Technische Hochschule Köln

Fakultät IME - NT
Bereich Regelungstechnik
Prof. Dr.-Ing. R. Bartz

SIG: Signalverarbeitung

Teampartner (Name):	**Praktikum**	**Laborplatz:**
Fatima Al khttabi	**Versuch 2**	

Name: Al Housseini	**Studiengang /-richtung** ☒ **Bachelor TIN (IE)** ☐ **Sonstige:**
Vorname: Ahmad	**Versuchs-Datum:** 13.01.2022
Matr.-Nr.: 11145964	**Abgabe-Datum:** 13.01.2022

Task-Nr: 25

Signalverarbeitung mit DSP

VORTESTAT: (erfolgreiche Versuchsdurchführung)	
☐ **Bitte den Versuchsbericht erneut vorlegen.**	
ENDTESTAT: (Vortestat & anerkannter Versuchsbericht)	

Anmerkung: Erteilung des Endtestates bedeutet nicht, dass der Versuchsbericht fehlerfrei ist.

1 Einleitung und Vorbereitung

1.1 Einleitung

Dieser Versuch dient dazu, die Funktionalität eines DT-Systems mit Hilfe eines digitalen Signalprozessors (DSP) zu implementieren.
Diese Versuchsanleitung stellt dazu in Abschnitt 2 die wesentlichen für die Implementierung wichtigen Eigenschaften des DSP und der verwendeten Entwicklungsumgebung dar. Lesen Sie diesen Abschnitt bitte sehr gründlich durch.
In Abschnitt 3 und 4 werden dann die Teilschritte dieses Versuchs beschrieben.

Da Ihnen im Praktikum PC-basierte Tools zur Verfügung stehen (u.a. Word, Excel, MATLAB) sollten Sie **einen eigenen USB-Stick** mitbringen. Auf diesen können Sie Ihre Ergebnisse speichern und ggf. zuhause weiter bearbeiten.

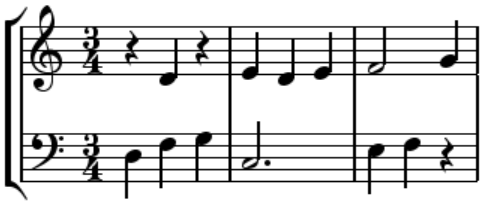
Es hat sich als zweckmäßig erwiesen, die **Unterlagen** (Vorlesung, Übung, Lehrbuch, Hilfsblätter, ... soweit vorhanden) zu "SIG: Signalverarbeitung" **mitzubringen**, da das Wissen in den meisten Fällen sonst nicht verfügbar ist.

Jedem Team wird am Versuchstermin eine konkrete Aufgabenstellung ausgehändigt (sog. **Task-Definition**), die die Details der zu implementierenden Funktionalität definiert. Diese Task-Definition wird an verschiedenen Stellen dieser Versuchsanleitung referenziert; sie trägt eine Task-Nummer.

Tragen Sie diese Task-Nummer auf der Titelseite ein!

1.2 Vorbereitung

Gegeben ist nebenstehende kurze Notensequenz:
Für alle Noten-Unkundigen: Hinweise über Noten und Pausen finden Sie im Anhang (Hilfsblatt 2).
Dort finden Sie auch die zugehörigen Frequenzen.



Ermitteln Sie hierfür die mindestens notwendige Größe des Feldes `to[][]` und bestimmen Sie die Werte der einzelnen Feldelemente (siehe hierzu die Erläuterungen zu einer möglichen Ablagestruktur für Song-Definitionen im Anhang, Hilfsblatt 3). Aus den Erfahrungen der vergangenen Jahre wird keine von Hilfsblatt 3 abweichende Ablagestruktur mehr akzeptiert.

Für die Amplituden der einzelnen Töne soll gelten, dass die Amplitude der tiefen Töne (untere Notenreihe) etwa **dreimal** so groß sein soll wie die der hohen Töne (obere Notenreihe).

Diese Vorbereitung ist **vor dem Versuchstermin** zu bearbeiten und zum Versuchstermin mitzubringen. Sie müssen auf Nachfrage in der Lage sein, die Herleitung zu erläutern.
Das Ergebnis dieser Vorbereitung muss mit dem Versuchsbericht abgegeben werden.
Fehlende Vorbereitung oder mangelndes Verständnis der Ablagestruktur kann zum Ausschluss von der Versuchsteilnahme führen!

2 (Sehr kurze) Einführung in wichtige Aspekte des DSP

Ein **DSP** (**D**igital **S**ignal **P**rocessor) ist ein spezieller Prozessor, dessen interner Aufbau ausgelegt ist für sehr effiziente Multiplikations- und Additions-Aufgaben. Im Praktikum wird der DSP-Typ 6713 (offizielle Bezeichnung TMS320C6713) von Texas Instruments (TI) eingesetzt, integriert in ein Experimentierboard (**DSK**; **D**SP **S**tarter **K**it) DSK6713 der Firma Spectrum Digital. Von den diversen Möglichkeiten dieses Boards wird im Praktikum lediglich die Audio-Input und -Output Funktionalität verwendet.
Software für den DSP wird im Praktikum in C geschrieben. Dafür steht das **CCS** (**C**ode **C**omposer **S**tudio), eine Eclipse-basierte Entwicklungsumgebung von Texas Instruments, in Version 6.1.3 zur Verfügung. Sie bietet neben integrierten Editor-, Compiler- und Linker-Funktionen auch einen leistungsfähigen Debugger. (Für industrielle Anwendungen an der Performance-Grenze würde man jedoch auf eine teilweise oder komplette Implementierung in Assembler wechseln.)

Zum verwendeten DSP, dem Experimentierboard und der Entwicklungsumgebung gibt es im Internet eine große Anzahl frei verfügbarer Beschreibungen und Tutorials, die Sie gerne für die Vorbereitung hinzuziehen können; erste Anlaufstellen ist dabei sicher die Webseite

- https://www.ti.com/product/TMS320C6713B

Die folgenden Erläuterungen fassen die wichtigsten Aspekte kurz zusammen; mit den hier beschriebenen Aspekten des DSP, des DSK und des CCS sollten Sie in der Lage sein die Praktikums-Aufgaben erfolgreich zu implementieren.

CCS Projekt:
Die Installation des CCS ist bereits für den Versuch vorbereitet. Sie finden im Eclipse Workspace drei Projekte jeweils mit den elementaren Einstellungen und einem rudimentären main.c vor. Das erste Projekt ist für die drei Unteraufgaben der Filterung gedacht, das zweite Projekt für den Test-Ton und das dritte Projekt für Test-Song und Real-Song (s. nächste Kapitel).
Obwohl für die Durchführung des Praktikums nicht erforderlich, dürfen Sie gerne weitere Projekte für sich einrichten; sollten Sie dabei auf Probleme stoßen, wenden Sie sich an Ihren Betreuer.

CCS Round Trip Engineering:
Der übliche Ablauf bei der Entwicklung Ihrer Software ist:

1) Sie editieren Ihren C-Code (und speichern ihn hin und wieder), bis Sie der Meinung sind, ein weiterer Schritt ist implementiert und sollte getestet werden.
   Da Sie nicht beabsichtigen Ihren Code vorzukompilieren und als IP-haltige Library geschützt weiterzugeben, ist eine Trennung in Header (.h) und Source-Files (.c) nicht erforderlich; Sie können die Deklarationen direkt in den .c-File schreiben.
   Die Praktikums-Aufgaben erfordern keinen sehr umfangreichen Code; in der Regel genügt es, wenn Sie in ein und demselben .c-File sowohl das main-Programm als auch die eventuell von Ihnen erstellten Unterprogramme halten. Sollten Sie dennoch Unterprogramme in separate Dateien auslagern wollen, geht das über das Menu 'File→New→Source File'.
2) Sie initiieren den Generierungs-Prozess "Compile→Link→Download" durch Betätigen des Debug-ToolButtons.
   
   Falls dabei Fehler gemeldet werden, beheben Sie die Ursache und wiederholen 1) und 2).
3) Bei fehlerfreier Übersetzung wechselt CCS automatisch in die Debug-Ansicht. Dort stehen Ihnen entsprechende Hilfsmittel zur Verfügung die das Testen und die potentielle Fehlersuche erleichtern:
   - Ein Code-Fenster zeigt den Source-Code und erlaubt Änderungen vorzunehmen; diese werden allerdings erst wirksam wenn der Generierungs-Prozess erneut durchlaufen ist.
   - Ein Debug-Fenster erlaubt Interaktionen mit dem Programm; die Wichtigsten sind:
     Ein ToolButton 'Resume' startet das Programm an der Stelle an der es aktuell steht; zu Anfang wird es somit mit dem ersten Befehl in main.c beginnen.
     
     Ein ToolButton 'Suspend' hält ein laufendes Programm an; es bleibt dort stehen wo es sich gerade zufällig befindet; das kann auch eine Stelle in von Ihnen aufgerufenen Library-Funktionen sein, für die der Source-Code dann meist nicht angezeigt werden kann. Sollte der Source-Code verfügbar sein wird die Zeile markiert, **vor** der das Programm stehengeblieben ist.
     
     Ein ToolButton 'StepOver' führt den Befehl der aktuell markierten Zeile aus und positioniert das Programm vor die nächste auszuführende Zeile (die dann die Markierung erhält).
     
     Ein ToolButton 'StepInto' ist sinnvoll bei Zeilen die einen Funktionsaufruf beinhalten; er bewirkt, dass die aufgerufene Funktion im Codefenster angezeigt wird (sofern der Source Code verfügbar ist) und positioniert das Programm vor die erste darin ausführbare Zeile. Wird 'StepIn' auf Zeilen ohne Funktionsaufruf ausgeführt verhält es sich wie 'StepOver'.

Ein ToolButton 'Restart' positioniert das Programm wieder an den Anfang des main.c. Das ist allerdings mit Vorsicht zu benutzen, da Variablenwerte ggf. erhalten bleiben.

- Im Code-Fenster können **breakpoints** gesetzt werden; dies sind Code-Stellen, bei deren Erreichen das Programm anhalten soll. Sie werden durch Doppelclick auf die graue Leiste in Höhe der Zeile gesetzt, **vor** der das Programm anhalten soll. Ein nachfolgendes 'Resume' startet das Programm und führt es aus bis der nächste Breakpoint erreicht ist.
- Ein Anzeigebereich auf der rechten Seite erlaubt die lokalen Variablen (auf dem Tab 'Variables') sowie globale Variable und weitere Ausdrücke (auf dem Tab 'Expressions') anzuzeigen. In der Spalte 'Value' kann der jeweils aktuelle Wert betrachtet und auch verändert werden. Eine Aktualisierung erfolgt wegen der Echtzeitforderungen nicht bei laufendem Programm, sondern nur wenn es steht und auf weitere Schritte wartet.

4) Wenn der Testlauf abgeschlossen ist und Sie genügend Informationen gewonnen haben um Ihre Software zu verändern bzw. zu erweitern, gehen Sie zu 1); dies wiederholen Sie so lange bis die jeweilige Praktikumsaufgabe gelöst ist.

Eingebundene Fremdsoftware:
Für die Durchführung des Praktikums greifen wir auf einige wenige Definitionen und Funktionen der TI **CSL** (**C**hip **S**upport **L**ibrary) und der Spectrum Digital **BSL** (**B**oard **S**upport **L**ibrary) zu.

Diese sind:

- Die Definition der Audio-Codec Konfiguration mit geeigneten Einstellwerten. Empfohlen ist:

```
static DSK6713_AIC23_Config config = { \
  0x0017,  /* 0 DSK6713_AIC23_LEFTINVOL  Left line input channel volume 0dB*/ \
  0x0017,  /* 1 DSK6713_AIC23_RIGHTINVOL Right line input channel volume 0dB*/\
  0x00f9,  /* 2 DSK6713_AIC23_LEFTHPVOL  Left channel headphone volume 0dB*/  \
  0x00f9,  /* 3 DSK6713_AIC23_RIGHTHPVOL Right channel headphone volume 0dB*/ \
  0x0011,  /* 4 DSK6713_AIC23_ANAPATH    Analog audio path control    */      \
  0x0000,  /* 5 DSK6713_AIC23_DIGPATH    Digital audio path control    */     \
  0x0000,  /* 6 DSK6713_AIC23_POWERDOWN  Power down control    */            \
  0x0043,  /* 7 DSK6713_AIC23_DIGIF      Digital audio interface format    */ \
  0x0081,  /* 8 DSK6713_AIC23_SAMPLERATE Sample rate control    */           \
  0x0001   /* 9 DSK6713_AIC23_DIGACT     Digital interface activation    */   \
};
```

- Die Definition des Codec-Handle: `DSK6713_AIC23_CodecHandle`
  Damit kann ein weiter unten benötigter Handle für den Audio Codec erzeugt werden:
  `DSK6713_AIC23_CodecHandle hCodec;`
- Die Initialisierungs-Funktion für die Board Support Library (BSL); sie wird verwendet um die Register des DSP auf für das Board angepasste Werte zu setzen, und muss aufgerufen worden sein, bevor der Audio-Codec initialisiert wird:
  `DSK6713_init();`
- Die Funktion zum Initialisieren des Audio-Codec; sie sorgt dafür dass der Codec Baustein über die SPI-Schnittstelle des DSP angesprochen und seine Register mit den entsprechenden Werten geladen werden:
  `hCodec = DSK6713_AIC23_openCodec(0, &config);`
- Die Funktion zum Lesen des aktuellen Eingangs-Sample `in` vom Audio-Codec:
  `DSK6713_AIC23_read(hCodec, &in);`
  Sie sorgt dafür, dass der Sample-Wert über die SPI-Schnittstelle des DSP vom Audio-Codec entgegengenommen wird, und liefert im Erfolgsfall ein `true` zurück. Der Sample-Wert findet sich dann in der Speicherstelle von `in`; die Variable `in` muss vom Typ `Uint32` sein. Darin enthalten allerdings nur die unteren 16 bit den vorzeichenbehafteten Integer-Wert des Samples. Um diesen zu extrahieren ist zunächst eine Maskierung der unteren 16 Bits vorzunehmen und das Ergebnis auf eine `short`-Variable zu casten; das bewahrt die Vorzeichen-Information.
  Die Konfiguration von SPI und Codec ist gerade so gewählt, dass durch Abfrage auf `true` eine zu f_s=48000/s passende Sample-Zeit T erreicht wird; eine mögliche Code-Sequenz zum Einlesen des nächsten Sample in eine Variable `sample` vom Typ `short` wäre dann:

```
while(!DSK6713_AIC23_read(hCodec,&in));
sample=(short)in & 0x0000FFFF;
```

  Da der Audio-Codec in Stereo arbeitet, wird mit dieser Funktion abwechselnd ein Sample-Wert für den linken und den rechten Kanal empfangen; sollen beide Stereo-Kanäle verarbeitet werden, muss diese Funktion also jeweils abwechselnd den linken und den rechten Kanal verarbeiten (oder z.B. denselben Sample-Wert in entsprechende Variablen `inL` und `inR` füllen); ansonsten muss das Ergebnis jedes zweiten Aufrufs ignoriert werden.

- Die Funktion zum Übertragen des nächsten auszugebenden Sample-Wertes `out` an den Audio-Codec:
  ```
  DSK6713_AIC23_write(hCodec, out);
  ```
  Sie sorgt dafür dass der Sample-Wert über die SPI-Schnittstelle des DSP an den Audio-Codec übertragen wird, und liefert im Erfolgsfall ein `true` zurück. Die Variable `out` ist vom Typ `short`.
  Die Konfiguration von SPI und Codec ist gerade so gewählt, dass durch Abfrage auf `true` eine zu f_s=48000/s passende Sample-Zeit T erreicht wird:
  ```
  while(!DSK6713_AIC23_write(hCodec,out));
  ```
  Da der Audio-Codec in Stereo arbeitet, wird mit dieser Funktion abwechselnd ein Sample für den linken und den rechten Kanal übertragen; sollen beide Stereo-Kanäle dasselbe Audio-Signal erzeugen, muss diese Funktion also zweimal hintereinander mit demselben Wert für `out` aufgerufen werden.
- Die Funktion zum Schließen des Audio-Codec; sie sollte aufgerufen werden, wenn der Audio-Codec nicht mehr benötigt wird (also am Ende des Programms):
  ```
  DSK6713_AIC23_closeCodec(hCodec);
  ```

Problembehebung:

- Manchmal meldet der Compiler Fehler obwohl offensichtlich alle Fehler beseitigt wurden. Dies kann evtl. behoben werden durch 'Clean Project' aus dem Kontext-Menu des Projektes und nachfolgendem erneuten "Compile→Link→Download" Versuch.
- Wenn das Programm durch den ToolButton 'Suspend' angehalten wurde, kann es u.U. an einer Stelle stehen, für die kein Source-Code gegeben ist. Wenn trotzdem eine Einzelschritt-Analyse durchgeführt werden soll, hilft in solchen Fällen meist das (ggf. mehrmalige) Betätigen des ToolButton 'StepOver'. Alternativ kann im Code-Fenster auf den Tab mit dem SourceCode Ihres main-Programms gewechselt und dort ein geeigneter Breakpoint gesetzt werden; ein nachfolgendes 'Resume' führt dann dazu, dass das Programm in Ihrem SourceCode anhält, wenn es das nächste Mal den Breakpoint erreicht.
- In seltenen Fällen lässt sich das Programm im DSP nicht (mehr) starten (z.B. wenn relevante Systemregister oder -speicherstellen durch Ihr Programm versehentlich überschrieben wurden). Dies kann meist behoben werden durch den ToolButton 'CPU Reset' gefolgt vom ToolButton 'Load'.
- Ausgesprochen selten lässt sich das DSK nicht mehr ansprechen. In solchen Fällen hilft meist, das DSK für einige Sekunden aus- und dann wieder einzuschalten. Danach sollte man noch etwas auf die Wiederherstellung der USB-Verbindung warten, und dann den Debug-Prozess erneut in die Wege leiten.
- Im Zweifelsfall fragen Sie Ihren Betreuer.

3 Teil 1: Filterung eines Audio-Streams

In diesem Versuchsteil soll ein DSP-Programm erstellt und angewendet werden, das ein DT-System implementiert. Dabei soll Echtzeit-Verhalten erreicht werden: jedes eingehende Sample des Audio-Stream soll unter Berücksichtigung der vergangenen Samples verarbeitet werden und unmittelbar ein Ausgangs-Sample erzeugen.

Dazu soll das System in DF1 implementiert werden. Eine kurze Übersicht über die DF1-Implementierung findet sich im Anhang (Hilfsblatt 1).

Folgende Notation soll eingehalten werden:
Die Variable `xL` soll das aktuelle Sample des linken Eingangs-Kanals beinhalten, die Variable `xR` das des rechten Eingangs-Kanals.
Die Variable `yL` soll das berechnete Sample des linken Ausgangs-Kanals beinhalten, die Variable `yR` das des rechten Ausgangs-Kanals.
Die Koeffizienten des Nenners der Übertragungsfunktion sollen in einem Array `a[]` gespeichert werden.
Die Koeffizienten des Zählers der Übertragungsfunktion sollen in einem Array `b[]` gespeichert werden.

Hinweis: Obwohl vom Audio-Codec für Eingangs-Signale effektiv ein 16 bit signed Integer Wert empfangen wird und das berechnete Ausgangs-Sample schließlich auch als 16 bit signed Integer Wert an den Audio-Codec übergeben wird, sollten Eingangswerte **frühestmöglich** in eine double-Repräsentation überführt werden; die Berechnungen sollten grundsätzlich mit Variablen vom Typ double durchgeführt werden und der typecast auf short für die Ausgabe an den Audio-Codec sollte zum **spätestmöglichen** Zeitpunkt erfolgen.

Als Audio-Quelle wird ein MP3-Player/Stick verwendet.

3.1 Test Audio-In und Audio-Out

Schreiben Sie ein DSP-Programm, das den Audio-Stream beider Kanäle einliest und mit einem individuell einstellbaren Faktor (`double` Variable) für die Verstärkung wieder ausgibt; Eingangs- und Ausgangs-Signale müssen weitgehend gleich sein.
Bestimmen Sie den Variablenwert so, dass Eingangs- und Ausgangs-Signale etwa gleich laut erscheinen.
Führen Sie das Ergebnis Ihrem Betreuer vor, und dokumentieren Sie den Faktor.

3.2 Filter-Implementierung und Test

a) Schreiben Sie ein DSP-Programm, das die Filterung entsprechend obiger Anforderungen implementiert. Dimensionieren Sie dabei die Arrays `a[]` und `b[]` bereits so, dass Sie damit auch die in Ihrer Task-Definition gegebene Filter-Definition implementieren könnten.
b) Verifizieren Sie das Programm indem Sie es mit der Konfiguration
   `b[0]=1.0;`
   `a[i]=b[i]=0.0;` für i>0
   testen; Eingangs- und Ausgangs-Signale müssen dabei weitgehend gleich sein.
   Führen Sie das Ergebnis Ihrem Betreuer vor.

3.3 Filterung mit spezieller Filtercharakteristik

a) Passen Sie das Programm auf die in Ihrer Task-Definition gegebene Filter-Definition an (die Filter haben höchstens die Ordnung 10).
b) Starten Sie die Filterung und beobachten Sie das Ergebnis anhand der Ausgabe auf dem Lautsprecher. Beschreiben Sie Ihre Beobachtungen in kurzen Worten.
c) Lassen Sie Ihre Implementierung durch Ihren Betreuer verifizieren; dafür wird ein Testsignal an den Eingang des Systems gelegt und der Ausgang auf einem Oszilloskop beobachtet.

4 Teil 2: Song-Generator

Unter einem Song soll hier eine polyphone (mehrstimmige) Sequenz von Tönen verstanden werden. Implementierungen beschränken dabei üblicherweise die Anzahl gleichzeitig möglicher Töne.
Die Tonsequenzen jeder einzelnen Stimme werden jeweils einer Spur (Track) zugeordnet.
In jedem Track wird zu jedem Zeitpunkt immer nur ein Ton verarbeitet: seine Sample-Werte werden berechnet und bereitgestellt. Das gesamte Ausgangssignal entsteht dann, indem zu jedem Abtastzeitpunkt die Sample-Werte aller Tracks aufsummiert und der Summenwert an den Audio-Codec übergeben wird. Dieser erzeugt dann über einen Lautsprecher ein zeitkontinuierliches Schall-Signal.

Mit Hilfe des DSP sollen in diesem Versuchsteil derartige Songs generiert werden. Dabei wird auf die Definition eines Song mit Hilfe von Noten und Pausen zurückgegriffen.

Ein einzelner Ton lässt sich als sinusförmiges Signal darstellen: $x[nT]=A\cdot\sin(2\pi f_0 nT+\theta)$; darin ist A die Amplitude, f_0 die Frequenz und θ die Phase.
Eine Note beschreibt die Frequenz und die zeitliche Dauer eines Tons. Die Zuordnung zwischen der Note und ihrer Frequenz / ihrer Dauer kann dem Anhang (Hilfsblatt 2) entnommen werden.
Die Amplitude des Tons ist zunächst frei. Allerdings führt bei gleichzeitig klingenden Tönen ein zu großer Unterschied ihrer Amplituden dazu, dass die 'leisen' Töne u.U. nicht mehr wahrgenommen werden. Weiterhin gilt: tiefe Töne benötigen eine höhere Amplitude um als 'gleich laut' wahrgenommen zu werden. Schließlich sollte darauf geachtet werden, dass die Summe aller gleichzeitig klingender Töne nicht den maximal zulässigen Ausgangs-Wert von etwa $\pm$1V (entspricht einem Sample-Wert `out` von $\pm$32767) überschreitet, da sich sonst Verzerrungen bemerkbar machen. Das erzeugte Audio-Signal kann z.B. mit einem Oszilloskop beobachtet werden und liefert dann einen Hinweis ob die Amplituden zu groß oder zu klein gewählt sind.
Die Phase soll in diesem Versuch vernachlässigt werden (indem θ=0 gesetzt wird). Die Verwendung des sin() führt dann gegenüber dem cos() bei Beginn des Tons zu geringeren Störgeräuschen und ist zu bevorzugen.

Ein Song setzt sich aus mehreren Tracks zusammen, die jeweils Noten und Pausen enthalten. Eine mögliche strukturierte Definition ist im Anhang (Hilfsblatt 3) erläutert; es wird empfohlen sich daran zu orientieren.

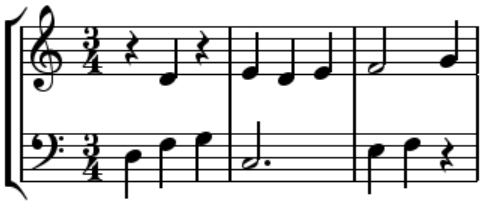
Hinweis: Obwohl an den Audio-Codec schließlich ein 16 bit signed Integer Wert übergeben wird, sollte grundsätzlich mit Werten vom Typ double gerechnet werden und der typecast auf short zum **spätestmöglichen** Zeitpunkt erfolgen.

4.1 Test-Ton

a) Schreiben Sie ein DSP-Programm, das einen Ton erzeugt. Frequenz und Amplitude entnehmen Sie Ihrer Task-Definition. Dabei sollen die Samples nicht in einem Array gespeichert werden; der jeweils nächste Sample des Tons soll allein durch Erhöhung der Zeit (nT) in der sinus-Funktion erfolgen. Um numerische Überläufe zu vermeiden soll zudem das Argument der sinus-Funktion immer im Intervall [0, 2π] gehalten werden.
b) Spielen Sie den Ton auf dem Lautsprecher ab, analysieren Sie den Signalverlauf mit dem Oszilloskop, und stellen Sie das Ergebnis Ihrem Betreuer vor.

4.2 Test-Song

a) Schreiben Sie ein DSP-Programm, das die nebenstehend dargestellte kurze zweistimmige Tonsequenz erzeugt und in einer Endlos-Schleife wiederholt.
Eine **Viertel-Note** soll **150 BPM** aufweisen.
b) Spielen Sie den Song auf dem Lautsprecher ab und stellen Sie das Ergebnis Ihrem Betreuer vor.



4.3 Real-Song

a) Schreiben Sie ein DSP-Programm, das die ersten Takte eines längeren Song generiert und in einer Endlos-Schleife wiederholt. Die Noten des Song werden Ihnen am Versuchsdatum ausgehändigt.
b) Spielen Sie den Song auf dem Lautsprecher ab und stellen Sie das Ergebnis Ihrem Betreuer vor.

5 Versuchs-Bericht

Jeder Teilnehmer hat einen eigenen Versuchs-Bericht in elektronischer Form per email an den Betreuer abzugeben. Zippen Sie alle unten genannten Dateien in **12345678.zip** (darin steht 12345678 für Ihre Matr.-Nr.) - achten Sie darauf dass es tatsächlich ein **zip** Format ist (nicht Formate wie rar, 7z, …).

Zum Versuchs-Bericht gehören (mit den hier angegebenen Dateinamen und -typen!):
1. das Deckblatt dieser **Versuchsanleitung** (vollständig ausgefüllt, gescannt) [v2_Deck.pdf]
2. Ihre **Task-Definition** (vollständig ausgefüllt, gescannt) [v2_TaskDef.pdf]
3. die Ihnen bereitgestellten **Noten** des Song (gescannt) [v2_Song.pdf]

sowie die **Dokumentation Ihrer Versuchsdurchführung**:
4. Vorbereitungs-Schritt zu 1.2: Beschreibung des Array `to[][]` (gescannt) [v2_Array_12.pdf]
5. Source Code des Programms zu 3.1 [v2_code_31.c]
6. Source Code des Programms zu 3.2a [v2_code_32.c]
7. Source Code des Programms zu 3.3a [v2_code_33.c]
8. Beschreibung der Beobachtungen aus 3.3b (gescannt) [v2_notes_33.pdf]
9. Source Code des Programms zu 4.1a [v2_code_41.c]
10. Source Code des Programms zu 4.2a [v2_code_42.c]
11. Source Code des Programms zu 4.3a [v2_code_43.c]

Bitte prüfen Sie unbedingt vor Abgabe, ob sich im zip-File 11 Dateien mit den gegebenen Namen befinden!

6 Hinweise

Denken Sie daran, Ihre Daten stets auf **einem eigenen USB-Stick** zu sichern. Es kann nicht davon ausgegangen werden, dass die zuletzt von Ihnen erarbeiteten Ergebnisse nach Ende Ihres Labortermins noch auf dem Laborrechner vorliegen.

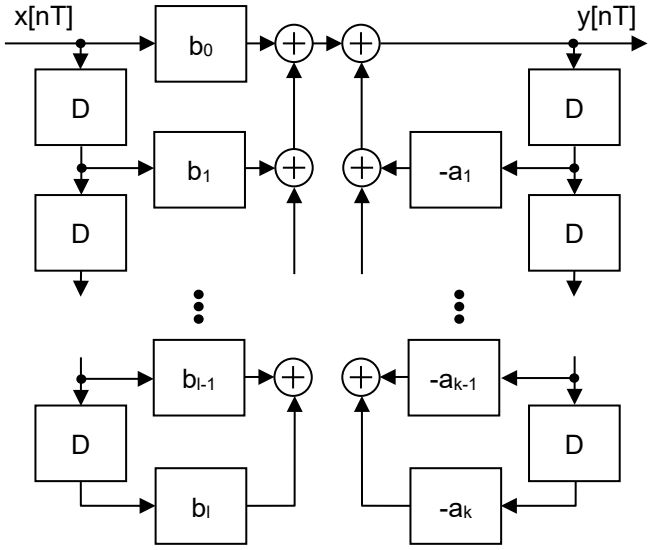
Hilfsblatt 1

DF1-Implementierung eines DT-Systems

Ein DT-LTI-System kann u.a. durch eine Differenzengleichung oder eine Übertragungsfunktion H(z) beschrieben werden; beide Beschreibungsarten lassen sich einfach ineinander überführen.

$$y[nT] = \sum_{i=0}^{l} (b_i) \cdot x[(n-i)T] + \sum_{i=1}^{k} (-a_i) \cdot y[(n-i)T] \qquad H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + ... + b_{l-1} z^{-(l-1)} + b_l z^{-l}}{1 + a_1 z^{-1} + a_2 z^{-2} + ... + a_{k-1} z^{-(k-1)} + a_k z^{-k}}$$

Aus der Differenzengleichung ist unmittelbar ein systematischer Ansatz für die Implementierung eines solchen Systems ableitbar: die **DF1** (**D**irect **F**orm **1**). Sie ist in Form eines Blockschaltbildes im folgenden Bild dargestellt. Ein Beispielcode für k=l=1 ist ebenfalls angegeben. Die Implementierung entspricht dabei der in der Vorlesung behandelten rekursiv numerischen Methode.




```
//Beispiel f. 1.Ordnung DF1
void main(void){
  double a1=3.0,b0=1.0,b1=2.0;
  double x,x1=0.0,y,y1=0.0;
  while(1){
    x=getx();//next sample
    y=x*b0+x1*b1-y1*a1;
    x1=x;
    y1=y;
    puty(y);
    wait();
  }
}
```

Die im Code verwendete Funktion `getx()` steht symbolisch für den Vorgang, mit dem ein Sample ermittelt werden kann; die Funktion `puty(..)` steht symbolisch für den Vorgang, mit dem das Ergebnis-Sample abgeliefert wird.

Für eine flexiblere Implementierung können die Koeffizienten a_i und b_i sowie die erforderlichen Samples der Signale x[nT] und y[nT] jeweils in Arrays abgelegt werden.

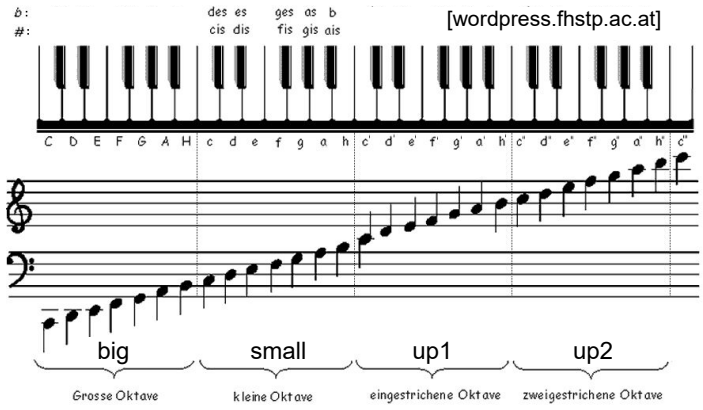
Einen weiteren Performancegewinn erzielt man, wenn nicht die Sample-Werte zwischen den Speicherstellen umkopiert werden, sondern der Zugriff auf die Sample-Werte über eine Manipulation der Array-Indizes erfolgt. Dies ist im Praktikum allerdings nicht erforderlich.

Hilfsblatt 2

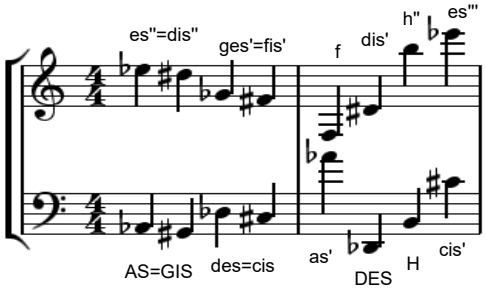
Noten und Pausen

Noten: Tonhöhen

Die Grafik stellt einen Ausschnitt aus dem Tastensystem eines Klaviers und die den weißen Tasten zugeordneten Noten dar. Schwarze Tasten sind sog. Halbtonschritte zwischen benachbarten weißen Tasten; benachbarte weiße Tasten ohne zwischenliegende schwarze Taste sind selber nur einen Halbtonschritt voneinander entfernt.
In der Notenschrift erreicht man die schwarzen Tasten durch ein vor die Note der linken Taste gesetztes #-Zeichen oder ein vor die Note der rechten Taste gesetztes b-Zeichen.



Allgemein erhöht ein #-Zeichen die jeweilige Note um einen Halbtonschritt während das b-Zeichen die Note um einen Halbtonschritt erniedrigt (s. Beispiele im nebenstehenden Bild). Diese sog. Vorzeichen können auch zwischen/auf einer Linie am Anfang einer Notenzeile stehen und beziehen sich dann auf alle Noten, die auf derselben Linie stehen (sowie auf alle um Vielfache einer Oktav davon entfernte Noten).

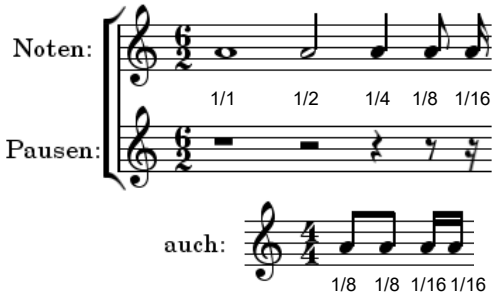


Die Frequenzen zu den verschiedenen Noten sind (bei der sog. gleichstufigen Stimmung) in folgender Tabelle dargestellt; in Halbtonschritten benachbarte Noten haben ein Frequenzverhältnis von $2^{1/12}$.

	sub2	sub1	big	small	up1	up2	up3	up4	up5	up6	up7
c	16.351598	32.703196	65.406391	130.812783	261.625565	523.251131	1046.502261	2093.004522	4186.009045	8372.018090	16744.036179
cis=des	17.323914	34.647829	69.295658	138.591315	277.182631	554.365262	1108.730524	2217.461048	4434.922096	8869.844191	17739.688383
d	18.354048	36.708096	73.416192	146.832384	293.664768	587.329536	1174.659072	2349.318143	4698.636287	9397.272573	18794.545147
dis=es	19.445436	38.890873	77.781746	155.563492	311.126984	622.253967	1244.507935	2489.015870	4978.031740	9956.063479	19912.126958
e	20.601722	41.203445	82.406889	164.813778	329.627557	659.255114	1318.510228	2637.020455	5274.040911	10548.081821	21096.163642
f	21.826764	43.653529	87.307058	174.614116	349.228231	698.456463	1396.912926	2793.825851	5587.651703	11175.303406	22350.606812
fis=ges	23.124651	46.249303	92.498606	184.997211	369.994423	739.988845	1479.977691	2959.955382	5919.910763	11839.821527	23679.643054
g	24.499715	48.999429	97.998859	195.997718	391.995436	783.990872	1567.981744	3135.963488	6271.926976	12543.853951	25087.707903
gis=as	25.956544	51.913087	103.826174	207.652349	415.304698	830.609395	1661.218790	3322.437581	6644.875161	13289.750323	26579.500645
a	27.5	55.0	110.0	220.0	**440.0**	880.0	1760.0	3520.0	7040.0	14080.0	28160.0
ais=b/bes	29.135235	58.270470	116.540940	233.081881	466.163762	932.327523	1864.655046	3729.310092	7458.620184	14917.240369	29834.480737
h/b	30.867706	61.735413	123.470825	246.941651	493.883301	987.766603	1975.533205	3951.066410	7902.132820	15804.265640	31608.531280

Noten und Pausen: Dauer (zeitliche Länge)

Die Dauer einer Note bzw. einer Pause wird durch das Notensymbol selber dargestellt. Zwischen Noten können Pausen eingefügt sein, deren Dauer ebenfalls durch ihr Symbol dargestellt wird. In beiden Fällen handelt sich dabei immer um die Angabe relativ zur Dauer einer Referenz-Note.
Wenn ein kleiner Punkt hinter dem Symbol erscheint, verlängert sich die Dauer um 50%.



Die zeitliche Dauer einer Referenz-Note wird für jedes Musikstück individuell vorgegeben, in der Regel als **BPM** (**B**eats **P**er **M**inute); häufig wird als Referenz-Note die Viertelnote verwendet. Manchmal findet sich diese Angabe über dem ersten Takt des Musikstücks.

Mit diesen Informationen lässt sich der Ausschnitt in 4.2 wie folgt interpretieren:

- die Noten im oberen System sind: d'-e'-d'-e'-f'-g'; vor und nach der ersten Note befindet sich eine Pause; die meisten Noten und beide Pausen sind je 1/4 lang; die vorletzte Note hat die Länge 1/2.
- die Noten im unteren System sind: d-f-g-c-e-f; die Pause, die ersten drei und die letzten beiden Noten sind je 1/4 lang, die vierte Note hat die Länge 3/4.

Hilfsblatt 3

Strukturierte Definition eines Song: ein möglicher Ansatz

Zwei Vorüberlegungen:

- Ein Song besteht aus einer Vielzahl von Noten die jeweils eine bestimmte Länge haben. In der Regel lässt sich daraus ein Zeitintervall T_{TSL} bestimmen, so dass die Länge jeder Note als ein ganzzahliges Vielfaches dieses Zeitintervalls dargestellt werden kann. Dieses Zeitintervall sei mit **Tone-Slot** bezeichnet. Häufig ist T_{TSL} die Dauer der kürzesten Note im Song.
  Über die zeitliche Dauer der Referenz-Note und die Abtastfrequenz f_s=48000Hz kann die zu einem Tone-Slot gehörende Anzahl Samples `slotLen` bestimmt werden. Dabei ist zu beachten, dass `slotLen` auf eine ganze Zahl gerundet werden muss; ggf. wird dadurch die Geschwindigkeit des Song leicht verändert.
  Der Song kann dann zeitlich aufgeteilt werden in eine Sequenz von Tone-Slots, die für die Song-Generierung nacheinander abgearbeitet werden können; sie können über einen Index `actSlot` gekennzeichnet werden. Die Anzahl erforderlicher Tone-Slots sei `numToneSlots`; sie bestimmt sich aus der Länge des Songs (wobei eventuell wiederkehrende Songteile mit berücksichtigt werden müssen).
  Somit ergibt sich eine for-Schleife: `for(actSlot=0; actSlot<numToneSlots; actSlot++);`

Jede Note des Song beginnt zu Anfang eines Tone-Slot und endet am Ende desselben oder eines späteren Tone-Slot. Da im Praktikum davon ausgegangen wird, dass für **eine Note** genau **ein Ton** generiert wird, genügt zur Charakterisierung einer Note (neben Beginn- und Ende-Slot) die Angabe ihrer Frequenz und Amplitude. Damit kann man eine Struktur für eine durch einen Ton implementierte Note definieren:

```
struct tone{
  int bSlot;  //Beginn-Slot; erster Slot in der die Note gespielt wird
  int eSlot;  //Ende-Slot (>=bSlot); letzter Slot in der die Note gespielt wird
  double freq;
  double amp;    };
```

Wenn `actSlot>=bSlot` und `actSlot<=eSlot` erklingt die Note.

- In einem Song können mehrere Noten zur gleichen Zeit auftreten. Dies ist z.B. der Fall wenn mehrere Stimmen gleichzeitig klingen oder mehrere Instrumente zur gleichen Zeit spielen sollen.
  Um dies zu beschreiben kann der Song in mehrere parallele **Track**s aufgeteilt werden.
  Jeder Track ist dann wie oben beschrieben in dieselben Tone-Slots aufgeteilt.
  Die Anzahl erforderlicher Tracks für einen Song sei `NUM_TRACKS`; sie bestimmt sich aus der Bedingung, dass einem Tone-Slot eines Track höchstens eine Note zugeordnet werden darf. In der Regel kann `NUM_TRACKS` aus der höchsten Anzahl zeitgleicher Noten des Song abgelesen werden.

In jedem Track gibt es die gleiche Anzahl von Tone-Slots. Die Zahl der Noten in den einzelnen Tracks kann aber verschieden sein, da ja eine Note über mehr als einen Tone-Slot erklingen kann und Pausen in einem Track auftreten können.
Dies kann abgebildet werden durch ein zweidimensionales Array `to`, in dem jedes Array-Element vom Struct-Typ `tone` ist:

```
struct tone to[NUM_TRACKS][MAX_TONES_PER_TRACK];
```

Dabei ist `MAX_TONES_PER_TRACK` die Anzahl der Noten im längsten Track (dem mit den meisten Noten).
Das Element `to[i][j]` beschreibt dann die j-te Note im i-ten Track; im längsten Track werden alle Array-Elemente verwendet, in den anderen Tracks können unbenutzte Array-Elemente auftreten, die dann geeignet initialisiert sein müssen.

Ein Song wird somit vollständig beschrieben durch die Angabe des Arrays `to` und der Anzahl Samples `slotLen` pro Tone-Slot.

Laufzeit-Verhalten:
Beim Start des Song wird mit Tone-Slot 0 begonnen; im Array `to` werden alle Tracks daraufhin untersucht, ob sie in diesem Tone-Slot eine Note enthalten. Wenn ja wird ihre Frequenz und ihre Amplitude verwendet um daraus über alle Samples im Tone-Slot einen Signalanteil für das Ausgangssignal zu erzeugen (wie schon in 4.1 implementiert); die Beiträge aller Tracks werden zu jedem Sample-Zeitpunkt addiert und ausgegeben.
Sobald die Zeit für Tone-Slot 0 vorüber ist wird für Tone-Slot 1 und folgende dieselbe Prozedur angewandt.
Sollte eine Note sich über mehrere Tone-Slots hinziehen, ist es hilfreich das Argument des sin() fortzuführen und nicht erneut bei 0 beginnen zu lassen; das reduziert potentielle Störgeräusche.
Wenn der Song zu Ende ist soll er wieder automatisch neu starten.