Karlsruher Institut für Technologie

IIIT

Prof. Dr.-Ing. habil. K. Dostert

Praktikum: „Mikrocontroller und digitale Signalprozessoren“

Versuch 2

Digitale Frequenzsynthese mit dem µC ADuC7026

Inhaltsverzeichnis

2.1 Einleitung

Die digitale Frequenzsynthese beschäftigt sich mit der Erzeugung kontinuierlicher Signale aus diskreten Werten. Diese Aufgabenstellung hat einen durchaus realen Hintergrund. Häufig werden verschiedene elektrische Signale zum Testen von Filtern, zur Übertragung von Informationen und zum Messen von Frequenzgängen benötigt. Je nach Anwendung sind die Anforderungen an die Signale bezüglich Form und Frequenz sehr unterschiedlich. Da sich nicht alle Formen und Frequenzen mit angemessenem Aufwand analog erzeugen lassen, wird häufig der Weg über die digitale Frequenzsynthese beschritten. Er bietet einerseits die Möglichkeit, Signale in nahezu beliebiger Form und Frequenz zu erzeugen, andererseits ist die Langzeitstabilität der Signale nicht mehr von sich verändernden Bauteilkoeffizienten abhängig, wie es bei der analogen Erzeugung mit Hilfe von Schwingkreisen der Fall ist.

Zunächst wird ein kleiner Exkurs in die digitale Signalverarbeitung unternommen. Anschließend wird auf die Implementierung der digitalen Frequenzsynthese auf dem Mikrocontrollersystem eingegangen.

2.2 Grundlagen

Die Synthese kontinuierlicher Signale aus diskreten Werten beruht auf der digitalen Signalverarbeitung. Die für das Verständnis notwendigen mathematischen Grundlagen sind die kontinuierliche Fouriertransformation sowie die diskrete Fouriertransformation (DFT) bzw. die schnelle Fouriertransformation (FFT), die sich leicht auf Rechnern implementieren lässt. Eine umfassende Erläuterung der Theorie der digitalen Signalverarbeitung ist hier nicht möglich, da dies den Rahmen eines Praktikumsversuchs sprengen würde. An dieser Stelle sei deshalb auf die zahlreiche Literatur zu diesem Thema verwiesen. Für die Durchführung des Versuchs werden jedoch keine Kenntnisse in digitaler Signalverarbeitung vorausgesetzt, so dass sich das folgende Kapitel auf die wesentlichen Zusammenhänge, die für das Verständnis der Aufgaben notwendig sind, beschränkt.

Ziel dieses Abschnittes ist es, zu zeigen, wie aus diskreten Werten ein kontinuierliches Signal erzeugt werden kann. Dazu wird, aus didaktischen Gründen, zunächst einmal der umgekehrte Weg beschrieben, also vom kontinuierlichen Signal hin zu den diskreten Werten.

2.2.1 Zeitbereich

Ausgangspunkt sei ein kontinuierliches Signal, das zu äquidistanten Zeitpunkten abgetastet wird.

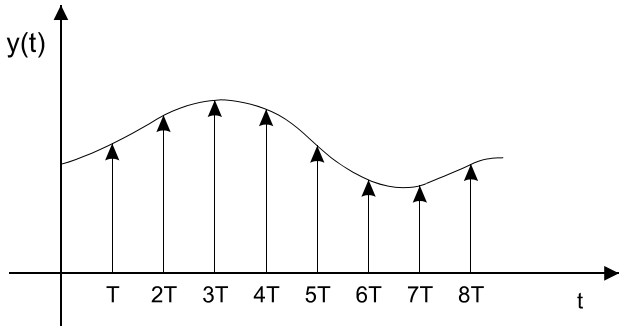


Abbildung 1 Abtastung eines kontinuierlichen Signals zu äquidistanten Zeitpunkten

Anschaulich kann man sich die Funktionswertentnahme so vorstellen: das kontinuierliche Signal wird mit einer Impulsfolge $i_T(t)$ multipliziert.

$$y_g(t) = y(t) \cdot i_T(t) \qquad \text{Gl.1.1}$$

Dabei ist die Impulsfolge $i_T(t)$ eine Folge von Dirac-Impulsen[1], die einen zeitlichen Abstand von T besitzen.

$$i_T(t) = \sum_{n=-\infty}^{+\infty} \delta(t - nT) \qquad \text{Gl. 1.2}$$

oder in der Schreibweise diskreter Signale:

$$i_T(k) = \sum_{n=-\infty}^{+\infty} \delta(k - n) \qquad \text{Gl. 1.3}$$

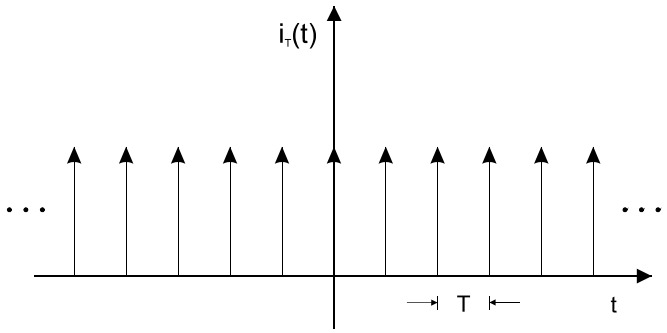


Abbildung 2: Impulsfolge

Hinweis: Für das zeitdiskrete Signal soll folgende Schreibweise verwendet werden:

$$x(t)\,|_{t=nT} = x(nT) := x(n) \qquad \text{Gl. 1.4}$$

[1] Der Dirac-Impuls diskreter Systeme weist bei der Definition nicht die Probleme seines kontinuierlichen Pendants – unendlich große Amplitude, unendlich kurze Zeitdauer, Fläche vom Maß eins – auf. In diskreten Systemen lautet die Definition des Dirac-Impulses

$$\delta(k) = \begin{cases} 1 & k = 0 \\ 0 & k \neq 0 \end{cases}$$

wobei T das Abtastintervall und $f_A = 1/T$ die Abtastfrequenz bezeichnet. Der Wert n ist eine ganze Zahl im Bereich $-\infty < n < +\infty$. Da $x(n)$ für ein bestimmtes n eine feste Zahl darstellt, für laufendes n aber eine Folge von Zahlen, wäre für das diskrete Signal die Bezeichnung $\{x(n)\}$ angemessener. Zur Vereinfachung der Schreibweise wird im Folgenden die Bezeichnung $x(n)$ verwendet. Mit Hilfe der Ausblendeigenschaft der Impulsfolge $\delta(k)$ kann das abgetastete, zeitdiskrete Signal $y_g(k)$ in der Form

$$y_g(k) = \sum_{n=-\infty}^{+\infty} y(n) \cdot \delta(k-n) \qquad \text{Gl. 1.5}$$

geschrieben werden. Die Folge $y_g(k)$ besteht also aus einer unendlichen Folge äquidistanter Deltafunktionen, deren Gewichte den Funktionswerten von $y(n)$ bei $k = n$ entsprechen. Hierbei bezeichnet $y(n)$ einen einzigen Amplitudenwert, $y_g(k)$ dagegen das gesamte Signal.

2.2.2 Frequenzbereich

Transformiert man Gl. 1.5 mit Hilfe der Fouriertransformation in den Frequenzbereich, so erkennt man, dass das Frequenzspektrum des abgetasteten Signals um die Frequenz $f_A = 1/T$ periodisch ist.

$$\Im\{y_g(k)\} = Y_g(k) = \sum_{k=-\infty}^{+\infty} y(k) \cdot e^{-j2\pi kf/f_A} = f_A \cdot \sum_{k=-\infty}^{+\infty} Y(f - kf_A) \qquad \text{Gl. 1.6}$$

Die Wahl der Abtastfrequenz f_A ist für die weitere Verwendung der Signale von entscheidender Bedeutung. Wird das Abtastintervall T größer gewählt, verkleinert sich der Abstand $f_A = 1/T$ der einzelnen Bänder im Frequenzbereich. Das kann dazu führen, dass sich diese Bänder überlappen. Diese Verzerrung der Fouriertransformierten des Abtastsignals wird als Aliasing bezeichnet und entsteht grundsätzlich, wenn eine Zeitfunktion $y(t)$ nicht mit einer ausreichend hohen Frequenz abgetastet wird. Zur Klärung des Sachverhalts diene ein bandbegrenztes Signal $Y(f)$. Die höchste auftretende Frequenz sei $B/2$, wobei B die Bandbreite des Signals sei (siehe Abbildung 3).

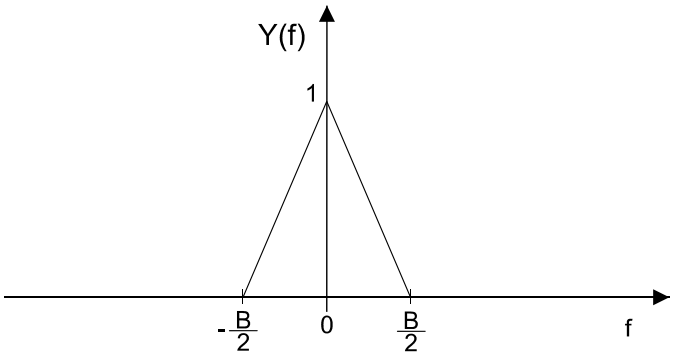


Abbildung 3: Spektrum eines bandbegrenzten kontinuierlichen Signals

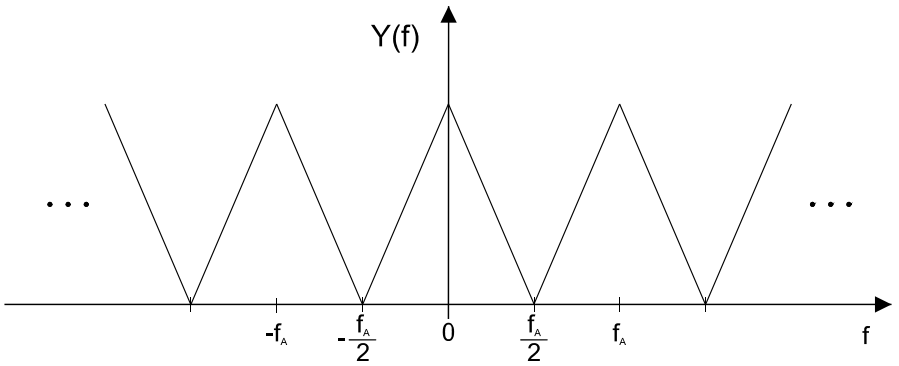


Abbildung 4: Spektrum des abgetasteten Signals. Die Abtastfrequenz ist doppelt so groß wie die höchste Signalfrequenz

Das Spektrum des abgetasteten Signals ist um die Abtastfrequenz f_A periodisch. In Abbildung 4 ist die Abtastfrequenz doppelt so groß wie die höchste Signalfrequenz. Dadurch grenzen die einzelnen Bänder aneinander. Sie überlappen sich nicht.

Abtastung mit $f_A > B$

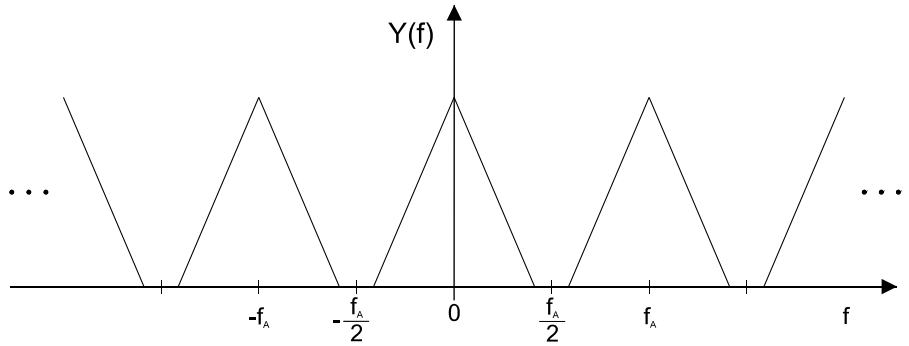


Abbildung 5: Die Abtastfrequenz ist mehr als doppelt so groß wie die höchste Signalfrequenz

Die einzelnen Bänder sind durch Abstände voneinander getrennt. Sie überlappen sich nicht. Ist die Abtastfrequenz kleiner als das Doppelte der Signalfrequenz, so bewegen sich die Frequenzbänder aufeinander zu.

Abtastung mit $f_A < B$

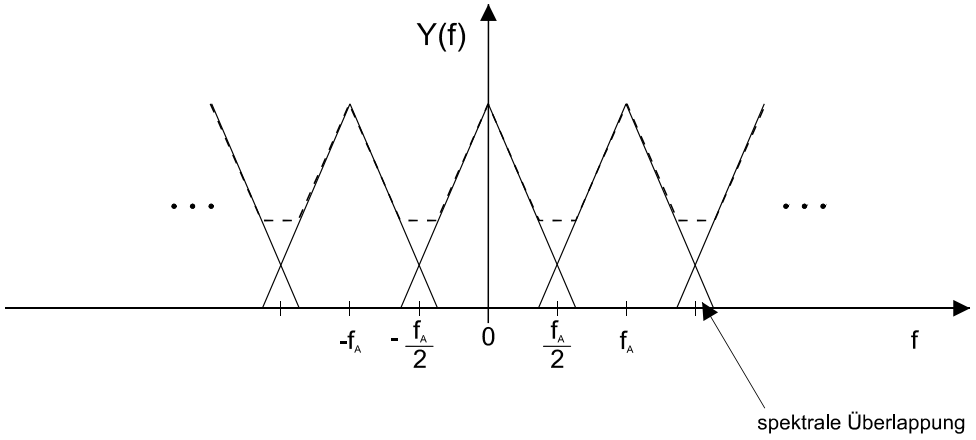


Abbildung 6: Die Abtastfrequenz ist kleiner als das Doppelte der höchsten Signalfrequenz

Die punktierte Linie ergibt sich aus der Superposition der Bänder. Es kommt zu spektralen Überschneidungen der Frequenzbänder (Aliasing), die zu Verfälschungen im interessierenden Spektralbereich führen.

Zur Rekonstruktion wird das Signal $y_g(t)$ durch einen Tiefpass mit der Grenzfrequenz $B/2$ gefiltert. Die Frequenzen oberhalb der Grenzfrequenz werden abgeschnitten. Zurück bleibt das Signal im Bereich $f = -B/2$, ... , $f = +B/2$, das bei Erfüllung der Abtastbedingung $f_A \geq B$ bis auf einen hier nicht interessierenden Amplitudenfaktor dem Originalsignal entspricht. In Abbildung 6 ist durch das Aliasing das kontinuierliche Signal aus dem diskreten Signal jedoch nicht mehr rekonstruierbar.

2.2.3 Das Abtasttheorem

Das Abtasttheorem besagt folgendes:

> Ist $y(t)$ ein mit $B/2$ bandbegrenztes Signal und wird $y(t)$ mit einer Abtastfrequenz $f_A \geq B$ abgetastet, so lässt sich $y(t)$ durch die Abtastwerte eindeutig rekonstruieren. Man sagt: „Das Signal ist auf das **Nyquist-Band** begrenzt".

Wurde das Abtasttheorem bei der Abtastung eines Signals erfüllt, so kann aus den diskreten Werten das abgetastete kontinuierliche Signal vollständig wiederhergestellt werden.

Die Frage, die sich nun stellt, ist: Wie kann die Abtastfrequenz gewählt werden, wenn das Spektrum des Signals nicht bekannt ist?

Da Signale praktisch nie wirklich bandbegrenzt sind, ist die Erhöhung der Abtastfrequenz bis an die Grenze des technisch Machbaren sicherlich keine Lösung. Es gibt zwei Möglichkeiten, mit dem Aliasing-Effekt umzugehen:

1. Man nimmt den Aliasing-Fehler als gegeben hin und berücksichtigt den dabei entstehenden Fehler in der weiteren Verarbeitung der Werte.
2. Man begrenzt das Signal mit Hilfe eines Tiefpasses auf das Nyquist-Band, um spektrale Überschneidungen zu vermeiden. Dieser Tiefpass wird auch als Anti-Aliasing-Filter bezeichnet. Das begrenzte Signal kann abgetastet und vollständig mit Hilfe eines

Rekonstruktionsfilters reproduziert werden. Der Fehler entsteht durch die Begrenzung des Signals auf das Nyquist-Band.

Wird das Abtasttheorem beachtet, so stellt die Funktionsreihe aus Gl. 1.5 das Signal dar. Nun läuft aber der Summationsindex in Gl. 1.5 über den gesamten Zeitstrang von $-\infty$ bis $+\infty$, denn über die Dauer des Zeitsignals ist keine Aussage gemacht. Das bringt für die Verarbeitung folgende Schwierigkeiten mit sich:

- Es können nur endlich lange Signale beobachtet werden. Eine Beobachtungszeit T_0 wird festgelegt.
- Die Wertemenge ist durch die Darstellung einer endlichen Anzahl von Bits beschränkt. Die Summation der diskreten Fouriertransformation muss auf endlich viele Summanden begrenzt werden.

Durch die Einführung einer Beobachtungszeit T_0 wird ein Zeitfenster über das Signal gelegt, innerhalb dessen Funktionswerte zu äquidistanten Zeitabständen T entnommen werden. Am Ende einer Beobachtungszeit liegt eine Stichprobe von $N = T_0 / T$ Elementen vor.

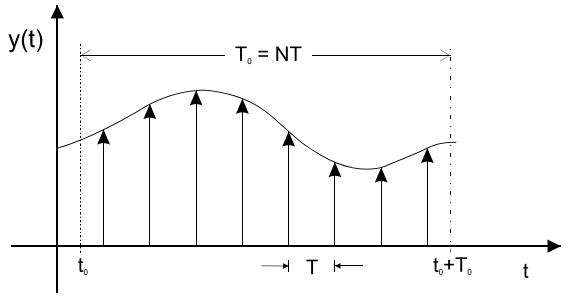


Abbildung 7: Größen der Signalerfassung im Zeitbereich

Wird der Kehrwert der Beobachtungszeit T_0 als Beobachtungsfrequenz F_0 eingeführt, so lauten die Zusammenhänge im Einzelnen:

N	Umfang der Stichprobe
$T_0 = N \cdot T$	Beobachtungszeit
$F_0 = 1 / T_0 = 1 /(N \cdot T) = (1 / N) \cdot F$	Beobachtungsfrequenz
$T = 1 / N \cdot T_0$	Abtastzeit
$F = N \cdot F_0 = 1 / T$	Abtastfrequenz

Wird Abbildung 7 in den Frequenzbereich transformiert, so liegen folgende Verhältnisse vor:

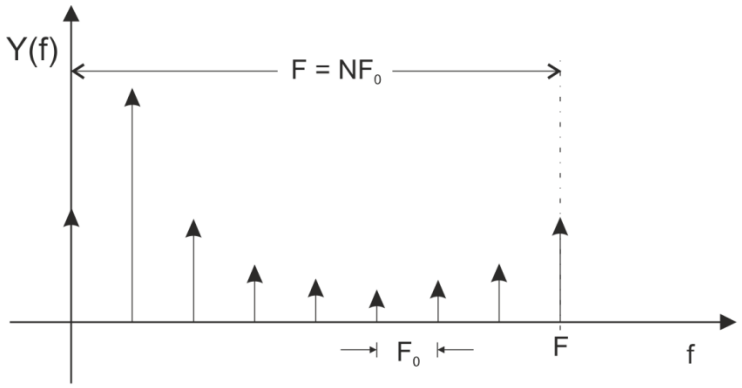


Abbildung 8: Größen der Signalerfassung im Frequenzbereich

Das zunächst analoge, kontinuierliche Signal wurde über eine Impulsfolge diskretisiert. Das diskretisierte Signal besitzt ein periodisches Spektrum. Wird mit einer zu niedrigen Frequenz abgetastet, kann dies zu einem Aliasing-Effekt führen, der eine korrekte Rekonstruktion des ursprünglichen Signals unmöglich macht. Somit ist auf die Wahl der Abtastfrequenz größte Sorgfalt zu legen.

Im Versuch selbst werden zunächst allerdings keine Signale abgetastet, sondern es sollen mit Hilfe von digitalen Werten analoge kontinuierliche Signale erzeugt werden, wobei die Rekonstruktion mittels eines D/A-Wandlers und eines Filters vorgenommen wird.

Ein D/A-Wandler interpoliert aus diskreten Werten ein kontinuierliches Signal, das aber aufgrund seiner Periodizität Frequenzanteile enthält, die in dem ursprünglichen Signal nicht enthalten waren. Sie zu beseitigen, ist die Aufgabe eines nachgeschalteten Tiefpasses, dessen Grenzfrequenz bei der halben Abtastfrequenz liegt.

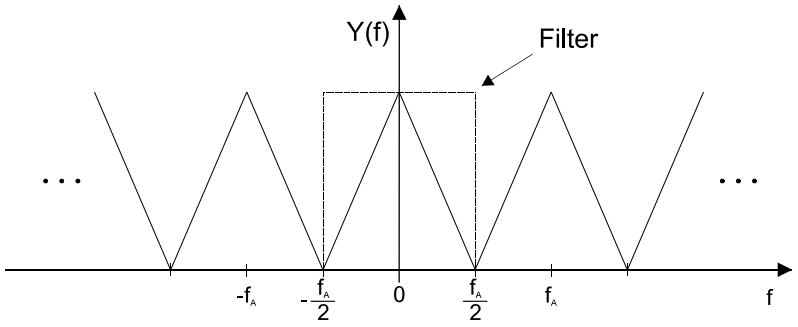


Abbildung 9: Ausgangssignal des D/A-Wandlers

Soweit die Theorie. Die reale Rekonstruktion ist Gegenstand des nächsten Abschnittes.

2.3 ARM-Assembler-Sprache

Das Schreiben von Code in der ARM Assembler Sprache: Eine Assemblersprache (oft abgekürzt als ASM bzw. asm) ist eine spezielle Programmiersprache, welche die Maschinensprache einer spezifischen Prozessorarchitektur in einer für den Menschen lesbareren Form repräsentiert.

Im Folgenden ist ein Programmausschnitt für die Takt-Konfigurierung dargestellt. Die Programmierung der jeweiligen Register wird im Datenblatt des Mikrocontrollers ADuC7026 (page 53 of 92) erläutert.

```
        GET  ADUC7026.s                ; include file ADUC7026.s

        AREA beispiel_1, CODE, READONLY      ; declaration of code segment
        ENTRY                          ; entry point of code segment
start
        LDR   r1,=POWKEY1              ;Load the address of MMR POWKEY1 into r1
        MOV   r2,#0x01                 ;Move the value 0x01 into r2
        STR   r2,[r1]                  ;Store the value 0x01 to MMR POWKEY1

        LDR   r1,=POWCON               ;Load the address of MMR POWCON into r1
        MOV   r2,#0x04                 ;Move the value into r2
        STR   r2,[r1]                  ;Store the value 0x01 to MMR POWCON

        LDR   r1,=POWKEY2              ;Load the address of MMR POWKEY2 into r1
        MOV   r2,#0xF4                 ;Move the 0xF4 into r2
        STR   r2,[r1]                  ;Store the value 0xF4 to MMR POWKEY2

        END
```

Der `AREA`-Befehl definiert entweder einen Datenblock, oder wie in diesem Fall, einen Programmblock, wobei `beispiel_1` den Blocknamen definiert. Der Befehl `Code` bedeutet, dass der Abschnitt Maschinenbefehle als `READONLY` enthält. `ENTRY` definiert einen Eintrittspunkt für das Programm. Ein Label ist ein Symbol, das eine Adresse irgendwo im Speicher repräsentiert, in unserem Beispiel `start`. Die Adresse wird während der Assemblierung vom Assembler erstellt. `LDR` steht für „load instruction“. Der Befehl nimmt einen einzelnen Wert vom Speicher und schreibt ihn in ein *General-Purpose*[2] *Register*. Mit dem Befehl `MOV` können bestimmte Bits im zugehörigen Register gesetzt werden. `STR` - „store instruction“ liest einen Wert vom general-purpose Register und legt ihn im Speicher ab. Das Ende des Datenblocks wird durch den Befehl `END` deklariert.

Table 34. POWCON MMR Bit Designations

Bit	Value	Name	Description
7			Reserved.
6:4		PC	Operating Modes.
	000		Active Mode.
	001		Pause Mode.
	010		Nap.
	011		Sleep Mode. IRQ0 to IRQ3 and Timer2 can wake up the part.
	100		Stop Mode. IRQ0 to IRQ3 can wake up the part.
	Others		Reserved.
3			Reserved.
2:0		CD	CPU Clock Divider Bits.
	000		41.78 MHz.
	001		20.89 MHz.
	010		10.44 MHz.
	011		5.22 MHz.
	100		2.61 MHz.
	101		1.31 MHz.
	110		653 kHz.
	111		326 kHz.

[2] Siehe Datenblatt des ADuC 7026 page 32 of 92.

Das POWCON Register kann nur in einer Schreibsequenz verändert werden. Hierzu wird POWKEY1 auf den Wert 0x01 und POWKEY2 auf den Wert 0xF4 gesetzt. Im oben beschriebenen Beispiel ist nur das 2. Bit von POWCON gesetzt, somit arbeitet die CPU bei einer Taktfrequenz von 2,61 MHZ.

2.3.1 Speicherzuweisung

`DCB` weist ein oder mehr Bytes dem Speicher zu und bestimmt die Anfangslaufzeit des Speichers. Der Befehl `DCW` stellt einem oder mehreren Halbwörtern einen 2-byte großen Speicherplatz bereit, wohingegen `DCD` einem oder mehreren Wörtern einen 4-byte großen Speicherplatz zuweist.

```
DCD     0x0800
DCD     0x0CB3
```

2.3.2 Logik und Arithmetik

Zu Beginn eine Einführung der Flags, grundlegende arithmetische Befehle und ein Überblick über gebrochene Arithmetik.

2.3.2.1 Flags und ihre Benutzung

Das *CPSR* (*Current Program Status Register*) enthält den aktuellen Status der Maschine, der Flags und dem Modus mit denen Bits festgelegt werden wie in folgender Abbildung:

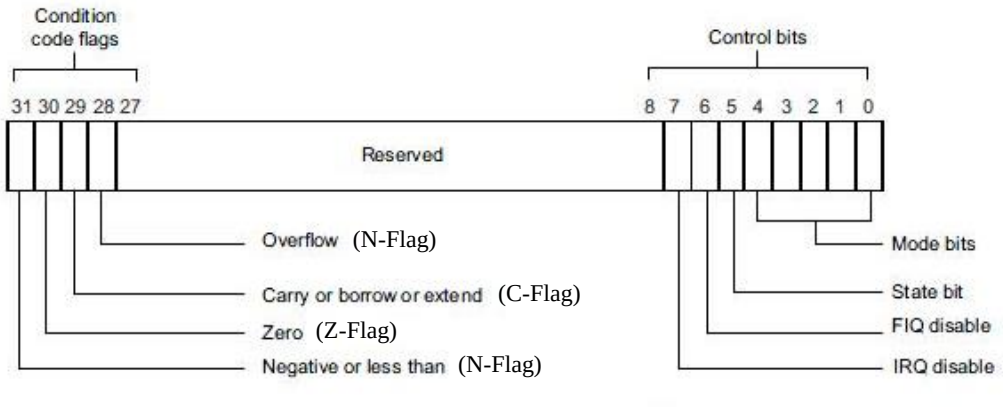


Abbildung 10: Current Program Status Register

Die 4 höherwertigen Bits helfen bei der Bestimmung, ob ein Befehl bedingt ausgeführt wird oder nicht. Die Flags sind gesetzt oder Null basierend auf einer von zwei Möglichkeiten: entweder wird ein Befehl der Flags setzt und löscht, benutzt oder Befehle, die Flags setzen aufgrund der Ergebnisse des Befehls und anhängen eines „S“ an das Mnemonic (Gedächnis). Zum Beispiel würde EORS eine exklusive Oder-Operation durchführen und die Flags im Nachhinein setzen, da das S Bit im Befehl gesetzt ist. Dies kann mit allen ALU Befehlen durchgeführt werden, so dass die Aktualisierung der Flags kontrolliert werden kann.

Das N-Flag

Dieses Flag kontrolliert, ob das Ergebnis negativ ist. Die Definition einer negativen binären Zahl wird im Kontext der 2er Komplement Zahlen erklärt. Falls das höchstwertigste Bit (MSB) einer 2er Komplement Zahl gesetzt ist, ist das Ergebnis negativ.

Das Z-Flag

Das Z-Flag zeigt an, ob das Ergebnis einer Operation nur Nullen erzeugt, also alle 32 Bits sind Null.

Das C-Flag

Das Carry-Flag oder Überlauf Bit ist gesetzt, wenn das Ergebnis einer Addition größer gleich 2^{32} ist, das Ergebnis einer Subtraktion positiv oder als das Ergebnis einer Inline-Barrel-Shifter Operation eines MOVE oder logischen Befehls. Das Carry ist ein nützliches Flag, welches Operationen mit größerer Präzision auszuführen erlaubt, z.B. um ein Programm zu schreiben, das 64-bit Zahlen addiert.

Das V-Flag

Das V-Flag, oder auch Overflow-Flag genannt, wird bei einem Überlauf gesetzt, also wenn das Ergebnis einer Addition, Subtraktion oder eines Vergleichs größer gleich 2^{31} oder weniger als -2^{31} beträgt. Mit dem V-Flag wird somit – im Gegensatz zum C-Flag – ein vorzeichenbehafteter Überlauf gekennzeichnet.

2.3.2.2 Vergleich Befehle

`CMP` – *compare*. `CMP` subtrahiert ein Register oder einen direkten Wert von einem Registerwert und aktualisiert den Bedingungscode. Falls `Rn` > `operand2`, wird das C-Flag des `CPSR` Registers gesetzt; falls `Rn` = `operand2`, werden beide das C und Z-Flag gesetzt; falls `Rn` < `operand2`, wird das N-Flag gesetzt. `CMP` wird benutzt um schnell den Inhalt eines Registers auf einen bestimmten Wert zu prüfen, z.B. am Anfang oder am Ende einer Schleife.

`CMN` – *compare negative*. `CMN` addiert ein Register oder einen direkten Wert zu einem anderen Register und aktualisiert den Bedingungscode. `CMN` kann auch schnell den Inhalt von Registern überprüfen. Dieser Befehl ist die Inverse zu `CMP`.

`TST` – *test*. `TST` logisch UND-verknüpft den arithmetischen Wert mit einem Registerwert und aktualisiert den Bedingungscode ohne das V-Flag zu beeinflussen. `TST` kann benutzt werden für die Feststellung, ob viele Bits eines Register Null sind oder ob mindestens ein Bit eines Registers gesetzt ist.

`TEQ` - *test equivalence*. Der Befehl Oder-verknüpft den arithmetischen Wert mit dem Register Wert und aktualisiert die Schleife. Er wird benutzt um die Gleichheit zweier Werte, festzustellen.

Ein Beispiel für einen typischen Befehl eines Vergleichs ist unten dargestellt:

```
CMP r8, #0          ;r8==0?
TEQ r9, r4, LSL #3
```

Hierbei wird vergleichen, ob das Register `r8` gleich 0 ist. Wenn `r9` und `r4` gleich sind, wird `r4` mit $2^3 = 8$ multipliziert.

2.3.3 Schleifen und Sprünge

Ein Sprungbefehl ändert den Fluss der Ausführung oder wird zum Aufrufen einer Routine benutzt.

`B` – Branch, einfachster Sprungbefehl. Bedingungen können genutzt werden, um zu entscheiden, ob zu einer neuen Adresse im Code gesprungen wird.

`BX` – Branch and exchange. Mit diesem Befehl kann von einer 32-bit ARM Instruction zu einem 16-bit THUMB Befehl gewechselt werden.

`BL` – Branch and link. Das Linkregister wird benutzt um eine Rückkehradresse nach der Branch Instruction einzuhalten. Falls wir eine Subroutine ausführen und zurückkehren möchten, muss der Prozessor bloß den Wert des Linkregisters in das Programmfenster am Ende der Subroutine schreiben.

Fast jeder eingebettete Code bietet die Konstruktion von Schleifen an, besonders falls ein Betriebssystem läuft oder falls die Anwendung den Prozessor fordert die Eingabe oder die Peripherie periodisch zu kontrollieren.

2.3.3.1 While-Schleifen

```
        B test          ; jump to condition evaluation
loop
        ADD r0,r0,#1    ; instruction example
test    TEQ r0,r1       ; if r0=r1, set Z flag, else clear Z flag
        BNE loop        ; branch back to loop begin if Z flag is "0",
                        ; else finish WHILE loop
```

Der obige Code zeigt die Basis Struktur der While-Schleife – realisiert mit dem Branch-Sprungbefehl. Die Register `r0` und `r1` beinhalten vorher definierte Werte. Zunächst springt die gezeigte Codesequenz zum Label `test`, wo die Schleifen-Bedingung, hier: ein Vergleich der beiden Registerwerte, ausgewertet wird. Der Wert des Registers `r0`, wird solange erhöht, bis die beiden Register den gleichen Wert besitzen. Falls `r0=r1` wird das Z-Flag des `CPSR` Registers gesetzt, ansonsten gelöscht. Der `BNE`-Befehl beeinflusst das Z-Flag, wenn es gelöscht ist, springt das Programm zurück zum `loop` Label. Ansonsten wird die Schleife beendet.

2.3.3.2 For Schleifen

Eine weitere übliche Schleife ist die for-Schleife, eine Variation der While-Schleife.

```
        MOV   r1,#0             ; r1=0
loop    CMP   r1,#10            ; r1<10?
        BGE   done              ; wenn r1>=10, Ende, springe zu done
                                ; sonst, kein Sprung, nächster Befehl
        ADD   r1,r1,#1          ; j++
        B     loop              ; branch back to loop
done
```

Die erst Zeile des Programmcodes (r1=0) löscht die Variable bevor die Schleife beginnt. Die zweite Zeile (r1<10) bewertet bei jedem Durchlauf, ob die Schleife verlassen wird oder nicht. Der Index wird am Ende der Schleife inkrementiert, bevor wieder zum Anfang der Schleife gesprungen wird. Nach dem letzten Scheifendurchlauf springt der Programm-Counter wieder zum Scheifenanfang. Die CMP-Vergleich setzt das V-Flag, weil die Bedingung r1<10 nicht mehr erfüllt ist. Daraufhin wird der Programm-Counter in der nächsten Zeile veranlasst zum done-Label zu springen und somit die Schleife zu veralssen.

Eine bessere Möglichkeit eine for-Schleife zu realisieren ist unten dargestellt:

```
        MOV   r1,#10            ; r1=10
loop
        SUBS r1,r1,#1           ; r1= r1-1, and set Z flag if r1=0.
        BNE   loop              ; jump to loop, if Z flag is not set.
```

Hierbei wird im Gegensatz zum vorigen Beispiel das Zählregister dekrementiert. Die for-Schleife springt am Ende der Schleife zum Label `loop` zurück, solange der Zählwert im Register r1 nicht gleich eins ist. Dieses Beispiel ist effizienter als das Vorige, weil es nach dem letzten Scheifendurchlauf keinen Rücksprung zum Scheifenanfang gibt.

2.3.3.3 Do…While-Schleifen

```
loop    ..              ; loop body
        ..              ; evaluate condition
        BNE     loop
        exit
```

Hier wird die Schleifenkonstruktion ausgeführt bevor die Bedingung überprüft wird. Die Struktur ist die Gleiche wie die der While-Schleife ohne den Sprung am Anfang.

2.3.4 Tabellen

Eine weitere übliche Aufgabe für Mikroprozessoren ist die Suche nach Daten im Speicher von einer Liste von Elementen. Elemente könnten abgetastete Werte sein, die von Sensoren oder Analog-Digital-(A/D)Wandlern übertragen wurden, oder sogar Daten in einem Puffer, die in ein externes Gerät übertragen werden. Umsetzungstabellen sind manchmal ein effizienter Ersatz für komplizierte Routinen, wenn Logarithmus oder Tangens-Funktionen gebraucht werden. Der Nachteil ist, dass man Speicherplatz und Präzision gegen Schnelligkeit eintauschen muss.

Zunächst werfen wir einen kurzen Blick auf einige elementare Verwendungszwecke von Tabellen und Listen.

2.3.4.1 Umsetzungstabellen (Lookup Tables)

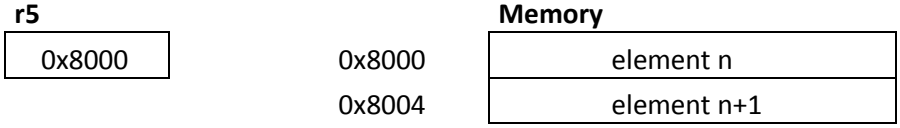
Berücksichtigt man eine Liste von Elementen, die im Speicher ab einer bestimmten Adresse angeordnet sind, wird die Addressierung eines bestimmten Elements in der Liste ziemlich einfach, da der ARM Adressierungsmodus eine Pre-Index-Adresse mit einem Offset erlaubt. Genauer, wenn die Startadresse im Register `r5` wäre, könnte ein bestimmtes Element entweder einer Adresse durch Hinzufügen eines Offsets in einem anderen Register oder die Elementnummer kann genutzt werden um einen Offset durch Skalierung zu erhalten. Das dritte Element in der Liste könnte abgerufen werden durch

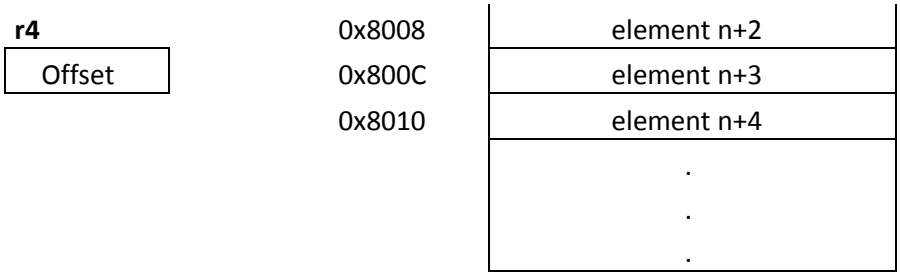
```
        LDR     r6,[r5,r4]
```

oder

```
        LDR     r6,[r5,r4, LSL #2]
```

hier würde `r4` den Wert 8 enthalten, der aktuelle Offsetwert entspricht im ersten Fall der Elementnummer eins weniger wie im 2. Fall (für unsere Diskussion, ist das erste Element die Nummer 0). Der letzte Addressierungsmodus prüft die Menge an Daten durch Skalierung der Elementnummer mit 4.





2.3.4.2 Jump Tables

Jump Tabellen sind ein anderer Tabellentyp. Sie enthalten Adressen, anstatt Daten zu beinhalten. Wenn der Prozessor einen Input erhält und in Folge dessen ein bestimmtes Programm ausführt, sind Jump Tabellen nützlich um eine Serie von Vergleichen und Sprüngen zu ersetzen. Z.B. könnte ein Controller ein Input von einem Keypad erhalten und auf Grundlage der gedrückten Nummer springt er zu einer bestimmten Subroutine um die Register und Displaydaten zu konfigurieren oder Variablen zu initialisieren.

2.3.5 Data Processing Operations

Aus fundamentalen Operationen wie Addition, Subtraktion und Shift können kompliziertere Operationen wie Division, Multiplikation und Wurzel entwickelt werden.

Der ARM unterstützt boolsche Algebra wie z.B. `AND`, `ORR`, `EOR` (exklusives ODER), `MOVN` (Move Negative), `BIC` (Bit Clear, für ausgewählte Bits). Ein Beispiel ist unten dargestellt:

```
AND  r1, r2, r3                    ; r1 = r2 AND r3
ORR  r1, r2, r3                    ; r1 = r2 OR r3
EOR  r1, r2, r3                    ; r1 = r2 exclusive-OR r3
BIC  r1, r2, r3                    ; r1 = r2 AND NOT r3
```

Die ersten drei Operationen sind selbsterklärend, das sind grundlegende logische Funktionen. Die vierte Operation ist die Bit-Clear Operation, welche genutzt wird um ausgewählte Bits zu löschen. Jede 1 des zweiten Operands löscht das zugehörige Bit des ersten Operands (ein Register) und eine 0 lässt das Bit unverändert. Bezüglich des Befehlsformats für Daten, kann ein direkter Wert für den 2. Operand geschrieben werden.

```
BIC  r2, r3, #xFF000000
```

In diesem Beispiel wird das obere Byte des Register `r3` gelöscht und das Ergebnis in das Register `r2` geschrieben.

2.3.5.1 Shift

Der ARM-Prozessor ist in Lage fünf verschiedene Shift-/Rotierungs-Typen auszuführen. Obwohl nicht alle erläutert werden, sind in Abbildung 11 der Vollständigkeit wegen, alle Typen aufgezeigt.

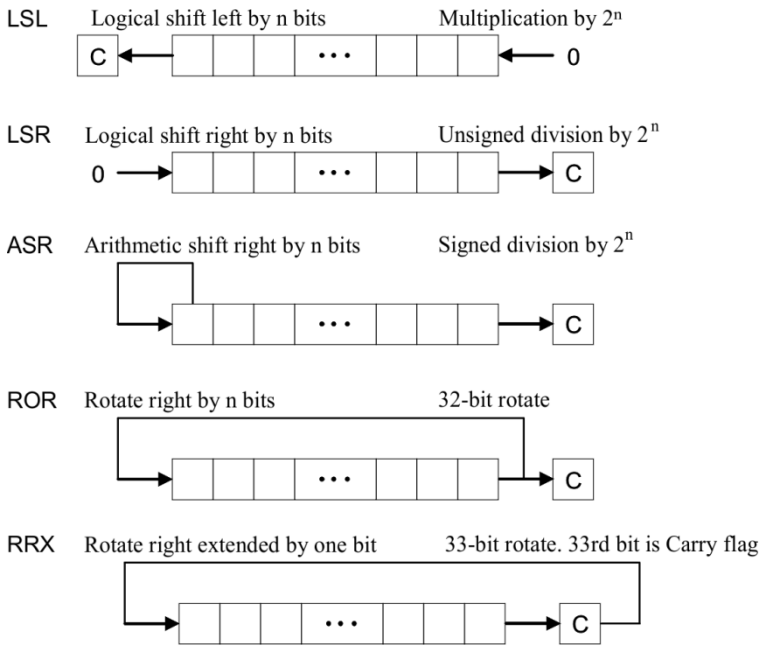


Abbildung 11: Shifts and Rotates

Beim ASL (arithmetic shift left) Befehl muss das Vorzeichen Bit bewahrt werden. Man muss also sehr auf den Überlauf aufpassen, sonst wird aus einer negativen eine positive Zahl.

Wenn Daten nur geschrieben werden sollen, ohne eine andere Operation auszuführen wird der MOV Befehl benutzt. Mit MOV können Daten von einem Register in ein anderes übertragen werden, auch wenn ein zusätzlicher Shift Befehl ausgeführt werden soll. Einfache Shift Befehle können wie folgt geschrieben werden.

```
MOV   r4, r6, LSL #4                ; r4 = r6 << 4 bits
MOV   r4, r6, LSL r3                ; r4 = r6 << # specified in r3
```

Alle Shift Operationen benötigen einen Zyklus für die Ausführung, außer der Register-spezifische Shift (zweite Code-Zeile des obigen Beispiels), diese benötigen einen extra Zyklus.

Die Shifts und logischen Operationen können auch dazu benutzt werden um Daten von einem Byte zum anderen zu schieben. Angenommen man möchte das höchste Byte vom Register `r2` zum niedrigsten des Registers `r3` schieben. Der Inhalt von Register `r3` muss als erstes um 8 Bits nach links geschoben werden. Als Beispiel folgender Programmcode:

```
MOV   r0, r2, LSR #24               ; extract top byte from r2 into r0
ORR   r3, r0, r3, LSL #8            ; shift up r3 and insert r0
```

2.3.5.2 Addition und Subtraktion

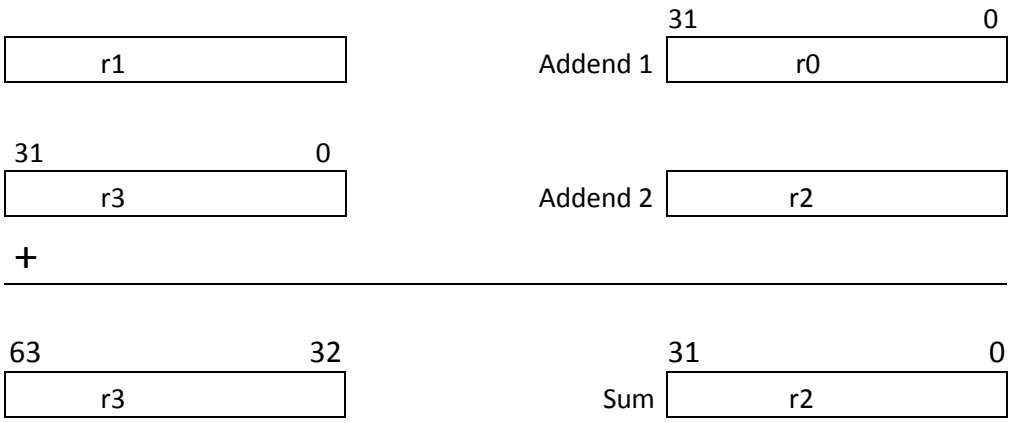
Auch können Addition, Subtraktion mit oder ohne Carry ausgeführt werden:

```
ADD   r1, r2, r3                ; r1 = r2 + r3
ADC   r1, r2, r3                ; r1 = r2 + r3
SUB   r1, r2, r3                ; r1 = r2 - r3
SBC   r1, r2, r3                ; r1 = r2 - r3 + C - 1
RSB   r1, r2, r3                ; r1 = r3 - r2
RBC   r1, r2, r3                ; r1 = r3 - r2 + C - 1
```

Das Carry-Flag könnte in diesem Fall zeigen, ob eine Operation ein Carry-Bit im MSB des Ergebnisses erzeugte. Die `ADC`, `SBC`, `RSC` Befehle benutzen dieses Flag, indem sie das Carry-Flag in die Operation integrieren.

Der folgende Befehl addiert eine 64-bit Zahl, die in Register `r2` und `r3` steht, zu einer anderen 64-bit Zahl in den Registern `r0` und `r1`. Das Ergebnis wird in die Register `r4` und `r5` geschrieben:

```
ADDS r4, r0, r2          ; adding the least significant words
ADC  r5, r1, r3          ; adding the most significant words
```



Die 64-bit Subtraktion kann durch einen ähnlichen Code durchgeführt werden. Im Folgenden werden die unteren Hälften des 64-bit Werts subtrahiert und das Carry-Flag aktualisiert und danach die oberen Hälften einschließlich des Carry-Flag.

```
SUB 64  SUBS r0, r0, r2          ; substract lower halves, set Carry flag
SBC r1, r1, r3                   ; subtract upper halves, and set Carry
```

2.3.5.3 Multiplikation mit einer Konstanten

Beispiele der Multiplikation mit einer Konstanten sind nachfolgend dargestellt:

```
MOV  r1, r0, LSL #2          ; r1 = r0*4
```

Multiplikation mit einer Zahl, die kein Vielfaches von 2 ist.

```
ADD  r0, r1, r1, LSL #2          ; r0 = r1 + r1*4
RSB  r0, r2, r2, LSL #3          ; r0 = r2*7
```

Durch die Shiftoperation zur Multiplikation wird nur der 32-bit Addierer und ein Barrel-Shifter[3] benötigt. Der ARM Prozessor kann mithilfe von Schiebeoperationen Multiplikationen von 2^n, $2^n - 1$, $2^n + 1$ in einem einzigen Zyklus generieren ohne ein Multiplizierer benutzen zu müssen, was Ausführungszeit spart.

```
ADD  r0, r1, r1, LSL #1          ; r0 = r1*3
SUB  r0, r0, r1, LSL #4          ; r0 = (r1*3) - (r1*16) = (-13)*r1
ADD  r0, r1, r1, LSL #7          ; r0 = (-13)*r1 + r1*128 = r1*115
```

2.3.5.4 Division

Die meisten ARM Mikroprozessoren besitzen keine Hardware für einen Dividierer, weil die Division so selten benutzt wird (und deshalb ein Softwareprogramm dafür geschrieben wird). Ein Dividierer verbraucht zu viel Speicherplatz und/oder Energie um ihn als eingebetteten Prozessor zu erwägen und es gibt immer eine Möglichkeit um die Division gänzlich zu vermeiden. Bei der Wahl eines geeigneten Programmcodes für einen Dividierer muss der zu verarbeitende Datentyp (Fraktalzahlen oder ganze Zahlen) ins Auge gefasst werden.

[3] Führt die Schiebeoperationen LSL, LSR, ASR, ROR und RRX durch.

2.3.6 Makros

Makrodefinitionen erlauben dem Programmierer Definitionen für Funktionen oder Operationen einmalig zu deklarieren, einen Namen hinzu zufügen, sodass Schreibarbeit gespart wird. Es gibt Vor- und Nachteile Makros zu benutzen. Im Allgemeinen kann der Quellcode dadurch verkürzt werden. Doch wenn die Makros ausgebaut werden, verschlingen sie den Speicherplatz durch eine ständige Ausführung, Makros können sehr groß werden. Durch die Benutzung von Makros kann der Code schneller geändert werden, da oft nur ein Block geändert werden muss. Man kann auch eine neue Operation im Code als Makro definieren und sie aufrufen, sobald sie gebraucht wird. Aber die neue Operation sollte ausreichend kommentiert sein, falls Jemand unvertrautes den Code liest.

```
        MACRO
$Label  mmr $Register, $Value
        LDR   r5,=$Register
        LDR   r6,=$Value
        STR   r6,[r5]
        MEND

        ⋮

SET1    mmr GP3CON,0x00000001          ;configure Pin 3.0 in PWM0H mode
SET2    mmr PWMCON,0x01                ;enable PWM
```

Obige Beispiel äquivalent zu:

```
        LDR   r5, =GP3CON
        MOV   r6, #0x00000001
        STR   r6,[r5]

        LDR   r5, =PWMCON
        MOV   r6, #0x01
        STR   r6, [r5]
```

Die Makrofunktion dient dazu, die Werte der Register schneller/einfacher setzen zu können.

2.3.7 Exception Handling

Große Anwendungen müssen oft Inputs von verschiedenen Quellen, z.B. Keyboards, Mäuse, USB Anschlüsse bewältigen, oder auch Teile des Energiemanagement übernehmen, falls die Batterie fast leer ist. Manchmal hat ein integrierter Mikrocontroller nur einen oder zwei externe Anschlüsse (z.B. von einem Sensor zum Motor), aber es könnte immer noch dezentrale Bauteile enthalten, die von Zeit zu Zeit Aufmerksamkeit benötigen, wie z.B. die Laufzeigüberwachung. Universale asynchrone Receiver/Transmitter (UARTs), Warnalarm, Analog-Digital (A/D) Wandler und PC Geräte können alle die Zeit des Prozessors beanspruchen. In diesem Kapitel geht es um die Interrupts.

2.3.7.1 Interrupts

Interrupts sind sehr verbreitet in Mikroprozessoren. Dadurch kann ein Gerät, wie z.B. ein Timer oder ein USB Anschluss direkt die Aufmerksamkeit des Prozessors erhalten. Miktrocontroller sind in der Tat kleinere Versionen von vollständigen Systemen, wovon Motorcontroller, Timer, Real-Time-Clocks und serielle Anschlüsse den Prozessor für eine relativ kurze Dauer beanspruchen müssen. Was ist die beste Möglichkeit, dass der Prozessor seine Aufgaben erledigt, während auch andere Schnittstellen bedient?

Angenommen man hat einen UART[4] der an den Prozessor angeschlossen ist und gleichzeitig Input von einem anderen Gerät erhält. Wenn bei einem ankommenden Datenstrom ein Eingangswert im UART eintrifft, sitzt er im Grunde genommen an einem Speicherplatz der dem UART zugeordnet ist und wartet, dass der Prozessor diesen Wert annimmt. Es gibt 3 Möglichkeiten wie der Prozessor diese Aufgabe verarbeiten kann. Die erste und ineffizienteste Lösung wäre, wenn der Prozessor nichts tut, außer in einer Schleife auf einen Input vom UART zu warten. Bedenkt man die Schnelligkeit mit der Prozessoren arbeiten – heutzutage können Milliarden von Befehlen in einer Sekunde abgearbeitet werden – bedeutet das Warten von einem Bruchteil einer Sekunde auf ein Gerät, das Daten übertragen möchte, die Verschwendung von Millionen Zyklen der Bandbreite. Als zweite Möglichkeit kontrolliert der Prozessor nur gelegentlich die Speicherstelle auf neue Daten, genannt „polling". Hier kann der Prozessor noch andere Arbeiten durchführen, während er wartet, doch benötigt er immer noch eine Zeitunterbrechung um den (vielleicht leeren) Speicherplatz zu überprüfen. Die dritte Möglichkeit und beste Möglichkeit besteht darin, dass das Gerät dem Prozessor mitteilt, wenn neue Daten abrufbar sind. So, kann der Prozessor seine anderen Funktionen erfüllen, z.B. den Display aktualisieren oder MP3 Daten in einen analogen Schwingungsverlauf umzuwandeln, solange es auf ein langsameres Gerät, das seine Aufgabe abschließt, wartet. Durch den Interrupt kann dem Prozessor durch eine sehr effiziente Methode mitgeteilt werden, dass etwas anderes (meist ein Gerät oder eine Schnittstelle) Beachtung benötigt.

Ein Interrupt mit niedriger Priorität heißt IRQ, einer mit hoher Priorität FIQ. Zusätzlich zu Hardware Unterbrechungen besitzt auch Software Interrupts, genannt SWI. FIQ's haben auf zwei Arten eine höhere Priorität als IRQ's: sie werden als Erstes bedient wenn mehrere Interrupts auftauchen und wenn ein FIQ bedient wird, ist der IRQ gesperrt. Sobald der FIQ Betrieb beendet ist, kann der IRQ bearbeitet werden. FIQ Interrupts besitzen den letzten Eintrag in der Vektrotabelle, wodurch eine schnelle Methode bereitgestellt wird um in den Steuerungsbetrieb einzutreten, sowie 5 extra Register die für den Ausnahmefall benutzt werden.

2.4 Frequenzsynthese mit dem Mikrocontroller

2.4.1 Ausgabe der Werte über den D/A-Wandler

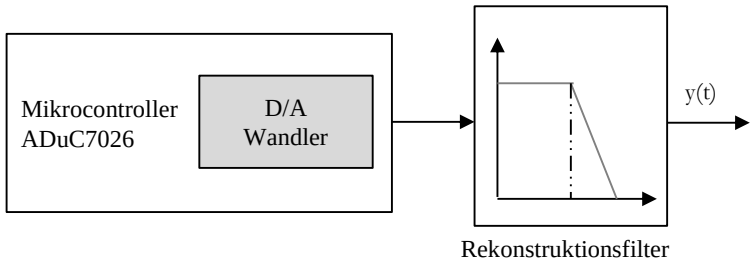


Abbildung 12 Übergang von diskreten Werten zum kontinuierlichen Signal

[4] Die Kurzform für *Universal Asynchronous Receiver Transmitter* bezeichnet eine serielle Schnittstelle zum Senden und Empfangen von Daten.

Obige Abbildung veranschaulicht das Prinzip der Umwandlung diskreter Werte in ein kontinuierliches Signal. Die diskreten Werte werden über dem D/A-Wandler des Mikrocontrollers herausgegeben. Das Ausgangssignal des D/A-Wandlers wird durch einen Tiefpass gefiltert, damit sich das rekonstruierte Signal ergibt.

Es gibt prinzipiell drei Möglichkeiten, periodische Signale auf einem Mikrocontroller auszugeben. Bei allen drei Verfahren ist die Anzahl der möglichen Werte begrenzt. Auch bei der Berechnung der Werte durch den Mikrocontroller ist die mögliche Anzahl der Werte durch den Algorithmus vorgegeben, so dass sich die Werte zyklisch wiederholen. Die dadurch generierten kontinuierlichen Signale sind periodisch. Dies bedingt einen – wie auch immer gearteten – Sprung des Programms vom letzten zum ersten Wert, bei dem es zu keiner zeitlichen Verzögerung der Ausgabe kommen darf. Daraus resultiert, dass die Zeitspanne zwischen zwei Ausgaben immer gleich lang sein muss, unabhängig von der Lage der Werte im Speicher. Um eine Aussage über die Zeitspanne machen zu können, werden in den folgenden Beispielen dieses Abschnitts die Zyklen angegeben und summiert, die zur Ausgabe eines einzelnen Bytes notwendig sind.

2.4.1.1 Werte liegen innerhalb des Mikrocontrollers als Konstanten vor

Bei diesem Verfahren werden die Werte in das Programm eingebunden und von dort direkt an das entsprechende Register des D/A-Wandlers geschrieben. Da das Signal periodisch sein soll, muss nach der Ausgabe des letzten Wertes wieder an den Anfang gesprungen werden. Dieser Sprung muss so ausgeführt werden, dass die Ausgabe der Werte an den D/A-Wandler periodisch bleibt, d.h. der Weg vom Ende der Ausgabe zurück zum Anfang muss zeitlich so bemessen sein, dass im Signal kein unerwünschter Phasensprung entsteht.

Im folgenden Beispiel werden die Werte 80h, F9h, CBh, 34h und 06h zyklisch ausgegeben.

```
      LDR   r6,=DAC0DAT
;-----------------------------------
loop
      MOV   r2,#0x08000000                    ;1
      STR   r2,[r6]                           ;4
      NOP                                     ;1
      NOP                                     ;1
      NOP                                     ;1
      NOP                                     ;1
;-------- Start einer Ausgabe ------ Summe = 9 Zyklen
      MOV   r2,#0x0F9000000                   ;1
      STR   r2,[r6]                           ;4
      NOP                                     ;1
      NOP                                     ;1
      NOP                                     ;1
      NOP                                     ;1
;-------- Ende einer Ausgabe ------- Summe = 9 Zyklen
      MOV   r2,#0x0CB00000                    ;1
      STR   r2,[r6]                           ;4
      NOP                                     ;1
      NOP                                     ;1
      NOP                                     ;1
      NOP                                     ;1
;----------------------------------- Summe = 9 Zyklen
      MOV   r2,#0x03400000                    ;1
      STR   r2,[r6]                           ;4
      NOP                                     ;1
      NOP                                     ;1
      NOP                                     ;1
      NOP                                     ;1
;----------------------------------- Summe = 9 Zyklen
      MOV   r2,#0x00600000                    ;1
      STR   r2,[r6]                           ;4
      B     loop                              ;4
```

Beispiel 1 Zyklische Ausgabe von als Konstanten gespeicherten Werten

In Beispiel 1 wird in jedem neunten Zyklus ein Wert zum D/A Wandler ausgegeben. Damit ergibt sich im obigen Beispiel mit einem Core-Takt von 5,22 MHz alle 1,72 µs ein neuer Wert auf den D/A-Wandler geschaltet wird. Dies entspricht einer Tastfrequenz von 580 kHz. Durch das Abtasttheorem wird die maximal generierbare Frequenz f_s auf 290 kHz begrenzt.

Die *No-Operation-Befehle* (*NOPs*) zwischen den einzelnen Ausgaben sind notwendig, um eine Phasenverschiebung beim kontinuierlichen Signal zu verhindern. Würden die Werte unmittelbar hintereinander ausgegeben werden, so wäre zwar eine höhere Tastfrequenz realisiert, aber durch den Rücksprung an die Adresse Loop wäre die zyklische Ausgabe unterbrochen.

Das Verfahren hat mehrere Nachteile. Zum einen muss für jedes auszugebende Signal ein nahezu gleiches Programmteil geschrieben werden, zum anderen werden die Quellcodes dadurch sehr lang. Diese Nachteile vermeidet das nächste Verfahren.

2.4.1.2 Ausgabe von Tabellenwerten

Die Werte werden in Form einer Tabelle im Speicher abgelegt. Das Auslesen der Tabelle und das Ausgeben der Werte können durch verschiedene Algorithmen erfolgen, von denen einer exemplarisch vorgestellt werden soll.

Für die Arbeit mit Tabellen bietet sich der *LDR* Befehl mit Pre-Indexed Adressierung

```
ldr       r2,[r4, r3, lsl #2]
```

an. Die Ausgabe an den D/A-Wandler erfolgt über den *STR* Befehl. Wendet man diese Schritte iterativ an, kann eine ganze Tabelle ausgelesen werden. Das folgende Beispiel demonstriert den Sachverhalt:

```
        LDR   r6,=DAC0DAT              ;Adresse von DAC in r6 laden
        MOV   r3,#3                    ;r3+1 ist di Anzahl der Tabellenwerte
        ADR   r4,tabelle               ;Basisadresse von Tabelle in r4 laden

;-------- Start eine Ausgabe --------
ausgabe
        NOP                            ;1
        NOP                            ;1
anf
        LDR   r2,[r4,r3,lsl #2]        ;5
        STR   r2,[r6]                  ;4
        SUBS  r3,r3,#1                 ;1
        BGE   ausgabe                  ;4 oder 1

;-------- Ende einer Ausgabe ---------
        MOV   r3,#3                    ;1
        B     anf                      ;4
;------------------------------------ Tabelle
tabelle
   DCD 0x03400000, 0x0F9000000, 0x03400000, 0x00600000
```

Beispiel 2 Auslesen einer Tabelle

Um einen Wert auszugeben, werden 16 Zyklen benötigt. Mit einem Core-Takt von 5,22 MHz entspricht es einer Tastfrequenz von 326 kHz. Die dadurch maximal generierbare Signalfrequenz ist somit auf 163 kHz beschränkt.

Die Verfahren aus Beispiel 1 und 2 haben eines gemeinsam: Die Werte zur Darstellung der Signale müssen vor ihrer Verwendung errechnet und abgespeichert werden. Falls aber der Algorithmus einfach zu implementieren bzw. das Signal sehr niederfrequent ist, kann die Berechnung vom Mikrocontroller in Echtzeit durchgeführt werden.

2.4.1.3 Berechnung der Werte in Echtzeit

Als einfaches Beispiel der Berechnung eine Rampenfunktion. Das Register R0 wird auf 00h gesetzt und ausgegeben. Anschließend wird das Register inkrementiert und erneut ausgegeben. Führt man diese Schritte hintereinander aus, so ändert sich der auszugebende Wert von 000h bis FF0h. Das entspricht einer langsam ansteigenden Funktion mit steilem Abfall beim Übergang von FF0h zu 000h.

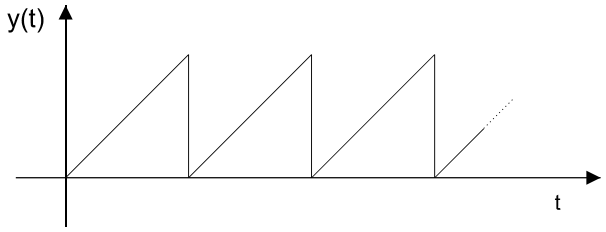


Abbildung 13: Rampen- oder Sägezahnfunktion

Das Programm könnte so aussehen:

```
      LDR     r6,=DAC0DAT
      MOV     r0,#0x00000000
      STR     r0,[r6]
;---- Beginn der Ausgabeschleife -----
loop
      ADD     r0,r0,#0x00010000        ;1
      STR     r0,[r6]                  ;4
      CMP     r0,#0x0ff00000           ;1
      MOVEQ   r0,#0x00000000           ;1
      B       loop                     ;4
;----- Ende der Ausgabeschleife ------
```

Beispiel 3 Berechnung der Werte in Echtzeit

Ist der Algorithmus zur Berechnung der Werte komplizierter, so steigt die Anzahl der benötigten Zyklen stark an, während die höchste noch generierbare Signalfrequenz absinkt.

Bei den Ausgaberoutinen Beispiel 1 und Beispiel 2 wurden die diskreten Werte als gegeben vorausgesetzt. Wie man zu den Werten gelangen kann, beschreibt der nächste Abschnitt.

2.4.2 Errechnen der digitalen Werte

Um die digitalen Werte zu erhalten, muss nicht erst ein analoges Signal abgetastet werden. Die Werte können selbstverständlich numerisch ermittelt werden.

Die Überlegung dazu ist folgende: Um eine zyklische Ausgabe zu erreichen, müssen sich die Werte früher oder später wiederholen. Wie viele Perioden bzw. Werte notwendig sind, hängt vom Verhältnis der zu realisierenden Frequenz f_S zur Abtastfrequenz f_A ab:

$$\frac{f_A}{f_S} = V \qquad \text{Gl. 1.7}$$

Existiert ein Verhältnis

$$N = P \cdot V \qquad \text{Gl. 1.8}$$

wobei N für die Anzahl der Werte und P für die Anzahl der Perioden steht, so ist eine Synthese möglich, vorausgesetzt, dass N und P aus dem Bereich der natürlichen Zahlen $\mathbb{N}$ stammen. Es ist leicht einzusehen, dass eine nicht ganzzahlige Anzahl von Werten nicht realisierbar ist. Ist die Anzahl P der Perioden nicht ganzzahlig, so kann das Signal nicht periodisch wiederholt werden.

Die Wahl der Quantisierung geht ebenfalls in die Berechnung ein. Dazu wird die Nulllinie des gewünschten Funktionsverlaufs in die Mitte des Zahlenbereichs gelegt (bei einer 8 bit-

Quantisierung also auf $2^8/2 = 256 \rightarrow$ 80h). Somit errechnet sich beispielsweise eine Sinusfunktion folgendermaßen:

$$A(i) = \frac{2^8}{2} \cdot \left[1 + \sin\left(2\pi \frac{f_S}{f_A} \cdot i\right)\right] \text{ mit } i = 0, \dots, N-1 \qquad \text{Gl. 1.9}$$

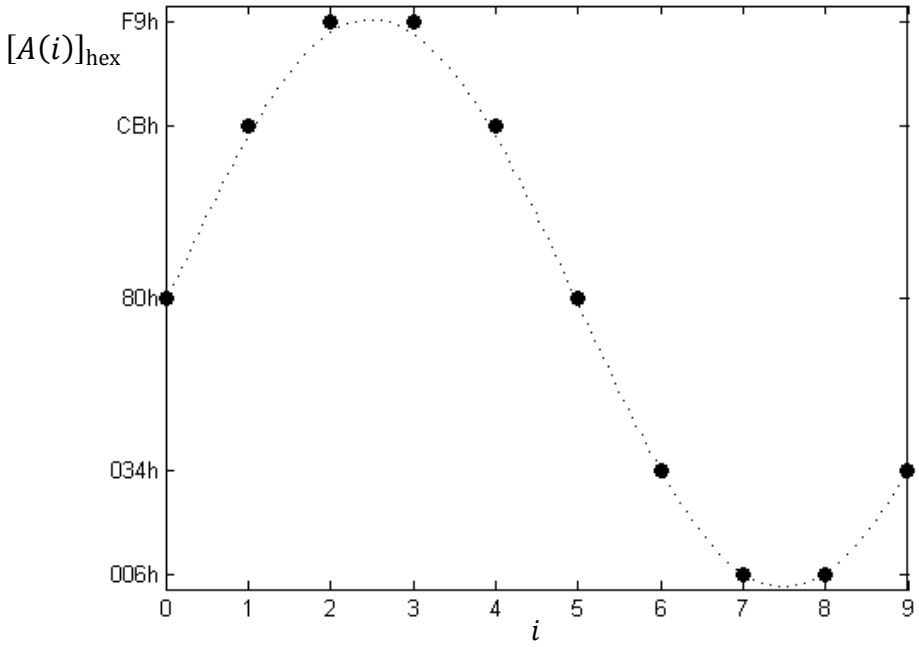


Abbildung 14: Quantisierungsbeispiel einer 8 bit-Quantisierung mit $\boldsymbol{P = 1}$, $\boldsymbol{f_S = 10}$ kHz und $\boldsymbol{f_A = 100}$ kHz, $\boldsymbol{V = f_A/f_S = 10}$, $\boldsymbol{N = P \cdot V = 20}$. Die gepunktete Linie zeigt den dazu korrespondierenden kontinuierlichen Verlauf an.

2.5 Praktikumsversuch

Für den Versuch zur digitalen Frequenzsynthese mit dem ADuC7026 Development System steht am Versuchstag eine komplette Entwicklungsumgebung bereit. Die Software wird auf einem PC mit Hilfe des Entwicklungstools *Keil µVision IDE* erstellt. Im Folgenden wird der Versuchsaufbau beschrieben.

Der Aufbau besteht aus einem PC, einem Signalgenerator, einem µC ADuC7026 mit integrierten D/A- und A/D-Wandler, einem Rekonstruktionsfilter vierter Ordnung mit einer festen Grenzfrequenz von 100 kHz und einem einstellbaren Tiefpassfilter aus geschalteten Kondensatoren (switched-capacitor-filter: SC-Filter).

Die Grenzfrequenz des SC-Filters lässt sich durch Variation einer Schaltfrequenz ändern und die Schaltfrequenz wird mit dem PWM-Modul des µCs erzeugt. Die vom Mikrocontroller generierten Werte werden über den D/A-Wandler ausgegeben. Das Ausgangssignal des Wandlers gelangt zu den Rekonstruktionsfiltern. Verschiedene Analogsignale, die von einem Signalgenerator erzeugt werden, werden ohne Tiefpassfilterung direkt von einem im µC integriertem A/D-Wandler abgetastet. Aliasing-Effekte können deshalb beobachtet und eine unbekannte Abtastrate des A/D-Wandlers damit ermittelt werden. Mit Hilfe des Flashtools

ARMWSD.exe kann die vom Compiler generierte Hex-Datei über die serielle RS232- oder JTAG-Schnittstelle in den Flashspeicher des Mikrocontrollers geladen werden.

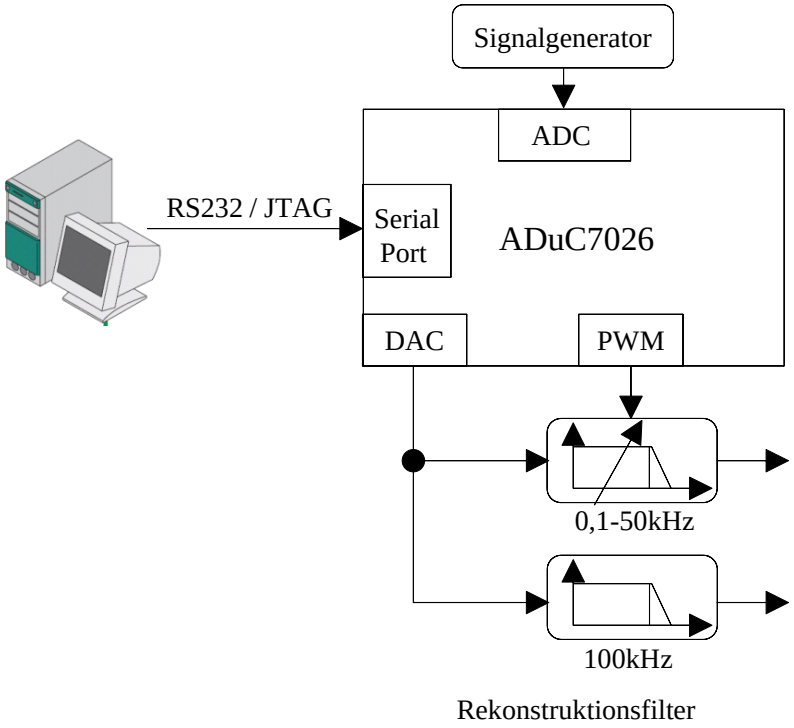


Abbildung 15: Versuchsaufbau

2.6 Aufgaben

In diesem Kapitel werden einige Aufgaben gestellt, die zum Verständnis und zur Vertiefung des Stoffes dienen sollen. Sie sollten nach Möglichkeit vor dem Versuchstag bearbeitet werden. Die Aufgaben 1 bis 3 dienen der Vorbereitung, während die Aufgaben 4 bis 6 am Versuchstag selbst zu erledigen sind. Die Programme sollen in der Programmiersprache Assembler erstellt werden. Für jeden Aufgabenteil ist ein Projekt vorhanden, in dem die notwendigen Einstellungen zur Codegenerierung durchgeführt wurden.

Verinnerlichen Sie sich zudem das detallierte Blockdiagramm des ADuC7026. Falls Sie es bei einer Abfragung zeichnen müssen, dann können die Pinne zusammengefasst werden.

2.6.1 Aufgabe 1

a) Beschreiben Sie ARM-Register, die Organisation des Speichers sowie die einzelnen Speicherblöcke (Flash/EE, SRAM, MMR).

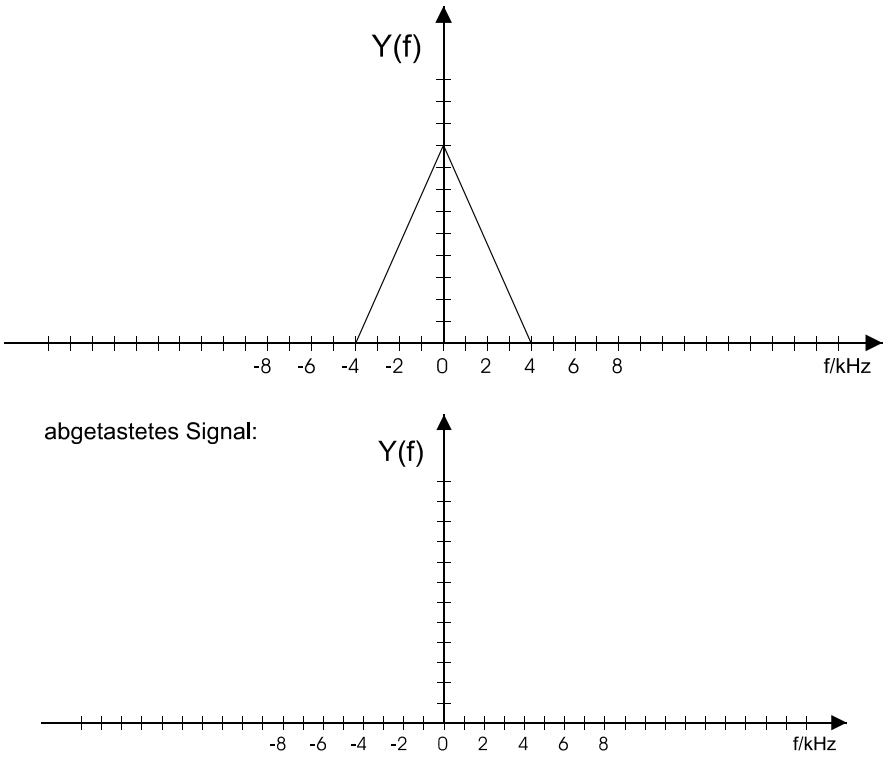
b) Mit welchem MMR-Register (Name und Adresse) wird der D/A-Wandler DAC2 konfiguriert? Schreiben Sie zwei alternative Befehle in *ARM Assembly Programming*, der den DAC2 im Bereich 0 V bis V_{REF} (2,5 V) arbeiten lässt. Schreiben Sie dann einen Befehl, der den höchsten Wert Digital-Analog wandelt (Am DAC2-Pin sind 2,5 V zu messen, nachdem der Befehl ausgeführt wird).

c) Schreiben Sie im Assembler-Code eine Code-Sequenz mit der Sie *core clock frequency* mit 20,89 MHz konfigurieren. Warum wird das Register POWCON in einer bestimmten Sequenz geändert?

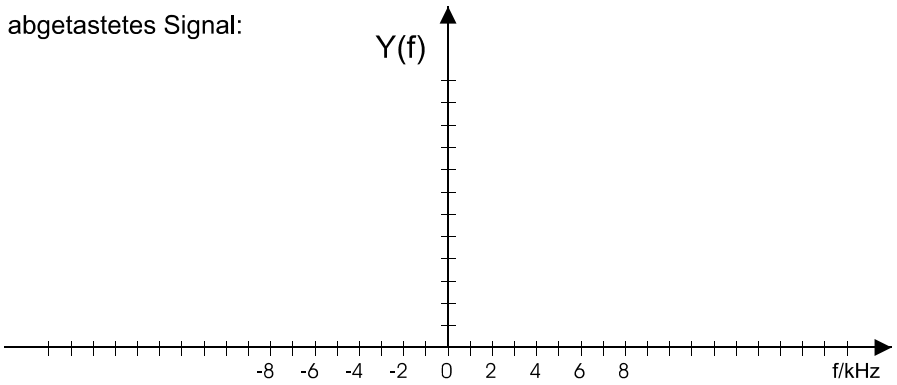
d) Erklären Sie folgenden Codeausschnitt:

```
            …
      LDR     r6,=DAC0DAT
      MOV     r3,#3
      ADR     r4,valuetable

loop
      LDR     r2,[r4,r3,lsl #2]
      STR     r2,[r6]
      SUBS    r3,r3,#1
      MOVMI   r3,#3
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
B     loop
; ------------------------------; sum Cycles = 17 + 9 nops

      stop    B  stop

valuetable
      DCD   0x00000000,0x07F00000,0x0FF00000,0x07F00000
```

e) Ein Signal mit folgendem Spektrum werde mit einer Frequenz von 10 kHz abgetastet. Wie sieht das Frequenzspektrum des abgetasteten Signals aus?



f) Nun wird das Signal aus Aufgabe 1 e) nur noch mit einer Frequenz von 6 kHz abgetastet. Wie sieht jetzt das Spektrum des abgetasteten Signals aus?



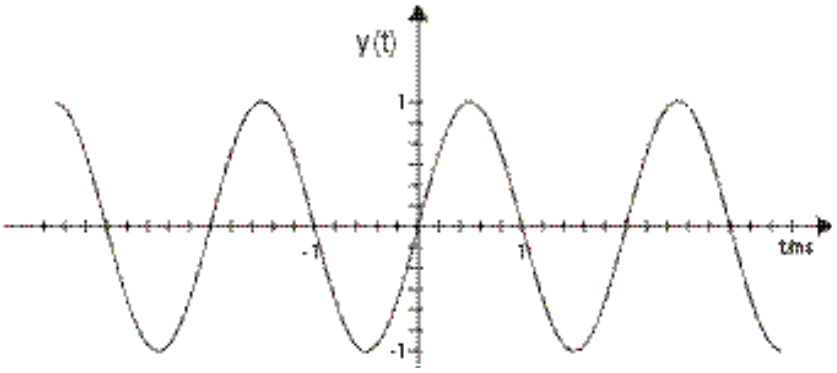
g) Wie nennt man den Effekt, der sich in 1 f) bemerkbar macht? Was besagt das Abtasttheorem?

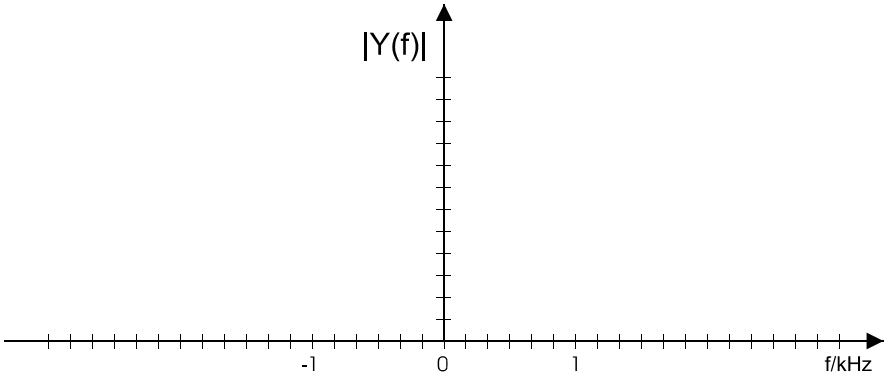
h) Genügt ein D/A-Wandler zur Rekonstruktion eines Signals? Begründen Sie Ihre Antwort!

2.6.2 Aufgabe 2

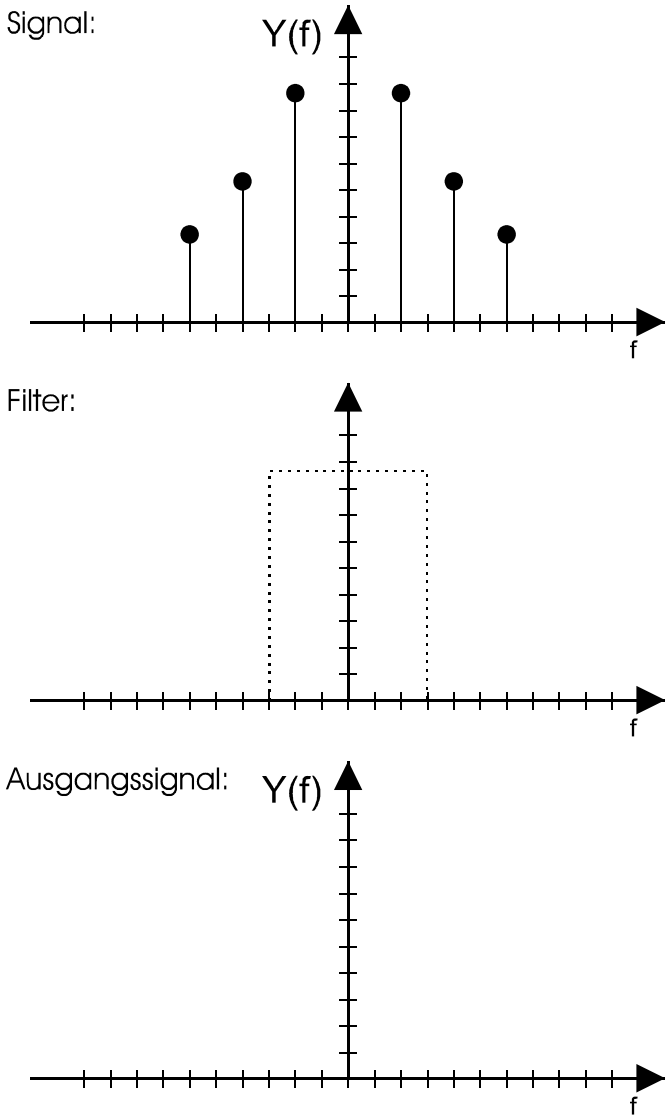
a) Wie sieht das Spektrum des folgenden sinusförmigen Zeitsignals aus?

Signal y(t):





b) Ein Signal mit folgendem Frequenzgang durchlaufe ein Filter. Wie sieht das Ausgangssignal des Filters aus?



c) Ein Signal soll generiert werden. Es wird alle 5µs ein Wert am D/A-Wandler ausgegeben. Welcher Abtastfrequenz entspricht dies und welche Frequenz kann maximal erzeugt werden?

2.6.3 Aufgabe 3

Ein sinusförmiges Signal der Frequenz 10 kHz und einer Abtastfrequenz von 100 kHz soll mit 8 bit quantisiert und ausgegeben werden.

a) Wie lautet die Formel zur Berechnung der diskreten Werte?

b) Wie viele Perioden und Werte müssen errechnet und abgespeichert werden?

Anzahl der Perioden:

Anzahl der Werte:

c) Errechnen Sie die Werte und tragen Sie sie in die Tabelle ein.

Index	0	1	2	3	4	5	6	7	8	9	10	11
$[A]_{\text{hex}}$												

Es soll ein sinusförmiges Signal mit einer Abtastfrequenz von 100 kHz und einer Signalfrequenz von 16 kHz ausgegeben werden.

d) Wie viele Perioden und Werte sind notwendig?

Periodenzahl P:

Anzahl der Werte N:

e) Berechnen Sie die Werte für eine 8 bit-Quantisierung und tragen Sie sie in die Tabelle ein!

Index	0	1	2	3	4	5	6	7	8	9	10	11
$[A]_{\text{hex}}$												
Index	12	13	14	15	16	17	18	19	20	21	22	23
$[A]_{\text{hex}}$												
Index	24	25	26	27	28	29	30	31	32	33	34	35
$[A]_{\text{hex}}$												

2.6.4 Aufgabe 4

a) In Aufgabe 3 c) sollten Sie die für die Ausgabe eines sinusförmigen Signals mit einer Signalfrequenz von 10 kHz und einer Abtastfrequenz von 100 kHz notwendigen Werte berechnen. Erstellen Sie mit diesen Werten ein Flussdiagramm zur Ausgabe des Signals und setzen Sie dieses in ein Programm um.

Öffnen Sie hierzu das Projekt c:\versuch2\aufgabe4a\Aufgabe4a.Uv2. Falls noch nicht geschehen, binden Sie den Programmrahmen Aufgabe4a_h.s ein: Project Workspace → rechte Maustaste auf Source Group 1 klicken → Add Files to Group 'Source Group 1' → Datei Aufgabe4a_h.s wählen, Add → Close.

- Überprüfen Sie, ob die Taktfrequenz des Mikrocontrollers auf 2,61 MHz eingestellt ist.
- Öffnen Sie Aufgabe4a.s (aber nicht ins Projekt einbinden) und fügen Sie Ihr Programm zur Initialisierung des D/A-Wandlers ein (`add code 1`).
  - ✓ DAC0 muss eingesetzt werden
  - ✓ Spannungsbereich des Ausgangs: 0 V - AV_{DD}
  - ✓ DAC-Werte werden durch Systemtakt (core clock) aktualisiert

- Fügen Sie die berechneten Werte unter dem Label ValueTable ein (add code 2). Benutzen Sie dabei Direktive DCD. Die Position des 8 bit-Wertes innerhalb eines Wortes (32 bit) für den D/A-Wandler ist im Datenblatt des µCs vorgegeben.
- Kompilieren Sie das Projekt und programmieren Sie den µC über die serielle Schnittstelle. Betrachten Sie die Signale mit Hilfe eines Oszilloskops. Wie erklären Sie die unterschiedlichen Signale hinter dem SC-Filter und dem 100 kHz-Filter?

b) Entwerfen Sie ein Flussdiagramm zur Ausgabe eines rechteckförmigen Signals mit einer Signalfrequenz von 16,3 kHz und Tastverhältnis von 50%. Öffnen Sie das Projekt c:\versuch2\aufgabe4b\Aufgabe4bUv2. Falls es noch nicht geschehen, binden Sie den Programmrahmen Aufgabe4b_h.s (siehe oben) ein.

- Stellen Sie die Taktfrequenz des Mikrocontrollers auf 653 kHz ein (add code 1).
- Falls es noch nicht geschehen, binden Sie Aufgabe4b_50.s unter Verwendung von Direktive GET in den Programmrahmen ein.
- Öffnen Sie Aufgabe4b_50.s (aber nicht einbinden) und fügen Sie Ihr Programm ein (add code 2 und add code 3).
- Kompilieren Sie das Projekt und programmieren Sie den µC über die serielle Schnittstelle.
- Binden Sie nun Aufgabe4b_30.s unter Verwendung von Direktive GET in den Programmrahmen ein.
- Öffnen Sie Aufgabe4b_30.s, ändern Sie das Programm so, dass das Rechtecksignal ein Tastverhältnis von 30% aufweist.

2.6.5 Aufgabe 5

In dieser Aufgabe sind 580 Werte für eine Periode des Sinussignals in der Datei SineGen.s ab dem Label ValueTable gespeichert und sollen zur Erzeugung von Sinussignalen verschiedener Frequenzen mit einer Abtastrate von 580 kHz über den D/A-Wandler ausgeben werden. Um die Grenzfrequenz des SC-Filters auch zu ändern, soll die Frequenz dieses Tiefpassfilters über PWM-Signal entsprechend eingestellt werden. Die resultierende Grenzfrequenz des SC-Filters soll zum Schluss mittels des Sinussignals untersucht werden. Öffnen Sie das Projekt c:\versuch2\aufgabe5\Aufgabe5.Uv2. Falls noch nicht geschehen, binden Sie den Programmrahmen Aufgabe5_h.s ein (siehe Aufgabe 4).

a) Öffnen Sie Aufgabe5_h.s. Stellen Sie die Taktfrequenz des Prozessors auf 10,44 MHz ein (add code 1).

b) Binden Sie SineGen.s unter Verwendung von Direktive GET in den Programmrahmen ein. Öffnen Sie nun SineGen.s.

- DAC0 soll wie in Aufgabe 4 konfiguriert werden (add code 2).
- Die *For*-Schleife dauert 17 Zyklen. Um eine Abtastrate von 580 kHz zu realisieren, sollen nun *NOP*-Befehle eingefügt werden. Überlegen Sie sich, wie viele *NOP*s verwendet werden müssen (add code 3).

c) Berechnen Sie die realisierbaren größten bzw. kleinsten Frequenzen und die Frequenzauflösung.

d) Wie lässt sich die Frequenz des Sinussignals einstellen?

e) Schließen Sie den Ausgang des SC-Filters an ein Oszilloskop an, ermitteln Sie die Grenzfrequenz (Frequenz bei 3 dB Dämpfung) des Filters mit Hilfe der erzeugten Signale.

f) Der Programmschnitt Nr. 3 in Aufgabe5_h.s ist für die Steuerung des PWM-Moduls zuständig. Welche Register kann man zur Konfiguration der Pulsperiode verwenden?

g) Konfigurieren Sie die Register so, dass das PWM-Signal eine Periode von $t_s = 1{,}92$ µs aufweist.

h) Ermitteln Sie die Grenzfrequenz des Filters wie in e).

2.6.6 Aufgabe 6

Bei diesem Versuch sollen die Effekte, die durch Aliasing entstehen, im Zeitbereich untersucht werden. Dazu wird ein Programm benötigt, das ein Zeitsignal mit einer festen Abtastrate f_A abtastet und wieder ausgibt. Verwenden Sie zur Bearbeitung der Aufgabe das Projekt …\versuch2\aufgabe6\Aufgabe6.Uv2.

a) Binden Sie den Programmrahmen Aufgabe6_h.s in dieses Projekt (siehe Aufgabe 4). Stellen Sie den Systemtakt auf 2,61 MHz ein (add code 1).

b) Binden Sie ADCDAC.s unter Verwendung von Direktive GET in den Programmrahmen ein. Öffnen Sie die Datei ADCDAC.s und ergänzen Sie sie wie folgt:

- DAC0 soll wie in Aufgabe 4 konfiguriert werden (add code 2).
- A/D-Wandler (add code 3):
  - ✓ interne Referenzspannung (2,5V) soll mit Pin VREF verbunden werden (Register REFCON)
  - ✓ kontinuierliche Software-Umwandlung
  - ✓ single-ended Modus
  - ✓ ADC einschalten und freischalten
  - ✓ Aufnahmedauer: 4 Takte
  - ✓ ADC Taktfrequenz: $f_{ADC}/4$
  - ✓ ADC0 als positiver Eingang (Register ADCCP) freigeben

c) Programmieren Sie die ADC und DAC so, dass der µC ständig den Wert aus ADC ausliest und gleich über den DAC ausgibt (add code 4).

Der Analogeingang des µC wird mit dem Frequenzgenerator verbunden, der ein Sinussignal mit einer Amplitude von ca. 1V und einem Offset von 1V erzeugt. Das Offset ist notwendig, da der Analogeingang des µC nur Spannungen zwischen 0V und 2.7V verarbeiten kann. Der Ausgang des D/A-Wandlers wird mit dem Oszilloskop verbunden.

d) Was können Sie sehen wenn Sie die Frequenz des Sinussignals kontinuierlich erhöhen?

e) Wie kommen die beobachteten Effekte zustande?

f) Wie kann man die Abtastfrequenz mit dem Sinusgenerator und dem Oszilloskop ermitteln, wie groß ist die Abtastfrequenz des Programms wirklich?