HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

DIPLOMARBEIT

Herr
Amadeus Alfa

Implementierung und Untersuchung von Algorithmen der Sprachsignalcodierung auf dem TMS320DM6446

2010

Fakultät Informationstechnik & Elektrotechnik

DIPLOMARBEIT

Implementierung und Untersuchung von Algorithmen der Sprachsignalcodierung auf dem TMS320DM6446

Autor:
Amadeus Alfa

Studiengang:
Elektrotechnik

Seminargruppe:
ET06wK1

Erstprüfer:
Prof. Dr.-Ing. habil. Hans- Joachim Thomanek

Zweitprüfer:
Prof. Dr. rer. nat. Sergej Alekseev

Mittweida, November 2010

Bibliografische Angaben

Autor:	Alfa, Amadeus
Titel:	Implementierung und Untersuchung von Algorithmen der Sprachsignalcodierung auf dem TMS320DM6446
Herausgabe:	November 2010 Mittweida, Hochschule Mittweida (FH), University of Applied Sciences, Fakultät Informationstechnik & Elektrotechnik, Diplomarbeit
Seitenzahl:	94

Referat

Ziel der vorliegenden Diplomarbeit ist es, den digitalen Signalprozessor des Typs „TMS320DM6446“ der Firma Texas Instruments und dessen Softwareumgebung zu untersuchen. Zu diesem Zweck soll eine umfassende Dokumentation von der Installation, über die Benutzung, bis hin zur Entwicklung eigener Programme erstellt werden.
Die Arbeit umfasst detaillierte Vorschriften zur Verwendung der Softwarekomponenten, sowie selbst erstellte Beispiele und darüber hinaus einen Praktikumsversuch für Studenten.

Abstract

The following paper was created to research Texas Instruments' digital signal processor „TMS320DM6446“ and its software components. For future use setup and installation steps plus software requirements have been specified as well as development issues.
This document provides detailed information about using shipped software files in order to run custom applications such as pre-built tutorials for students which have been attached to this file.

Inhaltsverzeichnis

Abkürzungsverzeichnis

Advanced Audio Coding High Efficiency	AAC-HE
Advanced Audio Coding Low Complexity	AAC-LC
Advanced Audio Coding Low Delay	AAC-LD
Advanced Linux Sound Architecture	ALSA
Advanced RISC Machine	ARM
Apache Subversion	SVN
Application Programming Interface	API
Basic Input/Output System	BIOS
Code Generation Tool	CGT
Codec Engine	CE
Davinci Multimedia Application Interface	DMAI
Digital Video Evaluation Module	DVEVM
Digital Video Software Development Kit	DVSDK
Digitaler Signalprozessor	DSP
eXpress DSP Components	XDC
eXpressDsp Algorithm Interoperability Standard	xDAIS
eXpressDSP Digital Media	xDM
Internet Protocol	IP
Joint Photographic Experts Group	JPEG
Linux Support Package	LSP
Microprocessor without Interlocked Pipeline Stages	MIPS
MontaVista Linux	MVL
Network File System	NFS
Open Sound System	OSS

Operating System Abstraction Layer	OSAL
Puls-Code-Modulation	PCM
Redhat Package Manager	RPM
Secure Shell	SSH
Software Development Kit	SDK
SSH File Transfer Protocol	SFTP
Texas Instruments	TI
Trivial File Transfer Protocol	TFTP
Universal Boot	U-Boot
Virtual Instrument Software Architecture	VISA
Voice over IP	VoIP
Windows Media Audio	WMA

Vorwort

Die vorliegende Diplomarbeit wurde im Zeitraum der Monate Juni bis November des Jahres 2010 angefertigt und ist zugleich der Abschluss des Studiengangs Elektrotechnik Fachrichtung Kommunikationstechnik an der Hochschule Mittweida (FH).

Für die Unterstützung im Entstehungsprozess dieser Arbeit durch Beantwortung von Fragen, Überprüfung der Quellcodes, sowie für das Durchsehen und Ausprobieren der Softwaredokumentation möchte ich mich ausdrücklich bedanken bei:

Prof. Dr.-Ing. habil. Hans- Joachim Thomanek

Prof. Dr. rer. nat. Sergej Alekseev

1 Einleitung

1.1 Motivation der Arbeit

Die digitale Signalverarbeitung ist mit ihren Algorithmen und Datenstrukturen ein fester Bestandteil der heutigen Kommunikationsanwendungen. Analoge Filter bestehend aus klassischen elektronischen Bauteilen, wie Kondensatoren und Spulen, bieten nur mit hohem Schaltungsaufwand und damit verbunden unter Einsatz nicht unerheblicher finanzieller Mittel, etwa bei Präzisionsteilen, die Möglichkeit zu einer gezielten Implementierung einer Vorschrift zur Veränderung von Signalanteilen für Tiefpässe oder Echo-Unterdrückung.
Zur Verarbeitung von Nachrichten mit Hilfe digitaler Rechentechnik kommen digitale Signalprozessoren (DSPs) oder Mikrokontroller auf Embedded Systems zur Anwendung, welche durch ihre frei programmierbare Softwareumgebung im Vergleich zu einer gelöteten Platine eines analogen Filters sehr viel flexibler in ihrer Anwendung und Skalierbarkeit sind. Es können viele Algorithmen auf dem gleichen DSP genutzt werden. Neue Algorithmen lassen sich ebenso, insofern sie einmal vorhanden sind, in kurzer Zeit implementieren oder anpassen.
Auf einem analogen Filter durchquert ein Signal analoge elektronische Bauteile in deterministischer Laufzeit. Digitale Algorithmen sind hingegen nur ein Abbild einer mathematischen Gesetzmäßigkeit, welche auf einem Prozessor mit einer bestimmten Betriebsfrequenz, definiertem Interrupthandling und begrenztem Speichermanagement umgesetzt wird. Der dadurch entstehende Overhead und die zusätzliche Zeitkomponente müssen bei der Entwicklung eines DSPs berücksichtigt werden, sodass die Echtzeitfähigkeit des Systems weiterhin gewährleistet ist, um das Ausgabesignal in endlicher Zeit zur Verfügung stellen zu können.

In der vorliegenden Diplomarbeit soll ein DSP-Board des Typs „TMS320DM6446“ der Firma Texas Instruments (TI) auf seine Möglichkeiten der Programmierung über ein Application Programming Interface (API) in der Programmiersprache C hin untersucht werden. Damit verbunden ist gleichermaßen die Systemumgebung für den Zugriff auf den DSP über eine Linux-Plattform unter ARM-Architektur sowie das Software Development Kit (SDK) auf einer externen Linux-Maschine unter Benutzung der Virtualisierungsumgebung „VMware“. Unter Vorbereitung eines Praktikumsversuchs für Studenten der Lehrveranstaltung „Digitale Signalverarbeitung“ soll zudem die Arbeit mit einem DSP und das Bewusstsein gegenüber Sprachcodecs nähergebracht werden.

1.2 Zielsetzung

Ziel dieser Diplomarbeit ist die Erstellung einer Dokumentation über den Aufbau, die Installation und Benutzung des DSPs, sowie die Untersuchung der Möglichkeiten im Bereich von Codierungsaufgaben und die Benutzung der Softwareumgebung.
Bei der Arbeit mit dem DSP-Board unter Linux ist ein Grundverständnis der UNIX-Philsophie und erweiterte Anwenderkenntnisse über Kernel und Systemwerkzeuge erforderlich. Als notwendiges Ziel steht die Vertiefung dieser Kenntnisse in Kombination mit der Abstimmung auf die Eigenschaften des Boards im Vordergrund. Zur Verwaltung der zahlreichen Coderevisions kam das Versionskontrollsystem Apache Subversion (SVN) zum Einsatz, welches jederzeit das Verändern eines Quellcodes erlaubt, ohne diesen endgültig mit einer neuen Version zu überschreiben und damit die Entwicklung von Software erheblich vereinfacht.

Die im Zuge der Benutzung des Systems entstandenen Fragen und Probleme, sowie der Gebrauch des Systems selbst soll für spätere Zwecke gut dokumentiert werden. Einige ausgewählte Code-Beispiele für Codierungsaufgaben sollen erstellt und kommentiert werden, um die Arbeitsweise des APIs zu verstehen.

2 Das Digital Video Software Development Kit

2.1 Installation unter Redhat Linux

2.1.1 Herunterladen der benötigten Installationsdateien

Eine neue Version des Digital Video Software Development Kits (DVSDK) oder einfach eine zusätzliche VMware-Workstation als Board-Arbeitsplatz erfordern eine Komplettinstallation der Softwareumgebung. Die dafür notwendigen Dateien für die Version 2.00 des DVSDKs werden auf den Websites von Texas Instruments zur Verfügung gestellt [1].
Die folgenden Binär-Installer sind für ein vollständiges Setup notwendig (Versionsangaben können abweichen):

- **mvl_5_0_0801921_demo_sys_setuplinux.bin**
  mvl_5_0_0_demo_lsp_setuplinux_02_00_00_140.bin
  Installationspakete für MontaVista Linux (MVL) der Version 5 (Filesystem und Kernel für das Board). Später werden die Dateien per Network File System (NFS) und Trivial File Transfer Protocol (TFTP) bereitgestellt. Die Installation erfolgt üblicher Weise außerhalb des DVSDKs, etwa in „/opt“.
- **dvsdk_setuplinux_2_00_00_22.bin**
  Das DVSDK stellt die Basis der Entwicklungswerkzeuge dar, enthalten sind CodecEngines, eXpressDsp Algorithm Interoperability Standard (xDAIS) und eXpressDSP Digital Media (xDM) Komponenten sowie Kernel-Module für CMEM.
- **xdctools_setuplinux_3_10_05_61.bin**
  Die eXpress DSP Components (XDC) Tools dienen der Verarbeitung von XDC-Metadaten, etwa in Codec-Packages enthaltene Informationen über Bitraten oder Wasserzeichen. Besondere Bedeutung kommt ihnen bei der Erstellung eigener x64P Combofiles zu.
- **bios_setuplinux_5_33_03.bin**
  Für die Kommunikation mit dem DSP müssen die Basic Input/Output System (BIOS) Tools installiert sein. Dabei ist darauf zu achten, dass diese Version mit der Version des DVSDKs abgestimmt sein muss (in diesem Package der Fall).
- **TI-C6x-CGT-v6.0.21.1.bin**
  Die Code Generation Tools (CGTs) dienen der dynamischen Erzeugung von Quellcodes für verschiedene Zielplattformen in C64x+.

– **dm6446_codecs_setuplinux_2_00_00_22.bin**
  Die Codecs müssen in dieser Version separat vom DVSDK installiert werden. Darin enthalten sind die Standard-Packages, welche auch im Lieferumfang beiliegen. Zusätzliche Codecs sind auf den Websites von TI zum Download verfügbar (weitere Informationen im Kapitel 2.2.1).

Zur Sicherheit befinden sich alle Installationsdateien auch noch einmal im Pfad „/home/praktikum/dvsdk_praktikum/dvsdk_setup/“ um eventuellen Veränderungen der Websites vorzubeugen.

2.1.2 Starten des Setups

Die nachfolgende Anleitung geht davon aus, dass alle benötigten Setup-Dateien unter Redhat-Linux im Pfad „/tmp/setup“ abgespeichert wurden (vgl. [2] Kapitel 4.3.1 ff).

Nach dem Öffnen eines Terminals in der VMware kann die Installation begonnen werden. Zunächst müssen alle heruntergeladenen Dateien „ausführbar“ gemacht werden:

Setup-Dateien ausführbar machen

```
$ cd /tmp/setup
$ chmod +x *.bin
```

Nun kann ein neues Verzeichnis für „MontaVista Linux“ der Version 5 erstellt und die notwendigen Installer als Benutzer „root“ gestartet werden. Bei der Wahl des Zielverzeichnisses soll „/opt/mv_pro_5.0“ angegeben werden:

MontaVista Linux wird installiert

```
$ su
# mkdir /opt/mv_pro_5.0
# ./mvl_5_0_0801921_demo_sys_setuplinux.bin
# ./mvl_5_0_0_demo_lsp_setuplinux_02_00_00_140.bin
```

Nach erfolgreicher Installation in den Pfad „/opt/mv_pro_5.0“ liegen an dieser Stelle 2 Archiv-Dateien bereit (Überprüfung mit „ls“). Beide Archive müssen entpackt werden:

Entpacken der installierten Archiv-Dateien

```
# cd /opt/mv_pro_5.0
# ls -lh
insgesamt 1,1G
-rw-r--r-- 1 root root  57M DaVinciLSP_02_00_00_140.tar.gz
-rw-r--r-- 1 root root 982M mvltools5_0_0801921_update.tar.gz
-rwxr-xr-x 1 root root 1,2M uninstall
# tar xfzv DaVinciLSP_02_00_00_140.tar.gz
# tar xfzv mvltools5_0_0801921_update.tar.gz
```

Das Filesystem, welches dem Board später per NFS zugänglich gemacht wird, ist nun unter „/opt/mv_pro_5.0/montavista/pro“ installiert. Die beiden Archiv-Dateien können nach dem Entpacken ohne Auswirkung gelöscht werden, etwa bei Speicherplatzbedarf.

Die folgende Installation des eigentlichen DVSDKs nach „/home/praktikum“ darf nicht als „root“ erfolgen („exit“ verlässt die Root-Shell):

Installation des DVSDKs in das Home-Verzeichnis

```
# exit
$ cd /tmp/setup
$ ./dvsdk_setuplinux_2_00_00_22.bin
```

Im Anschluss können die XDC-Tools sowie die BIOS-Tools und die Standard Codec-Packages installiert werden. Dabei muss der Installationspfad jeweils auf das im vorhergehenden Schritt erstellte Verzeichnis „/home/praktikum/dvsdk_2_00_00_22“ umgestellt werden:

XDC, BIOS und Codec-Packages in das DVSDK installieren

```
$ ./xdctools_setuplinux_3_10_05_61.bin
$ ./bios_setuplinux_5_33_03.bin
$ ./dm6446_codecs_setuplinux_2_00_00_22.bin
```

Die zentrale Konfigurationsdatei „Rules.make“ muss nach der Installation noch angepasst werden, da die Plattformbezeichnung und einige Pfade angepasst werden müssen. Eine entsprechend vorkonfigurierte Version kann aus dem Praktikums-Ordner einfach kopiert werden (Anhang G):

Kopieren der Datei „Rules.make“

```
$ cp /home/praktikum/dvsdk_praktikum/configs/dvsdk_Rules.make
      ␣ /home/praktikum/dvsdk_2_00_00_22/Rules.make
```

Der vorerst letzte Schritt installiert die CGTs in ein zuvor erstelltes Verzeichnis „/home/praktikum/dvsdk_2_00_00_22/cg6x_6_0_21“, welches im Wizard ausgewählt werden muss. Es müssen zudem für spätere Zwecke die Environment-Variablen der Linux-Shell angepasst werden:

Installation der CGTs

```
$ mkdir /home/praktikum/dvsdk_2_00_00_22/cg6x_6_0_21
$ ./TI-C6x-CGT-v6.0.21.1.bin
$ export C6X_C_DIR=/home/praktikum/dvsdk_2_00_00_22/cg6x_6_0_21/
      lib:/home/praktikum/dvsdk_2_00_00_22/cg6x_6_0_21/include
```

Zur dauerhaften Speicherung muss die folgende Änderung der Umgebungsvariable „$PATH“ an das Ende der Datei „/home/praktikum/.bashrc“ eingefügt werden (erfolgt mittels „echo“):

Umgebungsvariablen der Shell setzen

```
$ echo "PATH=\"/opt/mv_pro_5.0/montavista/pro/devkit/arm/v5t_le/
      bin:/opt/mv_pro_5.0/montavista/pro/bin:/opt/mv_pro_5.0/
      montavista/common/bin:$PATH\"" >> /home/praktikum/.bashrc
$ source /home/praktikum/.bashrc
```

Wenn das gesamte Setup vollständig durchlaufen wurde, können alle Installationsdateien entfernt werden:

```
$ rm -rf /tmp/setup
```

2.1.3 Konfiguration des NFS-Servers

Das Root-Filesystem, von dem das Board später booten soll, muss an geeigneter Stelle plaziert werden. Um Speicherplatz zu sparen reicht es dafür „Symlinks“ zu verwenden, eine symbolische Verknüpfung:

```
$ mkdir /home/praktikum/workdir
$ ln -s /opt/mv_pro_5.0/montavista/pro/devkit/arm/v5t_le/target
      ␣ /home/praktikum/workdir/filesys
```

Falls der Ort „/home/praktikum/workdir“ aufgrund früherer Installationen schon existieren sollte, genügt es „workdir“ zu entfernen oder umzubenennen bevor die Verknüpfung angelegt wird.
Wenn keine Symlinks verwendet werden sollen, können die benötigten Dateien auch mit dem System-Befehl „cp“ kopiert werden (siehe dazu [2] Kapitel 4.3.7).

Der NFS-Server muss nun, insofern noch nicht geschehen, über das neue Verzeichnis per Konfigurationsdatei im Pfad „/etc/exports“ in Kenntnis gesetzt und neu gestartet werden:

Konfiguration des NFS-Servers

```
$ su
# echo "/home/praktikum/workdir/filesys *(rw,no_root_squash,
      no_all_squash,sync)" > /etc/exports
# /usr/sbin/exportfs -av
# /sbin/service nfs restart
```

2.1.4 Einrichten eines TFTP-Servers

Die Umstellung auf die Version 2.00 des DVSDKs macht es erforderlich auf dem Board von einem neuen Kernel zu booten. Die folgende Skizze dient der Verdeutlichung des veränderten Ablaufs links:

Veränderter Bootvorgang durch TFTP

Anders als in älteren Setups muss also der Kernel von einer externen Quelle auf das Board geladen werden, statt wie vorher üblich direkt von der Festplatte entpackt zu werden. In der VMware muss dazu ein TFTP-Server konfiguriert werden, welcher dem Board beim Starten das Kernel-Image über das Netzwerk liefert (vgl. [2] Kapitel A.3).
Zunächst muss geprüft werden, ob unter Redhat bereits ein TFTP-Server installiert ist (im Standard-Setup ist dies der Fall):

```
$ rpm -q tftp-server
tftp-server-0.39-1
$ /sbin/chkconfig --list | grep tftp
        tftp:   Ein
```

Sind diese beiden Abfragen negativ, muss das Paket aus den Quellen des Redhat Package Managers (RPM) manuell nachinstalliert werden. Andernfalls kann man den folgenden Schritt überspringen. Als Beispiel soll hier die Version „tftp-server-0.39-1“ für die Architektur „i386“ eingerichtet werden:

Installation eines TFTP-Servers

```
$ cd /tmp
$ wget "ftp://ftp.univie.ac.at/systems/linux/kernel.org/software/
      network/tftp/RPMS/i386/tftp-server-0.39-1.i386.rpm"
$ su
# rpm -ivh tftp-server-0.39-1.i386.rpm
```

Ist der TFTP-Server installiert, kann er einfach gestartet werden. Der Standard-Pfad in der VMware für Dateien, die mit TFTP transportiert werden sollen, ist „/tftpboot“:

```
# /sbin/chkconfig tftp on
```

2.1.5 Compilieren eines neuen Kernels

Der neue Kernel, welcher bei jedem Bootvorgang mit TFTP übertragen wird, muss nun aus den Kernelquellen compiliert werden.
Alternativ dazu befindet sich ein vorcompilierter Kernel der Version 2.6.18 im Verzeichnis „/home/praktikum/dvsdk_praktikum/dvsdk_setup/uImage“, welcher einfach kopiert werden kann. In diesem Fall kann man die folgende Schritte mit Ausnahme des letzten (abschließender Kopiervorgang) überspringen.

Zur Compilierung des Kernels müssen die Quellcodes zunächst per Symlink an die richtige Stelle im Filesystem gelinkt werden:

```
$ mkdir /home/praktikum/workdir/lsp
$ ln -s /opt/mv_pro_5.0/montavista/pro/devkit/lsp/ti-davinci
      ␣ /home/praktikum/workdir/lsp/
```

Nun kann man den Kernel mit den Standard-Optionen, welche bereits unter der Datei „davinci_dm644x_defconfig“ konfiguriert sind, cross-compilieren. Dabei wird unter einer i386er-Maschine ein Image für die ARM-Architektur erstellt:

```
$ cd /home/praktikum/workdir/lsp/ti-davinci/linux-2.6.18_pro500
$ su
# make ARCH=arm CROSS_COMPILE=arm_v5t_le- davinci_dm644x_defconfig
```

Bei speziellen Anforderungen ist es möglich die Kernelkonfiguration zu verändern. Dazu kann entweder direkt die Datei „.config“ editiert werden, oder das grafische Menü benutzt werden, welches sich mit „make ARCH=arm CROSS_COMPILE=arm_v5t_le-menuconfig“ aufrufen lässt.

Schließlich kann das eigentliche uImage, welches zum Booten benötigt wird, compiliert werden, was einiges an Zeit in Anspruch nimmt. Danach müssen noch die Kernelmodule an die richtige Stelle kopiert werden, da es sich um einen Kernel mit Modulsupport handelt:

Kernel und -module compilieren

```
# make ARCH=arm CROSS_COMPILE=arm_v5t_le- uImage
# make ARCH=arm CROSS_COMPILE=arm_v5t_le- modules
# make ARCH=arm CROSS_COMPILE=arm_v5t_le- INSTALL_MOD_PATH=/home/
      praktikum/workdir/filesys modules_install
```

Nach dem Compilevorgang befindet sich das fertige Kernel-Image im Unterverzeichnis „arch/arm/boot“ und muss nurnoch nach „/tftpboot“ kopiert werden:

Fertiges Kernel-Image kopieren

```
# cp /home/praktikum/workdir/lsp/ti-davinci/linux-2.6.18_pro500/
      arch/arm/boot/uImage /tftpboot
```

2.1.6 Konfiguration des Bootloaders

Die Einrichtung der VMware ist abgeschlossen, nun folgt die Umstellung des Bootloaders „U-Boot“ (Universal Boot) auf dem Board, um von TFTP zu booten (vgl. [2] Kapitel A.4.4). Dazu muss in der VMware eine Verbindung mittels „minicom“ zum seriellen Terminal aufgebaut werden (weitere Informationen im Kapitel 3.3).
Danach kann man das Board einschalten und muss während des Startens eine beliebige Taste gedrücken, um den automatischen Bootvorgang zu stoppen:

```
U-Boot 1.1.3 (Dec  4 2006 – 12:05:38)

U-Boot code: 81080000 -> 81097628  BSS: -> 810A0018
RAM Configuration:
[...]
Hit any key to stop autoboot: 0
DaVinci EVM #
```

Im Bootmenü angelangt, kann man mit der Einstellung der neuen Parameter beginnen. Dazu sollte das Terminalfenster, in welchem „minicom“ läuft, maximiert werden, da die serielle Konsole keinen automatischen Zeilenumbruch beherrscht und lange Zeilen somit

verschwinden.

Die hier verwendeten IP-Adressen dienen nur als Beispiel und unterliegen folgender Konfiguration: VMware: 141.55.242.175 DVEVM: 141.55.242.191

Setzen der Bootparameter

```
DaVinci EVM # setenv bootcmd  'tftp;bootm'
DaVinci EVM # setenv ipaddr   141.55.242.191
DaVinci EVM # setenv serverip 141.55.242.175
DaVinci EVM # setenv bootfile uImage
DaVinci EVM # setenv nfshost  141.55.242.175
DaVinci EVM # setenv rootpath /home/praktikum/workdir/filesys
DaVinci EVM # setenv nfsroot  $(nfshost):$(rootpath)
DaVinci EVM # setenv bootargs console=ttyS0,115200n8 noinitrd
      rw ip=dhcp nfsroot=$(nfsroot),nolock mem=120M
DaVinci EVM # saveenv
Saving Environment to Flash...
[...]
Writing to Flash...\done
Protected 1 sectors
```

Um aus dem Konfigurationsmenü zu gelangen kann man entweder den Resetknopf direkt am Board betätigen, oder den Befehl „reset“ in den Bootloader eingeben. Danach sollte das Board mit dem neuen Setup booten.

2.1.7 Anlegen eines Benutzers und Rechtevergabe

Nachdem das Board gestartet ist, kann sich als Benutzer „root“ (ohne Passwort) eingeloggt werden. Bei Bedarf kann ein Passwort mit dem System-Befehl „passwd“ gesetzt werden.
Anschließend soll ein neuer Benutzer „praktikum“ angelegt werden. Um Konflikte zwischen den Rechten an Dateien auf dem Board und der VMware auszuschließen, sollten die User-IDs dieses Benutzers auf beiden Systemen jeweils identisch sein. Dazu muss zunächst auf der VMware als Benutzer „praktikum“ der Befehl „id“ ausgeführt werden (IDs können vom Beispiel abweichen):

```
$ id
```

```
uid=500(praktikum) gid=500(praktikum) [...]
```

Auf dem Board muss nun als „root“ der Benutzer „praktikum“ mit der im oberen Schritt gewonnen User-ID angelegt und Rechte auf das Verzeichnis „/opt“ erteilt werden:

Erstellen eines neuen Benutzers auf dem Board

```
# adduser --uid 500 praktikum
Adding user praktikum...
Adding new group praktikum (500).
[...]
Is the information correct? [y/N] y
# chown -R praktikum:praktikum /opt
```

Nun kann man sich als „praktikum“ auf dem Board einloggen und in das Home-Verzeichnis „/home/praktikum“ schreiben, welches gleichmaßen in der VMware unter „/home/praktikum/workdir/filesys/home/praktikum“ zu finden ist.
Aufgrund der identischen User-ID haben beide Benutzer „praktikum“ sowohl auf dem Board als auch auf der VMware gleiche Rechte auf dieses Verzeichnis, sowie auf „/opt“.

Eventuell soll der Benutzer „praktikum“ und sein Home-Verzeichnis aus bestimmten Gründen wieder entfernt werden:

Löschen eines Benutzers und dessen Home-Verzeichnis

```
# deluser --remove-home praktikum
```

2.1.8 Kopieren der Kernelmodule und Demofiles

In der Standardinstallation ist das Verzeichnis „/opt“, welches die Kernelmodule „cmemk“ und „dsplink“, sowie einige Demos bereitstellen soll, leer. Aus dem Praktikums-Ordner können von der VMware aus die Dateien einfach kopiert werden:

Kopieren des /opt-Verzeichnisses

```
$ cp -R /home/praktikum/dvsdk_praktikum/opt/*
       ␣ /home/praktikum/workdir/filesys/opt/
```

2.1.9 Automatisches Laden der Kernelmodule

Nach jedem Start des Boards müssen die notwendigen Kernelmodule zur Zusammenarbeit mit dem DSP neu geladen werden. Ein vorkonfiguriertes Skript wurde dazu bereits im Kapitel 2.1.8 mit kopiert. Auf dem Board kann die Datei nun zur Bootzeit (über Runlevel 3, Priorität 30) ausführbar gemacht werden:

Konfigurieren der Bootreihenfolge auf dem Board

```
# chmod +x /opt/dvsdk/dm6446/loadmodules.sh
# ln -s /opt/dvsdk/dm6446/loadmodules.sh /etc/rc.d/rc3.d/S30dvsdk
```

Der Symlink auf das Verzeichnis „rc3.d“ indiziert den Zugriff auf das Runlevel 3, welches als das Standardrunlevel des Systems für normales Booten in die Konsole definiert ist. Linux kennt auch andere Runlevels, wie etwa 5 für das Booten bis zum X-Window-System (grafische Oberfläche).

2.2 Erzeugung eigener Combofiles

Um beliebige Codecs gleichzeitig in einem Programm nutzen zu können, müssen sich diese alle innerhalb einer Codec Engine befinden. Das dazu verwendete x64P-File („Codec-Combofile“) wird beim Aufruf der API-Funktion „CERuntime_init()“ auf den DSP als eine Art Gesamtpaket geladen und kann zur Laufzeit nicht mehr verändert werden. Zudem kann immer nur genau ein x64P-File in Benutzung sein. Darum muss für beliebige Codecs in Kombination ein eigenes Combofile erstellt werden.

2.2.1 Auswahl an Codecs

Von TI sind Codecs in den folgenden Gruppen auf deren Websites verfügbar:

- Speech [3]
  ENC/DEC: G.711, G.722, G.722.1, G.722.2, G.726
- Audio [4]
  ENC/DEC: AAC-LC, AAC-HE, AAC-LD
  ENC: WMA8
  DEC: MP3, WMA9

- Video/Image [5]
  ENC/DEC: MPEG4 (SP), H.264 (BP), JPEG
  DEC: MPEG2

Zur Sicherheit befindet sich eine Vielzahl der Packages auch noch einmal im Pfad „/home/praktikum/dvsdk_praktikum/codec_packages/“ um eventuellen Veränderungen der Websites vorzubeugen.

Die Bezeichnung „C64x+“ umfasst die Modelle DM644x, DM646x, OMAP3530, DM648, C6466 und C674x. Alle Codec-Packages sind mit XDC-Metadaten versehen, um Informationen für die XDC-Tools bereitzustellen (Namen, Versionsangaben, Wasserzeichen und einiges mehr).

2.2.2 Integration neuer Codecs

Im Unterverzeichnis „dm6446_dvsdk_combos_2_05/packages/ti/sdo/codecs/“ befinden sich jeweils in getrennten Ordnern alle derzeit verfügbaren Codec-Packages. Wesentliche Bestandteile sind dabei die XDC-Files sowie das Library-File „l64P“ im Ordner „lib/“, welchem bei der späteren Erstellung des x64P-Files große Bedeutung zukommt. Fehlt jene Bibliothek, ist das gesamte Codec-Package für diesen Zweck unbrauchbar. Spezifikationen über die Parameter des Coders sind häufig im Ordner „docs/“ zu finden.

Bei den im vorhergehenden Kapitel heruntergeladenen Codec-Packages handelt es sich um ausführbare Binärdateien, welche neben dem eigentlichen Codec einen kompletten Installer beinhalten. Ziel der Installation ist es das eingebettete Archiv-File mit den entsprechenden Nutzdateien an einen beliebigen Ort im Dateisystem abzulegen und anschließend zu entpacken (*.tar).

Die nachfolgende Anleitung geht davon aus, dass das gewünschte Codec-Package unter Redhat-Linux im Pfad „/tmp“ abgespeichert wurde.

Nach dem Öffnen eines Terminals in der VMware kann die Installation des neues Codecs begonnen werden. Als Beispiel wird hier die Installation eines G.711-Encoders vollzogen:

Setup-Datei ausführbar machen und starten

```
$ cd /tmp
```

```
$ mkdir codec
$ chmod +x ./c64xplus_g711_1_12_00_000_production.bin
$ ./c64xplus_g711_1_12_00_000_production.bin
```

Beim Ausführen der Binärdatei öffnet sich ein Installations-Menü. Als Zielpfad wird in diesem Beispiel das zuvor angelegte Verzeichnis „/tmp/codec“ verwendet:

Installations-Wizard durchlaufen

```
- Please select the installation language: German → OK
- Weitermachen → Ja
- Weiter → Lizenzbestimmungen akzeptieren (Häkchen) → Ja
- Durchsuchen → +/ → +tmp → codec → OK → Weiter
- Typical → Weiter
- Weiter → Installation läuft → Beenden
```

Nach erfolgreicher Installation folgt das Entpacken und Kopieren des Codec-Packages in die entsprechenden Ordner im DVSDK:

Gewonnene Archiv-Dateien entpacken und in den DVSDK-Ordner kopieren

```
$ cd /tmp/codec
$ ls
dm6446_g711dec_1_12_00_000_production.tar
dm6446_g711enc_1_12_00_000_production.tar
uninstall
$ tar xfv dm6446_g711enc_1_12_00_000_production.tar
$ cd dm6446_g711enc_1_12_00_000_production
$ cp -r packages/ti/sdo/codecs/g711enc/ /home/praktikum/
      dvsdk_2_00_00_22/dm6446_dvsdk_combos_2_05/
      packages/ti/sdo/codecs/
```

2.2.3 Vorbereiten der XDC-Tools

Sind alle gewünschten Codecs für das neue Combofile in das DVSDK integriert, kann mit dem nächsten Schritt begonnen werden. Dabei kommen die XDC-Tools zum Einsatz, welche dazu dienen die installierten Codecs automatisch zu erkennen und nach entsprechend manueller Auswahl mit Hilfe der CGTs Quelldateien erzeugen, aus denen sich im späteren Verlauf das eigentliche Combofile compilieren lässt.
Weitere Informationen zum Aufbau und zur Funktionsweise von XDC befinden sich im

Kapitel 5.1.

Ein globales XDC-Makefile dient dem Setzen der Pfade zu den Komponenten innerhalb des DVSDKs, etwa wo sich entsprechende Codec-Packages befinden. Zudem steuert die Operation „all“ innerhalb des Makefiles einen Compiliervorgang, welcher die grafische Oberfläche zur Auswahl der Codecs erzeugt.
Wird XDC zum ersten Mal nach einer Neuinstallation des DVSDKs ausgeführt, so muss vorher das passende Makefile aus dem Praktikums-Ordner kopiert werden (Anhang H):

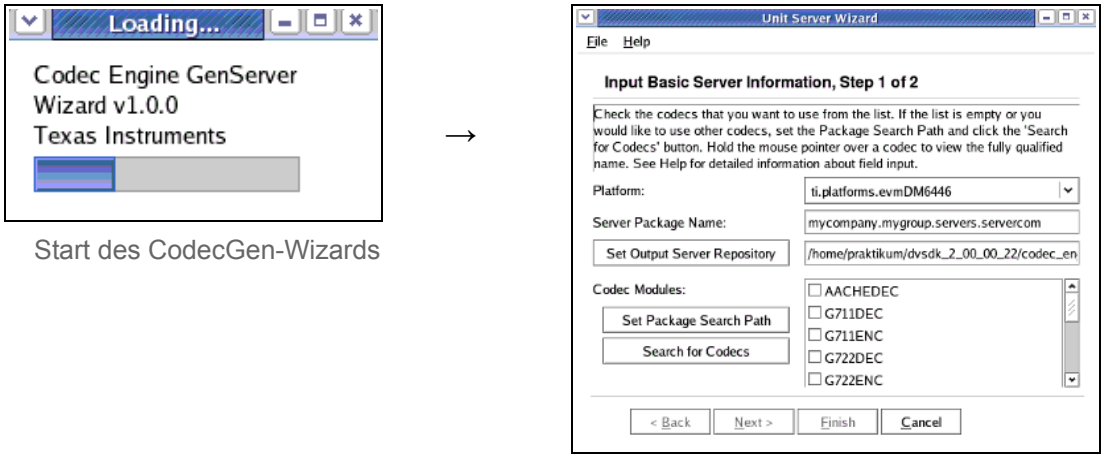
```
$ cp /home/praktikum/dvsdk_praktikum/configs/xdctools_Makefile
     ␣ /home/praktikum/dvsdk_2_00_00_22/xdctools_3_10_05_61/
     Makefile
```

Nun kann die grafische Oberfläche des GenServer Wizards gestartet werden. Dieser sucht im Pfad „/home/praktikum/dvsdk_2_00_00_22/dm6446_dvsdk_combos_2_05/packages/ti/sdo/codecs/“ nach Codec-Packages:

Codec Engine GenServer starten

```
$ mkdir /tmp/combo
$ cd /home/praktikum/dvsdk_2_00_00_22/xdctools_3_10_05_61/
$ make
```

Das Laden der Oberfläche kann durch den Compiliervorgang im Terminal verfolgt werden:



Start des CodecGen-Wizards

Oberfläche des CodecGen-Wizards

Anschließend kann die Einstellung der gewünschten Parameter erfolgen. In diesem

Beispiel sollen die zu erzeugenden Dateien in den zuvor angelegten Ordner „/tmp/combo“ gespeichert werden:

CodecGen-Wizard durchlaufen

```
- Server Package Name: testcombo
- Set Output Server Repository: /tmp/combo
- Codec(s) auswählen
- Next → Finish
- Would you like to save the values entered into the Codec
      Engine GenServer Wizard? → Nein
```

Im Zielverzeichnis wird ein neuer Ordner entsprechend des Namens angelegt („/tmp/combo/testcombo/“). Zur erfolgreichen Compilierung des x64P-Files müssen noch zwei Änderungen vorgenommen werden:

Zunächst müssen die fehlende Konfigurationsdatei „config.bld“ (beinhaltet die Pfade zu CGT) sowie das Makefile aus dem Praktikums-Verzeichnis entnommen und in das Zielverzeichnis kopiert werden (Anhänge I und J). Zu beachten sind die angegebenen Pfade innerhalb der Dateien, welche nur in der Komplettinstallation des DVSDKs 2.00 korrekt gesetzt sind und andernsfalls geändert werden müssen:

```
$ cp /home/praktikum/dvsdk_praktikum/configs/integration_neuer_
      codecs_config.bld /tmp/combo/testcombo/config.bld
$ cp /home/praktikum/dvsdk_praktikum/configs/integration_neuer_
      codecs_Makefile /tmp/combo/testcombo/Makefile
```

Zum Schluss müssen in der Datei „codec.cfg“ für alle eingebundenen Codec-Module die Memory-Optionen von „undefined“ auf „'DDR2'“ umgestellt werden. Das muss nicht von Hand erfolgen, sondern man kann sich der Systemtools von Linux bedienen:

```
$ cd /tmp/combo/testcombo
$ sed -e "s/undefined/'DDR2'/g" -i "codec.cfg"
```

Nun ist das Verzeichnis bereit für die Erzeugung des Combofiles. Alle Compile- und Linkervorgänge werden angezeigt und sollten bis zum Ende sauber durchlaufen:

```
$ make
```

Das Resultat ist ein x64P-Combofile entsprechend des am Anfang festgelegten Namens (kann auch im Nachhinein ohne Konsequenzen beliebig umbenannt werden). Das fertige Combofile ist jetzt bereit, um auf das Filesystem des Boards kopiert zu werden:

```
$ cp /tmp/combo/testcombo/testcombo.x64P
      ␣ /home/praktikum/workdir/filesys/<gewünschtes Ziel>
```

2.2.4 Testen des neuen Combofiles

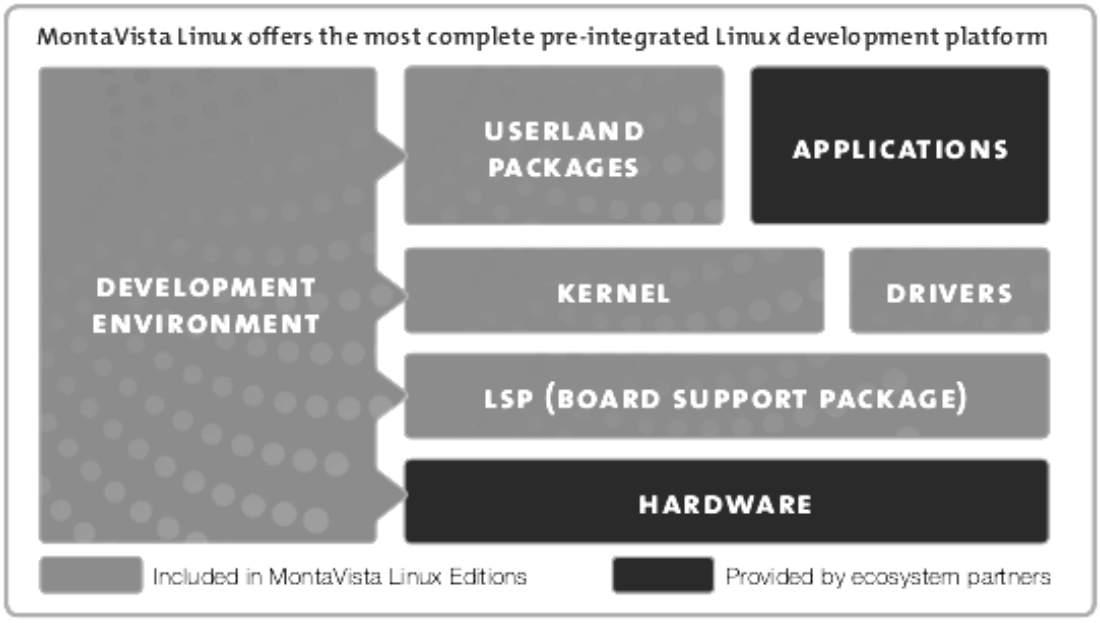
Für einen kurzen Funktionstest bietet sich die Erzeugung eines neuen Projekts an (siehe Kapitel 6.2). Darin könnten beispielsweise die eingebundenen Codecs gemeinsam initialisiert und wieder geschlossen werden.

3 MontaVista Linux

3.1 Allgemeine Betrachtung

3.1.1 Aufgaben des Betriebssystems

Die Linux-Umgebung des Boards dient als Arbeitsplattform für den Anwendungsentwickler. Dazu gehört zum einen die Herstellung einer Verbindung zum DSP über Shared Memory, als auch die Initialisierung der Hardware und externen Geräte durch den Kernel und dessen Treiber. Nicht zuletzt stellt MontaVista Linux auch Tools und Packages bereit oder erlaubt es diese zu installieren, wie etwa „gdb“ für Debuggingaufgaben.
Die in der folgenden Abbildung hellgrau dargestellte Bereiche werden durch MontaVista Linux als Gesamtpaket zur Verfügung gestellt:



MontaVista Linux Plattform
[6] „Most Powerful Linux Environment for Intelligent Devices“

Das Linux Support Package (LSP) wurde nicht von MontaVista entwickelt, jedoch zur Verwendung auf dieser Plattform verifziert. Es enthält eine Vielzahl von Packages für Zielumgebungen wie x86, ARM oder MIPS (vgl. [6] „Most Powerful Linux Environment for Intelligent Devices“).

3.1.2 Aufbau des Dateisystems

MontaVista Linux folgt als Distribution des Linux-Betriebssystems den klassischen Regeln der Verwaltung von Dateien- und Ordnerstrukturen. Ein Blick auf die oberste Ebene des Filesystems zeigt den gewohnten Aufbau einer Linuxumgebung:

```
$ ls /
bin   dev  home  media  opt   root  share  sys  usr
boot  etc  lib   mnt    proc  sbin  srv    tmp  var
```

Die Bedeutung der Ordner ist selbsterklärend. Das Stammverzeichnis des Benutzers „praktikum“ befindet sich in „home“. Vom Kernel erkannte Geräte werden unter „dev“ als Device-Nodes angelegt. In „tmp“ gespeicherte Dateien und Ordner werden bei einem Neustart automatisch gelöscht. Weiterführende Erklärungen zum Dateisystem von Linux finden sich auf den Websites der Universität Münster [7].

3.1.3 Neuerungen in Version 5

Mit der Version 5 des Betriebssystems MontaVista hat der gleichnamige Hersteller einige Verbesserungen implementiert. Dazu zählt zum einen die standardmäßige Installation eines Secure Shell (SSH/SFTP) Servers für den vereinfachten Zugriff auf das Board über ein Netzwerk, sowie die für DVSDK 2.00 notwendige Aktualisierung des Kernels auf die Version 2.6.18 und damit verbundene Neuerungen, wie etwa Support für die Advanced Linux Sound Architecture (ALSA).

3.2 Installation von Programmen

Für die Installation eines Programms gibt es unter zahlreichen Linux-Distributionen einen Paketmanager, welcher automatisch eine vorcompilierte Binärdatei und deren Abhängigkeiten installiert. Unter MontaVista Linux existiert eine solche paketbasierte Verwaltung von Software nicht, es müssen alle benötigten Programme selbst von Hand aus den Quellen compiliert werden.

Am Beispiel des Tools „gdbserver“ soll die Installation nachvollzogen werden. Zunächst muss das Archiv mit dem Sourcecode von „gdb“ aus dem Internet von beliebiger Quelle

heruntergeladen und entpackt werden:

Entpacken des Sourcecodes

```
$ cd /tmp
$ wget "http://ftp.gnu.org/gnu/gdb/gdb-7.2.tar.gz"
$ tar xfzv gdb-7.2.tar.gz
```

Anschließend muss die Configure-Routine für einen Cross-Compile konfiguriert und dabei das Installationsverzeichnis auf das NFS-Share gesetzt werden:

Konfigurieren für Cross-Compilierung

```
$ cd /tmp/gdb-7.2/gdb/gdbserver
$ CC=/opt/mv_pro_5.0/montavista/pro/devkit/arm/v5t_le/bin/
      arm_v5t_le-gcc ./configure --host=armv5tl-montavista-linux-
      gnueabi --prefix=/home/praktikum/workdir/filesys
```

Mit „make“ werden die Binärdateien erzeugt und anschließend als „root“ mit „make install“ auf das NFS-Verzeichnis kopiert:

Compilieren der Binaries

```
$ make
$ su
# make install
```

Nun befindet sich im angegebenen Prefix-Pfad „/home/praktikum/workdir/filesys“ das Binary mit seinen Libraries in den entsprechenden Unterverzeichnissen, etwa „/usr/sbin“ und „/lib“.
Als Hinweis sei gegeben, dass diese Anleitung nur Demonstrationszweck hat, um auf dem Board eine Fremdsoftware zu installieren. Soll gdb tatsächlich eingesetzt werden, so muss auf der VMware noch der passende gdb-Client compiliert werden (siehe dazu Kapitel 6.3.1).

3.3 Serielle Verbindung

Auf das Board kann nicht nur per SSH oder Telnet zugegriffen werden, sondern ebenso mittels einer seriellen Verbindung über den COM-Port. Dabei wird ab dem Bootloader ein virtuelles Terminal auf der seriellen Verbindung bereitgestellt, welches mit dem Systemtool „minicom“ angezeigt werden kann. Entsprechend des Setups des Bootvorgangs in Kapitel 2.1.6 müssen die Einstellungen des Modems, insbesondere Baudrate und Bitparameter, berücksichtigt werden. Die notwendige Konfiguration zu „minicom“ befindet sich in der Datei „/home/praktikum/.minirc.dfl“:

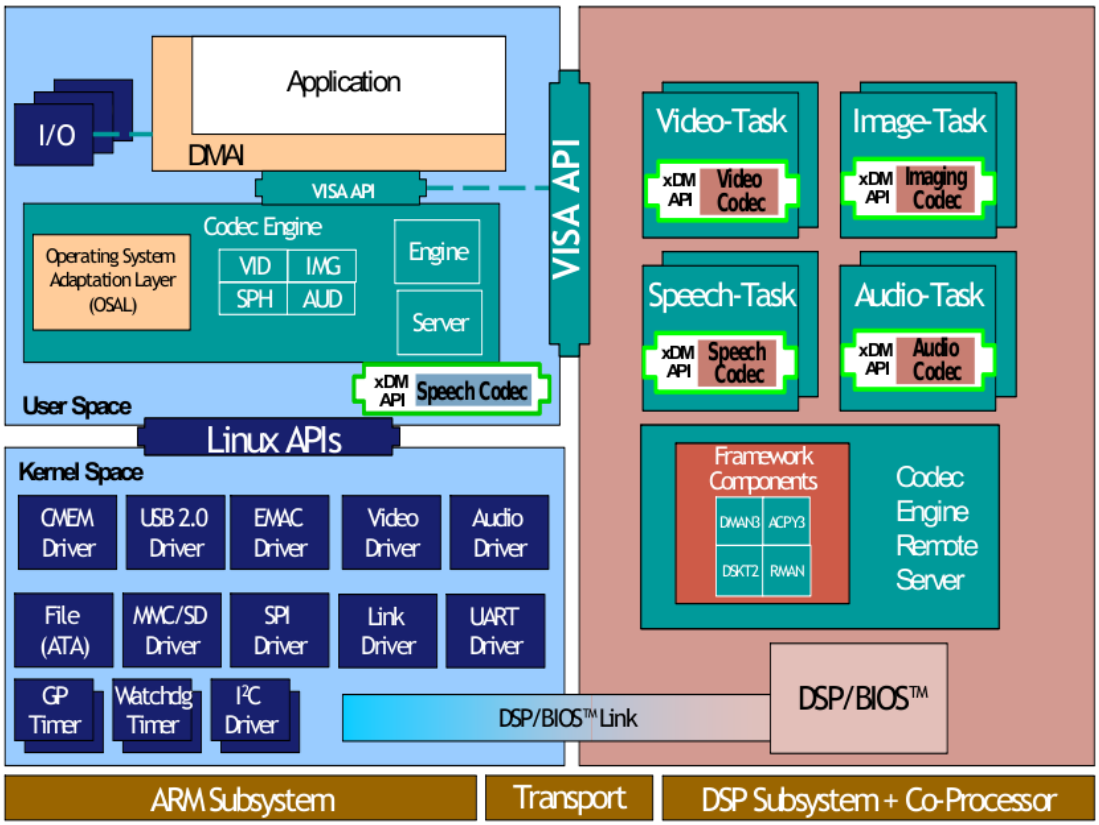
Minicom-Konfiguration

```
# Machine-generated file - use "minicom -s" to change parameters.
pr port             /dev/ttyS0
pu baudrate         115200
pu bits             8
pu parity           1
pu stopbits         1
pu minit
pu mreset
pu mdialsuf3
pu mhangup
pu rtscts           No
```

4 Programmieren mit dem VISA API

4.1 Einordnung des APIs

Ein API dient Anwenderprogrammen als Schnittstelle zu Systemen und Abläufen, welche sich im Hintergrund verbergen. Oftmals vereinfachen API-Funktionen komplexere Schritte aus dem Subsystem und stellen eine Bibliothek für Hochsprachen bereit (vgl. [8]).
Das VISA API von Texas Instruments dient der Kommunikation zwischen der Anwenderebene (Application) und der Codec Engine im „User Space“ des Betriebssystems (virtueller Speicher für Anwendungen). Die eigentliche Ein- und Ausgabe von Datenströmen über angeschlossene Geräte wie Mikrofon oder Lautsprecher steuert der Linux-Kernel über entsprechende Treiber im „Kernel Space“:



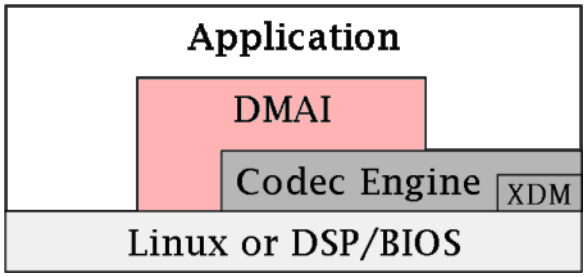
Einordnung des APIs zwischen DSP und Linux
[2] Kapitel 4.1.2

Der Transport zwischen beiden Subsystemen geschieht über „Shared Memory“ auf fest definierten Speicherplätzen im „Kernel Space“ mittels der zuvor geladenen Kernelmodule „cmemk“ und „dsplink“.

4.2 DMAI und Codec Engine Framework

4.2.1 Einordnung des DMAIs

In der im vorangegangenen Abschnitt gezeigten Grafik befindet sich zwischen dem Anwenderprogramm und der Codec Engine noch eine zusätzliche Komponente, das Davinci Multimedia Application Interface (DMAI, vgl. [9]), welches einen weiteren Abstraktionslayer beim Zugriff auf die Codec Engine darstellt:



DMAI Blockdiagramm
[9] „Introduction“

Verwendet ein Programmierer in seiner Application eine DMAI-Funktion, so wird diese beim Compilieren auf spezifische API-Funktionen für ein bestimmtes Ziel-Board übersetzt (vgl. [9] „Introduction“). Als Ziel verfolgt der DMAI-Ansatz eine verbesserte Portabilität von Programmen zwischen verschiedenen Plattformen, sowie eine insgesamt vereinfachte Benutzung der Codec Engine durch entsprechende Abstraktion der API-Funktionen. Entwicklungen mit dem DMAI sollen schneller zum Ergebnis führen, denn es müssen nicht alle Details selbst programmiert und durchdacht werden (vgl. [9] „Benefits of Using DMAI“).
In der Praxis hat sich gezeigt, dass das DMAI durchaus nützlich sein kann, um mit wenigen Zeilen C-Code eine Codierungsvorschrift zur Anwendung zu bringen. Andererseits bringt der hohe Abstraktionsgrad wenig Flexibilität, sodass gezielte Operationen auf Bitebene durch den indirekten Zugriff auf API-Komponenten nur erschwert möglich sind ([10] Abschnitt 1.2).

Zur Verdeutlichung des Grades der Abstraktionen soll das Gerät „/dev/dsp“ (die Soundausgabe per Open Sound System (OSS) unter Linux) initialisiert werden. Unter Verwendung von DMAI-Komponenten des Header-Files „<ti/sdo/dmai/Sound.h>“ gestaltet sich der Sourcecode sehr kurz:

Indirekter Hardwarezugriff mit DMAI-Komponenten

```
Sound_Handle hSound = NULL;
Sound_Attrs sAttrs = Sound_Attrs_MONO_DEFAULT;
hSound = Sound_create(&sAttrs);
```

Eine Einstellungsmöglichkeit des Gerätes gibt es über die Variable „sAttrs“, welche vordefinierte Konstanten vom Typ „Sound_Attrs“ erwartet. Nach der Öffnung steht eine Variable vom Typ „Sound_Handle“ zur weiteren Verarbeitung unter DMAI bereit. Der Wert „hSound“ lässt sich nicht als klassischer File-Deskriptor betrachten und ist zu anderen C-Funktionen nicht als solcher kompatibel.

Der direkte Zugriff auf die Hardware lässt im Gegensatz dazu deutlich mehr Spielraum und stellt mit „open()“ nach Unix-Philosophie einen Deskriptor vom Typ „int“ zur Verfügung:

Direkter Hardwarezugriff mit I/O-Settings

```
int hSound=0, channels=1, sampleRate=8000, format=AFMT_S16_LE;
hSound = open("/dev/sound/dsp", O_WRONLY);
ioctl(hSound, SNDCTL_DSP_CHANNELS, &channels);
ioctl(hSound, SNDCTL_DSP_SPEED, &sampleRate);
ioctl(hSound, SNDCTL_DSP_SETFMT, &format);
```

Je nach Anwendungsfall kann die eine oder die andere Art der Initalisierung von Vorteil sein. Das Mischen von File-Deskriptoren und Sound-Handles im weiteren Verlauf, etwa bei einer Codierungsaufgabe, ist nicht möglich.

4.2.2 Praktisches Beispiel

Zwei vollständige Demo-Programme mit jeweils identischer Funktion befinden sich im Praktikums-Ordner „/home/praktikum/dvsdk_praktikum/beispiele/g711dec_*“ (Anhänge B und C). Diese dienen der Herausarbeitung wesentlicher Unterschiede im Funktionsablauf und Umfang für das Beispiel der Decodierung einer G.711-ALAW codierten Datei. Der Ordner „g711dec_dmai“ enthält dabei ein Demo-Programm mit DMAI-Komponenten, während „g711dec_ce“ den klassischen Zugriff auf das API zeigt. Die Quelltexte dokumentieren mit Kommentaren die einzelnen Abläufe.

4.3 Wichtige Grundfunktionen

4.3.1 Runtime-Initialisierung

Bevor die Funktionen der Codec Engine API benutzt werden können, muss diese zusammen mit allen Modulen initalisiert werden ([11] Abschnitt „Functions“). Das gilt auch für Anwendungen mit DMAI-Komponenten. Im Fehlerfall ist der Rückgabewert der Funktion der NULL-Pointer:

```
#include <ti/sdo/ce/CERuntime.h>
CERuntime_init();
```

Analog dazu können mit „CERuntime_exit()“ die Module wieder geschlossen werden.

4.3.2 Engines

Um eine Codec Engine zu öffnen, steht die API-Funktion „Engine_open()“ bereit, welche einen Identifikator vom Typ „Engine_Handle“ zurückgibt. Die Funktion kann mehr als ein Mal auf die selbe Engine aufgerufen werden und erzeugt dabei immer wieder neue Identifikatoren. Bei der Arbeit mit Threads muss jedes Child mit einem eigenen Handle arbeiten ([11] Abschnitt „Engine_open“):

Codec Engine öffnen

```
#include <ti/sdo/ce/Engine.h>

Engine_Handle hEngine = NULL;
hEngine = Engine_open("Meine_Engine", NULL, NULL);
```

Im Fehlerfall ist der Rückgabewert der Funktion „Engine_open()“ der NULL-Pointer und bei Erfolg ist der Rückgabewert 0 (Konstante „Engine_EOK“). Zur besseren Fehleranalyse kann ein Handle vom Typ „Engine_Error“ übergeben werden, welches als Integer-Wert den Fehlercode enthält. Um diese Zahl in einen verständlichen Fehler aufzulösen, steht in der Header-Datei „Praktikum.h“ die Funktion „DSP_Errors()“ bereit:

Fehler-Codes abfangen

```
#include "Praktikum.h"
```

```
Engine_Error hEngineError = NULL;
hEngine = Engine_open("Meine_Engine", NULL, &hEngineError);
printf("Fehler: %s\n", DSP_Errors(hEngineError));
```

Geschlossen wird eine Engine mit der API-Funktion „Engine_close()“, welche als Parameter das Engine Handle erwartet.

4.3.3 Codecs

Nachdem in Kapitel 4.3.2 eine Codec Engine erfolgreich geöffnet wurde, können dieser eine Vielzahl von Codierungsalgorithmen zugewiesen werden. Das folgende Beispiel erstellt einen G.711-Decoder auf einem vorhandenen Engine-Handle „hEngine“ unter Verwendung des DMAIs.
Die verwendete Bezeichnung „g711dec“ für den Decoder muss zuvor im CFG-File definiert werden (siehe Kapitel 6.1.1):

G.711-Decoder mit DMAI-Komponenten

```
#include <ti/sdo/dmai/Dmai.h>
#include <ti/sdo/dmai/ce/Sdec1.h>

Sdec1_Handle hDecode = NULL;
SPHDEC1_Params defaultParams  = Sdec1_Params_DEFAULT;
SPHDEC1_DynamicParams defaultDynParams =
      Sdec1_DynamicParams_DEFAULT;
hDecode = Sdec1_create(hEngine, "g711dec", &defaultParams,
      &defaultDynParams);
```

Die DMAI-Funktion „Sdec1_delete()“ schließt den übergebenen Decoder.
Der gleiche Vorgang ohne die Verwendung des DMAIs für G711DEC-ALAW gestaltet sich ähnlich:

G.711-Decoder mit CE-Komponenten

```
#include <ti/sdo/ce/speech1/sphdec1.h>
#include <ti/xdais/dm/ispeech1_pcm.h>
```

```
SPHDEC1_Handle hDecode = NULL;
SPHDEC1_Params hDecodeParams;

hDecodeParams.size = sizeof(hDecodeParams);
hDecodeParams.compandingLaw = ISPEECH1_PCM_COMPAND_ALAW;
hDecode = SPHDEC1_create(hEngine, "g711dec", &hDecodeParams);
```

Die API-Funktion „SPHDEC1_delete()“ schließt den übergebenen Decoder.
Das Beispiel kann für einen Encoder in gleicher Weise genutzt werden. Dazu müssen die Komponentennamen nur von „dec“ durch „enc“ ersetzt werden.

4.3.4 Speichermanagement

Jede Ein- oder Ausgabe von Nutzdaten auf dem API erfolgt über Buffer. Diese müssen bei Verwendung des DMAIs mit der Funktion „Buffer_create()“ allokiert und mit „Buffer_delete()“ geschlossen werden. Die Angabe der Größe des anzufordernten Speichers erfolgt in Byte. Liegt bereits ein geöffnetes Code Handle aus Kapitel 4.3.3 vor, kann dessen Größe beispielsweise für einen Decoder mit „Sdec1_getOutBufSize()“ (Rückgabewert ist vom Typ „Int“) automatisch ermittelt werden:

Speicherallokierung mit DMAI

```
#include <ti/sdo/dmai/Buffer.h>

Buffer_Handle hOutBuf = NULL;
Buffer_Attrs bAttrs = Buffer_Attrs_DEFAULT;
hOutBuf = Buffer_create(Sdec1_getOutBufSize(hDecode), &bAttrs);
```

Die Größe des Ausgabepuffers beträgt bei einem Codec Handle „hDecode“, welches mit G711DEC-ALAW initialisieriert wurde, genau 160 Byte:

```
8 kHz Abtastrate → 8000 Samples/s
20ms Sprache ≙ 160 Samples
1 Sample G.711 ≙ 8 Bit PCM → 8*160 Bit = 160 Byte
```

(vgl. [12] „PCM-Standards der Sprach- und Audiosignalcodierung“)

Im folgenden Beispiel wird der benötigte Speicherplatz manuell ohne die Nutzung des

DMAIs allokiert. Der Pufferspeicher ist anders als bei DMAI kein Handle des Typs „Buffer_Handle“, sondern enthält echte Werte vom Typ „Int8“, was die Manipulation des Datenstroms später deutlich erleichert:

Blockweise Speicherallokierung über calloc()

```
#include <ti/sdo/ce/osal/Memory.h>

static XDAS_Int8 *hOutBuf;
hOutBuf = (XDAS_Int8 *)Memory_contigAlloc((160 * sizeof(Int8)),
      Memory_DEFAULTALIGNMENT);
```

4.3.5 Blockweises Lesen von Datenströmen

Ein Codierungsprozess unterliegt der Speicherung von Samples, also Proben eines analogen Signals. Bei der Wiedergabe werden diese Proben blockweise (je nach Abtastrate) wieder ausgegeben. Nach dem G.711-Standard entspricht die Länge eines solchen Blocks 20ms, also 160 Bytes PCM-Samples (vgl. Kapitel 4.3.4).
Unter DMAI erfolgt das Lesen der Datei „sample.g711“ über das eigene Handle „Loader“ auf einen bekannten Puffer „hInBuf“:

Lesen mit DMAI-Komponenten

```
#include <ti/sdo/dmai/Loader.h>
Loader_Handle hLoader = NULL;
Loader_Attrs lAttrs = Loader_Attrs_DEFAULT;

lAttrs.readSize = Sdec1_getInBufSize(hDecode);
lAttrs.readBufSize = lAttrs.readSize * 2;
hLoader = Loader_create("sample.g711", &lAttrs);
Loader_prime(hLoader, &hInBuf);
```

Mit Codec Engine Komponenten erfolgt das Lesen mit dem Systemruf „read()“. Der Vorteil dieser Methode ist, dass es sich bei den gelesenen Bytes um „echte“ Int8-Zahlen handelt und diese für Testzwecke manipuliert werden können:

Lesen mit Unix-Systemruf „read()“

```
#include <stdio.h>
```

```
int in;
static XDAS_Int8 *inBuf;

in = open("sample.g711", O_RDONLY);
read(in, inBuf, (160 * sizeof(Int8)));
```

Beim Lesen von Datenströmen, etwas aus Netzwerk-Sockets oder Arrays, kann zudem der Systemruf „memcpy()“ zum blockweisen Lesen mit einem Offset verwendet werden. Das folgende Beispiel liest innerhalb der Schleife immer wieder 160 Byte Characters aus „hSource“ nach „hInBuf“ (mit Typkonvertierung). Dazu wird zu der Adresse von „hSource“ ein Offset aus der Anzahl der bereits gelesenen Bytes, multipliziert mit deren Größe im Speicher, addiert:

Lesen mit Unix-Systemruf „memcpy()“

```
unsigned int hSamples=160, hOffset=0;
while (1) {
  memcpy(hInBuf, (XDAS_Int8 *) (sizeof(unsigned char) * hOffset
      + hSource), hSamples);
  hOffset += hSamples;
}
```

4.3.6 Codierungsprozess

Die DMAI-Funktion „Sdec1_process()“ kann auf einem gegebenen Decode-Handle „hDecode“ einen Eingabepuffer „hInBuf“ lesen und einen Ausgabepuffer „hOutBuf“ mit den decodierten Daten befüllen:

Decodieren mit DMAI-Komponenten

```
#include <ti/sdo/dmai/Dmai.h>
#include <ti/sdo/dmai/ce/Sdec1.h>

Sdec1_process(hDecode, hInBuf, hOutBuf);
```

Unter Verwendung der API-Funktionen für die Codec Engine ist deutlich mehr Initialisierungsaufwand notwendig. Für einen Decode-Prozess dienen Int8-Zahlen als PCM-Werte, während Char-Ketten den Raw-Datenstrom für die Soundausgabe bilden:

Decodieren mit CE-Komponenten

```
#include <ti/sdo/ce/speech1/sphdec1.h>

static XDAS_Int8 *hInBuf, *hOutBuf;
SPHDEC1_InArgs decInArgs;
SPHDEC1_OutArgs decOutArgs;
XDM1_SingleBufDesc inBufDesc, outBufDesc;
XDAS_Int32 bufSize = 160;

inBufDesc.bufSize = bufSize * sizeof(short);
inBufDesc.buf = hInBuf;
outBufDesc.bufSize = bufSize * sizeof(char);
outBufDesc.buf = hOutBuf;

read(hInput, hInBuf, bufSize);
SPHDEC1_process(hDecode, &inBufDesc, &outBufDesc, &decInArgs,
      &decOutArgs);
```

Die aus „hInput“ gelesenen 160 Byte werden in den Eingabepuffer „hInBuf“ kopiert und durch „SPHDEC1_process()“ in den Ausgabepuffer „hOutBuf“ geschrieben. Der File-Deskriptor „hInput“ kann von einer beliebigen Quelle stammen.

4.3.7 Audiogeräte

Bei der Arbeit mit einem Audiogerät gibt es aus Sicht der Betriebssystems zwei Möglichkeiten der Nutzung, es kann entweder gelesen oder geschrieben werden. Unter Linux kann dazu das Gerät „/dev/sound/dsp“ mit dem Systemruf „open()“ für Lesen oder Schreiben über entsprechende Flags geöffnet werden:

Schreiben auf das Gerät (Lautsprecher)

```
int hSound;
hSound = open("/dev/sound/dsp", O_WRONLY);
```

Lesen vom Gerät (Mikrofon)

```
int hSound;
hSound = open("/dev/sound/dsp", O_RDONLY);
```

Entsprechende Einstellungen etwa für Kanal und Samplerate befinden sich bereits im Kapitel 4.2.1.

4.3.8 Paralles Arbeiten mit Threads

Oftmals sollen unterschiedliche Codierungsaufgaben gleichzeitig ablaufen, etwa bei duplexer Übertragung von Echtzeitstreams in einem VoIP-System. Auf der gleichen Codec Engine müssen in diesem Fall eine Codierungs- sowie eine Decodierungsaufgabe parallel ausgeführt werden.
Unter Linux können von einem Prozess („Vater“) weitere Prozesse („Kinder“) mit dem Systemruf „fork()“ abgespaltet werden:

Erzeugen eines Forks unter Linux

```
#include <sys/wait.h>

pid_t pid_encode;
switch (pid_encode = fork()) {
      case -1:
            printf("Thread nicht gestartet.\n");
      case 0:
            printf("Thread gestartet.\n");
            encode();
            exit(0);
}
```

Nach der Erzeugung eines Kinder-Prozesses mit der Nummer „pid_encode“ wird die Funktion „encode()“ gestartet und nach deren Ausführung der Kinder-Prozesser mit „exit()“ beendet. Innerhalb der genannten Funktion kann dann die Codierungsaufgabe analog zu Kapitel 4.3.6 ausgeführt werden.

API-Aufrufe erfolgen auf dem DVSDK fast ohne Ausnahme blockierend. Wird dem DSP eine Aufgabe zugewiesen, so kehrt die API-Funktion erst nach deren Ausführung zurück. Für einen Kinder-Prozess stellt sich damit das Problem, dass der möglicherweise unendliche Datenstrom den Prozess blockiert und das Hauptprogramm nicht auf dessen Rückkehr warten kann, um sich selbst zu beenden, etwa weil der Benutzer das Programm

gern schließen möchte. Aus diesem Grund müssen Kinder-Prozesse vom Vater-Prozess aus mit einem Kernel-Signal geschlossen werden.
Im folgenden Beispiel würde ein Tastendruck den Kinder-Prozess beenden:

Beenden des Forks im Hauptprogramm

```
while (1) {
      if (getchar()) {
            kill(pid_encode, SIGKILL);
      }
}
```

Nach dem Schließen aller Threads ist darauf zu achten, dass sämtliche initialisierte API-Funktionen wieder sauber entfernt werden, um den DSP nicht zu blockieren.

4.4 Wesentliche Unterschiede in der Version 2.00

4.4.1 Allgemeine Verbesserungen

Das Changelog der Versionshistorie auf den Websites von Texas Instruments beinhaltet einige Neuerungen. Zudem treten beim praktischen Einsatz interessante Features auf:

- erweiterte Demos für CE und DMAI in „codec_engine_.../examples“
- Umbenennung der API-Funktionen (siehe Kapitel 4.4.2)
- getrennte Codec-Servers mit Libraries (*.l64P) für XDC
- XDC-Tools nutzbar für Erstellung eigener Combo-Files

4.4.2 Umbenennung der API-Funktionen

Ein Nebeneffekt der Einführung des DMAIs ist, dass die klassischen API-Funktionen für das Codec Engine-Framework umbenannt werden mussten. In der Version 1.00 des DVSDKs sah die Erstellung eines Decoders wie folgt aus:

Öffnen eines Decoders für G.711 nach DVSDK Version 1.00

```
#include <ti/sdo/ce/speech/sphdec.h>
SPHDEC_Handle hDecode;
```

```
SPHDEC_Params hDecodeParams;

hDecodeParams.size = sizeof(SPHDEC_Params);
hDecodeParams.compandingLaw = ISPEECH_ALAW;
hDecode = SPHDEC_create(hEngine, "g711dec", &hDecodeParams);
```

Für Codierungsfunktionen kann man grob davon ausgehen, dass der neue Funktionsname unter DVSDK Version 2.00 eine „1“ enthält:

Öffnen eines Decoders für G.711 nach DVSDK Version 2.00

```
#include <ti/sdo/ce/speech1/sphdec1.h>
#include <ti/xdais/dm/ispeech1_pcm.h>
SPHDEC1_Handle hDecode = NULL;
SPHDEC1_Params hDecodeParams;

hDecodeParams.size = sizeof(hDecodeParams);
hDecodeParams.compandingLaw = ISPEECH1_PCM_COMPAND_ALAW;
hDecode = SPHDEC1_create(hEngine, "g711dec", &hDecodeParams);
```

4.5 Praktikumsversuch

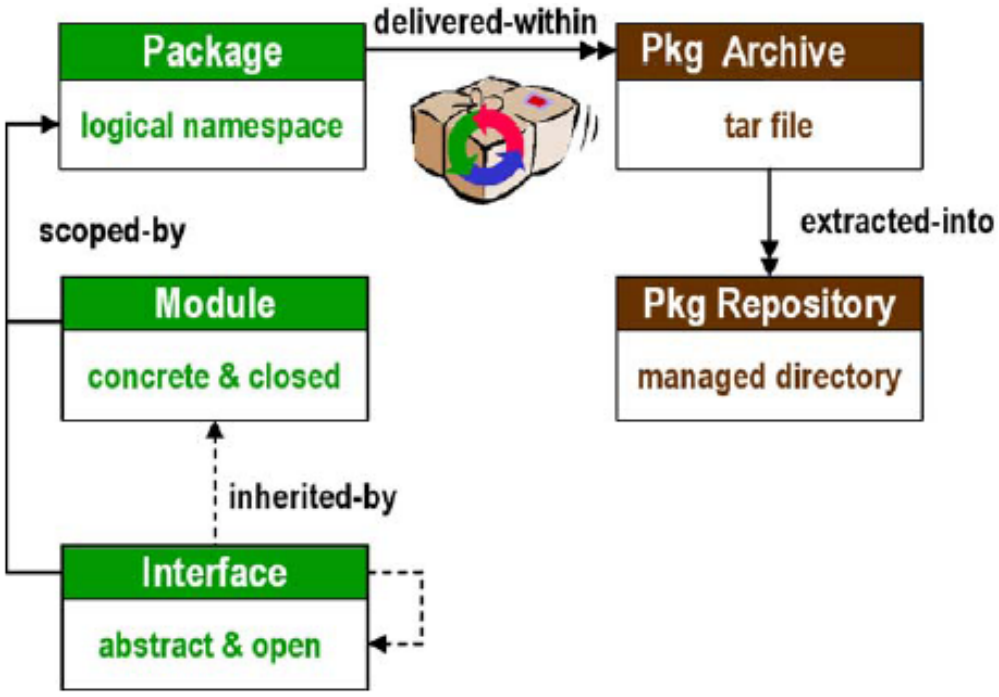
Das Praktikum „DVEVM“ soll Studenten die Möglichkeit bieten an einem DSP mit Codecs zu experimentieren und das Bewusstsein gegenüber der digitalen Signalverarbeitung zu erweitern. Grundlegendes Verständnis der Programmiersprache C sowie Kenntnisse über die Funktionsweise eines Sprachcodecs werden vorausgesetzt.
Die Versuchsunterlagen in Form von Quellcodes mit Lösungen sowie der Versuchsanleitung als PDF-Datei befinden sich im Praktikums-Ordner unter dem Pfad „/home/praktikum/dvsdk_praktikum/beispiele/g711dec_praktikum“ (Anhang D) bzw. „g711dec_praktikum_loesung“.

5 XDC

5.1 Definition und Aufbau

XDC bezeichnet eine Sammlung von Packages aus wiederverwendbaren Softwarekomponenten für einen Anwendungsentwickler, welche speziell für Embedded Systems und deren besondere Anforderungen im Bereich Echtzeit entwickelt wurden (vgl. [13]). Der modulare Aufbau der Pakete erlaubt es verschiedenen Produktentwicklern ihre jeweils eigenen XDC-Komponenten für Anwender bereitzustellen, wie etwa Codec-Packages. Die darin enthaltenen Metadaten zur Beschreibung des Pakets dienen dem XDC-Framework zur Erkennung bestimmter Eigenschaften wie Wasserzeichen oder Memoryoptions.

Das gesamte DVSDK ist aus XDC-Komponenten aufgebaut und bedient sich folgendem Aufbau:



XDC-Terminologie
[13] Kapitel 1.2

Folgt man etwa der Anleitung aus Kapitel 2.2.2 zur Integration eines neuen Codecs, so wird man genau dieses Verhalten nachvollziehen können. Die Setupdatei stellt ein Archiv im tar-Format bereit (Abbildung rechte Seite), welches in entsprechende Verzeichnisse entpackt werden muss.

5.2 Benutzung der Packages

XDC-Komponenten können in eigenen Projekten über das CFG-File mit den Funktionen *xdc.useModule()* oder *xdc.loadPackage()* eingebunden werden. Die Notierung der Pfade zu den Paketen erfolgt über Punkte, statt der üblichen Schrägstriche (vgl. Kapitel 6.1.1):

```
var G711ENC = xdc.useModule('ti.sdo.codecs.g711enc.ce.G711ENC');
```

Das Root-Verzeichnis für den angegebenen Pfad ist die im Makefile definierte Variable „XDC_PATH“, welche in das Verzeichnis „packages“ der jeweiligen XDC-Komponenten zeigt. Der Pfad zum XDC-Framework selbst wird in der globalen Datei „Rules.make“ im DVSDK-Verzeichnis durch die Variable „XDC_INSTALL_DIR“ festgelegt, welche vom Makefile des Projekts inkludiert wird.

5.3 XDC-Datentypen unter C

Bei der Arbeit mit dem DSP kommt das VISA API zum Einsatz, dessen Funktionen bestimmte Argumente, wie etwa Eingabepuffer, erwarten. Häufig werden dabei nicht klassische Variablentypen, wie sie unter C verwendet werden, erwartet:

Definition eines Eingabepuffers

```
static Int8 *inBuf;
XDM1_SingleBufDesc inBufDesc;
inBufDesc.buf = inBuf;
```

Die folgende Tabelle zeigt die wichtigsten Äquivalenzen zwischen XDC- (links) und den klassischen C99-Datentypen (rechts):

XDC-/C99-Datentypen

Int	int
Int8	int_least8_t oder int_fast8_t
Int16	int_least16_t oder int_fast16_t
Int32	int_least32_t oder int_fast32_t
Bool	unsigned short
Ptr	void *

String	char *
Char	char
Double	double
Float	float
Long	long

[13] Kapitel 2.3.4 Tabelle 2-1

Die Definition der Typen erfolgt in der Header-Datei „xdc/std.h“, welche einfach mittels der C-Direktive „#include“ benutzt werden kann.

6 Erstellen und Verwalten eigener Projekte

6.1 Aufbau eines Projektordners

6.1.1 CFG-File

Ein Projekt, welches die Komponenten des DSPs benutzt, wird sich in der Regel des APIs und damit der XDC-Funktionen bedienen. Das Einbinden der Header-Files innerhalb des C-Quellcodes reicht dazu nicht aus. Es muss eine Definition der XDC-Komponenten außerhalb in einer separaten Konfigurationsdatei (hier genannt „CFG-File“) vorgenommen werden.
Bei der Compilierung des Projekts werden die im CFG-File definierten Komponenten nachgeladen und daraus ein eigener Package-Ordner erzeugt. Der Name des CFG-Files muss zwingend dem Schema „<Projektname>.cfg“ entsprechen, da im Makefile (siehe Kapitel 6.1.2) das CFG-File mit der Variable „XDC_CFGFILE“ auf den Namen des aktuellen Ordners konfiguriert wird. Mehr Informationen zu XDC befinden sich im Kapitel 5.

Die Notierung des CFG-Files folgt üblichen Regeln einer Programmiersprache. Die Definition einer Variablen geschieht durch das Terminalsymbol „var“. Dieser können mit einem Punkt getrennt abhängige Eigenschaften zugewiesen werden, ähnlich des Typs „struct“ unter C:

```
var A = xdc.useModule('ti.sdo....');
A.server = „...“;
```

Eine vollständige Definition wird mit Semikolon beendet, Aufzählungen werden mit einem Komma abgetrennt und können innerhalb einer Gruppe auftreten, welche mit „{ }“ oder „[ ]“ beschrieben wird.

Beim Laden der Module durch „xdc.useModule()“ muss die Position des gewünschten Moduls angegeben werden. Dabei wird eine Notation verwendet, welche statt der im Dateisystem üblichen Schrägstriche einen Punkt erwartet:

```
var G711ENC = xdc.useModule('ti.sdo.codecs.g711enc.ce.G711ENC');
```

Die möglichen Präfixe zu der definierten Ordnerstruktur „ti/sdo/codecs/...“ werden im Makefile durch die Variable „XDC_PATH“ gesetzt. Mehr Informationen über den Aufbau des Makefiles befinden sich im Kapitel 6.1.2.

Das CFG-File folgt einem immer ähnlichen Aufbau. Am Anfang muss das XDC-Modul mit Support für den Operating System Abstraction Layer (OSAL) mit Option auf Linux geladen werden. OSAL ist eine Softwarebibliothek zur Trennung des Betriebssystems von der eigentlichen Embedded Software über ein API (vgl. [14]):

```
var osalGlobal = xdc.useModule('ti.sdo.ce.osal.Global');
var os = xdc.useModule('ti.sdo.ce.osal.linux.Settings');
osalGlobal.os = os;
```

Anschließend wird die CodecEngine mit der Anweisung die Version des DSP Links Linux zu nutzen definiert:

```
var ipc = xdc.useModule('ti.sdo.ce.ipc.Settings');
ipc.commType = ipc.COMM_DSPLINK;
```

In Vorbereitung auf die Konfiguration der „Engine“ können für eine übersichtlichere Notierung die zu verwendenden Codecs bereits im Vorfeld definiert werden. Im folgenden Beispiel wird auf die beiden Variablen ein G.711-Encoder/Decoder als XDC-Modul gesetzt:

```
var G711ENC = xdc.useModule('ti.sdo.codecs.g711enc.ce.G711ENC');
var G711DEC = xdc.useModule('ti.sdo.codecs.g711dec.ce.G711DEC');
```

Nun kann die Konfiguration der Engine erfolgen, welche später im C-Quellcode benutzt werden soll. Dabei muss der Name der Engine (hier: „Meine_Engine“) jeweils identisch sein. Die zuvor festgelegten Variablen „G711ENC“ und „G711DEC“ können einfach eingebunden werden:

```
var Engine_xdc = xdc.useModule('ti.sdo.ce.Engine');
var Engine = Engine_xdc.create("Meine_Engine", [
    {name: "g711enc", mod: G711ENC, local: false, groupId: 0},
```

```
    {name: "g711dec", mod: G711DEC, local: false, groupId: 0},
]);
```

Der Parameter „local“ mit dem Wert „false“ legt fest, dass die Engine nicht lokal, also unter der Linux-Umgebung des Boards, sondern „remote“ auf dem DSP laufen soll.
Die Gruppenzugehörigkeit „ groupId“ muss bei allen Codecs, welche sich innerhalb der selben Engine befinden sollen, gleich sein. Der Wert selbst spielt dabei keine Rolle.

Da die definierten Codecs direkt auf dem DSP zum Einsatz kommen sollen, muss ein x64P Combofile bereitgestellt werden, welches die angegebenen Codecs tatsächlich enthält:

```
Engine.server = "./g711encdec.x64P";
```

In diesem Beispiel befindet sich die Datei „g711encdec.x64P“ direkt im Projekt-Ordner. Mehr Informationen zur Erstellung eigener Combofiles finden sich im Kapitel 2.2.

Im letzten Schritt kann das Projekt zur Benutzung von DMAI-Komponenten zur vereinfachten Programmierung des APIs konfiguriert werden (nur bei Verwendung des DMAIs erforderlich):

```
var DMAI = xdc.loadPackage('ti.sdo.dmai');
```

Ein vollständiges CFG-File zur Benutzung eines G.711-Encoders/Decoders innerhalb einer Engine „Meine_Engine“ zur parallelen Nutzung unter DMAI-Support findet sich in Anhang A.

6.1.2 Makefile

In einem Projekt dient das Makefile der zentralen Steuerung. Immer wieder benötigte Operationen wie löschen oder compilieren werden hier verwaltet und müssen nicht immer wieder neu von Hand ausgeführt werden.
Das Makefile selbst wird mit dem Befehl „make“ gelesen und wenn keine Parameter übergeben werden mit der Standardoperation „all“ ausgeführt.

Ein Beispiel für die Operation „make clean“ könnte wie folgt aussehen:

```
[...]
clean:
      @echo Removing generated files..
      $(VERBOSE) -$(RM) -rf $(XDC_CFG) $(OBJFILES) $(TARGET)
```

Dabei werden die Variablen durch vorher definierte Zuweisungen ersetzt. Aus „$(RM)“ wird bei der Ausführung „rm“, der Terminalbefehl zum Löschen, mit Option „-rf“ für das Löschen von Ordnern. Das Löschen der vorher bei der Compilierung erzeugten Dateien geschieht also nun automatisch und es müssen nicht von Hand die generierten Binaries entfernt werden.

Für das Erstellen spezifischer Packages innerhalb eines Projekts, etwa bei der Benutzung von XDC-Komponenten, können für eine bessere Übersicht Pfade zu den entsprechenden Tools und Konfigurationen gesetzt werden:

```
ROOTDIR = /home/praktikum/dvsdk_2_00_00_22
include $(ROOTDIR)/Rules.make
```

Es können zudem andere Makefiles oder Dateien mit weiteren Definitionen („Rules.make“) wie im oberen Beispiel geschehen mit „include“ eingebunden werden.

Der wichtigste Teil eines Makefiles, die Konfiguration der Compiler und Linker, ist ebenso leicht zu definieren. Das folgende Beispiel verdeutlicht, wie dem Compiler/Linker „gcc“ entsprechende Flags übergeben werden können:

```
C_FLAGS += -Wall -g
LD_FLAGS += -lpthread -lfreetype -lasound
COMPILE.c = $(MVTOOL_PREFIX)gcc $(C_FLAGS) $(CPP_FLAGS) -c
LINK.c = $(MVTOOL_PREFIX)gcc $(LD_FLAGS)
SOURCES = $(TARGET).c
```

Die hier abgebildeten CPP_FLAGS (C++-Flags) haben im Makefile keine Bedeutung, da

ohnehin mit C gearbeitet wird. Im originalen Makefile von Texas Instruments wird diese Variable spezifiziert (jedoch ohne Bedeutung), weshalb sie hier ebenfalls aufgeführt wurde.

Die Funktionsweise des Beispiels lässt sich leicht erkennen: Das Tool „gcc“ wird unterhalb des Pfades „MVTOOL_PREFIX“ mit vorher definierten „C_FLAGS“ (Compiler-Optionen, etwa Debuglevel der Fehlerausgaben) und „LD_FLAGS“ (Linker-Optionen, zu linkende Bibliotheken) aufgerufen.
Dabei handelt es sich nicht um das gewöhnliche gcc-Tool von Redhat-Linux, sondern um einen Cross-Compiler aus den MontaVista-Tools, da die Binärdateien für ARM compiliert werden müssen und sich das Hostsystem auf einer i386-Architektur befindet.
Die Variable „SOURCES“ erfasst jene Datei innerhalb des Projekts mit dem Namen „<Projektname>.c“ und übergibt diese automatisch an gcc.

Ein vollständig konfiguriertes Makefile für ein Standardprojekt befindet sich im Ordner „/home/praktikum/dvsdk_praktikum/scripts/projekt/Makefile“ sowie im Anhang K.
Bei der Benutzung der vorkonfigurierten Skripte zur Erstellung eines neuen Projekts (siehe Kapitel 6.2) wird das Makefile automatisch kopiert und muss nicht von Hand eingefügt werden.

6.1.3 Änderungen in Version 2.00

Eine aktuelle Version des CFG-Files befindet sich in Anhang A. Im Vergleich dazu ist die wesentlichste Änderung, gegenüber alten CFG-Files aus der Version 1.00, die verschiedenen XDC-Pfade (oben neu, unten alt):

```
var G711DEC = xdc.useModule('ti.sdo.codecs.g711dec.ce.G711DEC');
var G711DEC = xdc.useModule('ti.sdo.codecs.g711dec.G711DEC');
```

Im Makefile ist ebenfalls eine Anpassungen der Pfade zu den verschiedenen Tools notwendig. Alte Makefiles sind daher nicht mehr nutzbar und sollten durch das Beispiel in „/home/praktikum/dvsdk_praktikum/scripts/projekt/Makefile“ komplett ersetzt werden.

6.2 Projekt automatisch erzeugen

Damit beim Anlegen eines neuen Projektes alle notwendigen Einstellungen nicht immer

wieder manuell vorgenommen werden müssen, liegt im Praktikumsverzeichnis unter dem Pfad „/home/praktikum/dvsdk_praktikum/scripts/NeuesProjekt.sh“ ein Bash-Skript bereit. Dieses muss innerhalb der VMware gestartet werden und es erwartet den gewünschten Projektnamen als Parameter. Anschließend erstellt das Skript die notwendigen Template-Files im konfigurierten Zielverzeichnis:

```
$ /home/praktikum/dvsdk_praktikum/scripts/projekt/neu.sh "Demo"
Erstelle das Projekt "Demo":
[...]
Fertig.
```

Der Standardpfad für angelegte Projekte befindet sich innerhalb des NFS-Shares der VMware unter „/home/praktikum/workdir/filesys/home/praktikum/Programme“.

6.3 Debuggen mit gdb

6.3.1 Compilieren der benötigten Dateien

Um den Ablauf von Programmen gezielt zu kontrollieren oder zu verändern, empfiehlt sich die Benutzung eines Debuggers wie etwa „gdb“. In der Standardinstallation von MontaVista Linux ist „gdbserver“ für das Remote-Target nicht vorinstalliert, sondern muss nach den Anweisungen in Kapitel 3.2 auf dem Board selbst compiliert werden.

Danach kann mit der Compilierung des Clients auf dem Host (VMware) fortgefahren werden. Dazu kann man die in Kapitel 3.2 benutzen Sourcecodes wiederverwenden, allerdings müssen diese vorher von den zuvor erstellen Dateien bereinigt werden:

Entfernen zuvor erstellter Dateien

```
$ cd /tmp/gdb-7.2/
$ make clean
$ make distclean
```

Im folgenden Schritt wird der Quellcode für die Compilierung auf i686 mit ARM-Target vorbereitet:

Konfigurieren des Makefiles auf ARM-Target

```
$ ./configure --host=i686-pc-linux-gnu --target=armv5tl-
      montavista-linux-gnueabi --prefix=/opt/mv_pro_5.0/
      montavista/pro/devkit/arm/v5t_le
```

Nun kann compiliert und anschließend als „root“ installiert werden:

Compilieren der Binaries

```
$ make
$ su
# make install
```

6.3.2 Benutzung des Debuggers

Ein Programm mit dem Namen „testapp“ könnte wie folgt unter dem Aufsatz von gdb auf Port 2000 auf dem Target (Board) gestartet werden:

```
$ gdbserver :2000 ./testapp
Process ./testapp created; pid = 900
Listening on port 2000
```

Dabei wartet gdbserver auf allen Netzwerk-Interfaces automatisch (IP „0.0.0.0“), weshalb die IP des Boards nicht angegeben werden muss. Der Port kann beliebig gewählt werden, jedoch muss beim Starten des Programms ohne Root-Rechte der Port oberhalb von 1024 gewählt werden.
Zu beachten ist außerdem, dass Debugging nur funktionieren kann, wenn das Programm mit Debugging-Symbols compiliert wurde (gcc-Option „-g“). Das Makefile aus dem Praktikums-Ordner compiliert Sourcecode automatisch mit dieser Option.

6.3.3 Grafische Debugging-Tools

Innerhalb der Entwicklungsumgebung „Eclipse“ kann das interne Debuggwerkzeug für Remote-Debugging benutzt werden. Dazu müssen innerhalb der Projektumgebung einige Einstellungen vorgenommen werden.
Ein anderes Debugging-Tool namens „ddd“ funktioniert auch im Standalone-Betrieb und enthält Funktionen, um sich Variablen, Speicher oder Register anzuschauen oder zu

manipulieren. Beim Start muss der Pfad des ARM-gdb-Tools angegeben werden:

```
$ ddd --gdb --debugger /opt/mv_pro_5.0/montavista/pro/devkit/arm/
      v5t_le/bin/armv5tl-montavista-linux-gnueabi-gdb
```

Danach kann über das Menü das gewünschte Binary geöffnet werden. Anschließend wird die Verbindung zum Target hergestellt, auf dem bereits „gdbserver“ gestartet wurde (siehe Kapitel 6.3.2). Dazu muss in das gdb-Terminal von ddd ein entsprechender Befehl eingetragen werden:

```
target remote <Board-IP-Adresse>:<gdb-Port>
```

7 Zusammenfassung

7.1 Ergebnisse

Das Zusammenspiel aller Software- und Hardwarekomponenten, welche mit dem DVEVM in Verbindung stehen, ist aufgrund der zahlreichen Versionsunterschiede und Eigenheiten der Systemumgebung keine Selbstverständlichkeit. Ziel der Diplomarbeit war es, eine vollständige Dokumentation über ein insgesamt funktionsfähiges Setup zu erstellen, welches sich jederzeit in Verbindung mit den Beispielprogrammen nachstellen lässt.
Der Aufbau eines solchen Setups, welches alle wichtigen Funktionen, wie das Erstellen eigener Combofiles oder parallele Codierungsaufgaben umfasst, kann mit dieser Arbeit schrittweise nachvollzogen werden.
Für die weiterführende Benutzung des Boards wurde ein Praktikumsversuch für Studenten erstellt, welcher die Möglichkeit bietet mit einem DSP auch ohne lange Einarbeitungszeit zu experimentieren.

7.2 Ausblick

Die Möglichkeiten des DVSDKs sind sehr umfassend und entscheidente Kleinigkeiten im Softwaresetup können schnell zu Problemen führen oder ein bestimmtes Verhalten des DSPs verursachen. Mit der vorliegenden Diplomarbeit kann ein funktionierendes Setup hergestellt werden, welches letztlich als Ausgangspunkt aller weiteren Entwicklungen steht. Als Ansatz sollen dazu die eigens erstellten Demoprogramme und der Praktikumsversuch dienen, welche grundlegend die Arbeitsweise des DVSDKs in Verbindung mit dem API widerspiegeln.
Die theoretischen Möglichkeiten zur Entwicklung von Software auf dem DSP sind sehr groß, erfordern jedoch für einen einzelnen Entwickler einiges an Zeit.

Anhang

A CFG-File G.711-Enc/Dec

CFG-File eines G.711-Encoders/Decoders mit DMAI-Support (optional) innerhalb einer Codec Engine unter Verwendung des Combofiles „g711.x64P“.

g711.cfg

```
var osalGlobal          = xdc.useModule('ti.sdo.ce.osal.Global');
var os                  = xdc.useModule('ti.sdo.ce.osal.linux.Settings');
osalGlobal.os           = os;

var ipc                 = xdc.useModule('ti.sdo.ce.ipc.Settings');
ipc.commType            = ipc.COMM_DSPLINK;

var G711ENC             = xdc.useModule('ti.sdo.codecs.g711enc.ce.G711ENC');
var G711DEC             = xdc.useModule('ti.sdo.codecs.g711dec.ce.G711DEC');

var Engine_xdc          = xdc.useModule('ti.sdo.ce.Engine');
var Engine              = Engine_xdc.create("Engine_G711", [
        {name: "g711enc", mod: G711ENC, local: false, groupId: 0},
        {name: "g711dec", mod: G711DEC, local: false, groupId: 0},
]);

Engine.server           = "./g711.x64P";
var DMAI                = xdc.loadPackage('ti.sdo.dmai');
```

B Demo G.711-Decoder mit DMAI-Komponenten

Programm zur Decodierung einer übergebener Datei im G.711-Format mit DMAI-Komponenten bestehend aus g711dec_dmai.c und g711dec_dmai.cfg (siehe g711.cfg in Anhang A).

g711dec_dmai.c

```
// C-Standardbibliotheken zur Ein-/Ausgabe
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// XDC-Komponente für die CFG-File-Konfiguration
#include <xdc/std.h>

// Codec-Engine-Komponenten zur DSP-Initialisierung
#include <ti/sdo/ce/CERuntime.h>
#include <ti/sdo/ce/Engine.h>

// DMAI-Komponenten zur Signalverarbeitung
#include <ti/sdo/dmai/Dmai.h>
#include <ti/sdo/dmai/Sound.h>
#include <ti/sdo/dmai/Buffer.h>
#include <ti/sdo/dmai/Loader.h>
#include <ti/sdo/dmai/ce/Sdec1.h>

// Hauptfunktion
int main(int argc, char** argv)
{
        // Definition und Initialisierung aller verwendeten Variablen
        Engine_Handle hEngine = NULL;
        Sdec1_Handle hDecode = NULL;
        SPHDEC1_Params defaultParams = Sdec1_Params_DEFAULT;
        SPHDEC1_DynamicParams defaultDynParams = Sdec1_DynamicParams_DEFAULT;
        Buffer_Handle hOutBuf = NULL;
        Buffer_Handle hInBuf = NULL;
        Buffer_Attrs bAttrs = Buffer_Attrs_DEFAULT;
        Loader_Handle hLoader = NULL;
        Loader_Attrs lAttrs = Loader_Attrs_DEFAULT;
        Sound_Handle hSound = NULL;
        Sound_Attrs sAttrs = Sound_Attrs_MONO_DEFAULT;

        // Codec-Engine und DMAI initialisieren
        CERuntime_init();
        Dmai_init();

        // Sound-Device initialisieren
        hSound = Sound_create(&sAttrs);
        if( hSound == 0 ) {
                printf("[ERR] Konnte die Soundausgabe nicht initialisieren!\n");
                goto cleanup;
        }
        printf("[OK]  Soundausgabe initialisiert.\n");

        // Verbindung zur Codec-Engine öffnen
        hEngine = Engine_open("Engine_G711", NULL, NULL);
        if( hEngine == 0 ) {
                printf("[ERR] Konnte die Codec-Engine nicht initialisieren!\n");
                goto cleanup;
```

```
        }
        printf("[OK]  Codec-Engine initialisiert.\n");

        // Codec (g711dec) initialsieren
        hDecode = Sdec1_create(hEngine, "g711dec", &defaultParams, &defaultDynParams);
        if (hDecode == NULL) {
                printf("[ERR] Konnte den Sprachdecoder nicht initialisieren!\n");
                goto cleanup;
        }
        printf("[OK]  Sprachdecoder initialisiert.\n");

        // Ausgabepuffer initialisieren
        hOutBuf = Buffer_create(Sdec1_getOutBufSize(hDecode) * 2, &bAttrs);
        if (hOutBuf == NULL) {
                printf("[ERR] Konnte den Ausgabepuffer nicht initialisieren!\n");
                goto cleanup;
        }
        printf("[OK]  Ausgabepuffer initialisiert.\n");

        // Eingabedatei nach "hLoader" initialisieren
        lAttrs.readSize = Sdec1_getInBufSize(hDecode);
        lAttrs.readBufSize = lAttrs.readSize * 2;
        hLoader = Loader_create(argv[1], &lAttrs);
        if (hLoader == NULL) {
                printf("[ERR] Konnte die Eingabedatei nicht laden!\n");
                goto cleanup;
        }
        printf("[OK]  Eingabedatei \"%s\" geladen.\n", argv[1]);

        // Erste Bytekette nach "hInBuf" kopieren
        Loader_prime(hLoader, &hInBuf);
        printf("[OK]  Beginne mit der Decodierung ...\n");

        // Decodiere in einer Endlosschleife
        for(;;) {
                // Decodiere "hInBuf" nach "hOutBuf" und Überprüfe, ob der Decodierungs-
                // prozess nicht erfolgreich war
                if (Sdec1_process(hDecode, hInBuf, hOutBuf) < 0) {
                        printf("[ERR] Fehler beim Decodieren des Puffers!");
                        goto cleanup;
                }

                // Decodierten Puffer auf "hSound" schreiben
                Sound_write(hSound, hOutBuf);

                // Laden der nächsten Bytekette nach "hInBuf"
                Loader_getFrame(hLoader, hInBuf);

                // Verlasse die Schleife, sobald "hInBuf" leer ist (auf NULL zeigt)
                if (Buffer_getUserPtr(hInBuf) == NULL)
                        goto cleanup;
        }

        // Setup schließen
        cleanup:
        Loader_delete(hLoader);
        Buffer_delete(hOutBuf);
        Sdec1_delete(hDecode);
        Engine_close(hEngine);
        Sound_delete(hSound);
        printf("[OK]  Setup beendet.\n");

        return 0;
}
```

C Demo G.711-Decoder mit CE-Komponenten

Programm zur Decodierung einer übergebener Datei im G.711-Format mit CE-Komponenten bestehend aus g711dec_ce.c, g711dec_ce.h, decode.c, sound.c und g711dec_ce.cfg (siehe g711.cfg in Anhang A).

g711dec_ce.c

```
// C-Standardbibliotheken zur Ein-/Ausgabe
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <malloc.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <pthread.h>
#include <linux/soundcard.h>

// XDC-Komponente für die CFG-File-Konfiguration
#include <xdc/std.h>

// Codec-Engine-Komponenten zur DSP-Initialisierung
#include <ti/sdo/ce/Engine.h>
#include <ti/sdo/ce/CERuntime.h>
#include <ti/sdo/ce/trace/gt.h>

// Codec-Engine-Komponenten zur Signalverarbeitung
#include <ti/sdo/ce/osal/Memory.h>
#include <ti/sdo/ce/speech/sphdec.h>
#include <ti/sdo/ce/speech1/sphdec1.h>

// XDAIS-Komponente für PCM-Konstanten
#include <ti/xdais/dm/ispeech1_pcm.h>

// Eigene Konstanten und Variablen
#include "g711dec_ce.h"

// Decodierungsfunktionen
#include "decode.c"

// Audiogerätfunktionen
#include "sound.c"

// Hauptfunktion
int main(int argc, char** argv)
{
        // Definition und Initialisierung aller verwendeten Variablen
        int in, out;
        Engine_Handle hEngine = NULL;
        SPHDEC1_Handle hDecode = NULL;
        SPHDEC1_Params hDecodeParams;

        // Codec-Engine initialisieren
        CERuntime_init();

        // Pufferspeicher allokieren
        inBuf = (XDAS_Int8 *)Memory_contigAlloc(IFRAMESIZE, BUFALIGN);
        outBuf = (XDAS_Int8 *)Memory_contigAlloc(OFRAMESIZE, BUFALIGN);
```

```
        // Codec-Einstellungen vornehmen
        hDecodeParams.size = sizeof(hDecodeParams);
        hDecodeParams.compandingLaw = ISPEECH1_PCM_COMPAND_ALAW;

        // Verbindung zur Codec-Engine öffnen
        hEngine = Engine_open("Engine_G711", NULL, NULL);
        if( hEngine == 0 ) {
                printf("[ERR] Konnte die Codec-Engine nicht initialisieren!\n");
                goto cleanup;
        }
        printf("[OK]  Codec-Engine initialisiert.\n");

        // Codec (g711dec) über CE initialisieren
        hDecode = SPHDEC1_create(hEngine, "g711dec", &hDecodeParams);
        if (hDecode == NULL) {
                printf("[ERR] Konnte den Sprachdecoder nicht initialisieren!\n");
                goto cleanup;
        }
        printf("[OK]  Sprachdecoder initialisiert.\n");

        // File-Deskriptor zur Eingabe öffnen ("argv[1]" = übergebene Datei)
        in = open(argv[1], O_RDONLY);
        if (in <= 0) {
                printf("[ERR] Konnte die Eingabedatei nicht laden!\n");
                goto cleanup;
        }
        else
                printf("[OK]  Eingabedatei \"%s\" geladen.\n", argv[1]);

        // File-Deskriptor zur Ausgabe öffnen (Soundkarte)
        out = initSoundDevice();
        if (out <= 0) {
                printf("[ERR] Konnte die Soundausgabe nicht initialisieren!\n");
                goto cleanup;
        }
        else
                printf("[OK]  Soundausgabe initialisiert.\n");

        // Decodierungsfunktion starten
        decode(hDecode, in, out);
        // Setup schließen
        cleanup:
        close(in);
        close(out);
        SPHDEC1_delete(hDecode);
        Engine_close(hEngine);
        printf("[OK]  Setup beendet.\n");

        return 0;
}
```

g711dec_ce.h

```
// Globale Pufferspeicher
static XDAS_Int8 *inBuf;
static XDAS_Int8 *outBuf;

// Speichereinstellungen
#define BUFALIGN Memory_DEFAULTALIGNMENT
#define NSAMPLES 160
#define IFRAMESIZE (NSAMPLES * sizeof(Int8))
#define OFRAMESIZE (NSAMPLES * sizeof(Int8))
#define MAXVERSIONSIZE 128

// Audiogerätkonfiguration
#define SOUND_DEVICE "/dev/sound/dsp"
#define NUM_CHANNELS 1
#define SAMPLE_RATE 8000
```

decode.c

```
void decode(SPHDEC1_Handle hDecode, int in, int out)
{
	// Definition und Initialisierung aller verwendeten Variablen
	Int n;
	Int32 status;
	SPHDEC1_InArgs decInArgs;
	SPHDEC1_OutArgs decOutArgs;
	SPHDEC1_DynamicParams decDynParams;
	SPHDEC1_Status decStatus;
	XDM1_SingleBufDesc inBufDesc;
	XDM1_SingleBufDesc outBufDesc;

	// Größe der Pufferspeicher festlegen
	inBufDesc.bufSize = IFRAMESIZE;
	inBufDesc.buf = inBuf;
	outBufDesc.buf = outBuf;
	decInArgs.size = sizeof(decInArgs);
	decOutArgs.size = sizeof(decOutArgs);
	decDynParams.size = sizeof(decDynParams);
	decStatus.size = sizeof(decStatus);

	// Zeiger auf Puffer initialisieren
	decInArgs.data.buf = NULL;

	printf("[OK]  Beginne mit der Decodierung ...\n");
	// Decodiere in einer Endlosschleife
	for (;;) {
		// Lese Bytekette der Länge "IFRAMESIZE" nach "inBuf"
		n = read(in, inBuf, IFRAMESIZE);

		// Verlasse die Schleife, sobald kein Byte mehr gelesen wurde
		if (n<=0) {
			printf("[OK]  Keine weiteren Bytes zu lesen.\n");
			break;
		}

		// Setzen der Größe des Ausgabepuffers
		outBufDesc.bufSize = OFRAMESIZE;

		// Decodiere Bytekette "inBuf" nach "outBuf"
		status = SPHDEC1_process(hDecode, &inBufDesc, &outBufDesc, &decInArgs,
			&decOutArgs);

		// Überprüfe, ob der Decodierungsprozess nicht erfolgreich war
		if (status != SPHDEC1_EOK) {
			printf("[ERR] Der Puffer konnte nicht decodiert werden!\n");
			break;
		}

		// Decodierten Puffer "outBuf" der Länge "outBufDesc.bufSize" auf das
		// Ausgabegerät schreiben
		write(out, outBuf, outBufDesc.bufSize);
	}
}
```

sound.c

```
static int initSoundDevice(void)
{
	int channels = NUM_CHANNELS;
	int sampleRate = SAMPLE_RATE;
	int format = AFMT_S16_LE;
	int hSound;

	// Sound-Device zum Schreiben öffnen
	hSound = open(SOUND_DEVICE, O_WRONLY);
	if (hSound == -1) {
		printf("[ERR] Soundgerät nicht verfügbar!\n");
		return -1;
	}

	// Ausgabe-Format per I/O-Control auf AFMT_S16_LE setzen (OSS, 16-bit signed little
	// endian)
	if (ioctl(hSound, SNDCTL_DSP_SETFMT, &format) == -1) {
		printf("[ERR] Konnte Soundgerät nicht konfigurieren (Format: %d)!\n", format);
		return -1;
	}

	// Anzahl der Kanäle auf 2 setzen (beide Lautsprecher aktiv)
	if (ioctl(hSound, SNDCTL_DSP_CHANNELS, &channels) == -1) {
		printf("[ERR] Konnte Mixer-Kanäle nicht auf %d setzen!\n", channels);
		return -1;
	}

	// Sample-Rate auf 8000 Hz setzen (entsprechend G.711)
	if (ioctl(hSound, SNDCTL_DSP_SPEED, &sampleRate) == -1) {
		printf("[ERR] Konnte die Samplerate nicht auf %d setzen!\n", sampleRate);
		return -1;
	}

	// Präparierten File-Deskriptor zurückgeben
	return hSound;
}
```

D Praktikumsversuch für Studenten

Versuchsanleitung mit Lösungen für einen Praktikumsversuch „DVEVM“.

Versuchsanleitung: DVEVM

1 Zielstellung des Versuchs

Digitale Signalprozessoren (DSPs) sind in einer Vielzahl elektronischer Geräte zu finden. Ob in Mobiltelefonen, PCs oder dem MP3-Player – analoge Filter haben zumeist ausgedient und Platz gemacht für die neuen Möglichkeiten der digitalen Signalverarbeitung. Adaptive Codecs mit Rauschunterdrückung und Dynamikkompression, Spracherkennung und Echounterdrückung bestimmen unsere Kommunikation.

Der folgende Versuch dient dazu das theoretische Wissen über Codecs auszuprobieren und einen DSP über ein Application Programming Interface (API) in der Programmiersprache C zu bedienen, ohne auf die tieferen Operationen und Befehlssätze einzugehen.

2 Versuchsgrundlagen

Ein Teil des Versuchs spielt sich in der Linux-Distribution „RedHat“ ab, welche in einer Virtualisierungsumgebung „VMware“ innerhalb des Windows-Betriebssystems gestartet werden kann. Innerhalb dieses Systems kann das Projekt bearbeitet und compiliert werden.
Auf dem Versuchsboard (DVEVM) selbst befindet sich eine weitere Linux-Umgebung auf einer ARM-Plattform (Advanced RISC Machine, Prozessorarchitektur), welche benötigt wird, um auf den DSP zuzugreifen. Dort können zuvor erstelle Binärfiles fertiger Programme ausgeführt werden.

Unter Linux werden auf beiden Systemen für diesen Versuch einige Standardbefehle benötigt:

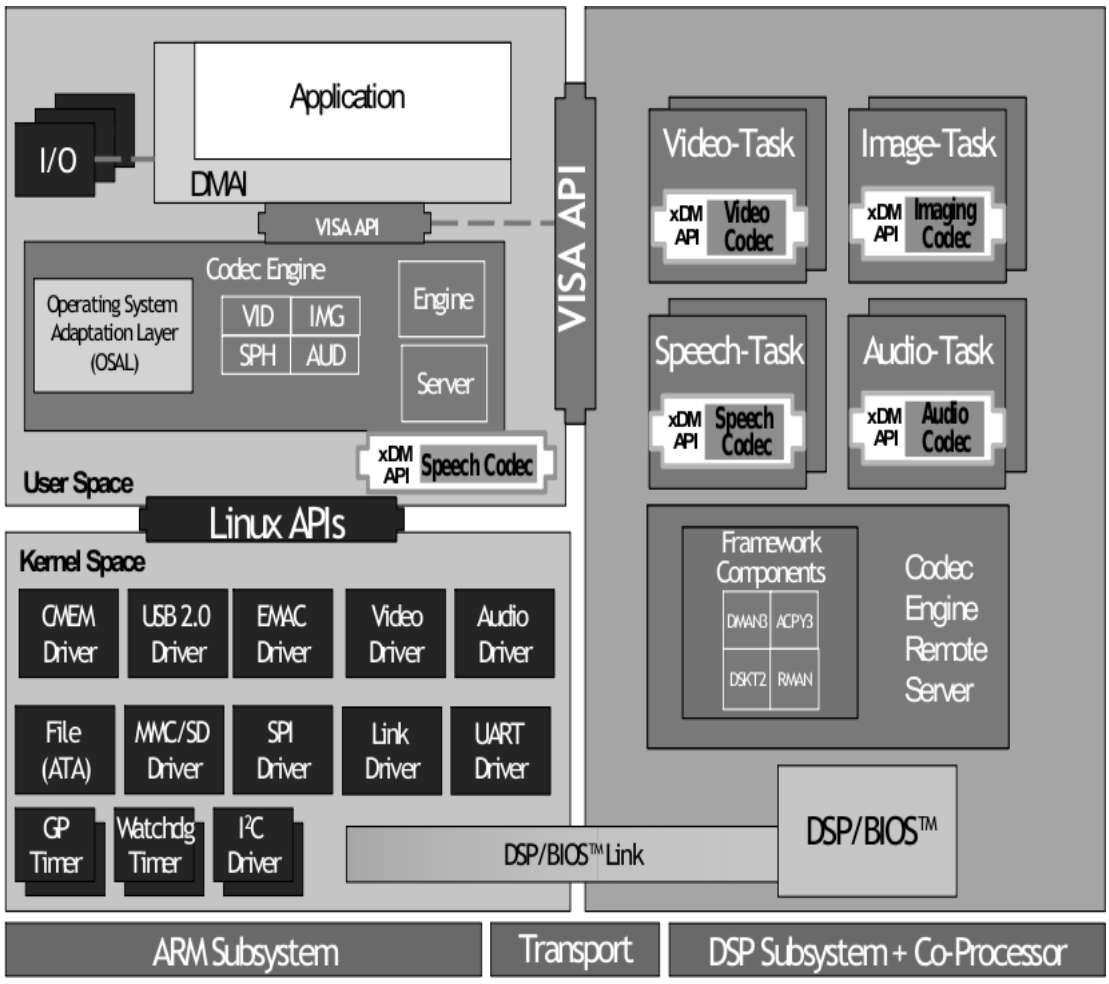
cat	Liest den Inhalt einer übergebenen Datei auf dem Terminal
ls	Listet alle Dateien innerhalb des aktuellen Verzeichnisses
make	Führt das „Makefile“ zum Compilieren eines Projektordners aus
xxd	Erstellt aus einer übergebenen Datei einen Hexdump auf dem Terminal oder schreibt Characters als C-Header

Wenn ein Befehl genauer nachgeschlagen werden soll, empfiehlt sich die Benutzung des Systemtools „man“ (z.B.: „man xxd“). Alternativ kann auch eine Onlineversion auf der Website http://linux.die.net genutzt werden (z.B.: http://linux.die.net/man/1/xxd).

Der Zugriff auf den DSP selbst erfolgt über ein C-API, welches über das Einbinden entsprechender Headerdateien zugänglich gemacht wird. Grundlegende API-Funktionen sind:

CERuntime_init	Initialisiert die Runtime der Codec-Engine auf dem DSP
Engine_open	Öffnet eine neue Codec-Engine
SPHDEC1_create	Erzeugt eine neue Instanz eines Sprachdecoders
SPHDEC1_control	Fragt den Zustand eines gültigen Sprachdecoders ab
SPHDEC1_process	Decodiert den Eingabepuffer eines gültigen Sprachdecoders auf einen Ausgabepuffer

Die folgende Abbildung zeigt noch einmal das Zusammenspiel zwischen DSP (rechts) und Linux (links) über das VISA API:



In diesem Versuch soll nur das „Application“-Feld betrachtet werden, welches als die Anwender- und Entwicklerebene auftritt. Der in C geschriebene Code hinter den API-Funktionen wird auf den spezifischen Befehlssatz dieses Prozessors von den Framework-Components übersetzt.

3 Versuchsvorbereitung

3.1 Signalcodierung

PCM-Standards

Machen Sie sich mit den PCM-Standards unter https://www.eit.hs-mittweida.de/index.php?id=1126 → Audiosignalcodierung (Blatt AC 2) vertraut (HS-Login zum Herunterladen erforderlich).

Bezug zur Versuchsdurchführung: Es sollen bitweise PCM/G.711-Samples untersucht werden.

ITU-T Standard zu G.711

Die ITU-T (International Telecommunication Union, Telecommunication Standardization Sector) ist Autor des G.711-Standards: http://www.itu.int/rec/T-REC-G.711-198811-I/en

Bezug zur Versuchsdurchführung: Es sollen vordefinierte Codeworte decodiert und das Ergebnis mit den offiziellen Tabellen aus dem Standard verglichen werden.

Fragen zur Vorbereitung:

1) Woraus besteht ein G.711-Sample?
2) Welche Bitfolge würde ein G.711-ALAW-Decoder für das folgende Codewort ausgeben: 00011010

3.2 C-Funktionen

memcpy()

Die C-Funktion *memcpy()* kopiert eine gegebene Anzahl von Bytes aus einem Quellpuffer in einen Zielpuffer: http://linux.die.net/man/3/memcpy

Bezug zur Versuchsdurchführung: Aus einem statischen Eingangspuffer von G.711-Samples sollen blockweise immer wieder 160 Byte zyklisch gelesen und decodiert werden:

rand()

Die C-Funktion *rand()* erzeugt pseudo-zufällige Zahlen. Für die Initialisierung mit einem „zufälligen“ Startwert kann etwa die aktuelle Systemzeit genutzt werden:

srand((unsigned) time(NULL));

Weitere Informationen zu *rand()* und *srand()* liefert die man-Page auf: http://linux.die.net/man/3/rand

Bezug zur Versuchsdurchführung: Einige Samples sollen mit zufälligen Bitfehlern verfälscht werden, um eine definierte Fehlerquote für den subjektiven Höreindruck zu simulieren.

Linux-Befehle (optional)

Nützliche Befehle für das Linux-Terminal: http://www.pc-erfahrung.de/linux/linux-befehle.html

C-Referenz (optional)

Ein empfehlenswertes C-Buch zum Online-Lesen gibt es kostenlos von Galileo Computing: http://openbook.galileocomputing.de/c_von_a_bis_z/

Fragen zur Vorbereitung:

1) Wie kann mit Hilfe von Pointern *memcpy()* so benutzt werden, dass die Adresse der Variablen des Eingangspuffers periodisch zu einem Offset von 160 Byte addiert wird?
2) Wie können mit der Funktion *rand()* zufällige Zahlen im Bereich von 0-7 erzeugt werden?
3) Was ist ein Cross-Compiler?

4 Versuchsdurchführung

4.1 Starten der VM-Ware

Starten Sie unter Windows „VMWare Workstation“ vom Desktop und loggen Sie sich nach dem Hochfahren mit folgenden Daten in das Linux-Betriebssystem ein:

Benutzer: praktikum Passwort: praktikum

4.2 Öffnen eines Samplefiles „musik.g711“

Öffnen Sie ein Terminal über das Menü *Anwendungen → Systemwerkzeuge → Terminal*. Wechseln Sie das Verzeichnis per „cd“ auf den Projektordner „g711dec“:

```
$ cd /home/praktikum/workdir/filesys/home/praktikum/DVEVM/g711dec
```

Erzeugen Sie auszugsweise einen Binärdump der dort befindlichen Datei „musik.g711“ von 160 Byte Länge ab Byte 8000 mit dem Befehl „xxd“!

1) Wie viele G.711-Samples sind zu sehen?
2) Welchem Zeitabschnitt (Sekunden von .. bis) des Musikstückes entspricht der Dump?

4.3 Decodieren des Samplefiles „musik.g711“

Erzeugen Sie für das komplette Musikstück „musik.g711“ ein Headerfile „musik.h“ von Characters mit dem Befehl „xxd“ im Projektordner! Machen Sie sich mit der angelegten Datei und ihren Variablen vertraut.

1) Von welchem Variablentyp sind die Samples?
2) Wie viele Samples befinden sich in der Datei „musik.h“ und wie lang ist daraus resultierend das Musikstück (in Sekunden)?

Öffnen Sie die Datei „decode.c“ und ergänzen Sie die Funktion *decode()* so, dass aus der Variable „musik_g711“ zyklisch ein Block von Samples nach „inBuf“ kopiert wird, sodass der Decodefunktion immer genau 160 Byte Musik übergeben werden, bis der Eingabepuffer vollständig gelesen wurde! Fangen Sie das Ende des zu lesenden Puffers ab! Editieren Sie die Datei ab dem Kommentar „*// Decodiere in einer Endlosschleife*“.

Speichern Sie die Änderungen ab und compilieren Sie das Projekt anschließend auf dem Terminal mit dem Befehl „make“.

4.4 Starten des Boards

Öffnen Sie ein weiteres Terminal und führen Sie den Befehl „minicom“ aus. Schalten Sie danach das Board ein – der Bootvorgang sollte zu beobachten sein. Anschießend können Sie sich auf dem Board mit den gleichen Daten aus Kapitel 4.1 einloggen.

4.5 Ausführen des Projekts

Wenn Sie auf dem Board eingeloggt sind, können Sie auf Ihr Projekt zugreifen und es starten:

```
$ cd /home/praktikum/DVEVM/g711dec
$ ./g711dec
```

Das Programm startet nun und sollte Ausgaben mit „[OK]“ liefern. Wenn Sie die Funktion *decode()* korrekt modifiziert haben, ist das Musikstück zu hören.

4.6 Erzeugen künstlicher Bitfehler

Gehen Sie in der VM-Ware zurück in den Projektordner und öffnen Sie die Datei „decode.c“ erneut. Implementieren Sie nach dem blockweisen Lesen der Samples mittels *memcpy()* eine For-Schleife, welche in diesem Block *inBuf[0..159]* zufällige Fehler mittels der bereits vorgefertigten Funktion *bitfehler()* erzeugt.
Die einzustellende Bitfehlerwahrscheinlichkeit soll 1% betragen. Die Funktion *bitfehler()* würde im folgenden Beispiel die Zahl 128 an Bitstelle 1 verfälschen: *bitfehler (128, 1);*

Codewort:	10000000	(128)
Fehler:	00000010	(2)
Ergebnis:	10000010	(130)

Speichern und Compilieren Sie das Projekt, starten Sie das Programm anschließend analog zu Punkt 4.5.

1) Wie hat sich der Klang des Musikstückes verändert?
2) Ab circa welcher Bitfehlerwahrscheinlichkeit ist das Hörempfinden gegenüber dem Musikstück bereits durch Knackgeräusche beeinträchtigt?

5 Hilfestellung und Anregungen

Arbeit mit memcpy() auf Adressbasis

Das folgende Codebeispiel kopiert nach A genau 10 Byte aus B ab Byte 100. Beide Variablen sind vom Typ „int“:

*memcpy(A, B + (sizeof(int) * 100), 10);*

Typecasting zwischen unterschiedlichen Variablentypen

Auf einer Variablen A vom Typ „int“ kann der Wert einer Variablen B vom Typ „char“ mittels Typecast abgespeichert werden:

```
int A; char B =  'z';
A = (char) B;   // Inhalt von A ist nun 122 (nach ASCII)
```

Zufallszahlen mit rand()

Zufällige Zahlen innerhalb eines bestimmtes Bereiches können mit der Modulo-Operation erzeugt werden (Divisionsrest). Das folgende Beispiel erzeugt zufällige Zahlen zwischen 10 und 15:

```
srand(time(NULL));
A = 10 + rand()%6;
```

Lösung zu den Aufgaben

3.1

1) Woraus besteht ein G.711-Sample?

8-Bit breites Wort (komprimiert aus 13-Bit linearem Eingabewert)

2) Welche Bitfolge würde ein G.711-ALAW-Decoder für das folgende Codewort ausgeben: 00011010

0000000110101111

3.2

1) Wie kann mit Hilfe von Pointern `memcpy()` so benutzt werden, dass die Adresse der Variablen des Eingangspuffers periodisch zu einem Offset von 160 Byte addiert wird?

Aus einem statischen Eingangspuffer E sollen periodisch 160 Byte in einen temporären Puffer „InBuf“ kopiert werden.
Syntax des Systemrufs: *memcpy(Zieladresse, Quelladresse, Byteanzahl)*

```
unsigned int Offset = 0;
for (;;) {
        memcpy(InBuf, (XDAS_Int8 *) (sizeof(unsigned char) * Offset + E), 160);
        Offset += 160;
        // Lesen ...
}
```

Ziel:	InBuf, Variable des Typs *static XDAS_Int8**
	Array der Breite 160 * sizeof(Int8)
Quelle:	Zu der Startadresse von E im Speicher wird ein Adressabstand der Breite Offset * sizeof(unsigned char) addiert
	Der Zielpuffer erwartet Werte des Typs Int8, daher muss eine Typ-konvertierung mittels (XDAS_Int8 *) durchgeführt werden
Byteanzahl:	160

2) Wie können mit der Funktion rand() zufällige Zahlen im Bereich von 0-7 erzeugt werden?

unsigned int wert = 0;
srand(time(NULL)); // Initialisierung mit einem „zufälligen“ Wert (Systemzeit)
wert = rand()%8; // wert = Rest von (Zufallswert/8) → Modulo-Operation

3) Was ist ein Cross-Compiler?

Compiler = Programm zur Übersetzung einer Programmiersprache (hier: C, Hochsprache) auf eine maschinenverständliche Zielsprache (z.B. Maschinencode)

„Cross“ (kreuzend, systemübergreifend) steht hierbei für die Fähigkeit Zielprogramme für andere Prozessorarchitekturen erstellen zu können, als die auf der der Compiler gerade arbeitet
Beispiel: Compilierung eines Programms für ARM auf einem x86-System

4.2

Erzeugen des Binärdumps

$ xxd -b -seek 8000 -l 160 ./musik.g711

Ausgabe:

```
0001f40: 01010101 01010101 01010101 01010101 01010101 01010101  UUUUUU
0001f46: 11010101 01010101 01010101 01010101 01010101 11010101  .UUUU.
0001f4c: 11010101 11010101 11010101 11010101 01010101 11010101  ....U.
0001f52: 11010101 11010101 11010101 01010101 01010101 11010101  ...UU.
0001f58: 11010101 11010101 11010101 11010100 11010100 11010100  ......
0001f5e: 11010100 11010100 11010100 11010100 11010100 11010100  ......
0001f64: 11010100 11010100 11010100 11010100 11010101 11010100  ......
0001f6a: 11010111 11010100 11010100 11010100 11010100 11010111  ......
0001f70: 11010111 11010111 11010111 11010100 11010111 11010111  ......
0001f76: 11010111 11010100 11010101 11010101 11010100 11010111  ......
0001f7c: 11010100 11010101 11010100 11010100 11010100 11010111  ......
0001f82: 11010111 11010111 11010100 11010111 11010111 11010100  ......
0001f88: 11010111 11010111 11010100 11010100 11010100 11010100  ......
0001f8e: 11010111 11010111 11010111 11010111 11010101 11010100  ......
0001f94: 11010100 01010101 01010101 11010101 01010101 11010101  .UU.U.
0001f9a: 11010101 11010101 11010100 01010101 01010100 11010101  ...UT.
0001fa0: 11010101 11010101 01010101 11010100 11010110 11010100  ..U...
0001fa6: 11010111 01010101 01011000 01000101 11010010 11000101  .UXE..
0001fac: 01010010 01010111 11010011 01010001 11011011 11000101  RW.Q..
0001fb2: 01011110 11010111 11011101 01011110 01010110 01010001  ^..^VQ
0001fb8: 01000101 01011100 01011101 01011100 11010101 11010111  E\]\..
0001fbe: 11010110 11011001 01010100 01001100 01000100 11011111  ..TLD.
0001fc4: 11011001 01011110 01000101 01010001 01010000 01000101  .^EQPE
0001fca: 01001010 01110111 01011011 11011111 11000101 11010010  Jw[...
0001fd0: 01000011 01001001 01011111 01000011 01110111 01001110  CI_CwN
0001fd6: 01000001 01001101 01110101 01110111 01001110 01011110  AMuwN^
0001fdc: 01001101 01110101 01011111 01011001                    Mu_Y
```

1) Wie viele G.711-Samples sind zu sehen?

160 Byte Binärdaten → 160 G.711-Samples

2) Welchem Zeitabschnitt (Sekunden von .. bis) des Musikstückes entspricht der Dump?

Offset von 8000 Bytes → 1 s
160 Byte Binärdaten → 20 ms
Zeitabschnitt: 1,00 – 1,02 s

4.3

Erzeugen der Datei „musik.h“

$ xxd -i musik.g711 musik.h

1) Von welchem Variablentyp sind die Samples?

Auszug aus „musik.h“:

```
unsigned char musik_g711[] = {
  0xd5, 0xd5, 0xd5, 0xd5, 0xd5, 0xd5, 0xd5, 0xd5, ...
```

Variablentyp: unsigned char

2) Wie viele Samples befinden sich in der Datei „musik.h“ und wie lang ist daraus resultierend das Musikstück (in Sekunden)?

Letzte Zeile der Datei „musik.h“: unsigned int musik_g711_len = 481600;
481600 Characters → 481600 Samples → 60,2s

Ergänzung der Funktion decode()

Es soll mittels *memcpy()* blockweise gelesen werden (siehe 3.2).
Das Ende des maximal zu lesenden Pufferspeichers ist erreicht, wenn die bereits gelesene Anzahl an Samples („offset“) addiert mit den nächsten 160 Byte („samples“) die Gesamtlänge von 481600 Samples („musik_g711_len“) überschreitet.

```
[...]
// Decodiere in einer Endlosschleife
unsigned int samples=160, offset=0;
for (;;) {
        // Blockweises Lesen
        memcpy(inBuf, (XDAS_Int8 *) (sizeof(unsigned char) * offset +
                musik_g711), samples);
        offset += samples;

        // Setzen der Größe des Ausgabepuffers
        outBufDesc.bufSize = OFRAMESIZE;
[…]
// Decodierten Puffer "outBuf" der Länge "outBufDesc.bufSize" auf das
// Ausgabegerät schreiben
write(out, outBuf, outBufDesc.bufSize);

// Beenden, wenn Sample-Ende erreicht ist
if ((offset+samples) > musik_g711_len)
        break;
[…]
```

4.6

Benutzung der Funktion *bitfehler()*

Es sollen Samples mit Bitfehlern verfälscht werden. Dazu steht eine Funktion *bitfehler()* bereit, welche einen Wert des Typs Int8 an einer angegebene Bitstelle zum jeweils anderen Bit verfälscht und den daraus entstandenen Wert wieder in Int8 konvertiert und zurückgibt.
Die Variable „fehlerbyte“ bestimmt mittels *rand()* welches der 160 Bytes verfälscht werden soll und „r*and()%8*“ gibt die zu verfälschende Bitstelle (0..7) an.

```
[...]
// Decodiere in einer Endlosschleife
unsigned int samples=160, offset=0, i, fehlerbyte;
float fehlerrate = 0.01;
srand(time(NULL));
for (;;) {
```

```
        // Blockweises Lesen
        memcpy(inBuf, (XDAS_Int8 *) (sizeof(unsigned char) * offset +
                musik_g711), samples);
        offset += samples;

        // Manipulation mit Zufallsfehlern
        for (i=0; i<(samples*8*fehlerrate); i++) {
                fehlerbyte = rand()%samples;
                inBuf[fehlerbyte] = bitfehler(inBuf[fehlerbyte], rand()%8);
        }

        // Setzen der Größe des Ausgabepuffers
        outBufDesc.bufSize = OFRAMESIZE;
[…]
```

1) Wie hat sich der Klang des Musikstückes verändert?

- deutliches Rauschen
- leise Sequenzen nicht mehr wahrnehmbar

2) Ab circa welcher Bitfehlerwahrscheinlichkeit ist das Hörempfinden gegenüber dem Musikstück bereits durch Knackgeräusche beeinträchtigt?

- ca. 10^{-4}

E Shellscript für neue Projekte

Automatisches Erstellen eines neuen Projekt-Ordners mit Template-Files.

neu.sh

```
#!/bin/bash

if [ "$(uname -m)" == "armv5tejl" ]; then
        echo "Bitte dieses Script nur in der VM-Ware starten!"
        exit 1
fi

if [ -n "$1" ]; then
        Projekt="$1"
        Prefix="/home/praktikum/workdir/filesys/home/praktikum"
        Pfad="$Prefix/Programme/$Projekt"
        Templates="$(dirname $0)"

        if [ ! -d "$Pfad" ]; then
                echo "Erstelle das Projekt \"$Projekt\":
                Erzeuge Ordner $Pfad ..."
                mkdir $Pfad
                echo "Kopiere Dateien ..."
                cp "$Templates/Makefile" "$Pfad/"
                cp "$Templates/Praktikum.h" "$Pfad/"
                cp "$Templates/Template.cfg" "$Pfad/$Projekt.cfg"
                cp "$Templates/Template.c" "$Pfad/$Projekt.c"
                echo "Fertig."
                exit 0
        else
                echo "*** Fehler: Ein Projekt mit dem Namen \"$Projekt\" existiert bereits!"
                exit 1
        fi
else
        echo "Benutzung: $(basename $0) \"MEIN_PROJEKTNAME\""
        exit 0
fi

exit 0
```

F Loadmodules

Laden der benötigten Kernelmodule und Setzen der Berechtigungen auf Geräte.

loadmodules.sh

```
#!/bin/sh

# Lade "cmemk", physikalischer Speicher von 118MB bis 128MB.
insmod cmemk.ko phys_start=0x87800000 phys_end=0x88200000 pools=1x3600000,
        5x829440,2x1244160,1x40960,2x8192

# Lade "dsplinkk"
insmod dsplinkk.ko

# Erzeuge "/dev/dsplink"
rm -f /dev/dsplink
mknod /dev/dsplink c `awk "\\$2==\"dsplink\" {print \\$1}" /proc/devices` 0

# Setze Rechte zum Lesen/Schreiben auf die Geräte
sleep 3
chmod 666 /dev/sound/audio
chmod 666 /dev/sound/dsp
chmod 666 /dev/sound/mixer
chmod 666 /dev/cmem
chmod 666 /dev/dsplink

exit 0
```

G Rules.make

Vorkonfigurierte Datei „Rules.make“ zum Setzen der Pfade zu DVSDK-Komponenten, welche in Makefiles inkludiert werden kann.

Rules.make

```
# This make variable must be set before the DVSDK components can be built.
PLATFORM=dm6446

# The installation directory of the DVSDK
DVSDK_INSTALL_DIR=$(HOME)/dvsdk_2_00_00_22

# For backwards compatibility
DVEVM_INSTALL_DIR=$(DVSDK_INSTALL_DIR)

# Where the DVSDK demos are installed
DEMO_INSTALL_DIR=$(DVSDK_INSTALL_DIR)/dvsdk_demos_2_00_00_07

# Where the Digital Video Test Bench is installed
DVTB_INSTALL_DIR=$(DVSDK_INSTALL_DIR)/dvtb_4_00_08

# Where the Davinci Multimedia Application Interface is installed
DMAI_INSTALL_DIR=$(DVSDK_INSTALL_DIR)/dmai_1_20_00_06

# Where the Codec Engine package is installed.
CE_INSTALL_DIR=$(DVSDK_INSTALL_DIR)/codec_engine_2_23_01

# Where the XDAIS package is installed.
XDAIS_INSTALL_DIR=$(DVSDK_INSTALL_DIR)/xdais_6_23

# Where the DSP Link package is installed.
LINK_INSTALL_DIR=$(DVSDK_INSTALL_DIR)/dsplink-1_61_03-prebuilt

# Where the CMEM (contiguous memory allocator) package is installed.
CMEM_INSTALL_DIR=$(DVSDK_INSTALL_DIR)/linuxutils_2_23_01

# Where the EDMA3 Low Level Driver is installed.
EDMA3_LLD_INSTALL_DIR=$(DVSDK_INSTALL_DIR)/edma3_lld_1_05_00
CODEC_INSTALL_DIR=$(DVSDK_INSTALL_DIR)/dm6446_dvsdk_combos_2_05

# Where the RTSC tools package is installed.
XDC_INSTALL_DIR=$(HOME)/dvsdk_2_00_00_22/xdctools_3_10_05_61

# Where Framework Components product is installed
FC_INSTALL_DIR=$(HOME)/dvsdk_2_00_00_22/framework_components_2_23_01

# Where DSP/BIOS is installed
BIOS_INSTALL_DIR=$(HOME)/dvsdk_2_00_00_22/bios_5_33_03

# BIOS Utilities
BIOSUTILS_INSTALL_DIR=$(DVSDK_INSTALL_DIR)/biosutils_1_01_00

# Additional RTSC package repositories to be picked up by components.
USER_XDC_PATH=$(CE_INSTALL_DIR)/examples

# Where the TI c6x code generation tools are installed
CODEGEN_INSTALL_DIR=$(HOME)/cg6x_6_0_21
```

```
# Platform Support Package installation directory.
PSP_INSTALL_DIR=$(DVSDK_INSTALL_DIR)/PSP_02_00_00_140

# The directory that points to your kernel source directory.
LINUXKERNEL_INSTALL_DIR=$(HOME)/workdir/lsp/ti-davinci/linux-2.6.18_pro500

# The prefix to be added before the GNU compiler tools (optionally including
# path), i.e. "arm_v5t_le-" or "/opt/bin/arm_v5t_le-".
MVTOOL_DIR=/opt/mv_pro_5.0/montavista/pro/devkit/arm/v5t_le
MVTOOL_PREFIX=$(MVTOOL_DIR)/bin/arm_v5t_le-

# Where to copy the resulting executables and data to (when executing 'make
# install') in a proper file structure. This EXEC_DIR should either be visible
# from the target, or you will have to copy this (whole) directory onto the
# target filesystem.
EXEC_DIR=/home/praktikum/workdir/filesys/opt/dvsdk/$(PLATFORM)
```

H Makefile (XDC-Tools)

Vorkonfiguriertes Makefile für die Benutzung der XDC-Tools und des grafischen Tools zur Erstellung eigener Combofiles.

Makefile

```
include ../Rules.make

XDC = $(XDC_INSTALL_DIR)/xdc

XDCPATH = $(CODEC_INSTALL_DIR);$(CODEC_INSTALL_DIR)/packages;$(CE_INSTALL_DIR)/
        packages;$(CE_INSTALL_DIR)/cetools/packages;$(BIOS_INSTALL_DIR)/packages;
        $(LINK_INSTALL_DIR)/packages

XDCOPTIONS=v

export XDCOPTIONS
export XDCPATH

all: .all-packages

.all-packages:
        $(XDC_INSTALL_DIR)/xs ti.sdo.ce.wizards.genserver
```

I config.bld

Konfigurationsdatei zur Verwendung der Build-Komponenten des XDC-Frameworks bei der Compilierung eines eigenen Combofiles unter Einsatz der CG-Tools.

config.bld

```
var Build = xdc.useModule('xdc.bld.BuildEnvironment');
var Pkg = xdc.useModule('xdc.bld.PackageContents');

var cgToolsDir = "/home/praktikum/dvsdk_2_00_00_22/cg6x_6_0_21";

var C64P = xdc.useModule('ti.targets.C64P');
C64P.rootDir = cgToolsDir;

C64P.platforms = [
        "ti.platforms.evmDM6446"
];

Build.targets.$add(C64P);
```

J Makefile (Combofiles)

Vorkonfiguriertes Makefile für die Compilierung eigener Combofiles.

Makefile

```
XDC_INSTALL_DIR=/home/praktikum/dvsdk_2_00_00_22/xdctools_3_10_05_61

XDCPATH=/home/praktikum/dvsdk_2_00_00_22/dm6446_dvsdk_combos_2_05/packages;/home/
        praktikum/dvsdk_2_00_00_22;/home/praktikum/dvsdk_2_00_00_22/codec_engine_2
        _23_01/packages;/home/praktikum/dvsdk_2_00_00_22/xdctools_3_10_05_61/packages;
        /home/praktikum/dvsdk_2_00_00_22/dsplink-1_61_03-prebuilt/packages;/home/
        praktikum/dvsdk_2_00_00_22/bios_5_33_03/packages;/home/praktikum/dvsdk_2_00_
        00_22/framework_components_2_23_01/packages;/home/praktikum/dvsdk_2_00_00_22/
        xdais_6_23/packages;/home/praktikum/dvsdk_2_00_00_22/biosutils_1_01_00/packages

all:
        "$(XDC_INSTALL_DIR)/xdc" --xdcpath="$(XDCPATH)" release

clean:
        "$(XDC_INSTALL_DIR)/xdc" clean
```

K Makefile (Projekt)

Vorkonfiguriertes Makefile für die Compilierung eigener Projekte.

Makefile

```
ROOTDIR = /home/praktikum/dvsdk_2_00_00_22
TARGET = $(notdir $(CURDIR))

include $(ROOTDIR)/Rules.make

VERBOSE = @

XDC_PATH = $(USER_XDC_PATH);../../packages;$(DMAI_INSTALL_DIR)/packages;
        $(CE_INSTALL_DIR)/packages;$(FC_INSTALL_DIR)/packages;$(LINK_INSTALL_DIR)/
        packages;$(XDAIS_INSTALL_DIR)/packages;$(CMEM_INSTALL_DIR)/packages;
        $(CODEC_INSTALL_DIR)/packages
XDC_CFG = $(TARGET)_config
XDC_CFLAGS = $(XDC_CFG)/compiler.opt
XDC_LFILE = $(XDC_CFG)/linker.cmd
XDC_CFGFILE = $(TARGET).cfg
XDC_PLATFORM = ti.platforms.evmDM6446
XDC_TARGET = gnu.targets.MVArm9
CONFIGURO = $(XDC_INSTALL_DIR)/xs xdc.tools.configuro

C_FLAGS += -Wall -g
LD_FLAGS += -lpthread -lfreetype -lasound -lm
COMPILE.c = $(VERBOSE) $(MVTOOL_PREFIX)gcc $(C_FLAGS) $(CPP_FLAGS) -c
LINK.c = $(VERBOSE) $(MVTOOL_PREFIX)gcc $(LD_FLAGS)
SOURCES = $(TARGET).c
OBJFILES = $(SOURCES:%.c=%.o)

.PHONY: clean install

all: dm6446

dm6446: dm6446_al

dm6446_al: $(TARGET)

install: $(if $(wildcard $(TARGET)), install_$(TARGET))

install_$(TARGET):
        @install -d $(EXEC_DIR)
        @install $(TARGET) $(EXEC_DIR)
        @install $(TARGET).txt $(EXEC_DIR)
        @echo
        @echo Installed $(TARGET) binaries to $(EXEC_DIR)..

$(TARGET): $(OBJFILES) $(XDC_LFILE)
        @echo
        @echo Linking $@ from $^..
        $(LINK.c) -o $@ $^

$(OBJFILES): %.o: %.c $(XDC_CFLAGS)
        @echo Compiling $@ from $<..
        $(COMPILE.c) $(shell cat $(XDC_CFLAGS)) -o $@ $<

$(XDC_LFILE) $(XDC_CFLAGS): $(XDC_CFGFILE)
        @echo
```

```
	@echo ======== Building $(TARGET) ========
	@echo Configuring application using $<
	@echo
	$(VERBOSE) XDCPATH="$(XDC_PATH)" $(CONFIGURO) -c $(MVTOOL_DIR) -o $(XDC_CFG)
	␣ -t $(XDC_TARGET) -p $(XDC_PLATFORM) $(XDC_CFGFILE)

clean:
	@echo Removing generated files..
	$(VERBOSE) -$(RM) -rf $(XDC_CFG) $(OBJFILES) $(TARGET) *~ *.d .dep
```

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Mittweida, den 1. November 2010

Literaturverzeichnis

[1] Texas Instruments
DVSDK_2_00 2_00_00_22 Product Download Page, Nov 2009
http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/S1SDKLNX/DVSDK_2_00/index_FDS.html
(Stand August 2010)

[2] Texas Instruments
TMS320DM6446 DVEVM v2.0 Getting Started Guide, Mai 2009
http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/S1SDKLNX/DVSDK_2_00/exports/docs/sprue66f.pdf
(Stand Jun 2010)

[3] Texas Instruments
C64XPlus_Speech_Codecs 1_00_001 Product Download Page, Mär 2010
http://software-dl.ti.com/dsps/dsps_public_sw/codecs/C64XPlus_Speech/index_FDS.html
(Stand Aug 2010)

[4] Texas Instruments
C64XPlus_Audio_Codecs 1_00_001 Product Download, Jul 2010
http://software-dl.ti.com/dsps/dsps_public_sw/codecs/C64XPlus_Audio/index_FDS.html
(Stand Aug 2010)

[5] Texas Instruments
C64XPlus_Video_CODECS 1_00_002 Product Download Page, Jun 2010
http://software-dl.ti.com/dsps/dsps_public_sw/codecs/C64XPlus_Video/index_FDS.html
(Stand Aug 2010)

[6] MontaVista Software, Inc.
MontaVista Linux Professional Edition 5.0, 2007
www.mvista.com/download/MontaVista-Linux-Pro-5-datasheet.pdf
(Stand Okt 2010)

[7] Stefan Ost, Mathias Grote
Das Linux-Dateisystem, Okt 2007
http://www.uni-muenster.de/ZIV.MathiasGrote/linux/Dateisystem.html
(Stand Okt 2010)

[8] PC Magazine
Definition of API, Juni 1996
http://www.pcmag.com/encyclopedia_term/0,2542,t=application+programming+interface&i=37856,00.asp
(Stand August 2010)

[9] Texas Instruments
Davinci Multimedia Application Interface, Jun 2010
http://processors.wiki.ti.com/index.php/Davinci_Multimedia_Application_Interface
(Stand Sep 2010)

[10] Texas Instruments Members
XDM 1.x Semantics, März 2010
http://processors.wiki.ti.com/index.php/XDM_1.x_Semantics

(Stand August 2010)

[11] Texas Instruments
Codec Engine API Reference, 2009
/home/praktikum/dvsdk_2_00_00_22/codec_engine_2_23_01/docs/html/group___c_o_d_e_c_e_n_g_i_n_e.html
(Stand Apr 2010)

[12] Prof. Dr. Ing. habil. Thomanek
Audiosignalcodierung, 2009
https://www.eit.hs-mittweida.de/index.php?id=1126
(Stand Jul 2010)

[13] Texas Instruments
XDC Consumer User's Guide, Jul 2007
http://focus.tij.co.jp/jp/lit/ug/spruex4/spruex4.pdf
(Stand Okt 2010)

[14] SourceForge Developer
OS Abstraction Layer Project Homepage, Feb 2010
http://osal.sourceforge.net/
(Stand Sept 2010)

Abbildungsverzeichnis