellt werden, in [1] werden alle Alternativ detailliert erklärt.

3.1 C/C++ S-Function

In diesem Abschnitt wird der Aufbau einer *S-Function* am Beispiel eines einfachen Blocks, der seinen Eingang mit einem fixen Wert multipliziert, erklärt. Dieser Block benötigt keine kontinuierlichen Zustände, somit müssen lediglich Funktionen zur Initialisierung, Berechnung der Ausgänge und Terminierung am Ende der Simulation implementiert werden.

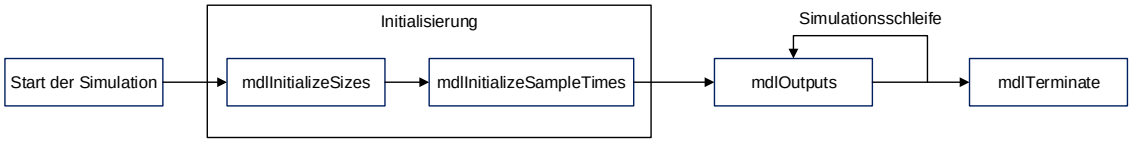

Abbildung 3: Interaktion eines diskreten Block mit der Simulink-Engine, Quelle: eigene Darstellung, Inhalt aus [1]

Die Implementierung in C++ beginnt mit der Definition der des Namen und Level der *S-Function*. Das Level einer *S-Function* sollte auf 2 gesetzt werden, bei dem Level 1 handelt es sich um einen veralteten *S-Function* Typ. Außerdem werden die benötigten Header-Dateien inkludiert.

```
1  #define S_FUNCTION_NAME CustomGain_SFunction
2  #define S_FUNCTION_LEVEL 2

4  #include "simstruc.h"
5  #include "matrix.h"
```

Anschließend werden die in **??** dargestellten Funktionen implementiert. Die Funktion *mdlInitializeSizes* legt die Anzahl der Parameter, Ein- und Ausgänge fest. Anschließend werden die Weiten der Ein- bzw. Ausgangssignale eingestellt und die Anzahl der Abtastraten festgesetzt. Hierfür wird das s.g. *SimStruct* verwendet. Hierbei handelt es sich um eine Baumstruktur, welche die Informationen über das gesamte Model und Blöcke enthält. Die Wurzel des *SimStruct* ist das gesamt Modell, welches in die verschiedenen Subsysteme und Blöcke verzweigt. Die *SimStruct*-Referenz die den Funktionen einer *S-Function* übergeben wird, verweist auf das jeweilige Blatt der Baumstruktur, welche den Block repräsentiert.

```
1  static void mdlInitializeSizes(SimStruct* S)
2  {
3      //Anzahl der S-Function Parameter festlegen
4      ssSetNumSFcnParams(S,1);
5      if(ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) return;

7      //Anzahl der Eingaenge und deren Weite festlegen und ueberpruefen
8      if(!ssSetNumInputPorts(S,1)) return;
9      ssSetInputPortWidth(S, 0, 1);
```

```
10      ssSetInputPortDirectFeedThrough(S, 0, 1);

12      //Anzahl der Ausgaenge und deren Weite festlegen
13      if(!ssSetNumOutputPorts(S,1)) return;
14      ssSetOutputPortWidth(S, 0, DYNAMICALLY_SIZED);

16      //Anzahl der Abtastsraten festlegen
17      ssSetNumSampleTimes(S, 1);
18  }
```

In der Initialisierungsphase müssen außerdem die Abtastraten spezifiziert werden. Dies geschieht in der Funktion *mdlInitializeSampleTimes*. Der Wert *INHERITED_SAMPLE_TIME* bedeutet, dass die Abtastrate der Blöcke, welche das Ausgangssignal erhalten, übernommen wird.

```
1   static void mdlInitializeSampleTimes(SimStruct* S)
2   {
3       //Festelegen der Abtastrate und Abtastoffset auf INHERITED
4       ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
5       ssSetOffsetTime(S, 0, 0.0);
6   }
```

Während der Simulation ruft die Simulink-Engine die *mdlOutputs* Funktion auf um den Ausgangswert des Blocks zu ermitteln. In diesem Fall wird der Eingangswert mit dem Block-Parameter multipliziert.

```
1   static void mdlOutputs(SimStruct* S, int_T tid)
2   {
3       //Eingang-, Ausgangsignale und Parameter von dem SimStruct abfragen
4       InputRealPtrsType inputPtrArr  = ssGetInputPortRealSignalPtrs(S, 0);
5       real_T* outputPtr              = ssGetOutputPortRealSignal(S, 0);
6       const mxArray* parameterPtr    = ssGetSFcnParam(S, 0);
7       //Ausgangswert berechnen
8       *outputPtr = (*(inputPtrArr[0])) * (*mxGetPr(parameterPtr));
9   }
```

Am Ende der Simulation wird die *mdlTerminate* Funktion ausgeführt. Hier müssen ggf. reservierte Ressourcen wieder freigeben werden.

```
1   static void mdlTerminate(SimStruct* S)
2   {
3   }
```

Am Schluss der C++-Datei befinden sich weitere Präprozessordirektiven, welche für die Codegeneration und MEX-Compilation benötigt werden. Diese beiden Ausführungsmodi werden in einem späteren Abschnitt näher erläutert.

```
1   #ifdef MATLAB_MEX_FILE
2   #include "simulink.c"
3   #else
4   #include "cg_sfun.h"
5   #endif
```

3.2 Codegeneration einer S-Function

Eine *S-Function* kann auch in Quellcode übersetzt werden. Die einfachste Form besteht in einer s.g. *non-inlined S-Function*, die darin besteht, dass die C/C++-Datei der *S-Function* direkt als Quelldatei verwendet wird. Somit muss allerdings das komplette *SimStruct* in Quellcode übersetzt werden und den *S-Function* während der Ausführung übergeben werden, wodurch die Größe und Ausführungszeit des Modelles erhöht wird.

Mit Hilfe von TLC-Dateien kann eine *S-Function* effizient in Quellcode übersetzt werden. In der TLC-Datei werden die Vorschriften festgehalten, wie die Ausgangswerte und Zustände eines Blocks in der gewählten Programmiersprache berechnet werden. Der folgende Codeausschnitt zeigt die TLC-Implementation der bereits gezeigten Gain-*S-Function*.

```
1   %implements "CustomGain_SFunction" "C"

3   %function Outputs(block, system) Output
4       %assign gainFactor  = LibBlockParameter(0, "", "", 0)
5       %assign inputValue  = LibBlockInputSignal(0, "", "", 0)
6       %assign outputValue = LibBlockOutputSignal(0, "", "", 0)
7       %<outputValue>      = %<inputValue> * %<gainFactor>
8   %endfunction
```

Für den Quellcode muss lediglich die Output-Funktion generiert werden, in welcher das Eingangsignal mit dem Wert des Blockparameter multipliziert wird. Auf die Syntax und Funktionalität von TLC wird hier nicht näher eingegangen, da Simulink mehrere Werkzeuge zur Verfügung stellt, die es ermöglichen *S-Functions* und die zugehörigen TLC-Dateien automatisch zu erzeugen. Diese werden in den nächsten Abschnitt genauer erklärt.

3.3 Kompilierung einer S-Function

C/C++ *S-Functions* müssen in MEX-Dateien übersetzt werden, damit sie in Simulink verwendet werden können. Matlab kann hierfür verschiedene Compiler wie z.B. *MinGW* oder *GCC* verwenden. Eine C/C++ Quelldatei kann über die Matlab Konsole mit dem folgenden Befehl in eine MEX-Datei übersetzt werden.

```
1  mex CustomGain_SFunction.cpp
```

Matlab wählt hier automatisch den Compiler, welcher der Dateiendung entspricht. In dem Fall, das mehrere MEX-kompatible Compiler installiert sind, kann mit dem folgenden Befehl der gewünschte Compiler ausgewählt werden.

```
1  mex -setup myCompiler
```

Die Verwendung von TLC-Dateien zur Codegeneration einer *S-Function* bringt hier weitere Vorteile mit sich. Da eine *non-inlined S-Function* ihre C/C++-Quelldatei sowohl für die MEX- als auch Target-Kompilierung verwendet wird, muss mit Hilfe von Makros der Quellcode geteilt werden. Der erste Teil dient als Quelle für die MEX-Datei. Der zweite Teil wird für die generierte Datei verwendet.

```
1  #ifdef MATLAB_MEX_FILE
2    /* Quellcode fuer die MEX-Datei */
3  #else
4    /* Quellcode fuer die Zielplattform */
5  #endif
```

Dadurch ensteht eine unnötig große und unübersichtliche Quelldatei. Bei der Verwendung einer TLC-Datei wird die C/C++-Quelle lediglich für die MEX-Kompilierung verwendet. Der Quellcode für die Zielplattform-Kompilierung wird mit Hilfe der TLC-Datei kompiliert.

Literaturverzeichnis

[1] TheMathWorks, "Writing S-Functions", März 2016

[2] TheMathWorks, "Target Language Compiler", März 2016

[3] TheMathWorks, SSimulink Coder User Guide", März 2016

[4] TheMathWorks, Ëmbedded Coder User Guide", März 2016