Universität Passau

Fakultät für Mathematik und Informatik

Bachelorarbeit

im Studiengang Mobile und Eingebettete Systeme

zur Erlangung des akademischen Grades
Bachelor of Science

Thema:	Bildverarbeitungs-Plugins auf einem Microcontroller
Autor:	Hann Lauritz Holze holzehan@fim.uni-passau.de MatNr. 67254
Version vom:	4. Oktober 2016
Betreuer:	Prof. Dr. Tomas Sauer

Zusammenfassung

Software für Mikrocontroller ist in der Regel ein einziger Block Code, welcher im Nachhinein sehr schwierig zu erweitern ist. Auf gewöhnlichen Computern gibt es daher das Prinzip der Plug-In Systeme. Diese können die Funktionalität einzelner Programme erweitern und ändern. In dieser Arbeit wird ein solches Plug-In System für Mikrocontroller entwickelt. Dazu wird als Proof of Concept ein Plug-In vorgestellt, welches die Wavelet-Transformation auf einem Mikrocontroller umsetzt. Diese Implementierung wird mit Blick auf die Sicherheit und die Performance analysiert.

Inhaltsverzeichnis

Abbildungsverzeichnis **3**

Listingverzeichnis **4**

1 Einleitung **5**

2 Grundlagen **6**
2.1 Mikrocontroller 6
2.2 Plug-In 7
2.3 Wavelet-Transformation 7

3 Entwicklungsumgebung **10**

4 Mikrocontroller und Plug-Ins **11**
4.1 Harvard-Architektur 11
4.2 Funktionszeiger 11
4.3 Funktionszeigertabelle 12
4.4 Ausgabe 13
4.5 API 13
4.6 Übersetzen 13
4.7 Installation 14
4.8 Konfiguration 14

5 Fast Wavelet Transformation als Plug-in **16**

6 Alternativen **17**
6.1 Scriptsprachen 17
6.2 Betriebssysteme 17

7 Evaluation **19**

8 Ausblick **21**

9 Fazit **22**

Literaturverzeichnis **23**

Anhang **24**

Eidesstattliche Erklärung **28**

Abbildungsverzeichnis

1 Vergleich der Zeit-/Frequenzauflösung von Short Time Fourier Transformation und Wavelet-Transformation [Quelle [Com13]] 8
2 Fast Wavelet Transformation als rekursive Filterbank [Quelle: [Com05]] 9
3 Das STM32F7G-EVAL Board [Quelle: [STM]] 10
4 Verwendung der Funktionstabelle. Das Hauptprogramm schlägt die Adresse der *Funktion 2* in der Tabelle nach und führt die Funktion dann aus. 12

Listingverzeichnis

1 API eines Plug-Ins wie in plugin.h implementiert 14
2 Installation eines Plugins wie in der Verwaltung implementiert 15
3 Die API wie in plugin_def.h 24
4 Das entwickelte Wavelet Plug-In 24

1 Einleitung

Durch den immer größeren Einsatz von vernetzten und intelligenten Systemen hat auch die Verbreitung von Mikrocontrollern in den letzten Jahren deutlich zugenommen. Diese sind so klein und kostengünstig, dass sie sehr einfach in alle möglichen Produkte integriert werden können. Im Gegensatz zu normalen Computern ist es im allgemeinen jedoch nicht möglich Mikrocontroller durch neue Programme zu erweitern, da auf diesen eine sehr spezialisierte Firmware ausgeführt wird. Die Entwicklungszyklen von Software werden gleichzeitig immer kürzer. Daher können nicht mehr alle Features direkt in der Firmware implementiert werden. Somit wäre es wünschenswert, nachträglich neue Funktionen ergänzen zu können. Dabei soll die Grundfunktionalität natürlich nicht geändert werden.

Die vorliegende Arbeit analysiert, wie ein Plug-In System auf einem Mikrocontroller umgesetzt werden kann, um eine größere Flexibilität zu erreichen. Diese Ergebnisse werden in Form eines Proof of Concepts umgesetzt. Hierfür werden eine Plug-In-Verwaltung, so wie ein beispielhaftes Plug-In entwickelt. Dieses implementiert den Algorithmus der schnellen Wavelet-Transformation.
Zunächst werden dafür die Konzepte Mikrocontroller, Plug-In und der Wavelet-Transformation eingeführt. Im Hauptteil werden die Möglichkeiten und Probleme betrachtet, die bei der Portierung auf einen Mikrocontroller entstehen. Dabei wird auch eine Beschreibung des entwickelten Proof of Concepts und der gewählten Lösungen gegeben. Abschließend folgt eine Einschätzung zur Sicherheit und Performance dieses Systems und ein kurzer Vergleich mit alternativen Systemen.

2 Grundlagen

In diesem Kapitel werden die Begriffe *Mikrocontroller*, *Plug-In* und *Wavelet-Transformation* eingeführt, welche für die Implementierung benötigt werden.

2.1 Mikrocontroller

Ein Mikrocontroller ist ein System bestehend aus einem Mikroprozessor und direkt daran angebundener Peripheriehardware, wie etwa einem RAM, einem Flash-Speicher oder verschiedene Bus-Schnittstellen. Diese sind in der Regel zusammen auf einem einzigen Chip umgesetzt. Durch die direkte Anbindung sind Mikrocontroller sehr flexibel einsetzbar. Dank der geringen Hardware Anforderungen sind sie meist sehr klein und kostengünstig. Sie eignen sich daher ideal zur Steuerung verschiedenster eingebetteter Systeme (vgl. [Ber15]).
Im Gegensatz zu Prozessoren in Computern sind die meisten Mikrocontroller nach der Harvard-Architektur aufgebaut. Das bedeutet der Befehlsspeicher und der Datenspeicher sind auf der Hardware Ebene voneinander getrennt. Dies sorgt für zusätzliche Sicherheit, da zufällig oder böswillig hinzugefügte Befehle aus dem Datenspeicher nicht als Anweisungen auf dem Prozessor ausgeführt werden können. Zusätzlich wird dadurch der gleichzeitige Zugriff auf Daten und Befehle ermöglicht wodurch die gesamte Geschwindigkeit erhöht wird. Ein weiterer Vorteil ist die daraus resultierende feste Größe der beiden Bereiche, wodurch sichergestellt ist, dass die Daten nicht den Programmcode überschreiben.
Der normale Ablauf beim Einsatz eines Mikrocontrollers sieht wie folgt aus: Zunächst wird die auszuführende Software, welche zumeist in der Programmiersprache C geschrieben ist, für die spezielle Architektur des Mikrocontrollers übersetzt. Dabei fügt der Compiler zusätzlich den Startcode für den verwendeten Mikrocontroller hinzu. Dieser initialisiert alle nötigen Variablen und startet die Timer. Im Anschluss wird das übersetzte Programm mit zusätzlicher Hard- und Software in den Flash-Speicher geschrieben. Während der eigentlichen Anwendungsphase wird nur noch der Datenspeicher durch Inputs manipuliert, nicht aber der Programmspeicher.
Die Software aus dem Flash-Speicher wird direkt auf der Hardware ausgeführt. Daher kommt in der Regel keinerlei Betriebssystem zum Einsatz. Ein solches würde einen zu großen overhead erzeugen. Die so ausgeführten Befehle sind deutlich schneller, als es mit einem Betriebssystem der Fall wäre. Allerdings gibt es dadurch auch keine der sonst üblichen Unterstützungen, wie automatisches Speichermanagement, Prozess-Scheduling oder eine Dateiverwaltung. Dies gibt dem Programmierer einiges an Freiheiten, kann jedoch auch schneller zu Fehlern führen. Es ist außerdem nötig, alle Ein- und Ausgaben direkt über die Hardware zu steuern. Dafür stehen zum Glück oftmals Bibliotheken zur Verfügung.

2.2 Plug-In

Plug-Ins stellen eine Schnittstelle zwischen einem Hauptprogramm und einer Erweiterung zur Verfügung. Diese Plug-Ins können vom selben Hersteller oder auch von fremden Entwicklern stammen [1]. Um dies zu ermöglichen, wird eine API definiert über die Erweiterungen mit dem Hauptprogramm kommunizieren können. Ein Plug-In System besteht in der Regel aus zwei Teilen, der Plug-In-Verwaltung und dem Plug-In selber.
Das Hauptprogramm implementiert eine Plug-In-Verwaltung, die den installierten Plug-Ins alle nötigen Funktionen zur Verfügung stellt. Sie ist zuständig für das Auffinden der Plug-Ins, entscheidet welche davon geladen werden und wann sie ausgeführt werden.
Das Plug-In greift auf die bereitgestellten Funktionen zu und kommuniziert so mit dem Hauptprogramm. Dazu implementiert es die API, welche von der Verwaltung vorgegeben ist. Die Funktionalität des Hauptprogramms kann so durch Dritte individuell erweitert werden.
Der Vorteil von Plug-Ins besteht darin, dass fremden Entwicklern die Möglichkeit gegeben wird, die Funktionalität des Programmes in einer sicheren Weise zu erweitern, ohne gleichzeitig das ganze Programm offen zulegen müssen (z.B. durch Veröffentlichung des Quellcodes). Außerdem kann der Entwickler des Hauptprogramms Einschränkungen in der API festlegen denen Plug-Ins unterliegen (z.B. eingeschränkter Zugriff auf sensible Daten). Ein großer Vorteil ist es auch das Programm erweitern und anpassen zu können, ohne dadurch das gesamte Programm neu Compilieren zu müssen.
Die meisten Programmiersprachen stellen Bibliotheken zum Erstellen eines Plug-In Systems bereit. Diese wiederum verwenden Betriebssystem Funktionen zum Auffinden und Verwalten der Plug-Ins.

2.3 Wavelet-Transformation

Die Wavelet-Transformation wird verwendet, um ein Signal vom Zeitbereich in den Frequenzbereich zu überführen. Dabei hat sie, im Gegensatz zur Short Time Fourier Transformation, eine höhere Genauigkeit bei der örtlichen Auflösung im Frequenzbereich bei höheren Frequenzen und eine bessere Frequenzauflösung bei tiefen Frequenzen, wie in Abbildung 1 dargestellt. Daher findet die Wavelet-Transformation häufig in den Bereichen der Bild- und Tonverarbeitung Verwendung (vgl. [Mal09]).
Bei der Wavelet-Transformation wird das zu analysierende Signal mit einer Wavelet-Funktion bzw. dessen Traslaten und Dilaten verglichen. Dabei werden in jeder Stufe von Dilaten weitere Details sichtbar. Die verwendete Wavelet-Funktion wird meist nach bestimmten Anforderungen aus dem Einsatzgebiet ausgewählt. Dafür steht eine große Zahl fest definierter Wavelets zur Verfügung, die sich als nützlich erwiesen

[1] Im Gegensatz dazu werden Addons in der Regel nur vom selben Hersteller geliefert. Die Abgrenzung ist allerdings nicht ganz eindeutig

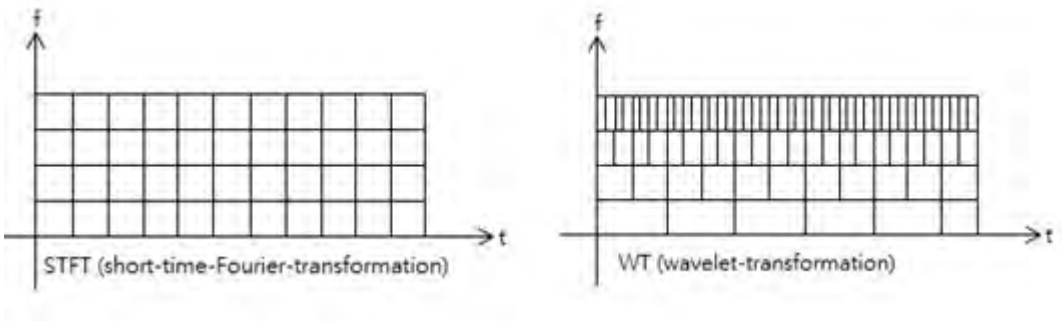

Abbildung 1: Vergleich der Zeit-/Frequenzauflösung von Short Time Fourier Transformation und Wavelet-Transformation [Quelle [Com13]]

haben. Zu jeder Wavelet-Funktion gehört auch immer eine Skalierungsfunktion. Diese berechnet die Approximations-Koeffizienten a_j, während die Wavelet-Funktion die Detail-Koeffizienten d_j berechnet. Diese repräsentieren das Signal im Frequenzbereich. Die Wavelet-Transformation hat eine inverse Funktion und kann damit verlustfrei wieder zurücktransformiert werden.
Für die digitale Berechnung wird die diskrete Wavelet-Transformation verwendet. Diese kann sehr schnell berechnet werden, da nur eine Reihe einfacher linearer Operationen ausgeführt werden muss. Im Folgenden wird der Algorithmus der Diskreten Wavelet-Transformation näher beschrieben (auch Fast Wavelet Transformation genannt).

1. Für die Berechnung der Detail-Koeffizienten d_j wird zunächst das Signal s_i mit den Wavelet-Koeffizienten g_k gefaltet $d_j = (s * g)(j) = \sum_x s_x * g_{x-j}$.

2. Das Ergebnis wird downgesamplet, es wird also nur jeder zweite Koeffizient gespeichert. Die anderen werden verworfen. Dies entspricht einer Stauchung um den Faktor zwei.

3. Für die Approximations-Koeffizienten a_j wird ebenfalls zunächst das Signal gefaltet, diesmal jedoch mit den Skalierungs-Koeffizienten h_k. Daraus ergibt sich dann $a_j = (s * h)(j) = \sum_x s_x * h_{x-j}$.

4. Auch hier wird das Signal downgesamplet.

Die Transformation kann dann immer wieder auf die Approximations-Koeffizienten angewandt werden, wie in Abbildung 2 dargestellt. Dadurch wird der Anteil dieser am Signal immer geringer, bis das Signal am Ende nur noch aus Detail-Koeffizienten besteht. Jede dieser Stufen wird Filterbank genannt. Die Rücktransformation zum Ausgangssignal sieht ähnlich aus.

1. Die Detail-Koeffizienten werden upgesamplet. Dabei werden die Detail-Koeffizienten a_j immer an die Stelle $2j$ geschrieben und die Stellen dazwischen mit 0 aufgefüllt. Dies entspricht einer Streckung um den Faktor 2.

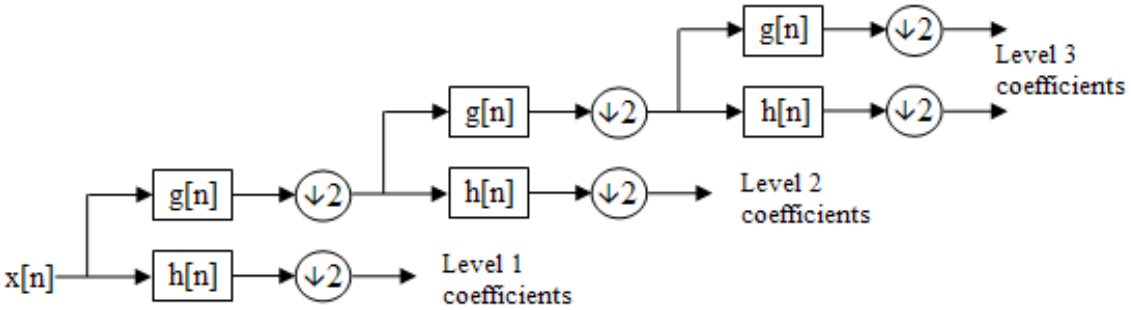

Abbildung 2: Fast Wavelet Transformation als rekursive Filterbank [Quelle: [Com05]]

2. Das gestreckte Signal wird mit den Wavelet-Koeffizienten gefaltet.
3. Auch die Approximations-Koeffizienten werden upgesamplet.
4. Dieses gestreckte Signal wird mit den Skalierungs-Koeffizienten gefaltet.
5. Zum Schluss werden die beiden Folgen von Koeffizienten addiert.

Da die Wavelet-Koeffizienten zumeist sehr klein sind, kann die Bitrate des Signals deutlich reduziert werden, mit Hilfe einer Quantisierung und einer anschließenden Codierung. Ebenso kann das Signal durch Thresholding von kleinen Koeffizienten entrauscht werden. Dabei erhält die Wavelet-Transformation Kanten deutlich besser als andere Verfahren. Des weiteren wird die Wavelet-Transformation eingesetzt, um Analysen direkt auf den Wavelet-Koeffizienten, also im Frequenzspektrum, durchzuführen.

Abbildung 3: Das STM32F7G-EVAL Board [Quelle: [STM]]

3 Entwicklungsumgebung

Die Software für diese Arbeit wurde für ein *STM32746G-EVAL* Board entwickelt [STM16], zu sehen in Abbildung 3, welches freundlicherweise von der Firma Micro-Epsilon zur Verfügung gestellt wurde. Dieses Board ist mit einem ARM Cortex M7 Mikroprozessor ausgestattet. Außerdem bietet es eine Vielzahl von Schnittstellen die zur Kommunikation eingesetzt werden können.
Die Programmierung selber ist in der *Stm Workench for STM32* Entwicklungsumgebung geschehen [AC6]. Diese bringt bereits die Hardware Abstraktion für das Entwickler-Board mit, die passende Toolchain zum compilieren, Tools zur Flash Programmierung und einen Debugger. Damit sind direkt alle benötigten Werkzeuge bereit.

4 Mikrocontroller und Plug-Ins

Die Software auf Mikrocontrollern ist in der Regel sehr unflexibel, wenn es um nachträgliche Anpassungen geht. Dies könnte durch den Einsatz eines Plug-In Systemes verbessert werden. In diesem Kapitel werden die Probleme und Besonderheiten aufgezeigt, die dabei auftreten. Dabei werden auch die Lösungsansätze gegeben, die zur Implementierung des Proof of Concept verwendet wurden.

4.1 Harvard-Architektur

Aufgrund der bei Mikrocontrollern eingesetzten Harvard-Architektur ist es im Allgemeinen nicht möglich Programmcode, der nachträglich hinzugefügt wurde, aus dem RAM auszuführen. Da RAM kein persistenter Speicher ist, würden ein dort gespeichertes Plug-In beim Abschalten der Stromversorgung außerdem verloren gehen. Daher ist es besser, neue Plug-Ins zunächst von der Plug-In-Verwaltung in den Flash-Speicher schreiben zu lassen. Der für diese Arbeit verwendete Mikroprozessor von STMicroelectronics stellt dafür die Funktion *HAL_FLASH_Program()* bereit. Mithilfe dieser können zur Laufzeit beliebige Binärdaten in den Flash-Speicher geschrieben werden. Passend dazu kann mit der Funktion *HAL_FLASHx_erase()* Flash-Speicher blockweise wieder gelöscht werden. Dabei ist es wichtig, dass im Vorhinein definiert wird, welche Speicherstellen für Plug-Ins zur Verfügung stehen und wo sich diese befinden. Ansonsten werden unter Umständen andere Teile des Programmes überschrieben. Ebenso muss die Plug-in-Verwaltung nachhalten, welche Speicherbereiche von anderen Plug-Ins belegt sind.

4.2 Funktionszeiger

In der Programmiersprache C werden Funktionszeiger verwendet, wenn im Programmfluss eine Funktion aufgerufen wird. Sie sind in der Regel für den Programmierer nicht direkt erkennbar, stehen jedoch hinter jedem Funktionsaufruf. Alternativ können Funktionszeiger auch von Hand verwendet und verwaltet werden. Bei unvorsichtiger Verwendung kann dies allerdings zu erheblichen Fehlern führen. Daher wird in der Regel auch davon abgeraten, Funktionszeiger einzusetzen, sofern sich diese vermeiden lassen. Im Falle eines Plug-Ins erstellt die Verwaltung zunächst einen leeren Funktionszeiger. Dieser wird dann mit der Flash-Adresse, an der das Plug-In gespeichert wurde, gefüllt. Von nun an kann das Plug-In wie jede andere Funktion aufgerufen werden, indem der Funktionszeiger wie eine Funktion aufgerufen wird. Dabei springt das Programm direkt an die erste Speicherstelle des Plug-Ins. Deshalb muss dieses dem in Abschnitt 4.5 beschriebenen Aufbau folgen. Wichtig ist es außerdem, auf die richtige Übergabe der Parameter zu achten, da diese zur Laufzeit nicht mehr überprüft werden.

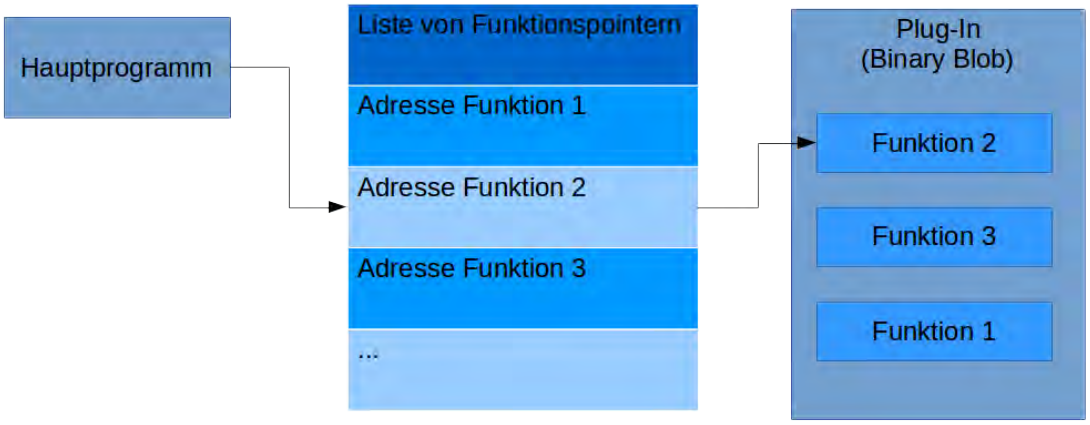

Abbildung 4: Verwendung der Funktionstabelle. Das Hauptprogramm schlägt die Adresse der *Funktion 2* in der Tabelle nach und führt die Funktion dann aus.

4.3 Funktionszeigertabelle

Damit das Plug-In auf ausgewählte Funktionen des Hauptprogrammes zugreifen kann, braucht dieses ebenfalls Funktionszeiger zu den Funktionen der Verwaltung. Dazu muss als Parameter bei der Initialisierung des Plug-Ins eine Tabelle mit Funktionszeigern an das Plug-In übergeben werden. In dieser stehen alle Funktionen, auf welche das Plug-In Zugriff haben soll. Auch das Plug-In schreibt dann die Adressen seiner eigenen Funktionen mit in die Liste. Damit kann auch die Verwaltung auf alle Funktionen zugreifen, die das Plug-In bereitstellt. Die genaue Definition, welche Funktionen an welcher Stelle gelistet ist, wird in der API in Abschnitt 4.5 spezifiziert.
Durch eine solche Funktionstabelle wird auch eine zusätzliche Abstraktion eingeführt. Das Plug-In greift nicht direkt auf Speicherstellen zu, sondern nutzt die dafür vorgesehenen Funktionen des Hauptprogramms, wie in Abbildung 4 dargestellt. Ebenso wird eine Kapselung erreicht, indem die bereitgestellten Funktionen beispielsweise bestimmte Speicherregionen nur lesen können. Die Sicherheit des Hauptprogramms kann damit besser gewährleistet werden.

4.4 Ausgabe

Für die Ausgabe eines Plug-Ins gibt es zwei Alternativen. Entweder wird die Rückgabefunktionalität der Funktion selber verwendet, oder das Plug-In modifiziert die Werte in einem gemeinsamen Speicher direkt, den es sich mit dem Hauptprogramm teilt. Ein Rückgabewert hat den Vorteil, dass dieser zu einem festen Zeitpunkt (der Beendigung der Funktion) erzeugt wird. Außerdem kann so schreibender Zugriff auf den Speicher des Hauptprogramms gänzlich vermieden werden. Gleichzeitig bedarf es aber einer deutlich komplexeren Definition der Rückgabewerte, damit das Hauptprogramm diese weiter verwenden kann. Das Überschreiben der Werte im Hauptprogramm kann ebenfalls von Vorteil sein. Insbesondere wenn die ausgegebenen Werte das selbe Format wie die eingegebenen haben, verringert sich dadurch der Aufwand und der Speicherbedarf. Da die Wavelet Transformation ein In-Place Algorithmus ist, wurde für den Proof of Concept die Lösung mit einem gemeinsamen Speicherbereich gewählt.

4.5 API

Um den Austausch zwischen der Plug-In-Verwaltung und einem Plug-In sicherstellen zu können, muss im Vorhinein eine API definiert werden. Diese legt fest, welche Funktionen von der Verwaltung und vom Plug-In zur Verfügung gestellt werden müssen. Außerdem definiert die API einen oder mehrere gemeinsame Speicherbereiche für den Datenaustausch. Diese API wird am sinnvollsten in einer C-header-Datei (.h) festgelegt. Sie muss dann von der Plug-In-Verwaltung und allen Plug-Ins eingebunden werden.
Die für diese Arbeit entworfene API sieht drei Funktionen für das Plug-In vor. Diese sind eine Intitialisierungsfunktion, welche direkt bei der Installation aufgerufen wird (*plugin_init()*), eine Funktion zum Ausführen der eigentlichen Funktionalität (*execute()*) und eine Funktion welche bei Bedarf reservierte Ressourcen zurück gibt, wenn das Plug-In deinstalliert wird (*stop()*). Für die Plug-In-Verwaltung sind die drei Funktionen *get_values()*, *get_last_value()* und *get_values_size()* vorgesehen. Außerdem wird ein gemeinsamer Speicher definiert, der zum Datenaustausch verwendet werden kann (*plugin_result* und *plugin_size_result*).

4.6 Übersetzen

Im Gegensatz zu einem eigenständigen Programm, welches direkt auf dem Mikrocontroller läuft, braucht ein Plug-In keinerlei Startcode. Diesen beinhaltet das Hauptprogramm bereits. Somit sind bereits vor der Ausführung des Plug-ins alle benötigten Werte initialisiert. Würde das Plug-In das System erneut initialisiert, liefe nur noch das Plug-In und das Hauptprogramm hätte keinerlei Kontrolle mehr. Damit der Compiler den Startcode nicht automatisch generiert, muss das entsprechende Compiler-Flag gesetzt werden. Außerdem muss festgelegt werden, mit welcher Funktion das Programm

```
typedef struct {
        //Functions in plug-in
        uint32_t (*plugin_init)();
        uint32_t (*plugin_execute)();
        uint32_t (*plugin_stop)();
        uint32_t *plugin_result;
        uint32_t plugin_size_result;

        //Functions in main
        uint32_t (*get_values)();
        uint32_t (*get_last_value)();
        uint32_t (*get_values_size)();
}plugin;
```

Listing 1: API eines Plug-Ins wie in plugin.h implementiert

beginnen soll. Normalerweise ist dies die *main()* Funktion. Bei einem Plug-In ist dies jedoch die Initialisierungsfunktion. So kann die Plug-In-Verwaltung ohne näheres Wissen über den internen Aufbau des Plug-Ins dieses installieren und initialisieren.

4.7 Installation

Zunächst muss das Plug-In an den Mikrocontroller übertragen werden. Dies ist entweder während des Flash-Vorganges des Mikrocontrollers möglich oder das Plug-In wird während des laufenden Betriebs gesendet. Soll das Plug-In während des Betriebs installiert werden, muss dieses über eine verfügbare Schnittstelle übertragen werden. Welcher Übertragungsweg dabei gewählt wird, ist in diesem Rahmen uninteressant. Dann wird es wie in Abschnitt 4.1 beschrieben in den Flash geschrieben. Von hier an sind beide Verfahren wieder identisch. Da das Plug-In ohne Startcode compiliert wurde, steht an der ersten Adresse die Initialisierungsfunktion. Diese benötigt als Parameter den Zeiger auf die Funktionszeigerliste. Dort schreibt die Verwaltung zuvor die Zeiger zu den bereitgestellten Funktionen hinein. In der Initialisierungsfunktion schreibt das Plug-In dann die Zeiger zu den Funktionen, die es selbst implementiert. Dadurch ist von diesem Zeitpunkt die Kommunikation zwischen Verwaltung und Plug-In möglich, da beide die Funktionen des Anderen aufrufen so wie den gemeinsamen Speicher nutzen können. Der entsprechende Code dazu ist in Listing 2 zu sehen. Dies ist die Funktion *add_plugin()* der Verwaltung.

4.8 Konfiguration

Je nach Plug-In kann es sein, dass für die korrekte Ausführung noch zusätzliche Parameter nötig sind. Bei einem Wavelet Plug-In sind dies zum Beispiel die Filter, welche verwendet werden. Diese Konfiguration kann zum einen direkt in der Header-Datei erfolgen, was gerade bei Konfigurationen sinnvoll ist, welche schon beim Compilieren feststehen. Zum anderen kann die Konfiguration über Variablen geschehen, welche zur

```
1  void add_plugin(uint16_t size, uint32_t *plugin)
2  {
3          static uint32_t next_address = PLUGIN_POS_1;
4          HAL_FLASH_Unlock();
5
6          int i;
7          for(i=0;i<size;i++)
8          {
9                  if(HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD, (
                       next_address + i), *(plugin+(i))))
10                 {
11                 print_char("FLASH write Error");
12                 Error_Handler();
13                 break;
14                 }
15         }
16         HAL_FLASH_Lock();
17
18         //plugin is in flash
19         //configure for use
20
21         //define the function pointer of main functions
22         plugins[plugins_nr].plugin_init = ((uint32_t (*)())(next_address
               + 1));
23         plugins[plugins_nr].get_last_value = get_last_value;
24         plugins[plugins_nr].get_values = get_values;
25         plugins[plugins_nr].get_values_size = get_values_size;
26
27         //run plugin_init
28         //this will set the function pointers of the plugin
29         plugins[plugins_nr].plugin_init();
30         plugins_nr++;
31
32 }
```

Listing 2: Installation eines Plugins wie in der Verwaltung implementiert

Laufzeit gesetzt werden. Dies erlaubt deutlich mehr Flexibilität und ist gerade bei Plug-Ins Sinnvoll, die erst nachträglich installiert werden.

5 Fast Wavelet Transformation als Plug-in

Als Plug-In für ein solches System lässt sich die Fast Wavelet Transformation sehr gut umsetzen, da sie aus mehreren eigenständigen Teilen besteht. So können die Analyse, die Verarbeitung der Koeffizienten und die Synthese als eigenständige Plug-Ins implementiert werden. Durch diese Modularisierung kann dann sehr einfach zum Beispiel nur die Verarbeitung der Koeffizienten angepasst werden, ohne direkt alle anderen Teile anpassen zu müssen. Alternativ kann beispielsweise die Synthese auch weggelassen werden. Dies ermöglicht dann das Fortfahren mit den Wavelet-Koeffizienten und somit die weitere Arbeit im Frequenzspektrum.
Die Wavelet-Transformation eignet sich außerdem ganz besonders für die Aufteilung in mehrere Plug-Ins, da diese auf einem festen Speicherbereich arbeitet und keinen zusätzlichen Speicher zur Rückgabe der Ergebnisse benötigt. So wird nur ein einziger Speicherbereich verwendet, der nacheinander von den Plug-Ins verändert wird.
Die Verwendung der DWT auf einem Mikrocontroller kann hilfreich sein, um Daten bereits vor dem Versenden zu komprimieren oder sogar zu analysieren.
In der Implementierung müssen alle Punkte aus dem zuvor in Kapitel 4 beschriebenen Verfahren beachtet werden. Insbesondere die Verwendung der API ist wichtig, um mit der Plug-In-Verwaltung kommunizieren zu können. Dazu muss auch hier die Header-Datei *Plugin_def.h* mit eingebunden werden.
Der Filter, welcher auf die Wavelet-Koeffizienten angewendet wird, ist in diesem Proof of Concept ein einfacher Threshold. Das heißt alle Werte unterhalb eines vorher definierten Thresholds werden auf Null gesetzt. Dies bewirkt dann ein einfaches Entrauschen des Eingangssignals.

6 Alternativen

Plug-Ins sind nicht die einzige Möglichkeit, nachträglich Software auf einen Mikrocontroller zu bringen. Die beiden Hauptalternativen sind zum einen Scriptsprachen und zum anderen ganze Betriebssysteme, in denen Plug-Ins installiert werden.

6.1 Scriptsprachen

Scriptsprachen wie zum Beispiel MicroPython [Geo] werden nicht direkt in Prozessoranweisungen übersetzt, sondern bleiben in der Form von Quellcode. Dieser wird dann zur Laufzeit von einem Interpreter ausgeführt. Dadurch ergibt sich eine deutlich größere Flexibilität. Für eine neue Hardware muss nur der Interpreter angepasst werden. Alle Programme können von nun an auf der neuen Hardware ausgeführt werden. Bei einer Sprache, welche compiliert wird, müsste hingegen jedes Programm erneut übersetzt werden, damit es auch der neuen Hardware funktioniert. Soll eine Scriptsprache auf einem Microcontroller verwendet werden, so wird neben der Grundfunktionalität noch ein Interpreter für die entsprechende Sprache hinzugefügt. Dieser interpretiert den Code der Plug-Ins dann zur Laufzeit. Diese Lösung ist sehr flexibel, da über den Interpreter die gesamte Mächtigkeit der Sprache zur Verfügung steht. Da der Code erst zur Laufzeit ausgewertet und interpretiert wird, ist das Programm allerdings deutlich langsamer im Vergleich mit einer Implementierung direkt in einer nativen Programmiersprache (wie z.B. C).

6.2 Betriebssysteme

Ein Betriebssystem, wie beispielsweise FreeRTOS [Bar], fügt eine Abstraktionsschicht zwischen der Hardware und der Software ein. Dabei implementiert es vor allem eine Speicherverwaltung und einen Scheduler. Die Speicherverwaltung achtet darauf, dass verschiedene Programme nur auf den jeweils ihnen zugewiesenen Speicher zugreifen können. Dies schützt sowohl vor Seiteneffekten durch andere Programme als auch vor böswilligen Angriffen durch Dritte.
Der Scheduler verwaltet die Ausführungsdauer der Programme auf dem Prozessor. Hier wird entschieden, welches Programm wann den Prozessor nutzen kann. Hält sich ein Programm nicht an diese Vorgabe, so kann es durch das Betriebssystem von außen beendet werden. Dadurch ist sichergestellt, dass innerhalb einer bestimmten Zeit alle Programme einmal ausgeführt werden. Andernfalls könnte ein blockiertes Programm das gesamte System zum Halten bringen.
Auch stellt das Betriebssystem oftmals einige Funktionen bereit, um diese Strukturen zu nutzen oder um mit anderen Programmen zu kommunizieren. Zusätzliche Program-

me können also einfach darauf aufbauen, nicht direkt mit der Hardware kommunizieren zu müssen. Das nachträgliche Installieren von Software ist also mit einer der Hauptgründe für die Nutzung eines Betriebssystems. Da Betriebssysteme in der Regel recht universell eingesetzt werden, beinhalten sie viele Funktionen, die im Anwendungsfall nicht genutzt werden. So ist zum Beispiel ein Scheduler auf einem System mit nur einem zusätzlichen Programm noch recht nutzlos. Dennoch werden für diese nicht genutzten Funktionalitäten Ressourcen benötigt, wodurch die effektive Leistung des Systems sinkt.

7 Evaluation

Wie in den vorherigen Kapiteln dieser Arbeit beschrieben, ist ein Plug-In System auf einem Mikrocontroller einsetzbar. Im Folgenden werden einzelne Aspekte dieser Lösung analysiert, insbesondere mit Hinblick auf die Sicherheit des Systems.
Ein Plug-In nachträglich in den Flash zu schreiben, umgeht die zusätzliche Sicherheit der Harvard-Architektur. Da der Programmcode des Plug-Ins von der Verwaltung nicht geprüft werden kann, besteht hier die Möglichkeit falschen Code auszuführen, also fehlerhafte oder sogar böswillige Plug-Ins. In der hier vorgestellten Implementierung können Plug-Ins nicht von der Verwaltung authentifiziert oder überprüft werden. Somit kann jeglicher Code als Plug-In von Dritten installiert werden. Dies ist nur möglich, wenn Plug-Ins zur Laufzeit nachgeladen werden können. Ist es hingegen lediglich möglich, Plug-Ins während des Flash-Vorganges zu installieren, wie in Abschnitt 4.7 beschrieben, besteht diese Gefahr durch nicht authentifizierte Software ohnehin nicht. Das nachträgliche installieren von Plug-Ins zu verbieten, kann also auch eine Entscheidung der Sicherheit sein. Die bessere Performance der Harvard-Architektur wird hingegen nicht beeinflusst und wirkt daher beschleunigend. So können auch bei einem Plug-In System Daten gleichzeitig aus dem Programm- und dem Datenspeicher gelesen werden.
Das Verwenden von Funktionszeigern sollte in der Regel bei der Programmierung in C vermieden werden, um Fehlern vorzubeugen, da der Programmfluss grundlegend beeinflusst werden kann. Besonders gefährlich ist das direkte Verändern der Zeiger durch arithmetische Operationen. Die Gefahr hierbei ist, Daten auszuführen, welche eigentlich kein Code sind, wie zum Beispiel der Inhalt von Variablen. Hierbei kann es zu unvorhersehbarem Verhalten und sogar zu Abstürzen kommen.
Um Funktionszeiger sicher einsetzen zu können, ist es wichtig, die API strikt umzusetzen. Diese beschränkt die Funktionszeiger auf ein nötiges Minimum. Darüber hinaus stellt sie die korrekte Kommunikation zwischen Verwaltung und Plug-In sicher. Sowohl die Verwaltung, als auch jedes Plug-In muss daher die Header Datei *Plugin_def()* beinhalten.
Bei der Installation eines Plug-Ins registriert sich das Plug-In bei der Verwaltung. Hierbei wird eine Liste mit Funktionszeigern erstellt in der alle verwendbaren Funktionen der Verwaltung und des Plug-Ins aufgelistet sind. So wird die Kommunikation zwischen Plug-In und Verwaltung ermöglicht.
Plug-Ins zu verwenden ist besonders sinnvoll, wenn einzelne Funktionen nachgerüstet werden sollen. Der Nutzer kann so entscheiden, welche Teile er wirklich braucht ohne das System mit unnötigen Plug-Ins zu belasten.
Als Plug-Ins eignen sich modularisierbare Aufgaben besonders gut. Diese können in mehrere Plug-Ins aufgeteilt werden. So kann eine große Zahl von Funktionen mit einer recht überschaubaren Grundlage an Plug-Ins realisiert werden.

Eine Besonderheit bei der Erstellung von Plug-Ins ist das Compilieren, da ein Plug-In nicht wie sonst als reguläres Programm für den Mikrocontroller übersetzt wird, sondern ohne einen Startcode umgesetzt werden muss, wie in Abschnitt 4.6 beschrieben.
Im Vergleich zu anderen flexiblen Lösungen für Mikrocontroller haben Plug-In Systeme deutliche Performance Vorteile. Sie schneiden besser ab, sowohl in der Geschwindigkeit als auch beim insgesamt benötigten Speicher. Das liegt auch daran, dass die hier vorgestellt Plug-In-Verwaltung fast selbst ein Betriebssystem ist, allerdings mit den minimalst möglichen Anforderungen. Es implementiert eine grobe Abstraktion der Hardware so wie eine grundlegende Speicherverwaltung durch die bereitgestellten Funktionen. Außerdem entscheidet die Verwaltung, wann welches Plug-In ausgeführt wird, was das Grundprinzip eines Schedulers ist. Diese Funktionen sind allerdings so klein gehalten, dass dennoch ein deutlicher Performancevorteil im Vergleich zu einem vollwertigen Betriebssystem besteht. Dafür müssen allerdings einige Nachteile bei der Sicherheit in kauf genommen werden, wie in Abschnitt 6 beschrieben. Hier ist also die Entscheidung zwischen Leistung und Sicherheit des Systems zu treffen.

8 Ausblick

Da diese Arbeit ein Proof of concept darstellt, sind in der Zukunft noch viele Weiterentwicklungen denkbar und auch sinnvoll. So ist die hier vorgestellte API mit drei Funktionen des Plug-Ins und zwei Funktionen der Verwaltung noch recht klein und auf das Beispiel angepasst. Gerade bei größeren Projekten ist es sehr wichtig, eine breitere Schnittstelle zu entwickeln, um mehr Möglichkeiten für Plug-Ins zu schaffen. Das Installieren von Plug-Ins zur Laufzeit wurde hier nur allgemein beschrieben. Dies ist jedoch vermutlich einer der Hauptanwendungsfälle. Daher wäre es sehr interessant ein Plug-In zum Beispiel per TCP/IP nachzuladen und auszuführen. Darüber hinaus wäre für die Zukunft eine alternative Implementierung in einer Scriptsprache und mit einem Betriebssystem wünschenswert. Diese könnten dann mittels Benchmarks mit der Plug-In Lösung verglichen werden. Hiermit wäre eine besser Einschätzung der Leistungsunterschiede möglich. Insgesamt hat sich gezeigt, dass Plug-Ins auf Mikrocontrollern durchaus sinnvoll sein können, gerade im Hinblick auf die immer größere Leistung von eingebetteten Systemen und einer immer mehr geforderten Flexibilität der Software und deren Entwicklung.

9 Fazit

Diese Arbeit zeigt, dass Plug-Ins auf Mikrocontrollern durchaus möglich sind und Potenzial haben, gerade in Zeiten der immer größeren Verbreitung von Mikrocontrollern und der immer kürzeren Entwicklungszyklen von Software.
Bei der Entwicklung eines solchen Systems sind aber teilweise deutliche Unterschiede zu Plug-In Systemen auf herkömmlichen Systemen zu beachten. So müssen auf einem Mikrocontroller Funktionszeigerlisten von Hand angelegt werden 4.3 und bestimmte Sicherheitsfunktionen der Harvard-Architektur umgangen werden 4.1. Besonders wichtig ist das korrekte Definieren einer API im Vorhinein, um eine sichere Kommunikation zwischen Plug-In und Verwaltung zu gewährleisten4.5.
Als Plug-In implementierte Funktionen sind deutlich performanter als die in Kapitel 6 vorgestellten Alternativen. Ein Plug-In System in der hier entwickelten Form bringt allerdings einige Sicherheitsrisiken mit, die das Gerät angreifbar machen können 7. Daher muss bei jedem Anwendungsfall entschieden werden, ob ein Plug-In System die richtige Lösung ist. Dabei ist auch die Entscheidung wichtig, ob Plug-Ins während der Laufzeit nachgeladen werden können. Dürfen Plug-Ins nur beim erstmaligen Flashen installiert werden, ist das System deutlich sicherer und erlaubt immer noch eine einfache Konfiguration 4.7.
Die Entwicklung für den STM-Mikrocontroller war zunächst recht kompliziert, da dieser zum Zeitpunkt der Entwicklung noch recht neu war und die dazu gehörige Dokumentation zunächst entsprechend überschaubar bzw. unvollständig. Bei einem solchen Projekt wäre es sinnvoller gewesen, auf ein etabliertes Board zu setzen.
Da diese Arbeit ein Proof of concept darstellt, besteht noch großes Potenzial für die Weiterentwicklung des Systemes. Gerade die Entwicklung eines universell einsetzbaren Plug-In Systems dürfte sehr nützlich sein 8.

Literaturverzeichnis

[AC6] AC6: *OpenSTM32 Entwicklungsumgebung*. online. http://www.openstm32.org/System+Workbench+for+STM32. – Aufgerufen: 24.08.2016

[Bar] Barry, Richard: *FreeRTOS*. online. http://www.freertos.org/. – Aufgerufen: 17.08.2016

[Ber15] *Kapitel* Grundlagen der Mikrocontroller. In: Bernstein, Herbert: *Mikrocontroller*. Wiesbaden : Springer Fachmedien Wiesbaden, 2015. – ISBN 978–3–658–02813–8, 1-6

[Com05] Commons, Wikimedia: *Discrete wavelet transform - a 3 level filter bank*. https://upload.wikimedia.org/wikipedia/commons/2/22/Wavelets_-_Filter_Bank.png. Version: 2005. – Aufgerufen: 25.08.2016

[Com13] Commons, Wikimedia: *Difference between short-time Fourier transformation and wavelet-transformation*. https://upload.wikimedia.org/wikipedia/commons/c/c4/STFT_and_WT.jpg. Version: 2013. – Aufgerufen: 25.08.2016

[Geo] George, Damien: *MicroPython*. online. https://micropython.org/. – Aufgerufen: 17.08.2016

[Mal09] Mallat, Stéphan: *a wavelet tour of signal processing. The Sparse Way*. third. Elsevier, 2009

[STM] STMicroelectronics: *stm327x6g-eval*. http://www.st.com/content/ccc/fragment/product_related/rpn_information/board_photo/a7/9a/e5/9f/57/95/45/ef/stm327x6g-eval.jpg/files/stm327x6g-eval.jpg/_jcr_content/translations/en.stm327x6g-eval.jpg. – Aufgerufen: 25.08.2016

[STM16] STMicroelectronics ; STMicroelectronics (Hrsg.): *UM1902 STM32746G-EVAL Evaluation board with STM32F746NG MCU*. Rev 2. Genf, Schweiz: STMicroelectronics, März 2016. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/DM00188496.pdf. – Aufgerufen: 10.03.2016

Anhang

```
/*
* plugin_def.h
*
*  Created on: 19.01.2016
*      Author: Hann Holze
*/

#ifndef PLUGIN_DEF_H_
#define PLUGIN_DEF_H_

#include <stdint.h>

typedef struct {
        //Functions in plug-in
        uint32_t (*plugin_init)();
        uint32_t (*execute)();
        uint32_t (*stop)();
        uint32_t *result;
        uint32_t size_result;

        //Functions in main
        uint32_t* (*get_values)();
        uint32_t (*get_last_value)();
        uint32_t (*get_values_size)();

}plugin;


 #endif /* PLUGIN_DEF_H_ */
```

Listing 3: Die API wie in plugin_def.h

```
/*
 *  Created on: 19.11.2015
 *      Author: Hann Holze
 *
 *  This Plugin implements a wavelet transformatin (and retransformation)
 *  It uses a filter bank cascade
 *  The filter between the analysis and the synthesis is a simple
    threshold filter
 */
#include <stdint.h>
#include "plugin.h"

#define CASCADE_DEPTH 4
#define SIZE_FILTER 5
#define SIZE_VALUES_MAX 30
```

```
15 #define THRESHOLD 3
16
17 plugin *config;
18
19 uint32_t filter_bank();
20 void fir_down(uint32_t[], uint32_t[], uint32_t[], uint32_t, uint32_t);
21 void up_fir(uint32_t[], uint32_t[], uint32_t[], uint32_t, uint32_t);
22
23 uint32_t tmp_1[SIZE_VALUES_MAX];
24 uint32_t tmp_2[SIZE_VALUES_MAX];
25 uint32_t result[SIZE_VALUES_MAX];
26
27 uint32_t plugin_init(plugin *conf)
28 {
29   config = conf;
30   conf->execute = filter_bank;
31   conf->stop = stop;
32   conf->result = result;
33   conf->size_result = SIZE_VALUES_MAX;
34   return 0;
35 }
36
37 /*
38  * If a plugin has allocated memory and needs to free it or needs to
        perform
39  * a last task before it is disabled, this can be done here
40  */
41 uint32_t stop()
42 {
43   return 0;
44 }
45
46 /*
47  * Filter bank cascade algorithm
48  * The result is saved in the original storage
49  */
50 uint32_t filter_bank()
51 {
52   uint32_t size = config->get_values_size();
53   if((size > SIZE_VALUES_MAX) || //Variables will be to small
54       (size <= 0) || //negative size would make no sense
55       !((size & (~size+1))==size)) //size mus be a power of two (2^k) for
            easier calculation
56   {
57     return 1;
58   }
59   uint32_t *storage = config->get_values();
60
```

```
61
62    uint32_t i;
63    //Analysis filter
64    uint32_t fhp[SIZE_FILTER] = {0};
65    uint32_t flp[SIZE_FILTER] = {0};
66    //Synthesis filter
67    uint32_t ghp[SIZE_FILTER] = {0};
68    uint32_t glp[SIZE_FILTER] = {0};
69    //In this PoC the filters are left blank but could be filled with any
          wavelet filter
70
71    //Analysis filter bank:
72    for(i=0; i<size;i++)
73    {
74      tmp_1[i]=storage[i];
75    }
76
77    for(i=0; i<CASCADE_DEPTH; i++)
78    {
79      fir_down(tmp_1, fhp, flp, (size/(2^i)), SIZE_FILTER);
80    }
81
82    //Wavelet coefficients are ready
83
84    apply_thershold(tmp_1, THRESHOLD, size);
85
86    //Synthesis filter bank:
87    for(i=CASCADE_DEPTH-1; i>0; i--)
88    {
89      up_fir(tmp_1, ghp, glp, (size/(2^i)), SIZE_FILTER);
90    }
91
92    for(i=0;i<size;i++)
93    {
94      result[i]=tmp_1[i];
95    }
96    config->size_result = size;
97    return 0;
98 }
99
100 /*
101  * Implements a FIR filter on a ring buffer
102  */
103 void fir(uint32_t values[], uint32_t filter[], uint32_t size_in, uint32_t
        size_filter)
104 {
105   uint32_t i,j;
106
```

```
107   for(i=0;i<(size_in);i++)
108   {
109     tmp_1[i]=0;
110     for(j=0;j<size_filter;j++)
111     {
112       tmp_1[(size_filter+i)%size_in] += values[(i+j)%size_in] * filter[j
              ];
113     }
114   }
115
116   for(i=0;i<size_in; i++)
117   {
118     values[i]=tmp_1[i];
119   }
120 }
121
122 /*
123  * call the FIR filter and downsample the result
124  */
125 void fir_down(uint32_t values[], uint32_t hp[], uint32_t lp[], uint32_t
      size_in, uint32_t size_filter)
126 {
127   uint32_t i;
128
129   for(i=0;i<size_in;i++)
130   {
131     tmp_1[i]=values[i];
132     tmp_2[i]=values[i];
133   }
134
135   //Apply FIR
136   fir(tmp_1, lp, size_in, size_filter);
137   fir(tmp_2, hp, size_in, size_filter);
138
139   //Downsample
140   for(i=0;i<size_in/2;i++)
141   {
142     values[i]=tmp_1[i*2];
143     values[i+(size_in/2)]=tmp_2[i*2];
144   }
145 }
146
147 /*
148  * Upsample and call the FIR filter
149  */
150
151 void up_fir(uint32_t values[], uint32_t hp[], uint32_t lp[], uint32_t
      size_in, uint32_t size_filter)
```

```
152 {
153   uint32_t i;
154
155   //Upsample
156   for(i=0; i<size_in/2;i++)
157   {
158     tmp_1[i*2] = values[i];
159     tmp_2[i*2] = values[(size_in/2) + i];
160   }
161
162   //Apply FIR
163   fir(tmp_1, lp, size_in, size_filter);
164   fir(tmp_2, hp, size_in, size_filter);
165
166   //sum the results
167   for(i=0;i<size_in;i++)
168   {
169     values[i] = tmp_2[i] + tmp_1[i];
170   }
171 }
172
173 void apply_threshold(uint32_t values[], uint32_t threshold, uint32_t
      size_in)
174 {
175   for(i=0; i<size_in; i++)
176   {
177     if(values[i] < threshold)
178     {
179       values[i] = 0;
180     }
181   }
182 }
```

Listing 4: Das entwickelte Wavelet Plug-In

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Bachelorarbeit

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

$Unterschrift:$ $Ort, Datum:$