



7. Assembler - Werkzeug zur maschinennahen Programmierung

- Überblick zum Keil-Assembler für den ARM-Cortex
- Assemblerdirektiven
- Phasen der Assemblierung
- Tipps und Tricks



7.1 Einführung

7.1.1 *Wie es begann*

Programmierung eines Microcomputers
in den Anfangsjahren:

1. Entwickeln des mnemonischen Programms (auf Papier).
2. Übersetzen der mnemonischen Befehle in die Maschinenbefehle (Binärmuster) von Hand (Binärcodierung).
3. Eingabe der Maschinenbefehle als Binär- oder Hexadezimalzahl .





7.1.2 Das Assemblerprogramm

Wie bereits gezeigt, erfolgt die Bestimmung der Maschinencodes aus der symbolischen Programmbeschreibung nach einfachen Regeln.

symb. Programmbeschreibung
→ **Text**

```
mov    r1, #1
mov    r2, #2
subs   r0, r1, r2
addpls r0, r0, #5
```

Assemblierung

codiertes Maschinenprogramm
→ **Binärdaten**

```
01 10 A0 E3
02 20 A0 E3
02 00 51 E0
05 00 90 52
```

Es liegt daher nahe, diesen Codierungsvorgang nicht von Hand durchzuführen, sondern diese Arbeit mit Hilfe eines Programms durchzuführen. → **Assembler**

Zusätzlich kann ein solches Programm weitere Erleichterungen ermöglichen:

- Bezeichner für Konstanten, Daten und Adressen
- Reservieren von Speicherbereichen für (Zwischen-)Ergebnisse
- Anlegen und Initialisieren von Daten und Datenstrukturen beim Programmstart



7.1.3 Begriffe: Assemblieren, Disassemblieren und Compilieren

Assemblersprache = symbolische, textorientierte Darstellung einer **Maschinsprache**.

- Sie ist strukturgleich zur Maschinsprache.
- Die Transformation "symbolische Darstellung → Maschinsprache" wird als **Assemblierung** bezeichnet.
- Aufgrund der Strukturäquivalenz ist auch mit einfachen Mitteln eine Rücktransformation des Maschinencodes in eine Assemblernotation möglich. Dies wird als **Disassemblierung** bezeichnet.

Die **Assemblierung** unterscheidet sich von einer **Compilierung** dadurch, dass eine Transformation zwischen strukturäquivalenten Notationen stattfindet und keine Übersetzung von einem Sprachniveau in ein anderes.

7.1.4 Crossassemblierung



Download

Übertragen des lauffähigen Programms vom Host- zum Target.



USB



Debugausgaben

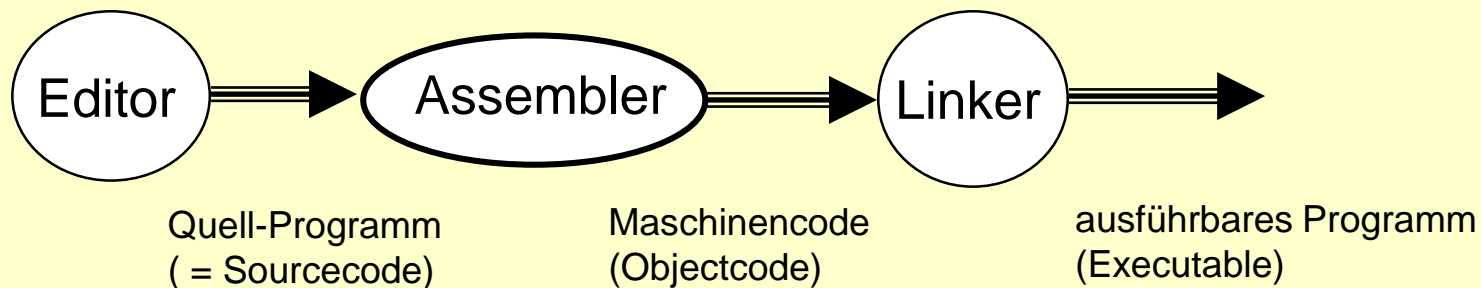


Target

Zielrechner, auf dem das Programm ausgeführt wird.

Host

Entwicklungsrechner, auf dem das Programm editiert, assembliert und gelinkt wird.





7.2 Assembler für ARM-Cortex—Prozessoren (Keil µVision)

7.2.1 Typische Fähigkeiten von Assemblersprachen

Mnemo = Text-Codierung der einzelnen Maschinenbefehle (*Mnemoniks*).

Beispiele:

```
mov  r0, #0x1A
ldr  r0, [r1, #4] !
```

Die **Operanden** können mit **symbolischen Bezeichnern** dargestellt werden. Die symbolischen Bezeichner stehen für die Operandenwerte, für Adressen von Speicherzellen bzw. für Prozessorregister.

Beispiele:

```
ldr    r0, =Messwerte
add    r1, [r0, #SpannungA]
```

Mit Hilfe von **Assemblerdirektiven** (Pseudobefehle)

- kann der Übersetzungsvorgang gesteuert werden,
- können Speicherbereiche für das Programm reserviert werden,
- können Datenstrukturen für das Programm angelegt und initialisiert werden.



7.2.2 Elemente der Assemblersprache

Assemblerdirektiven (*Pseudobefehle*) = Steueranweisungen an den Assembler
(werden zur Assemblierungszeit ausgeführt !!!!):

- **Äquivalenzdefinitionen** (EQU)
Hier werden Symbole eingeführt und mit konstanten Werten assoziiert
- **Speicher-Reservierungen** (COMMON, SPACE)
Hier werden Symbole eingeführt und mit einer Adresse assoziiert
- **Datendefinitionen** (DCB, DCW, DCD, SPACE)
Hier werden Symbole eingeführt, mit Adressen assoziiert und die adressierten Speicherzellen werden mit Datenwerten belegt
- **Befehlsersetzungen** (z.B. ldr r0, =Messwerte)
Es werden praktische, aber durch den Prozessor nicht unterstützte Befehle durch solche Befehle ersetzt, die der Prozessor unterstützt.
- **Definitionen von Makroanweisungen**
Definition von parametrisierten Textschablonen
- **Anweisungen zur bedingten Assemblierung**
Wahlweises Ausblenden von Anweisungsfolgen

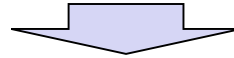


7.2.3 Schritte der Assemblierung

Die Arbeitsschritte eines Assemblers lassen folgendermaßen angeben:

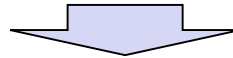
1. Makrogenerierung

Expandieren der Makro-Aufrufen durch ihre Textschablone und Einkopieren der aktuellen Parametersymbole



2. Aufbau der Symboltabelle

Eintragen aller definierten Symbole und ihre Wertentsprechungen in eine Symboltabelle (für Schritt 3)



3. Generierung des Maschinencodes

Ersetzen aller Befehls-Mnemoniks und der Operandenadressierungen durch ihre binären Codierungen



7.2.4 Befehlssyntax

Aufbau einer Zeile zur Formulierung eines Maschinenbefehls in BNF:

befehlszeile = *kommentarzeile* | *maschinenbefehl* | *direktive*

kommentarzeile = ; *Text*

maschinenbefehl = [*label*] *befehl* *operanden* [*kommentar*]

label = *buchstabe* { *buchstabe* | *ziffer* } :

befehl = *mnemonik*

operanden = *operand* [',' *operand*] [',' *operand*]

kommentar = ; *Text*



7.3 Symbole, Konstanten und Ausdrücke

7.3.1 Begriffsdefinitionen

Symbole = Bezeichner, die ganzzahlige numerischer Werte oder Zeichenketten symbolisieren. Symbolnamen werden vom Programmierer vergeben.

Beispiel: **WORDSIZE EQU 4**

Konstanten = Werte, die vom Assembler in binäre Maschinencodierungen umgewandelt werden (sind bereits zur Programmierzeit bekannt)

Beispiel: **TableSize EQU 0x100**

Ausdrücke = werden bei der Assemblierung ausgewertet und durch ihr Ergebnis ersetzt. Wegen der Auswertung zur Assemblierzeit dürfen in ihnen nur Symbole und Konstanten verwendet werden.

Als Operationen sind die 4 Grundrechnungsarten und Klammerausdrücke erlaubt.

Beispiel: **TableLength EQU WORDSIZE * 0x200**



7.3.2 Konstanten einen Namen geben

Mit der Direktive **EQU** können Konstanten mit einem Namen assoziiert werden.

Syntax: `symbol EQU (symbol | konstante | ausdruck)`

Zweck:

- Programme sind besser lesbar (keine „*magic numbers*“).
- Soll die Konstante einmal geändert werden, dann muss nur an einer Stelle die Änderung vorgenommen werden und nicht an vielen Stellen.

```
; → gut programmiert
; ---- Konstantendefinitionen ----
        SIZE EQU 100
        ....
; ---- Programm ----
        mov  r0, #SIZE
        ....
        add  r4, [r2, #SIZE]
```

```
; → schlecht programmiert !!!
; ---- Programm ----
        mov  r0, #100
        ....
        add  r4, [r2, #100]
```



Arbeitsweise: Die Namen (Symbole, Bezeichner) werden vor der Übersetzung in den Maschinencode durch ihre Zahlenwerte ersetzt.

EQU ist also eine Textverarbeitungsfunktion vor der Übersetzung (suche Name und ersetze durch Konstante).

Beispiele:

SIZE EQU 100	; SIZE = 100
NUM EQU 20	; NUM = 20
LEN EQU NUM * SIZE	; LEN = 2000
TABSIZE EQU SIZE	; TABSIZE = 100



7.4 Speicherreservierung

7.4.1 Reservierung von uninitialisierten Speicherblöcken

Mit der Direktive **COMMON** wird bei der Assemblierung ein zusammenhängender Speicherblock im Hauptspeicher (RAM) reserviert (*static allocation*).

COMMON Symbol, [size, [Alignment]]

Beispiel:

```
AREA MyCommonBlocks, COMMON, DATA ; ReadWrite-Data  
COMMON MyResultBlock, 80, 2
```

; *MyResultBlock* bezeichnet die Anfangsadresse des Speicherblocks
; reserviert 80 Bytes (Default 0)
; Halbwort Alignment (Default 4)

Die mit **COMMON** reservierten Speicherblöcke liegen im **COMMON-Speicherbereich** (folgt später).



7.4.2 Reservierung und Initialisierung von Speicher

Mit diesen Direktiven wird bei der Assemblierung Speicherplatz im Hauptspeicher reserviert und mit vorgebbaren Werten initialisiert.

[label]	DCB	(symb. konst. ausdr.) { , symb. konst. ausdr. }
[label]	DCW	(symb. konst. ausdr.) { , symb. konst. ausdr. }
[label]	DCD	(symb. konst. ausdr.) { , symb. konst. ausdr. }

Beispiel: Belegung mit Zeichenketten

	AREA MyData, DATA, align = 2
Wert1	DCB 0xfa, 12, 'A', 2_10001111
	; reserviert und initialisiert 4 Bytes
	; Wert1 bezeichnet die Anfangsadresse des Byte-Feldes
Text2	DCB "Fehler 15", 0
	; reserviert und initialisiert 10 Bytes (incl. 0-Terminator)
	; Text2 bezeichnet die Anfangsadresse des Strings

Die reservierten Speicherblöcke liegen im DATA-Segment (folgt später).



Wichtig: Label bezeichnen die Anfangsadresse des folgenden Datenblocks.

Im Programm benutzte Label werden vor der Übersetzung durch die konkreten Adressen ersetzt.

Label müssen linksbündig beginnen !!

Zweck:

- Programme sind besser lesbar (keine „*magic numbers*“).
- Die Adresse passt sich automatisch an und muss bei Veränderung nicht an mehreren Stellen des Programms manuell verändert werden.

Merke:

- DCB → Reserviere und Initialisiere Bytes
- DCW → Reserviere und Initialisiere **Halbworte** (=2 Byte) !!!!
- DCD → Reserviere und Initialisiere Worte (=4 Byte)



Wichtig: Alignment bei Byte- und Halbwortfeldern

- Zugriffe auf DCB-Felder müssen immer auf Wortgrenzen ausgerichtet (word-aligned) sein (Ausnahmen möglich, z.B. LDR, STR).
- Zugriffe auf DCW-Felder müssen immer auf Halbwortgrenzen ausgerichtet (hword-aligned) sein (Ausnahmen möglich, z.B. LDRH, STRH).

Bei Nichtbeachtung können völlig unerwartete Ergebnisse auftreten (keine Fehlermeldungen) !

Die korrekte Ausrichtung auf Wortgrenzen kann erzwungen werden, indem nach DCW- oder DCB-Feldern ein `ALIGN` ausgeführt wird.

Beispiel :

MyVar1	DCB	0x10, 22, 'A'	; Bytes
	ALIGN 2	;Ausrichtung auf Halbwortgrenze	
MyVar2	DCW	0x123f, 12000	; Halbworte
	ALIGN 4	; Ausrichtung auf Wortgrenze	
MyVar3	DCD	0x1234abcd	; Worte



Beispiel: Wie werden die Daten abgelegt ?

```

Var01          DCB      0xaa, 0xbb

                ALIGN 4

Var02          DCW      0x1234, 0x56ab
Var03          DCB      0xCD

                ALIGN 4

Var04          DCD      0x12345678
Var05          DCW      0x12ab, 0x22ff, 0xabcf
Var06          DCB      0x12, 0x23, 0x34
Text1          DCB      "ABC", 0, "012", 0
  
```

Address:



```

0x2000005C: AA BB 00 00 34 12 AB 56 CD 00 00 00 78 56 34 12
0x2000006C: AB 12 FF 22 CF AB 12 23 34 41 42 43 00 30 31 32
0x2000007C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  
```



7.5 Komfortable Befehlsersetzungen und Pseudobefehle

7.5.1 Laden von beliebigen 32-bit-Konstanten

Hintergrund: Der ARM/Cortex-Prozessor erlaubt nur sehr eingeschränkt die direkte (immediate) Angabe von Konstanten (s.o.).

Beispiel:

```
mov r0, #100           ; erlaubt  
mov r1, #0x1234ab      ; nicht erlaubt !!!!
```

Was kann man tun, um beliebige 32-bit-Konstanten in ein Register zu schreiben ?

Diese Folie nur zur Info



mögliche Lösung: (→ umständlich und anfällig)

1. Konstante im Nahbereich (-255 4095) des Aufrufs im Speicher ablegen.
2. Relativ zum Programmcounter PC (Adr.-art: „*immediate offset*“) darauf zugreifen.

Beispiel:

40 Byte tiefer

```

; ---- Programm ----
.....
ldr    r1, [PC, #32]      ; Die Konstante liegt 40 Byte (32+8)
.....                  ; unterhalb dieser Befehlszeile.
.....                  ; Angegeben wird immer:
.....                  ; → # Entfernung - 8

; ---- Programmende ----
DCD    0x1234A0          ; Konstante (32-Bit)
  
```

Anmerkung: Es erscheint zunächst merkwürdig, dass die Konstante 40 Byte entfernt ist, aber nur 32 angegeben werden muss.

Erklärung: Der PC ist zur Ausführungszeit (*execute-cycle*) wegen des Befehlspipelining bereits um 2 Befehle (=8 Byte) weiter.



bessere Lösung: Konstante unter einem Namen ablegen (→ umständlich)

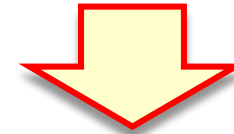
Beispiel:

```

; ---- Programm ----
ldr    r1, MyC           ; Name der Konstante angeben und ...
....
....
; ---- Programmende ----
MyC    DCD    0x1234A0    ; ... Konstante am Programmende ablegen

```

Der Assembler macht daraus **bei der Übersetzung** :



x Byte tiefer

```

┌ ldr    r1, [PC, #x-8]    ; Der Pseudobefehl wird durch einen
│ .....                  ; ARM-konformen Befehl ersetzt.
│ .....                  ; → immediate offset
└ A0 34 12 00 .. .. ..    ; am Programmende

```

Diese Folie nur zur Info



einfachste Lösung: Das explizite Ablegen der Konstante unter einem Namen kann entfallen, wenn der „=“–Pseudobefehl verwendet wird:

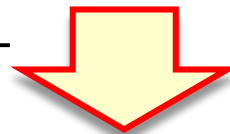
Beispiel:

; ---- Programm ----

.....
ldr r1, =0x1234A0

; oder noch viel einfacher

Der Assembler ersetzt den Pseudobefehl bei der Übersetzung in:



Beachte: Nicht mit **mov**, sondern mit **ldr**

.....

x Byte
tiefer

ldr r1, [PC, #x-8]

.....
.....

; Der Pseudobefehl wird durch einen
; ARM-konformen Befehl ersetzt.
; → *immediate offset*

A0 34 12 00 .. .

; Die Konstante wird vom Pseudobefehl
; während der Übersetzung autom. am
; Programmende abgelegt.



7.5.2 Zugriff auf initialisierte Daten (Variablen, Tabellen, Strings)

Prinzip:

1. Die Anfangsadresse der Daten wird wie eine Konstante geladen (Pseudobefehl).
2. Mit indirekter Adressierung (*immediate offset*) wird dann auf die Daten zugegriffen.

Beispiel:

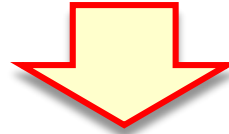
```

; ---- Initialisierte Daten ----
AREA    MyData, DATA, align = 2
MyTable DCD 1200, -1233, 0xffff3412, -150023 ; 32-Bit-Worte
;
; ---- Programm ----
;
; .....
ldr     r0, =MyTable      ; Adresse der Tabelle nach r0 laden
ldr     r1, [r0]           ; Zugriff auf 1. Element der Tabelle (1200)
ldr     r2, [r0, #4]       ; Zugriff auf 2. Element der Tabelle (-1233)
;
; .....
; ---- Programmende ----
```



Angenommen die Anfangsadresse der Daten (MyTable) liegt bei 0x40000000.

Der Assembler ersetzt
den Pseudobefehl **bei**
der Übersetzung durch :



Beispiel:

```

; ---- Initialisierte Daten ----
AREA    MyData, DATA, align = 2
MyTable DCD    1200, -1233, 0xffff3412, -150023 ; 32-Bit-Worte
;
; ---- Programm ----
;
; .....
ldr     r0, [PC, #x-8] ; Der Pseudobefehl wird durch einen
                       ; ARM-konformen Befehl ersetzt.
                       ; → immediate offset
x Byte  ldr     r1, [r0] ; Zugriff auf 1. Element der Tabelle (1200)
tiefer  ldr     r2, [r0, #4] ; Zugriff auf 2. Element der Tabelle (-1233)
;
; .....
; ---- Programmende ----
DCD     0x40000000 ; Die Anfangsadresse wird vom Pseudobefehl
                  ; am Programmende abgelegt.

```



7.5.3 Negative Konstanten

Hintergrund: Mit dem Befehl `mvn` wird die immediate angegebene Konstante bitweise negiert in das Register kopiert (= *Einerkomplement*).
Anm.: Für die Konstante gelten die gleichen Einschränkungen wie bei `mov` (0..255, und Linksverschiebungen um 0 ... 31)

Beispiel: ; mvn : bitweise negiert kopieren
`mvn r0, #0x01` ; [r0] ← FFFFFFFE (= -2)
`mvn r0, #2_11001100` ; [r0] ← FFFFFFF3
`mvn r0, #10` ; [r0] ← FFFFFFF5 (= -11)

Schreibt man (einfacher)

`mov r0, #-10` ; [r0] ← -10

So ersetzt der Assembler diesen Befehl durch (*Zweierkomplement*)

`mvn r0, #9` ; [r0] ← -10

Anm.: Einerkomplement = Zweierkomplement - 1



7.6 Phasen der Assemblierung

Assemblierung Pass 1:

- Bezeichner und Ausdrücke ersetzen
- Pseudobefehle ersetzen

```
; ----- Textersatz -----  
Start EQU 10  
Offs EQU 0x10  
  
AREA MyData, DATA, ...  
; ----- Speicherreserv. und Initialisierung -----  
MyDat DCB 0xf1, 0xf2, Start, Start+Offs  
MyStr DCB "ABC012", 0  
....  
AREA MyCode, CODE, readonly  
; ----- Start des Hauptprogramms -----  
main  
ldr r0, =MyDat  
ldrb r1, [r0]  
ldrb r2, [r0, #2]  
mov r3, #Start  
....  
; ----- Programmende -----
```



```
ldr r0, [PC, #0x0c]  
ldrb r1, [r0]  
ldrb r2, [r0, #2]  
mov r3, #0x0a  
....  
....  
DCD 0x40000000
```



Assemblierung Pass 2: Maschinencode erzeugen

```
0c 00 9f e5  
00 10 d0 e5  
02 20 d0 e5  
0a 30 a0 e3  
....  
00 00 00 40
```

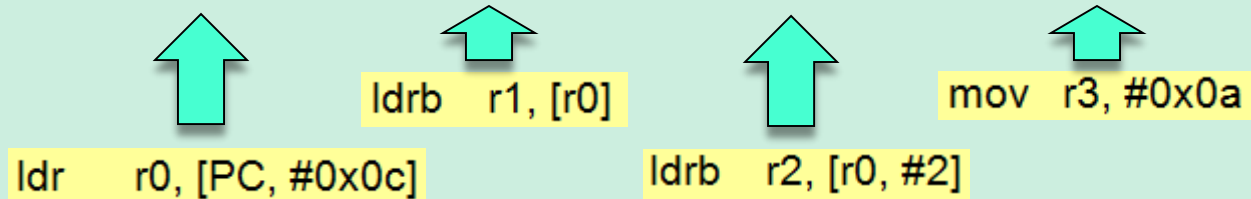


Beispiel: Was ist nach dem Download auf dem Zielrechner (ARM 7)?

Section „CODE“

Adresse Maschinencode

Address	Data
main	0C 00 9F E5 00 10 D0 E5 02 20 D0 E5 0A 30 A0 E3



Section „DATA“

Adresse Daten

Address	Data
0x40000000	F1 F2 0A 1A 41 42 43 30 31 32 00

MyStr DCB "ABC012", 0

MyDat DCB 0xf1, 0xf2, Start, Start+Offs

10

10 + 16



ÜBUNG: Assemblerdirektiven

Geben Sie die Symboltabelle und das Speicherbild (*Memorymap*) der folgenden Assemblersequenz an.

Das Datenfeld beginne bei Adresse 0x1000, der Datenblock beginne bei 0x2000.

```
; *** Konstanten ***
Val1      EQU      2
Val2      EQU      3
; *** Daten ***
        AREA      MyData, DATA, align=3 ; 2^3 = 8 Byte Alignment
Start
        DCD      2
        DCB      "AB 12",0
        ALIGN    4
        DCB      0, 1, 2
Zeit
        DCB      6, 10
        ALIGN    4
XCon
        DCD      Val1+Val2+Zeit-Start
XFeld
        DCD      Val1
; *** Datenblöcke ***
        AREA      MyBlocks, COMMON ; Default ist Wort-Aligned
        COMMON   Block 0x10
```



ÜBUNG: Assemblerprogramm mit Assemblerdirektiven

Geben Sie jeweils nach den Befehlen den Inhalt der Register an.
Der Datenblock beginnt bei Adresse 0x40000000.

Anm.: [r0] = [r1] = [r2] = 0.

```
; ***** Daten *****  
                AREA    MyData, DATA, align=3 ; 2^3 = 8 Byte Alignment  
in1             DCB     15  
in2             DCB     0x10  
                ALIGN    4  
pin1           DCD     in1  
; ***** Programm *****  
                AREA    MyCode, CODE,      readonly  
main            mov     r0, #10              ; 1  
                ldr      r1, =in1            ; 2  Adresse laden  
                ldrb     r2, [r1]            ; 3  Variable laden  
                add      r0, r0, r2          ; 4  
                ldr      r1, =in2            ; 5  Adresse laden  
                ldrb     r2, [r1]            ; 6  Variable laden  
                add      r0, r0, r2          ; 7  
                ldr      r1, =pin1           ; 8  
                ldr      r2, [r1]            ; 9  
                .....  
                .....
```



7.7 Tipps und Tricks

7.7.1 *Includieren von Quelldateien*

Mit dieser Direktive wird bei der Assemblierung das Einsetzen von Assembler-Quelltext aus einer anderen Datei veranlasst.

GET *Datei* *oder* **INCLUDE** *Datei*

Datei bezeichnet den Verzeichnispfad und den Dateiname der einzusetzenden Datei.

Beispiel:

```
GET    G:\LIB\SYSTEM.s
```

Auf diese Weise können z.B. Konstantendefinitionen (`EQU`) auch in mehreren Dateien bekannt gemacht werden.



7.7.2 Zugriff auf Variablen mit Hilfe einer Basisadresse

Durch Verwendung einer Basisadresse am Anfang eines Datenblocks muss das Adressregister nur einmal geladen werden.

Achtung: Die Daten dürfen nicht weiter als 4095 Byte von der Basisadresse entfernt liegen.

```
AREA    MyData, DATA, align=3

Base
Var1    DCD      123
Var2    DCD     8787
Var3    DCD    -34529
. . . .

AREA    MyCode, CODE,    readonly
ldr     r0, =Base          ; Basisadresse laden
ldr     r1, [r0, #Var2-Base] ; Var2 laden
str     r1, [r0, #Var3-Base]
```



7.7.3 Wahlfreier indizierter Zugriff auf Array- und Stringelemente

Es gibt es mehrere Möglichkeiten indiziert auf Tabellenelemente zuzugreifen:

a. Angabe der Adressdifferenz:

```
ldr    r0, =WortFeld    ; Arraystartadresse laden
ldr    r1, [r0, #4*2]    ; [r1] ← WortFeld[2]
```

b. Angabe der Adressdifferenz mit Hilfe eines Registers:

```
ldr    r0, =WortFeld    ; Arraystartadresse laden
mov     r1, #4*5          ; [r1] ← Wortgröße * Index
ldr     r2, [r0, r1]      ; [r2] ← WortFeld[5]
```

c. Angabe des Index über ein Register und die Wortgröße über einen Schiebewert :

```
ldr    r0, =WortFeld    ; Arraystartadresse laden
mov     r1, #6           ; [r1] ← Index
ldr     r2, [r0, r1, LSL #2] ; [r2] ← WortFeld[6]
```

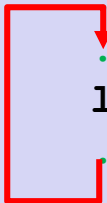
Vorteil von b./c.: Die Adresse kann während des Programmlaufs verändert werden.

Lösung c. ist am elegantesten, arbeitet aber nur für Elementgrößen 1,2,4,8,16,....



7.7.4 Abarbeitung von Feldern

Soll in einer Schleife Element für Element eines Arrays zugegriffen werden, dann bietet sich folgende Lösung an:



```
ldr    r0, =WortFeld    ; Arraystartadresse laden
      . . .
ldr    r1, [r0], #4      ; Wert n nach r1 laden
      . . .              ; Adresse in r0 um 4 erhöhen
      . . .              ; r0 „zeigt dann“ auf Wert n+1
```

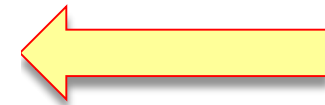



7.7.5 Sichtbar machen von Variablen im Debugger

Damit der Inhalt von Variablen im Debugger sichtbar wird, müssen die beobachteten Variablen global bekannt gemacht werden.

Beispiele

```
AREA MyData, DATA, align = 4  
GLOBAL MyData, in1, in2, pin1
```



```
in1      DCB      15  
in2      DCB      0x10  
         ALIGN    4  
pin1     DCD      in1 ; Zeiger auf in1
```

