Praktikumsbericht

für das Industriepraktikum

vom 2. April bis zum 29. Juni 2007

bei

Siemens Medical Solutions CO CHS CD

Erlangen

Praktikant: Michael Löffler
Betreuer: Volker Pritsching

(Unterschrift Praktikant)

(Unterschrift Betreuer)

Inhaltsverzeichnis

1. Die Entwicklungswerkzeugkette

Ein Hauptbestandteil meiner Arbeit bei Siemens Med bestand in der Programmierung von Mikrocontrollersystemen. Hierfür benötigt man eine ganze Reihe verschiedener Software, die man auf Englisch gewöhnlich „toolchain“ bzw. Werkzeugkette nennt. Dazu gehören ein auf Programmierung getrimmter Editor, ein Compiler, ein Linker, ein Programmer und häufig ein Debugger. Diese Bestandteile werden häufig unter dem Dach einer IDE, einer „Integrated Development Environment“, zusammengefasst.

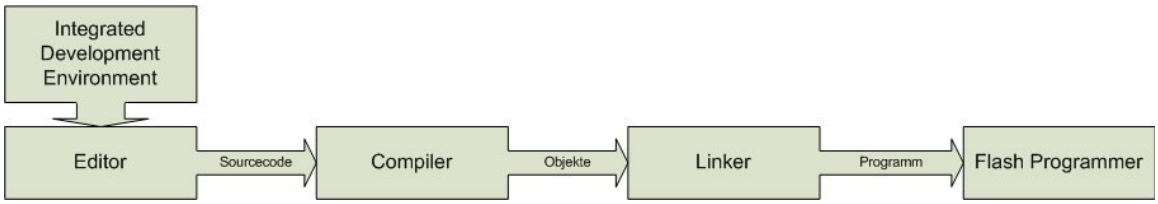

Abbildung 1: Typische Toolchain

Für den von mir verwendeten Mikrocontroller ARM STR750 gibt es z.B. die kommerzielle IDE µVision3 der Firma Keil, die alle Bestandteile enthält, und in unserer Abteilung rege verwendet wurde. Aufgrund der teuren Einzelplatzlizenzen bestand allerdings der Wunsch, Alternativen auf OpenSource-Basis zu suchen.

Editor
Als Editor und IDE kam dabei das bekannte Eclipse Framework zum Einsatz. Es ist intuitiv zu bedienen und bietet alle Funktionen, die man von einem Editor für Programmierzwecke erwarten kann. Zusätzlich installiert wurde das Plugin CDT für die C und C++ Entwicklung und dazu eine kleine Erweiterung, die das Cross-Development für ARM Chips erleichtert. Besonders arbeitserleichternd sind Funktionen wie die automatische makefile-Erzeugung, die gut gelungene Navigation in langen Quellcodedateien mit Zurück-Funktion, „Go to Definition“, „Go to Declaration“, die Markierung von Suchtreffern neben der Scrollleiste und die nahtlose Integration in CVS/SVN-Systeme,

Compiler
Für die ARM7TDMI Familie gibt es eine ganze Reihe proprietärer Compiler, aber auch vom GNU C Compiler (gcc) existiert dafür eine Crosscompiler Version. Um diesen unter Windows zum Laufen zu bekommen, benötigt man eine portierte Version der GNU toolchain, die neben dem Crosscompiler selbst auch noch Hilfsprogramme wie z.B. „make“ und eine für Mikrocontroller angepasste Version der libc (newlib) enthält.

Linker
Der Linker baut aus den einzelnen Objekten und den Bibliotheken ein lauffähiges Programm zusammen. Auch er muss an die Zielarchitektur angepasst werden,

um verschiedenen Stackgrößen, Speicheraddressen und Interruptvektoren gerecht zu werden. Er ist zusammen mit dem arm-gcc in einer auf Windows portierten Version in einem Paket namens „WinARM“ erhältlich. Lediglich ein passendes Linkerfile muss noch passend für den jeweiligen Chip organisiert werden.

Programmer
Hier war eigentlich die Verwendung eines Eigenbau-JTAGs am Parallelport angedacht, der von OpenOCD unterstützt wird, angedacht. Letztendlich blieb es aber beim proprietären Keil Programmer, der im Batch-Modus mit einer kostenlosen Evalutationsversion integriert in Eclipse betrieben wird.

Eclipse
Unter Eclipse ließen sich alle diese tools gut einbinden und zu einer produktiven Programmierumgebung vereinen. Der durchschnittliche Arbeitsdesktop sah dann etwa folgendermaßen aus:

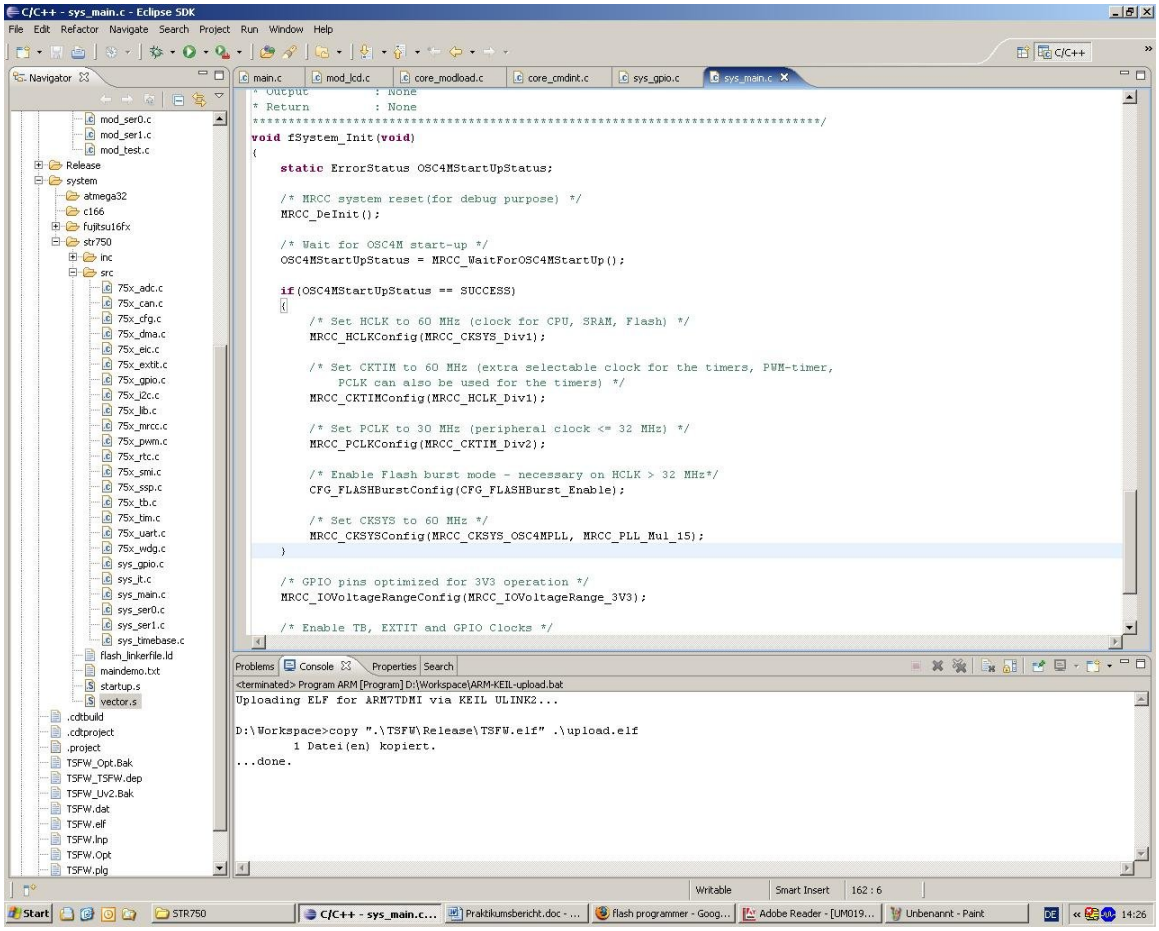

Abbildung 2: Eclipse Screenshot

2. Das Java TestApplet

Das TestApplet entstand während meiner vorherigen Werksstudentenzeit. Ziel war es, bestehende Hardware mit serieller Schnittstelle um einen Ethernetport zu erweitern. Um an der bestehenden Hardware möglichst wenig zu ändern, kam der XPort von Lantronics zum Einsatz. Er stellt auf der Größe einer herkömmlichen Ethernetbuchse bereits eine vollständig integrierte Lösung eines Wandlers von Ethernet nach seriell und zudem einen eigenen kleinen Webserver bereit. Das Applet kann nun entweder über das Netzwerk von diesem Server aus oder lokal gestartet werden. Befehle können von Hand eingegeben oder aus einer Liste aufgerufen werden. Zur automatisierten Durchführung von Testläufen kann es diese Listen selbstständig abarbeiten, die Antworten der Hardware auswerten und dabei Timeouts und Delays beachten. Zur lokalen Installation bringt es auch einen eigenen Installer mit.

Während meines Praktikums kam das Applet regelmäßig im lokalen, seriellen Betrieb zum Einsatz. Zudem waren einige kleine Erweiterungen nötig, um veränderten Anforderungen gerecht zu werden, z.B. die Wählbarkeit des Zeichens für den Zeilenumbruch oder der Wählbarkeit der Übertragungsrate.

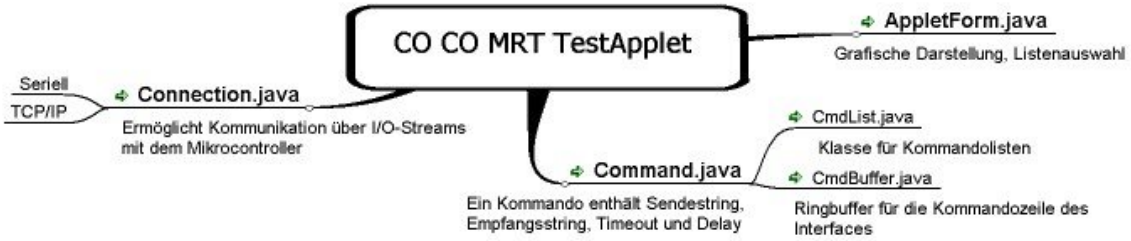

Abbildung 3: Programmstruktur des Applets

Befehlssyntax in den Listen

Befehlslisten werden als blanke Textdatei im Unterordner cmdlists des Installationsverzeichnisses erwartet. Zeilen beginnend mit # oder // sind Kommentare. Die Zeichen '\n' und '\r' werden escaped. Leerzeichen um | werden ignoriert. Alles andere muss der folgenden Syntax folgen:

Befehl [|[Antwort] [[|Verzögerung] |Timeout]]]

Parameter

- *Befehl*: Der Befehl, der an das Gerät geschickt wird.
- *Antwort*: Die vom Gerät erwartete Antwort, um mit der Listenverarbeitung fortzufahren. Kann leer sein.
- *Verzögerung*: Wartet die angegebene Zahl in ms, bevor mit dem nächsten Befehl fortgefahren wird.

- *Timeout*: Die maximale Antwortzeit für das Gerät in ms, bevor die Ausführung der Liste abgebrochen wird. Default: 100ms

Beispiele

- *idn?
- *idn? | *idn?\rOIS-FW 8867991-EFD-01S-VA01A
- *idn? | *idn?\rOIS-FW 8867991-EFD-01S-VA01A | 200
- :Set:Mode 0 0 || 2500 | 100

Bedienbarkeit

Gute Bedienbarkeit stand bei der Implementierung des Applets im Vordergrund. Wichtig war mir dabei insbesondere die Vereinfachung häufig wiederholter Vorgänge, wie das Senden eines Befehls mit nur leicht veränderten Parametern. Im Applet kann man Befehle in der Liste einfach durch Doppelklick oder mit Cursortasten und Entertaste senden. Ein einfacher Klick gefolgt von Tastaturinput führt direkt ans Ende der Kommandozeile zu den Parametern und ermöglicht deren Bearbeitung ohne weitere Mausklicks. Zudem werden alle gesendeten Befehle in einem Buffer zur Wiederverwendung gespeichert, äquivalent zur bash-history. Auch im nicht automatisierten Betrieb ist somit eine effiziente Steuerung der Zielhardware möglich.

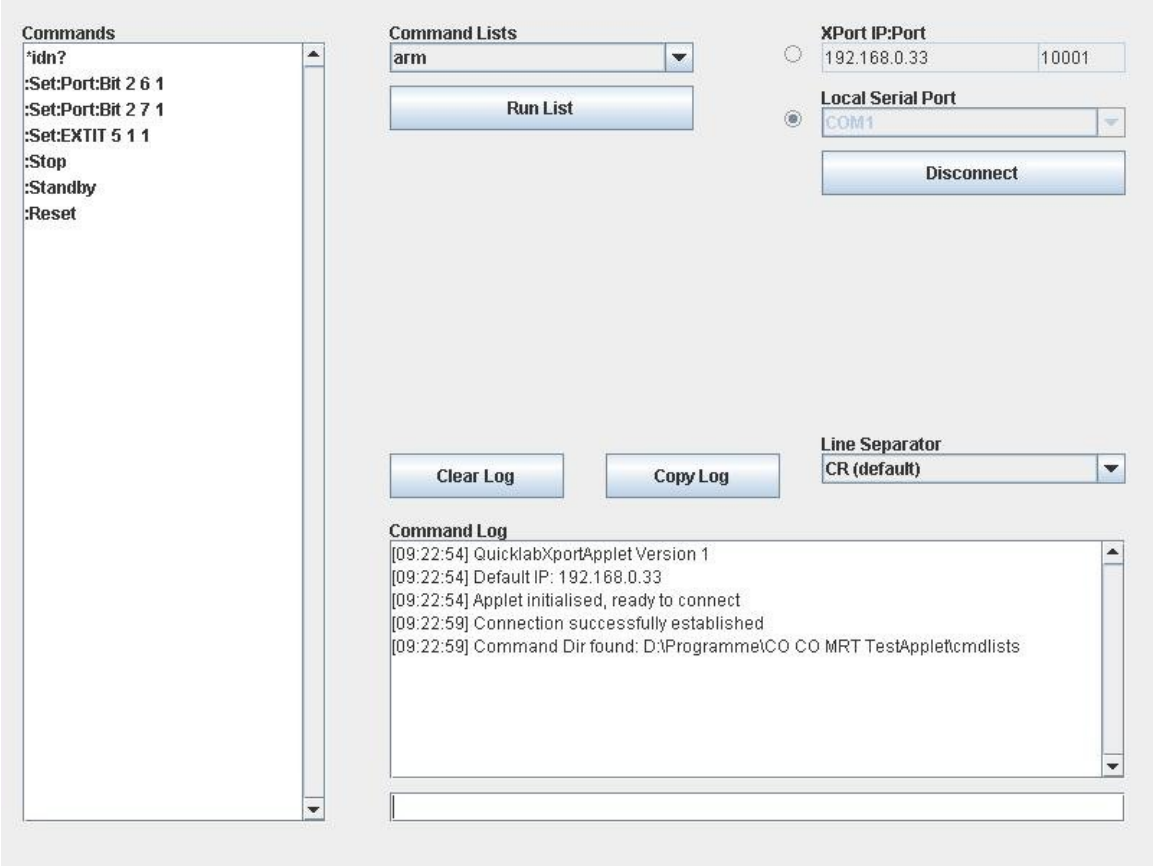

Abbildung 4: Screenshot des Applets

3. Standbymechanismen eines ARM STR750

Für den Antrieb einer Liege in einem Magnetresonanztomographen sollten mehrere Controller evaluiert werden. Während der Aufnahme des Tomographen sollte der Controller dabei in einen Standbymodus heruntergefahren werden, um die Messung des Tomographen nicht zu stören. Danach sollte der Chip durch einen Impuls über den CAN-Bus wieder geweckt werden. Zur Auswahl standen ein Altera NIOS Chip, ein Fujitsu 16FX und ein ARM STR750. Für den STR750 erstellte ich dabei Softwaremodule, die den Chip gezielt in verschiedene Tiefschlafmodi versetzte.

Low Power Modi
Der STR750 beherrscht mehrere Stromsparmodi für verschiedene Anwendungsfälle.

- Slow Modus: Reduzierte Taktrate
- PCG Modus: Peripheriemodule können separat deaktiviert werden (peripheral clock gating)
- WFI Modus: Der Kern hält an, aber die Peripherie wird weiter getaktet. Der Chip erwacht, sobald ein Interrupt ausgelöst wird (wait for interrupt)

- Stop Modus: Sämtliche Clocks können deaktiviert werden. Lediglich ein Impuls an einer externen Interruptleitung oder falls konfiguriert die interne Echtzeituhr kann den Chip wieder aufwecken. Daten im Speicher bleiben erhalten.
- Standby Modus: Nur noch ein einziges Pin und auf Wunsch die Echtzeituhr werden beschaltet. Alle Speicherzustände gehen jedoch verloren und der Resetzustand wird beim Aufwachen hergestellt.

Für den gegebenen Anwendungsfall sind insbesondere der Stop und der Standbymodus interessant, da nicht die Stromeinsparung, sondern die Reduzierung der Abstrahlung im Vordergrund steht. Beide Modi wurden evaluiert und sollen hier näher erläutert werden.

Stop Modus
Der Stop Modus bietet ein gewisses Spektrum an Konfigurationsmöglichkeiten. Die Echtzeituhr mit dem 32k-Quarz, der 4 MHz Oszillator und der Flashspeicher können wahlweise separat mit Strom versorgt bleiben. So konnten im Test sämtliche Speicherzustände trotz Deaktivierung aller HF-Quellen erhalten werden. Der Stop Modus wird durch folgende Prozedur initiiert:

- Externe Interrupts so konfigurieren, dass das Aufwachen möglich wird, also einen externen Interrupt auf die CAN-RX-Leitung legen
- Stop Modus durch setzen des LPMC Bits im Register MRCC_PWRCTRL wählen
- Im gleichen Register LP_PARAM Bits zum passenden Deaktivieren der Clocks und des Flash-Speichers setzen
- Low Power Bit Schreibsequenz durchführen (siehe unten)

Standby Modus
Wie bereits ersichtlich, sollte der Standby Modus keine weitere Verbesserung der Abstrahlung bringen, da sich schon im Stop Modus alle clocks deaktivieren lassen. Sicherheitshalber wurde aber auch dieser mitevaluiert und später in der HF-Kammer vermessen. Der Standby Modus bietet außer der Deaktivierung der Echtzeituhr keine weiteren Konfigurationsmöglichkeiten. Die Initiierung verläuft ähnlich zum Stop Modus. Der Stromverbrauch sinkt dabei auf beachtliche 20µA bei 3,3V, was dem Leckstrom und dem statischen Verbrauch des internen Spannungswandlers entspricht.

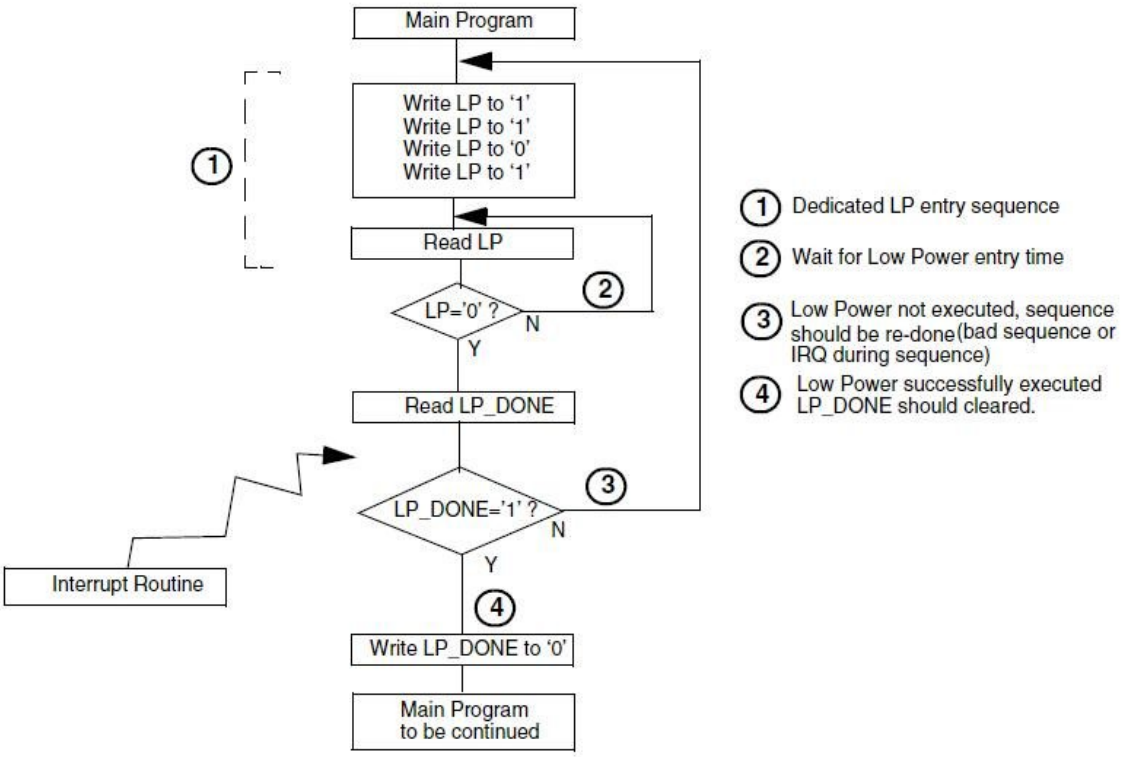

Abbildung 5: Ein- und Austritt in einen Low Power Modus

4. EMV Messungen

Nachdem die Mikrocontroller per Mausklick in verschiedene Schlafmodi versetzt und wieder aufgeweckt werden konnte, sollte verifiziert werden, dass die Abstrahlung den strengen Kriterien bei der Magnetresonanzaufnahme genügte.

Frequenzanalyze des Störsignals
Für eine grobe Überprüfung wurde zunächst in einem kleinen Versuchsaufbau das an die Netzspannung abgegebene Störsignal ausgekoppelt und im Frequenzanalyzer betrachtet. Die schwarze Kurve bildet dabei den eingeschalteten Zustand und die blaue den Standby Modus ab.

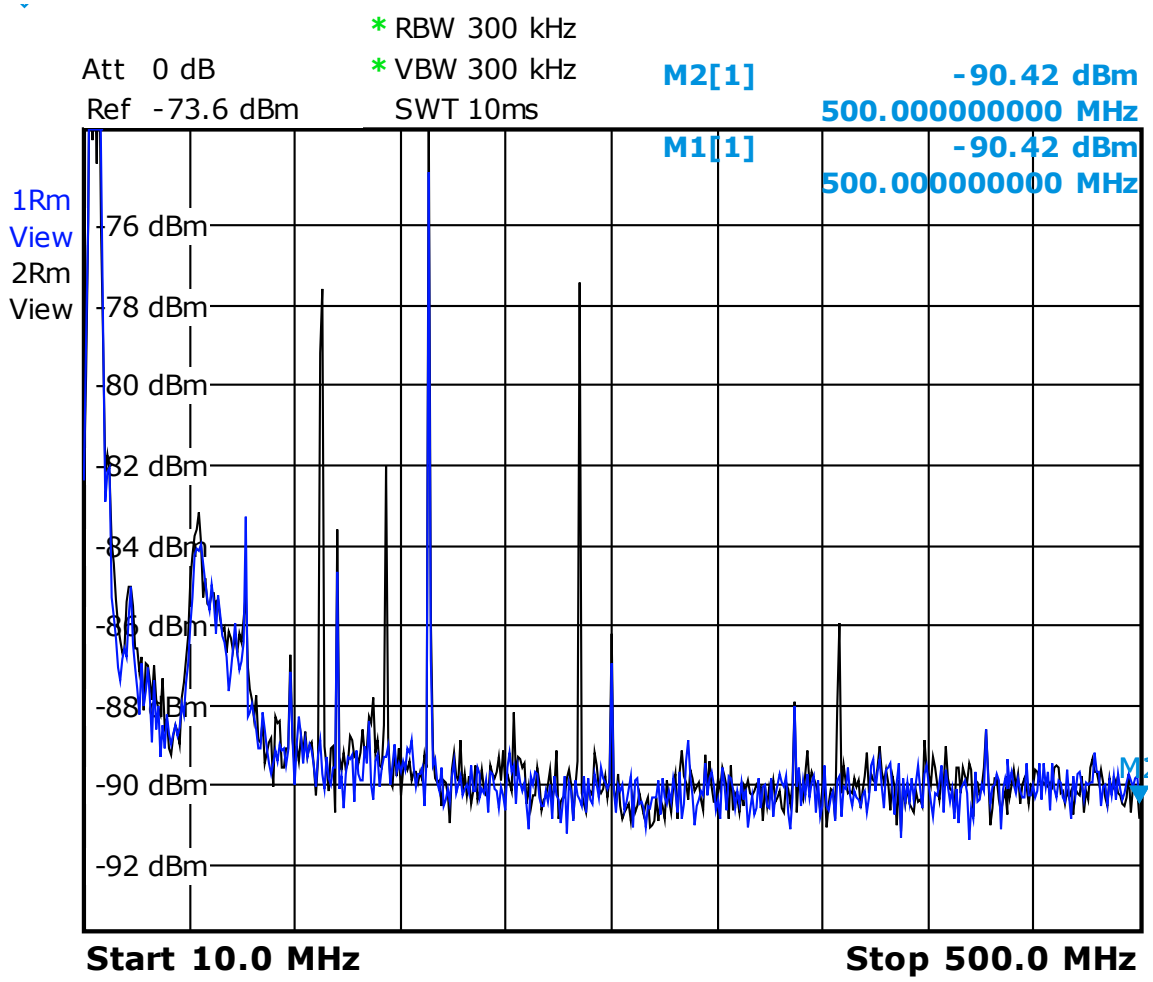

Abbildung 6: Störsignalmessung

Da der komplette Versuchsaufbau nicht abgeschirmt war, zeigen sich einige statische Träger, die auch bei deaktivierter Spannungsversorgung erhalten blieben. Im aktivierten Zustand sieht man einige Oberschwingungen der Taktfrequenz als exakte Vielfache von 60MHz. Am Analyzer waren diese noch wesentlich besser zu identifizieren, als auf dem Plot. Die Messung im Standby Modus enthält diese Oberwellen nicht. Abstrahlungen des 32kHz Quarzes konnten nicht gemessen werden. Sie lagen unterhalb des Rauschpegels.

Messung im HF-Labor

Im HF-Labor wurden die Evaluationsplatinen auf einen Drehtisch in der Absorberkammer gestellt. Jede Messung wurde pro Gerät vier Mal jeweils um 90° gedreht wiederholt. Dabei wurde das Emissionsspektrum zwischen 30 MHz und 1 GHz sowohl in horizontaler als auch vertikaler Polarisation gemessen und am Ende die Einzelergebnisse zu einem Gesamtbild überlagert. Die Antennen waren dabei etwa drei Meter von den Platinen entfernt.

Auch hier sind die Oberschwingungen wieder sehr gut als Vielfache von 60MHz bei 240 MHz (20,16 dBµV/m), 360 MHz (23,03 dBµV/m) und 480 MHz erkennbar, wenn der Mikrocontroller im Normalbetrieb läuft. Im Standby und Stop Modus sind sie verschwunden. Bemerkenswert ist hierbei noch die auch im Normalbetrieb sehr niedrige Abstrahlung unterhalb der Nachweisgrenze des Fujitsu Mikrocontrollers. Auch der Betrieb des Displays mitsamt Displaycontroller hatte überraschender Weise keinen Einfluss auf das Messergebnis.

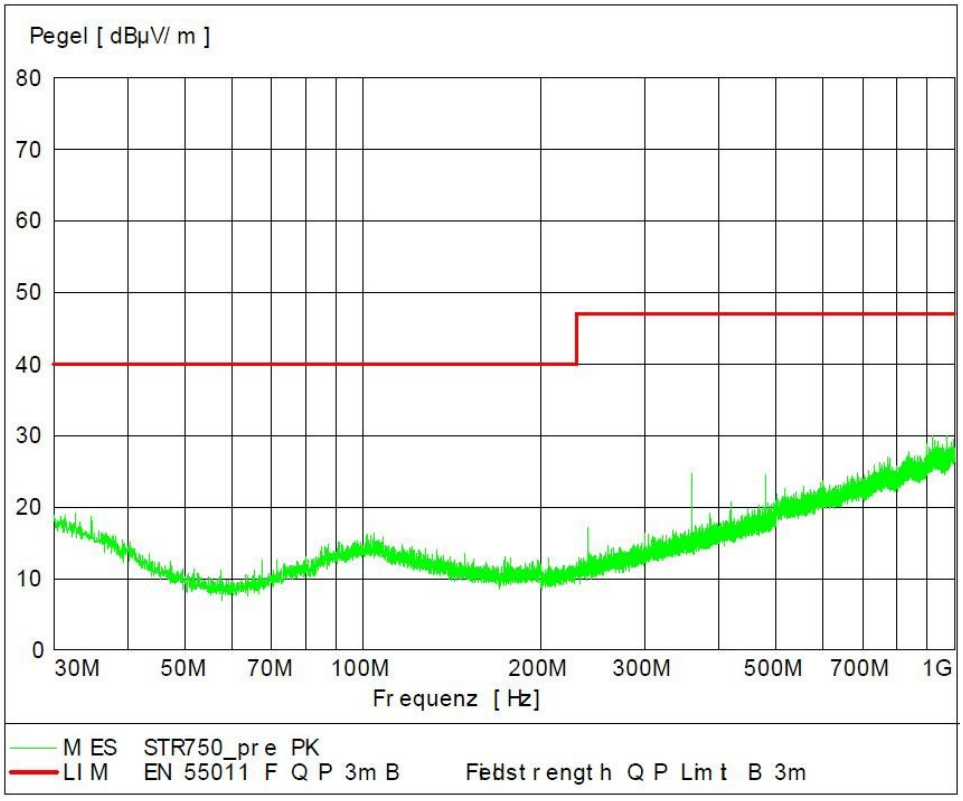

Abbildung 7: EMV Messung im eingeschalteten Zustand

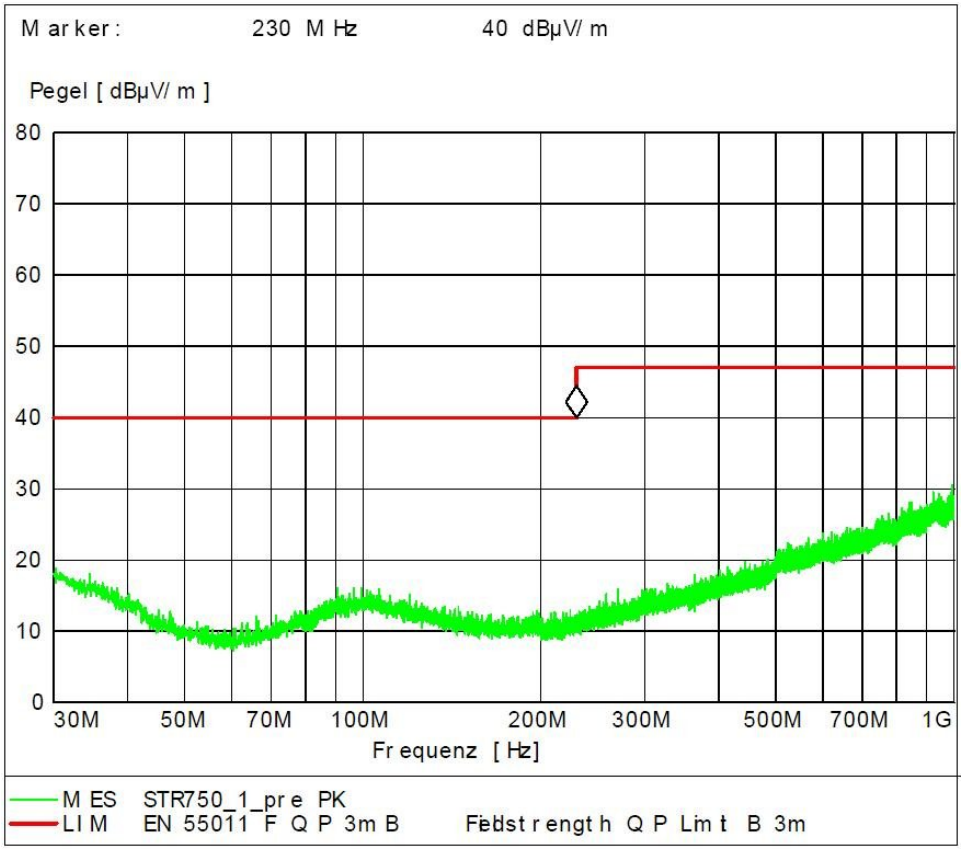

Abbildung 8: EMV Messung im Standby Modus

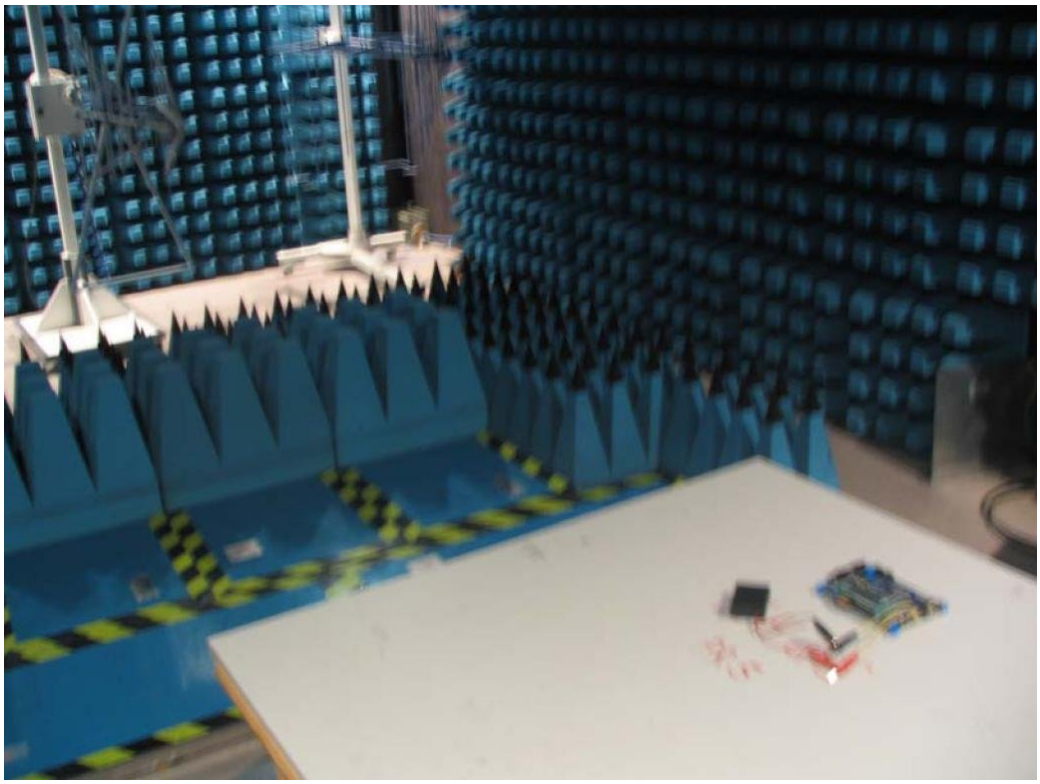

Abbildung 9: Messaufbau

5. Das Firmware-Framework

Ausgehend von einer bereits verwendeten modularen Testsoftware sollte ein Basissystem für zukünftige Entwicklungen der Abteilung geschaffen werden. Das Augenmerk lag dabei besonders auf leichter Portierbarkeit und einer modular erweiterbaren Struktur mit klar definierten Schnittstellen. Als Entwicklungsumgebung sollte dabei durchgängig Eclipse zum Einsatz kommen, um unabhängiger von proprietären Lösungen der einzelnen Hersteller zu werden und um eine einheitliche Umgebung für alle verwendeten Architekturen zu schaffen.

Struktur
Das Framework gliedert sich in drei Bestandteile, den Kern, die Abstraktionsschicht und Module.

Der Kern enthält systemunabhängige Logik für einen Kommandointerpreter, Debugfunktionen und stellt eine Schnittstelle zum Einhängen von Modulen bereit. Sie besteht aus mehreren speziellen Arrays mit Funktionspointnern für regelmäßig auszuführende Funktionen, Interruptroutinen und Intialisierung. Außerdem können Ein- und Ausgabefunktionen eines Moduls zu standardisierten Datenkanälen zusammengefasst werden, über die anschließend mit dem Chip kommuniziert werden kann.

In den Modulen steckt dann die eigentliche Anwendungslogik. Sie greifen wie der Kern auf eine Hardware-Abstraktionsschicht zu und sind nicht hardwareabhängig. Die meisten Module benötigen jedoch eigene Erweiterungen dieser Schicht, die auf die jeweils verwendete Hardware abgestimmt sind. Diese Treiber sind dabei möglichst so gehalten, dass sie ausschließlich die nötigsten Informationen und Routinen bereitstellen. Die eigentliche Logik, Funktionalität oder Datenverarbeitung der Anwendung steckt in den Modulen.

Die Abstraktionsschicht stellt dem Kern und den Funktionen für die verschiedenen Systeme bestimmte grundlegende Funktionen bereit. Beispielsweise An- und Abschalten einzelner Pins, blankes Schreiben und Lesen einer UART Schnittstelle oder Portdefinitionen für ein LCD Display. Die Schicht muss hoch genug liegen, um alle Unterschiede der Systeme darunter ausgleichen zu können, aber dennoch möglichst Tief, damit mehr Code systemunabhängig bleiben kann. Je schlanker die Schicht gehalten werden kann, desto einfacher ist das Framework portierbar.

Abbildung 10: Schema des Frameworks

6. Plugins für das CDT Managed Build System

Eclipse ist eine modular aufgebaute, leicht erweiterbare Entwicklungsplatform, deren Wurzeln in einer reinen Java IDE liegen. Durch Plugins wie die C/C++ Developement Tools (CDT) kann Eclipse auch als IDE für C und C++ verwendet werden. Auch für die Entwicklung von Webanwendungen in PHP, Fortran und andere exotischere Programmiersprachen existieren Erweiterungen. Dabei werden allerdings keine eigenen Compiler bereitgestellt.

Mit den CDT können C Projekte ganz regulär mittels selbst geschriebenem makefile erzeugt werden. Es bringt aber auch ein so genanntes Managed Build System mit. Dieses generiert aus Definitionen für die gesamte Werkzeugkette (Assembler, Compiler, Linker, Konverter) und benutzerdefinierten Eingaben automatisch ein solches makefile, mit dem dann die externe toolchain gefüttert wird. Die gesamte Ausgabe der toolchain des make-Aufrufs wird geparsed und als Fehlerliste an Eclipse zurückgegeben, so dass man Fehlerstellen im Programmcode ganz einfach via Doppelklick erreicht und korrigieren kann. Die Informationen über die Werkzeugkette und die Form der Fehlermeldungen können in einem eigenen kleinen Plugin definiert werden, um CDT an andere toolchains außer den üblichen GNU tools anzupassen.

Solche Plugins wurden im Zuge der Umstellung auf Eclipse als Ersatz für proprietäre IDEs (softune, µVision) für mehrere Mikrocontrollerarchitekturen erforderlich. Für den Einsatz von GCC als cross-compiler für den ARM existierte bereits solch ein Plugin, welches von mir noch etwas erweitert, von Bugs befreit und optimiert wurde. Für die Fujitsu toolchain musste das komplette Plugin auf Basis des ARM-Plugins plus eigenem error parser neu geschrieben werden. Für Atmel Chips existiert ebenfalls ein solches Plugin. Nach der jetzt vorhandenen Einarbeitung wäre auch eine weitere Portierung auf andere Architekturen wie z.B. den XC167, sofern die toolchain frei verfügbar und gut dokumentiert ist, kein großer Aufwand, womit dann alle in der Abteilung gängigen Architekturen abgedeckt wären.

Aufbau der Plugins
Die Plugins selbst bestehen aus einer sehr länglichen XML-Datei zur Beschreibung sämtlicher Werkzeuge, deren Parameter, Input und Output, passenden Defaultwerten, Angaben zum Parser und Variationen für verschiedene Buildsysteme (Windows/Linux/Mac/Solaris) und optional eigenen Java-Klassen wie dem errorparser zur Erweiterung der Funktionalität. Ein einzelnes Werkzeug wird etwa folgendermaßen definiert:

```
<tool
      id="org.eclipse.cdt.fujitsumcu.as" name="Fujitsu Assembler"
      outputFlag="-o"
      superClass="org.eclipse.cdt.fujitsumcu.targetselector">
      <optionCategory
            id="org.eclipse.cdt.fujitsumcu.as.debug" name="Debugging"
            owner="org.eclipse.cdt.fujitsumcu.as"/>
      <option
            category="org.eclipse.cdt.fujitsumcu.as.debug"
            id="org.eclipse.cdt.fujitsumcu.as.debug.output"
            command="-g" name="Write debug information to object (-g)"
            valueType="boolean" defaultValue="true"/>
```

```
        <option>
                …
        </option>
        <inputType
                dependencyExtensions="S,s,asm" sources="S,s,asm"
                id="org.eclipse.cdt.fujitsumcu.as.input" name="Input"
                sourceContentType="org.eclipse.cdt.core.asmSource" />
        <outputType
                buildVariable="OBJS" id="org.eclipse.cdt.fujitsumcu.as"
                name="Output" outputs="o" />
</tool>
```

Das Ergebnis sieht dann etwa folgendermaßen aus:

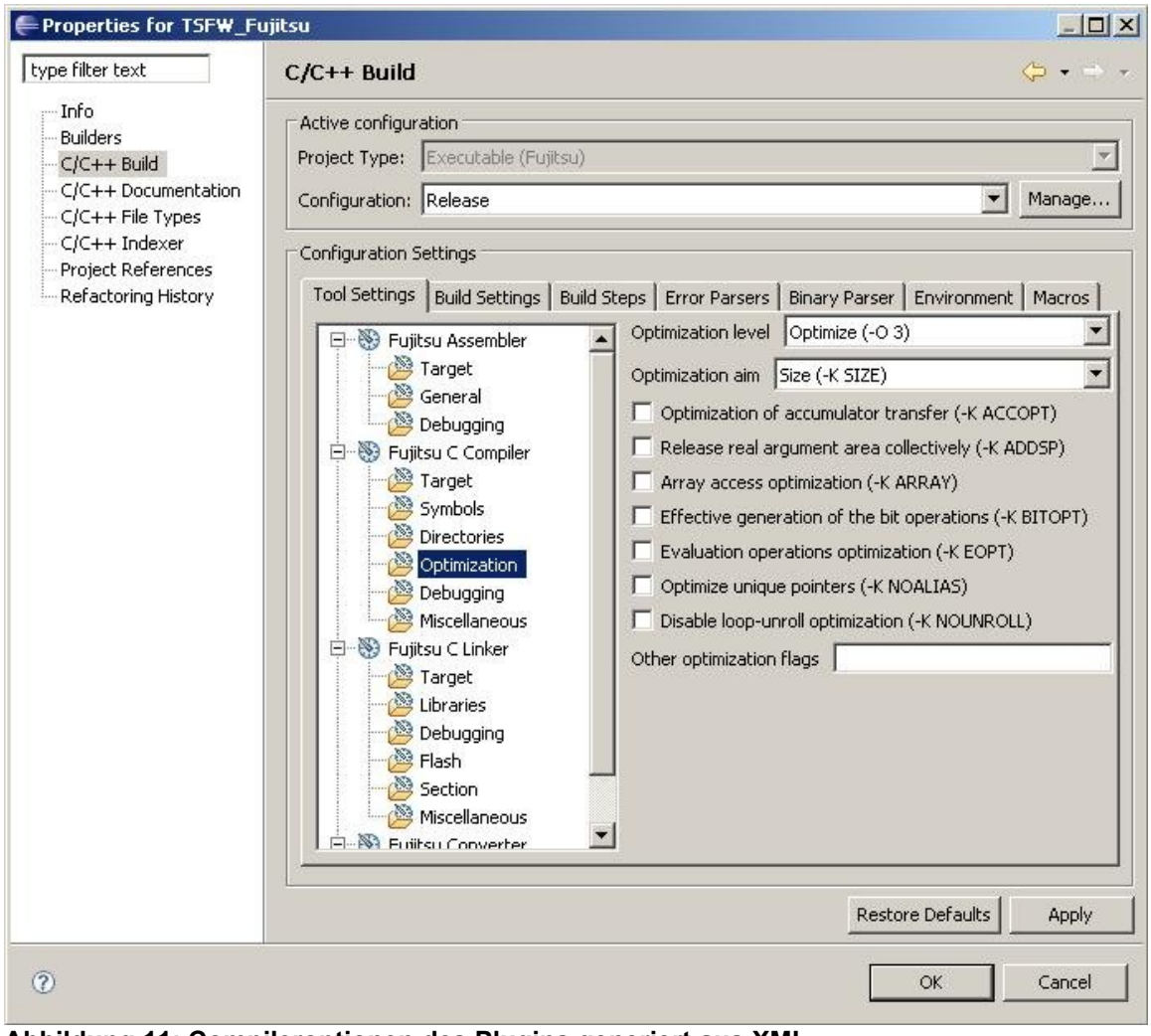

Abbildung 11: Compileroptionen des Plugins generiert aus XML

Man erkennt im Bild gut die einzelnen Werkzeuge. Darunter im Baum eingegliedert sind die optionCategories, denen jeweils die in der rechten Hälfte der Grafik eingeblendeten Optionen zugeordnet sind. Optionen können verschiedenen Typs für verschiedene Datentypen sein. Das Spektrum reicht von einfachen boolean-Checkboxen (Mitte) über string-Eingabezeilen (unten) und Dropdownboxen (oben) bis hin zu Dateiauswahlfenstern und string-Aufzählungen.

Errorparser
Der Parser ist wie auch Eclipse selbst in Java geschrieben. Er wird an einem so genannten extension point in die Klassenhierarchie von Eclipse eingehängt und kann so mit den anderen Modulen kommunizieren und auf deren Ressourcen zugreifen. Das Einhängen erfolgt wiederum durch Definition in der plugin.xml. Er definiert primär einige auf die Fehlerausgabe der toolchain angepasste reguläre Ausdrücke und deren Zerlegung in Parameter. Aus der Zeile

```
*** D:\Workspace\TSFW_Fujitsu\system\inc\sys_lcd.h(24) E4038P:
#include: cannot find file "75x_lib.h"
```

erhält man durch passende Zerlegung beispielsweise den Dateinamen, die Zeile, den Fehlercode und eine detaillierte Beschreibung des Fehlers. Wie schwerwiegend der Fehler ist, kann man aus dem ersten Buchstaben des Fehlercodes ablesen und in verschiedene Fehlertypen umsetzen. Unterschieden wird hierbei zwischen reiner Information (I), Warnungen (W), Fehlern (E) und fatalen Fehlern (F). Das für diesen Fehlertyp passende regex-pattern sieht folgendermaßen aus:

```
"\\*{3} (.+?)\\(([0-9]{1,4})\\) (([IWEF][0-9]{4}.): .*)"
```

Zu beachten ist dabei, dass doppelt escaped werden muss, da auch schon der Compiler beim Erstellen des Strings die Eingabe interpretiert und zum Beispiel “\n” in einen Zeilenumbruch verwandelt. Damit also der an den Patterninterpreter übergebene String einen Backslash enthält, muss hier jedesmal “\\” geschrieben werden.

Pluginentwicklung
Die komplette Verfügbarkeit des Quellcodes von CDT ist in Zusammenarbeit mit den umfangreichen Anzeigemöglichkeiten von Eclipse bei der Pluginentwicklung sehr hilfreich. Importiert man beispielsweise ein binäres CDT Package im Plugin, kann man jenes dennoch ganz automatisch durchsuchen und betrachten, als ob man dessen Code selbst geschrieben hätte. Ein Eingliedern des eigenen Plugins in das komplexe Framework gelingt erstaunlich leicht. Auch das Debugging und der Export eines Plugins sind vorbildlich gelöst. Mit einem Mausklick ist das in Entwicklung befindliche Plugin in einer neuen Eclipse-Instanz gestartet oder in die lokale Installation exportiert.

Zum Bearbeiten der Einstellungen des Plugins wie Versionsnummer, Autor, Abhängigkeiten und Erweiterungen stellt Eclipse zudem eine eigene Eingabemaske bereit, so dass man sich hier nicht selbst die Finger am XML schmutzig machen muss.

6. Der JTAG Adapter

JTAG steht für „Joint Test Action Group“ und bezeichnet normalerweise eine Debug- und Programmierschnittstelle für Mikroprozessoren. Sie wurde 1985/86 von einigen großen Halbleiterherstellern spezifiziert und letztendlich in der Norm IEEE 1149.1-1990 festgehalten. Auch für den ARM STR750 erfolgte die Programmierung über eine JTAG Schnittstelle. Folgende Leitungen werden benötigt:

1. Test Data Input (TDI) (positive Flanke)
2. Test Data Output (TDO) (negative Flanke)
3. Test Clock (TCK)
4. Test Mode Select Input (TMS)
5. Test Reset (TRST)

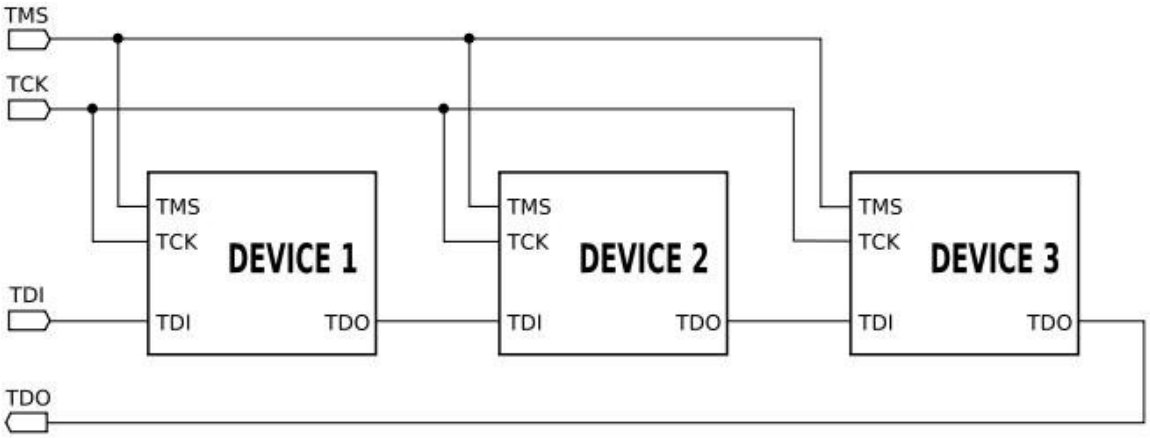

Abbildung 12: JTAG Verschaltung

Sowohl von ARM als auch von Atmel werden spezielle JTAG Schnittstellen definiert, die die oben genannten Pins einer Steckerbelegung zuweisen. Atmel definiert dabei einen kompakten 10-poligen Stecker, ARM unnötiger weise einen 20-poligen, wobei die Hälfte der Pins mit Masse belegt sind.

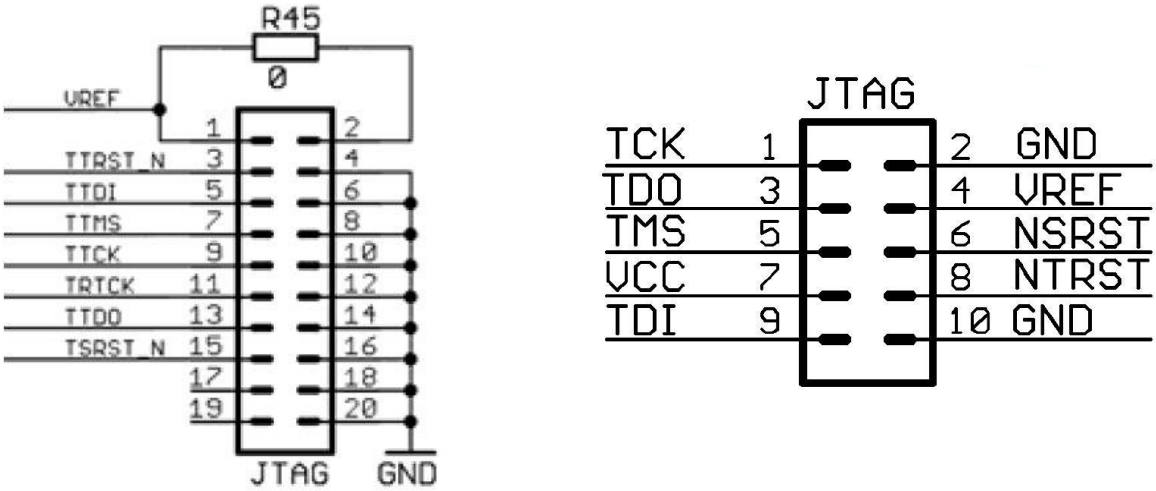

Abbildung 13: ARM- und AVR JTAG Pinbelegung

Auf der Seite des PCs existiert eine halbwegs definierte Pinbelegung, benannt nach dem sog. Wiggler von Macraigor Systems. Sie wird von einigen programmern unterstützt, obwohl sich hier nie ein wirklich einheitlicher Standard ergeben hat. Passende Adapter sind für etwa 50€ erhältlich. Aufgrund der relativ simplen Verdrahtung und der Verfügbarkeit aller benötigten Teile im Labor, entstand der Adapter dann im Eigenbau.

Leider funktionierte der Prototyp nicht wie geplant. Auch nach längerem Ausmessen der Leitungen und mehrfacher Überprüfung der Pinbelegung ließ sich der ARM damit nicht zur Zusammenarbeit bewegen. Um nicht noch mehr Zeit zu verlieren und da sich eine weitere kostenfreie Variante eröffnete, stellten wir das Projekt ein.

Als gut funktionierende Alternative stellte sich letztendlich der proprietäre Keil-Adapter dar. Mit Hilfe der abgespeckten Demoversion von µVision3 im Batchmodus, und einigen gut versteckten Kommandozeilenoptionen dafür, tat er klaglos, unabhängig von der IDE und vor allem ohne teure Lizenzen seinen Dienst.