Fakultät für Technik, Studiengänge EIT/TI
Dipl.-Ing.(FH) Felix Becker

Infoblatt

Code Composer Studio v5 (CCS v5.1)

C-Code für die C6000-Architektur

Grundlagenliteratur: Kernighan / Ritchie, Programmieren in C

Weiterführende Informationen: PDF-Dateien von TI
CCS Help

Das erste Projekt / Übersicht der eingebundenen Dateien:

Das erste Projekt (Laborversuch 1) enthält neben der eigentlichen C-Quellcode-Datei `sinetone1.c` die drei folgenden Dateien:

Header-Datei für das DSK6713:

`c6713dsk.h`

Enthält insbesondere die Registerdefinitionen (Alias) der CPU und der On-Chip Peripherie wie z.B. der McBSPs, der Timer, des EMIF und der EDMA sowie verschiedene DSK-Board-spezifische Adressdefinitionen.

Interrupt-Vektor-Tabelle für das DSK6713:

`vectors.asm`

Belegung der 14 möglichen CPU-Interrupt-Vektoren (Reset, NMI, INT4 ... INT15).

Wichtigster Vektor: Reset (belegt mit dem C/C++ Programmeinsprung zu `_c_int00` von wo aus dann u.a. auch `main()` aufgerufen wird)

Ein Interrupt-Vektor besteht aus acht 32-Bit-Instruktionen, die in einem Fetch-Packet FP (siehe VLIW bzw. VelociTI) geladen werden.

Prinzipiell lassen sich die folgenden beiden Fälle unterscheiden:

- die komplette Interrupt-Bearbeitung kann mit diesen 8 (Assembler-)Befehlen durchgeführt und abgeschlossen werden
- die Interrupt-Bearbeitung benötigt mehr als 8 Instruktionen, es erfolgt deshalb ein Sprung zu einer Adresse mit weiterem Interrupt-Programmcode (interrupt service routine code).

Linker-Datei für das DSK6713:

`dsk6713.cmd`

Der C6000 Compiler erzeugt Code und Daten für einen fiktiven, zusammenhängenden 32 Adressbit großen Speicherbereich, der in Code- und Datenunterbereiche (sog. sections) aufgeteilt ist. Der Compiler hat keine Kenntnis über den im jeweiligen System tatsächlich vorhandenen bzw. nutzbaren Speicher. Die vom Compiler erzeugten Code- und Datenunterbereiche sind deshalb verschiebbar (relocatable).

Es ist dann die Aufgabe des Linkers, die Abbildung der Code- und Datenbereiche auf die tatsächlich vorhandenen Speicherblöcke vorzunehmen.
Er erledigt diese Aufgabe anhand der Vorgaben in der Linker-Konfigurationsdatei. Diese Datei enthält zwei Gruppen von Einträgen:

- `memory {}` deklariert den im System vorhandenen Speicher (z.B. in- und externen)

und

- `sections {}` ordnet die Unterbereiche (.cinit, .const, .text, .bss ...) dem deklarierten Speicher zu

Library-Datei für das DSK6713:

Die bei der Compilierung eines DSK6713-Projekts benötigte Run-Time-Support-Bibliothek befindet sich zentral im Programmverzeichnis von CCS v5.

Run-Time-Support Object Library (for little endian C-code)

`rts6700.lib`

Inhalt:

- ANSI C/C++ standard library
- C I/O library
- Low-level-support Funktionen, die den Datenaustausch mit dem PC ermöglichen
- Intrinsic arithmetic routines
- System startup routine: _c_int00
- Funktionen und Makros, die C/C++ den Zugriff auf spezielle Funktionen erlauben

Datentypen (signed und unsigned):

`char`	8 Bit
`short`	16 Bit
`int`	32 Bit
`long`	40 Bit
`float`	32 Bit
`double`	64 Bit

Hinweise zur Benutzung der Datentypen:

Die **Standard-Datentypen** für Festkommazahlen sind **`int` / `unsigned int`** bzw. **`short` / `unsigned short`**. Für Fließkommazahlen verwenden Sie vorzugsweise **`float`** (siehe Infoblatt zum IEEE 32 Bit Gleitkommaformat).

Die Benutzung des Datentyps `long` anstelle von z.B. `int` hat möglicherweise einige Einschränkungen zur Folge. So können z.B. nicht alle Funktions-/Recheneinheiten des DSP 40 Bit breite Daten verarbeiten oder es müssen zusätzliche Instruktionen in den Assembler-Code eingefügt werden. Wählen Sie den Datentyp `long` also nur, wenn Sie die 40 Bit Datenbreite wirklich benötigen.

Wenn immer möglich, sollte für Festkomma-Multiplikationen der Datentyp `short` benutzt werden, da dann der effiziente 16-Bit Multiplizierer des DSPs eingesetzt werden kann. (Z.B. wenn die Daten von einem 16 Bit A/D-Wandler kommen und für die Koeffizienten die 16 Bit Genauigkeit ebenfalls ausreicht.).

Für **Schleifenzähler** benutzen Sie allerdings besser **`int`** oder **`unsigned int`**, da sonst eine unnötige sign-extension erfolgen muss.

Typkonvertierung: Gleitkomma -> 32 Bit Festkomma

```
(int)float_wert
```

Sinnvollerweise muss vor der Konvertierung dafür gesorgt / sichergestellt werden, das die Zahl im neuen Zahlenraum (-2^{31} ... $2^{31} - 1$) auch abbildbar ist.

Typkonvertierung: Gleitkomma -> 16 Bit Festkomma

```
(short)float_wert
```

Sinnvollerweise muss vor der Konvertierung dafür gesorgt / sichergestellt werden, das die Zahl im neuen Zahlenraum (-2^{15} ... $2^{15} - 1$) auch abbildbar ist.

Typkonvertierung: 16 Bit Festkomma -> 32 Bit Festkomma

```
(int)short_wert
```

Direkter Zugriff auf adressierbare Prozessor- und Peripherie-Control-Register:

Adressierbare Prozessor- und Peripherie-Control-Register sind in der Header-Datei `c6713dsk.h` über ihre Adresse definiert, z.B.: `#define McBSP1_DXR 0x01900004`

Der Zugriff auf ein solches Control-Register sieht in C dann wie folgt aus:

```
*(unsigned volatile int *)McBSP1_DXR
```

schreibend z.B.:

```
*(unsigned volatile int *)McBSP1_SPCR = 0x00410001;
```

oder

```
int out_data;
*(unsigned volatile int *)McBSP1_DXR = out_data;
```

lesend z.B.:

```
int in_data;
in_data = *(unsigned volatile int *)McBSP1_DRR;
```

Alternativ kann auch direkt die Speicheradresse des Control-Registers angegeben werden:

```
*(unsigned volatile int *)0x01900004
```

ist identisch zu:

```
*(unsigned volatile int *)McBSP1_DXR
```

Erläuterungen:

Zunächst wird durch `(unsigned volatile int *)0x01900004` ein Zeiger auf die Speicheradresse `0x01900004` erzeugt, der auf einen Wert vom Datentyp `unsigned int` zeigt. Durch den vorangestellten `*` wird der Zeiger dann sofort für den Zugriff auf den Wert an dieser Speicherstelle benutzt: `*(unsigned volatile int *)0x01900004`.

Die zusätzliche Angabe `volatile` ist ein Hinweis an den Übersetzer (Compiler), eine aus seiner Sicht möglicherweise sinnvolle Optimierung nicht durchzuführen.

Beispiel:

Die wiederholte Abfrage eines Registers (wie z.B. `McBSP1_DRR` - data receive register) in einer Schleife könnte durch den Übersetzer zu einer einzigen Abfrage optimiert werden, da ja kein ihm bekannter Zugriff (des Programms) auf das Register erfolgt, der den Inhalt verändern würde und so eine wiederholte Abfrage für ihn als sinnvoll erscheinen lässt.

Direkter Zugriff auf die internen Prozessor-Control-Register:

Interne Prozessor-Control-Register wie z.B. `CSR, IER, ICR` usw. sind in der Header-Datei `c6713dsk.h` über das Schlüsselwort `cregister` für den Zugriff unter C definiert, z.B.:

```
extern cregister volatile unsigned int CSR;
```

Das Setzen des GIE-Bits (Global Interrupt Enable - Bit = Bit 0) im Control-Status-Register sieht in C dann wie folgt aus:

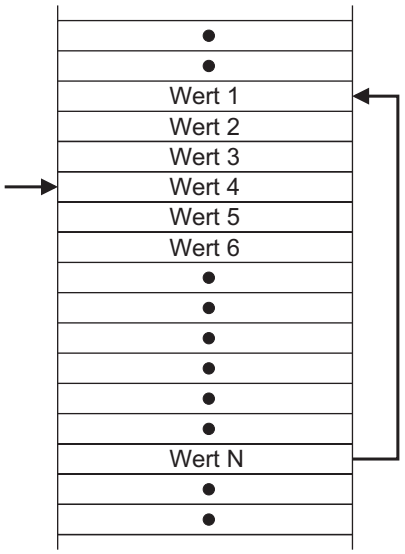
```
CSR = CSR | 0x00000001;
```

bzw.

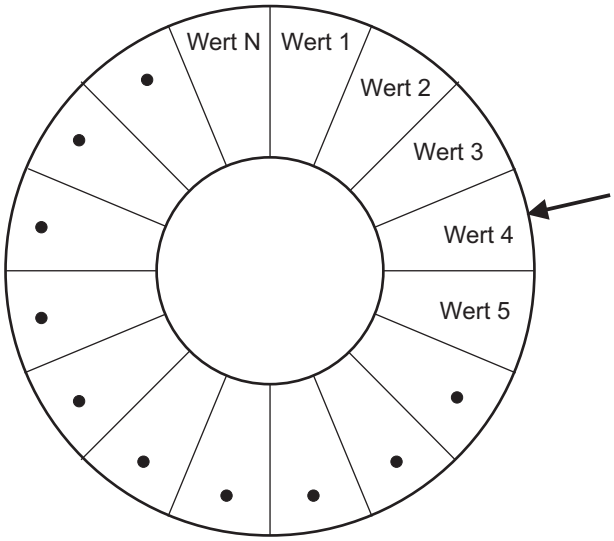
```
CSR |= 0x00000001;
```

Ringspeicher (-Adressierung) in C:

Unter einem Ringspeicher (Circular-Buffer) versteht man einen Speicherbereich der Größe N, der fortlaufend (wiederholt) ausgelesen oder beschrieben werden kann.
Dabei wird ein Zeiger z.B. nach jedem gelesenen Wert um 1 inkrementiert und zeigt somit auf das nächste Element. Ein Inkrementieren an der Stelle N führt dann bei der Ringspeicher-adressierung automatisch dazu, dass der Zeiger wieder auf das erste Element zeigt.
Anstelle von 1 kann selbstverständlich auch um höhere Werte inkrementiert werden.



Speicherabbild



Prinzipdarstellung – umlaufender Zeiger

Der TMS320C6713 unterstützt die Ringspeicheradressierung durch speziell dafür vorgesehene Adressierlogik, die entsprechend konfiguriert werden muss (als Zeiger verwendbare Register: `A4-A7, B4-B7`; Konfiguration über das Address Mode Register, `AMR`). Der Zugriff darauf erfolgt sinnvollerweise in Assembler. Unter C steht noch keine Implementierung zur Verfügung. Die Ringspeicherverwaltung erfolgt in C deshalb konventionell durch entsprechende Zeiger-abfragen bzw. durch Modulo-Zeigerarithmetik.
Die folgenden beiden Programm-Beispiele verdeutlichen dies.

Annahmen für die folgenden zwei Beispiele:

Es existiert ein `buffer` der Größe `buf_size`. Die Schrittweite beträgt `steps`.

C-Beispielcode für beliebige Ringspeichergrößen:

```
int k, offset = 0;
??? sample;  // float oder integer

for (;;)  // Endlos-for-Schleife
{
  for (k = offset; k < buf_size; k += steps)
  {
    sample = buffer[k];
    ...
    ...
    ...
  }
  offset = k – buf_size;
}
```

C-Beispielcode für Ringspeichergrößen entsprechend einer 2er-Potenz:
(Ringspeichergrößen z.B.: 128, 256, 512, ... 8192 ...)

```
int k = 0;
??? sample;  // float oder integer

for (;;)  // Endlos-for-Schleife
{
  sample = buffer[k];
  ...
  ...
  ...

  k += steps;
  k &= buf_size - 1;  // buf_size - 1
                      // z.B. 8191 -> 0x0000 1fff hex
}
```

Konkretes Beispiel:

Ausgabe eines Mono-Sinustones über den Stereo-Codec TLV320AIC23B auf dem DSK6713. Die Sinuswerte liegen als float-Zahlen im Bereich -1…+1 in einer z.B. 100 Elemente großen Tabelle `sine_table` im Speicher (sog. Look-up-table).

Die float-Zahlen müssen passend zum Wertebereich des Codecs korrigiert, d.h. mit einem Lautstärkefaktor multipliziert und dann ins Festkommaformat konvertiert werden.
Vor dem Transfer an die beiden D/A-Wandler im Stereo-Codec über den McBSP1 muss ein 32-Bit-Datenwort gebildet werden, dass das jeweilige 16-Bit-Sample zweimal enthält (Ausgabe eines Mono-Sinustones). Dies kann durch ‘verodern‘ der beiden Samples geschehen, wobei vorher eine entsprechende Verschiebung des linken Samples um 16 Stellen nach links erfolgen muss und das rechte Sample mit einer passenden Maske ‘verundet‘ werden muss.

Annahme für das folgende Beispiel:

Es existiert eine Funktion `mcbsp1_write()`, die 32-Bit-Werte an den McBSP1 übergibt.

Beispiel-Code:

```
float gain = 10000.0;  // maximal 32767.0 für Ausgabe auf TLV320AIC23
short sample;  // lautstärkeangepasste Zwischenvariable
               // (16 Bit K2 Format)
int k;
int offset = 0;  // Ausgabebeginn mit erstem Element des Ringspeichers
int tab_size = sizeof(sine_table) / sizeof(float);  // Anzahl samples
int steps = 1;  // Schrittweite mit der durch den Ringspeicher
                // gegangen wird (bestimmt die Frequenz als Vielfaches
                // der Grundfrequenz)

for (;;)  // Endlos-for-Schleife für Ausgabe eines Dauertones
{
  for (k = offset; k < tab_size; k += steps)
  {
    sample = (short)(sine_table[k] * gain);
    mcbsp1_write( ((int)sample)<<16 | ((int)sample & 0x0000FFFF) );
  }
  offset = k – tab_size;  // Ist steps z.B. 3, dann bekommt offset
                          // für den nächsten Speicherdurchgang den
                          // Wert 2 (2=102-100), für den
                          // übernächsten Durchgang den Wert 1 und
                          // danach dann wieder 0 usw.
}
```

Interrupts in C:

CPU-Interrupts:

Die CPU der C67xx-DSPs kann neben dem Reset und dem NMI (Non Maskable Interrupt) über 12 frei belegbare, maskierbare Interrupts in Ihrer augenblicklichen Programmabarbeitung unterbrochen werden.

Als auslösende Interrupt-Quellen können Teile der On-Chip-Peripherie wie z.B. Timer, McBSPs, EMIF, EDMA etc., aber auch ein angebundener Hostprozessor oder mehrere spezielle Interrupt-Pins des DSP's dienen (beim TMS320C6713 insgesamt 23 Interrupt-Quellen). Zusätzlich können die maskierbaren CPU-Interrupts auch per Software manuell ausgelöst werden.

Mit Hilfe der Interrupt-Multiplexerregister MUXL und MUXH können diese Interrupt-Quellen beliebig auf die 12 maskierbaren CPU-Interrupts (INT4 – INT15) gelegt werden. Die Interrupt-Priorität nimmt mit aufsteigender CPU-Interruptnummer ab (INT4 hat die höchste Priorität, INT15 die niedrigste).

DSP INTERRUPT NUMBER	INTERRUPT SELECTOR CONTROL REGISTER	DEFAULT SELECTOR VALUE (BINARY)	DEFAULT INTERRUPT EVENT
INT_00	–	–	RESET
INT_01	–	–	NMI
INT_02	–	–	Reserved
INT_03	–	–	Reserved
INT_04	MUXL[4:0]	00100	GPINT4†
INT_05	MUXL[9:5]	00101	GPINT5†
INT_06	MUXL[14:10]	00110	GPINT6†
INT_07	MUXL[20:16]	00111	GPINT7†
INT_08	MUXL[25:21]	01000	EDMAINT
INT_09	MUXL[30:26]	01001	EMUDTDMA
INT_10	MUXH[4:0]	00011	SDINT
INT_11	MUXH[9:5]	01010	EMURTDXRX
INT_12	MUXH[14:10]	01011	EMURTDXTX
INT_13	MUXH[20:16]	00000	DSPINT
INT_14	MUXH[25:21]	00001	TINT0
INT_15	MUXH[30:26]	00010	TINT1

TMS320C67xx Interrupts (MUXL, MUXH nach Reset)

INTERRUPT SELECTOR VALUE (BINARY)	INTERRUPT EVENT	MODULE
00000	DSPINT	HPI
00001	TINT0	Timer 0
00010	TINT1	Timer 1
00011	SDINT	EMIF
00100	GPINT4†	GPIO
00101	GPINT5†	GPIO
00110	GPINT6†	GPIO
00111	GPINT7†	GPIO
01000	EDMAINT	EDMA
01001	EMUDTDMA	Emulation
01010	EMURTDXRX	Emulation
01011	EMURTDXTX	Emulation
01100	XINT0	McBSP0
01101	RINT0	McBSP0
01110	XINT1	McBSP1
01111	RINT1	McBSP1
10000	GPINT0	GPIO
10001	Reserved	–
10010	Reserved	–
10011	Reserved	–
10100	Reserved	–
10101	Reserved	–
10110	I2CINT0	I2C0
10111	I2CINT1	I2C1
11000	Reserved	–
11001	Reserved	–
11010	Reserved	–
11011	Reserved	–
11100	AXINT0	McASP0
11101	ARINT0	McASP0
11110	AXINT1	McASP1
11111	ARINT1	McASP1

TMS320C6713 Interrupt-Quellen

Zusätzlich zu diesem Interrupt-Mapping muss der gewünschte CPU-Interrupt und der Interrupt-Betrieb generell freigegeben werden.

Die Freigabe der CPU-Interrupts erfolgt mit Hilfe des Interrupt-Enable-Registers IER und die generelle Freigabe des Interrupt-Betriebs über das GIE-Bit im Control-Status-Register CSR. Zuvor sollte allerdings das Interrupt-Flag-Register IFR über das Interrupt-Clear-Register ICR gelöscht werden.

Damit die CPU überhaupt auf die maskierbaren Interrupts reagiert, muss im IER zusätzlich das Non-Maskable-Interrupt-Enable-Bit (NMIE-Bit = Bit 2) gesetzt sein.
(Das NMIE-Bit wird zu Beginn eines Non-Maskable-Interrupts gelöscht, um zu verhindern, dass der NMI durch einen anderen Interrupt unterbrochen werden kann. Beim Verlassen der NMI-Interrupt-Service-Routine wird das NMIE-Bit wieder gesetzt. Nach einem Reset muss es allerdings von Hand gezielt gesetzt werden.)

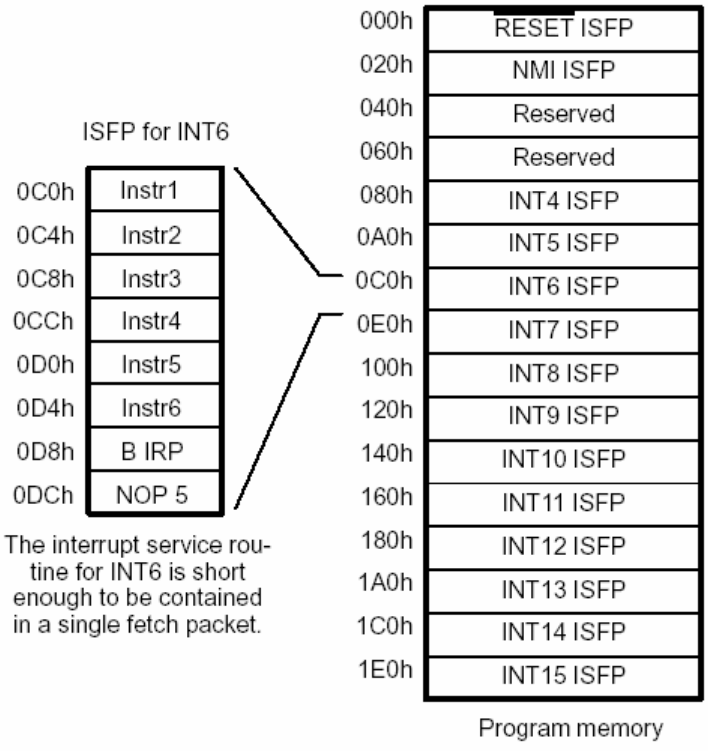
Interrupt Service Fetch Packet – ISFP:

Den CPU-Interrupts werden über die Interrupt-Vektor-Tabelle sogenannte Interrupt-Vektoren (Interrupt-Service-Fetch-Packets, 1 ISFP = acht 32-Bit-Instruktionen) zugeordnet.

(siehe auch Abschnitt „Interrupt-Vektor-Tabelle für das DSK6713“ am Anfang dieses Dokuments und die Datei: *vectors.asm*)

Diese Interrupt-Vektoren zeigen in der Regel auf Interrupt-Service-Routinen, die dann die eigentliche Interrupt-Bearbeitung übernehmen (für RESET: `_c_int00` – der Einsprung ins C-Programm oder für z.B. XINT1: `_isr11_xint1` – der Sprung zur C-Funktion `isr11_xint1` innerhalb eines C-Programms, zu beachten ist der dem Label (C-Funktionsnamen) voranzustellende Unterstrich).

Es ist allerdings auch möglich, die komplette Interrupt-Bearbeitung eines CPU-Interrupts mit nur den 8 zur Verfügung stehenden Assembler-Befehlen eines Interrupt-Service-Fetch-Packets (ISFP) durchzuführen, wenn die zu programmierende Aufgabe damit bewältigt werden kann.



Interrupt-Vektor-Tabelle mit Byte-Adressangaben
(„ISFP for INT6“ ist ein Beispiel für die Interrupt-Bearbeitung ohne separate ISR)

Interrupt Service Routine – ISR:

In C werden Interrupt-Service-Routinen als Funktionen vom Typ `void` geschrieben und mit dem Schlüsselwort `interrupt` versehen (z.B.: `interrupt void isr11_xint1(void)`).

Einer ISR werden keine Werte übergeben, sie kann allerdings auf global definierte Variablen des C-Programms zugreifen und diese verändern.

Über das Schlüsselwort `interrupt` wird dem C-Compiler mitgeteilt, dass alle Registerinhalte von in dieser Funktion benutzten Registern vor der Verwendung auf dem Stack gerettet und beim Verlassen der Funktion wieder zurückgeschrieben werden müssen. Für den einfachsten Fall ruft eine ISR deshalb auch keine weitere Funktion auf.

Zusammenfassung der für den Interrupt-Betrieb nötigen Maßnahmen und Einstellungen:

1. Zuordnung von Interrupts der (OnChip-) Peripherie zu den 12 CPU-Interrupts über die Interrupt-Multiplexer-Register MUXL und MUXH
2. Erstellen der gewünschten Interrupt-Vektoren in der Interrupt-Vektor-Tabelle
3. Erstellen der gewünschten Interrupt-Service-Routinen
4. Löschen des Interrupt-Flag-Registers IFR mit Hilfe des Interrupt-Clear-Registers ICR
5. Freigabe der gewünschten CPU-Interrupts über die entsprechenden Bits im Interrupt-Enable-Register IER

6. Freigabe der Non-Maskable-Interrupts über das NMIE-Bit im Interrupt-Enable-Register
7. Setzen des Global-Interrupt-Enable-Bits (GIE-Bit) im Control-Status-Register CSR.