Kapitel 15

Assembler-Programmierung

Entwicklungsmethode

Sehr ähnliche Richtlinien für Assembler- und HLL Programmentwicklung

- Klare Problembeschreibung

- **Entwurf (Design)**
  - Problemorientiert
  - Top-down Verfeinerung
  - Funktions- und Datenstrukturierung
  - Dokumentation, z.B.
    - Struktogramme
    - Flowcharts
    - Pseudocode
  - Fehlersituationen (relevant bei hardwarenahen Programmen)
  - Mit der Implementierung erst beginnen, wenn der Design wirklich fertig ist.
    - Die Korrektur eines Designfehlers wird umso teurer,
      je später sie durchgeführt wird.

- **Implementierung**
  - Abhängigkeiten zwischen Modulen minimieren
    - Mehrere Programmierer bei großen Projekten
    - Modultests werden einfacher
  - Konventionen für einheitliches Erscheinungsbild
    - Leichtere Einarbeitung in fremden Code
  - Dokumentation
    - Funktionsbeschreibungen
    - Beschreibung der Datenstrukturen
    - Input- und Output-Daten der einzelnen Module
    - aussagekräftige Kommentare im Code

- **Debug und Test**
  - Intensive Modultests für bestmögliche Codequalität
  - Testprozeduren für das Gesamtprojekt
    - Testfälle
    - Regression Tests
  - Effektives Versionmanagement

  ⇒ ***Auslieferung an Kunden***

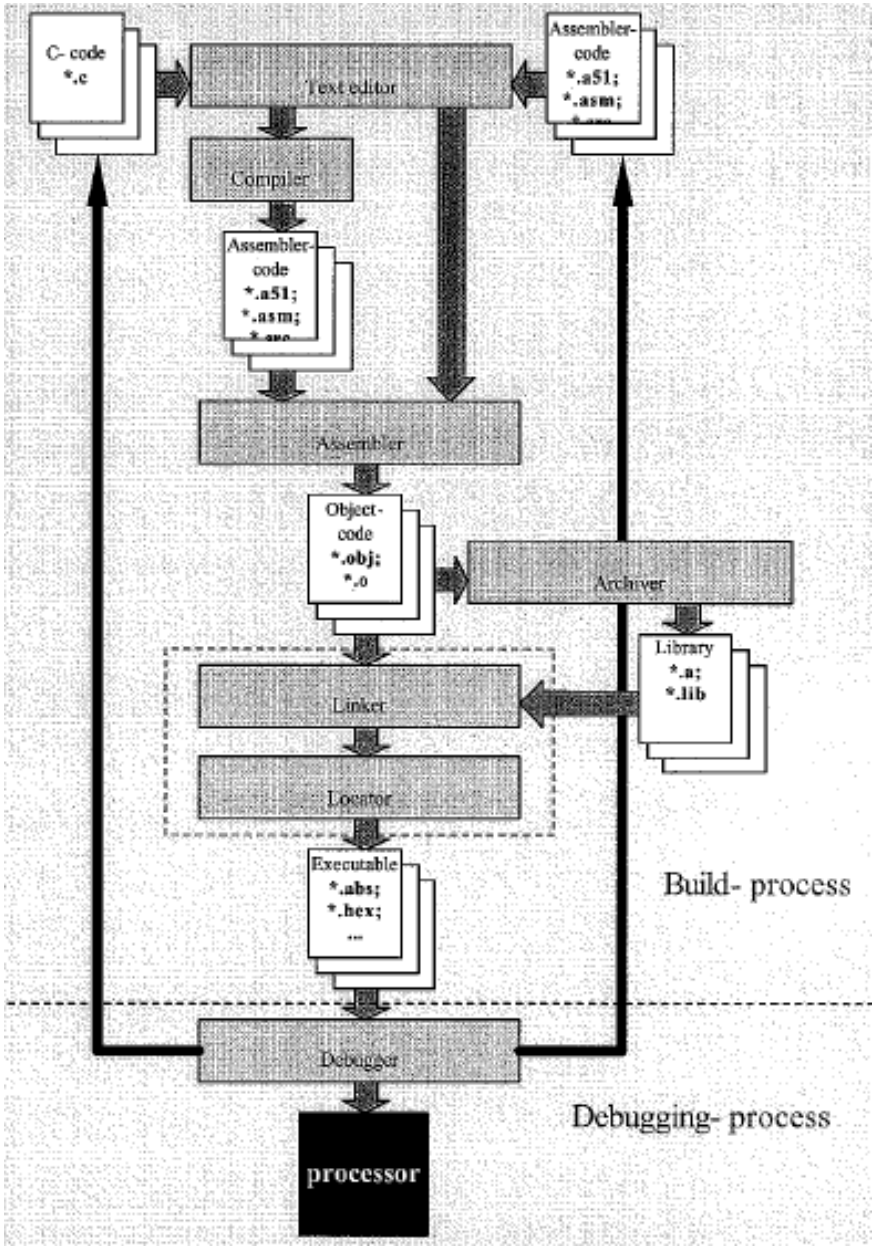
- **Programmwartung**
  - Fehlerbehebung
  - Programmanpassungen
  - Anderes Personal

Programmentwicklung

- Die manuelle Verwaltung der Entwicklung eines großen Programmsystems ist sehr aufwändig. Z.B.
  - müssen die Module von mehreren Programmierern kontrolliert zusammen geführt werden,
  - ist eine zeitliche Synchronisierung der Module erforderlich damit ein gemeinsamer Test durchgeführt werden kann.
- Deshalb hat man Hilfsmittel eingeführt, die erlauben,
  - dass einzelne Module eines Programmsystems unabhängig voneinander entwickelt werden können,
  - und dass die Konsolidierung dieser Module in einem Gesamtprogramm automatisch geschieht.

Vom Quellprogramm zum Maschinencode

- Mit einem beliebigen Text Editor werden die Module des **Quellcode**s erzeugt (C- und Assembler-Code).
- Assembler und Compiler übersetzen den Quellcode und erzeugen die **Objektcode** Module.
- Der Locator/Linker fügt alle Module zu einem **Maschinenprogramm** zusammen.
- In einem Archiv werden die einzelnen Versionen der Module verwaltet.



Skript Bild 61 : S87

Strukturierung von Assemblerprogrammen

Segmente

- Dass bei einem größeren Projekt mehrere Programmierer an einer Quellcode datei arbeiten, ist nicht praktikabel.
- Deshalb wurde in der 8051 Entwicklungsumgebung das Strukturierungselement **Segment** eingeführt.
- Ein Assemblerprogramm wird in mehrere Segmente aufgeteilt.
- Jedes Segment ist im Prinzip separat assemblierbar.
  - Der Assembler erzeugt aus jedem Segment ein Objektcode-Modul.
  - Der Linker setzt diesen Objektcode zusammen und erzeugt so das ausführbare Maschinenprogramm.
- Bei der Größe der Segmente gibt es ein Dilemma :
  - Zum einen möchte man die Segmente klein halten, um eine gute Anpassung an die Personalsituation zu erreichen.
  - Zum andern haben kleinere Segmente eine größere Anzahl von Schnittstellen zur Folge, die manuell zu behandeln sind
    - denn, es gibt mehr Sprünge und Unterprogrammaufrufe über Segmentgrenzen hinweg.
- Die Lösung : **Mehrere Segmente** können in eine Assembler-Quellcodedatei (z.B.: xyz.a51) gepackt werden.
  - Symbole in den Segmenten der selben Quellcodedatei werden als **global** betrachtet.
    - Sprünge und Unterprogrammaufrufe zwischen den Segmenten einer Datei bedürfen keiner besonderen Behandlung.
- Ein Segment beginnt mit dem Pseudobefehl, der das Segment spezifiziert (z.B.: CSEG ….) und erstreckt sich bis zum nächsten Segment-Pseudobefehl bzw. bis zum Dateiende.

Absolute Adressierung

- Für jedes Segment kann eine Startadresse angegeben werde, an der es im endgültigen Maschinenprogramm beginnen soll.
  - Syntax : **CSEG AT 0x1000** ;Programmsegment, das bei Adresse 0x1000 beginnt.
- Ein solches Segment wird **Absolutes Segment** genannt.
- Da der Assembler die Adressen der Befehle kennt, kann er Referenzen (z.B. die Adresse eines Sprungziels) korrekt einsetzen.
- Der Linker braucht dann den Objektcode dieser absoluten Segmente nur noch zum fertigen Maschinenprogramm zusammen zu setzen.
- Sollte sich beim Assemblieren heraus stellen, dass ein Segment länger wurde als erwartet, kann es zu einer Überschneidung mit dem nachfolgenden Segment kommen.
  - Der Linker wird dann eine solche Überschneidung als **Overlay** melden.
  - Overlays können durchaus beabsichtigt sein, wenn z.B. zu unterschiedlichen Zeiten jeweils andere Routinen in den selben Adressbereich geladen werden.
  - Eine unbeabsichtigte Überschneidung muss durch Anpassen der Startadressen behoben werden.
- Die Schwäche von absoluten Segmenten ist, dass man die Startadressen manuell behandeln muss.
  - Wählt man den Platz für die Segmente zu groß, verschenkt man Speicherplatz.
  - Wählt man den Platz für die Segmente zu knapp, müssen häufiger manuell Änderungen der Startadressen vorgenommen werden.
- Deshalb sollten absolute Segmente nur dort verwendet werden, wo es zwingend notwendig ist.

Relokative Adressierung

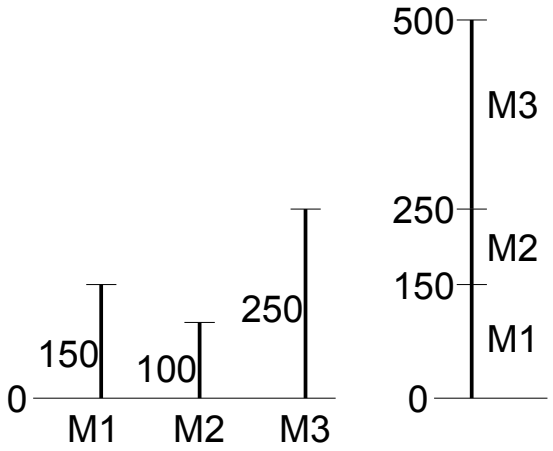
- Die Größe eines Segments stellt sich beim Assemblieren heraus.
- Der Linker kennt also die Segmentgrößen, und kann die Segmente dicht hinter einander packen.
- Ein so behandeltes Segment wird **Relokatives Segment** genannt.
- Syntax :

```
MyCode1   SEGMENT   CODE      ;Definition eine Programmsegments mit Namen MyCode1
          RSEG      MyCode1   ;Anfang des relokativen Programmsegments MyCode1
```

- Es wird keine Anfangsadresse spezifiziert.
  - Der Assembler verwendet für alle relokativen Segmente die Anfangsadresse 0.
  - Referenzen (z.B. ein Sprungziel) wären also nur korrekt, wenn das Segment bei Adresse 0 platziert wird.
- Da der Linker relokative Segmente an verschiedenen Anfangsadressen platziert, müssen die Referenzen während des Linkvorgang entsprechend angepasst werden. Diese Operation nennt man **Relokation**.
- Es ist durchaus möglich, absolute Segmente und relokative Segmente zusammen in ein Maschinenprogramm zu packen.
  - In diesem Fall wird der Linker zuerst alle absoluten Segmente platzieren und anschließend die relokativen Segmente in die Lücken füllen bzw. am Ende anfügen.
  - Alle Programme besitzen mindestens ein absolutes Programmsegment : dieses beginnt bei Adresse 0 und enthält den/die Befehl/e, die nach dem Einschalten bzw. nach Reset ausgeführt werden.
  - Bei nicht-trivialen Programmen ist es gute Praxis, nur den notwendigen Code in absolute Segmente, den Löwenanteil des Codes in relokative Segmente zu packen.

Relokation von Code

- Der Assembler übersetzt jedes relokative Segment des Quellcodes mit einer Anfangsadresse von '0'.
  (Beispiel : M1, M2, M3 im linken Bild)
- Damit das Programm ausgeführt werden kann, müssen aber alle Objektmodule gleichzeitig im Programmspeicher stehen.
- Der Linker verschiebt deshalb die Module so, dass sie hintereinander im Programmspeicher stehen. Der Linker fügt sie zu einem gemeinsamen Programm zusammen. (Beispiel : M1, M2, M3 im rechten Bild)
  - Das Verschieben nennt man **Relokation**.



- Für sequentiell ausgeführte Befehle ist diese Verschiebung unproblematisch.
  - Sie werden weiterhin sequentiell, allerdings an anderen Adressen ausgeführt.
- Probleme kann es mit Sprungbefehlen geben.
  - Eine absolute Zieladresse wird beim Assemblieren als Abstand zum Modulanfang bei '0' angegeben.
  - Durch die Relokation liegt der Modulanfang nicht mehr bei '0'.
    (Beispiel : Der Anfang von M2 liegt nach dem Linken bei Adresse 150)
  - D.h. alle **absoluten Zieladressen** von Sprungbefehlen müssen angepasst werden.
    - Die Zieladressen werden um den Wert der Anfangsadresse des Moduls erhöht (z.B.: in M2 um 150).
  - Relative Zieladressen geben den relativen Abstand zum Sprungbefehl an.
    - Der Wert des Operanden eines relativen 8051 Sprungbefehls (signed char) wird zum Augenblicklichen Wert des Befehlszählers addiert (subtrahiert bei negativem Wert).
    - Da die Zieladresse in gleichem Maße wie der Sprungbefehl verschoben wird, ist keine Anpassung nötig.

Strukturierung der Datenspeicher

- Wie der bisher betrachtete Programmspeicher werden auch die **Datenspeicher** mit Hilfe von **Segmenten** strukturiert.
- Ein Byte im internen 8051 Datenspeicher kann im Assemblercode auf mehrere Arten angesprochen werden.
  1. Die Adresse kann direkt als Zahl angegeben werden, z.B.:

```
MOV   0x33,#0x01
```

  2. Die Adresse kann als Symbol spezifiziert werden, dem per EQU eine Zahl zugewiesen wird, z.B.:

```
Adresse1   EQU    0x33
           MOV    Adresse1,#0x01
```

  3. Für ein Symbol wird im Datenspeicher ein Byte reserviert, z.B.:

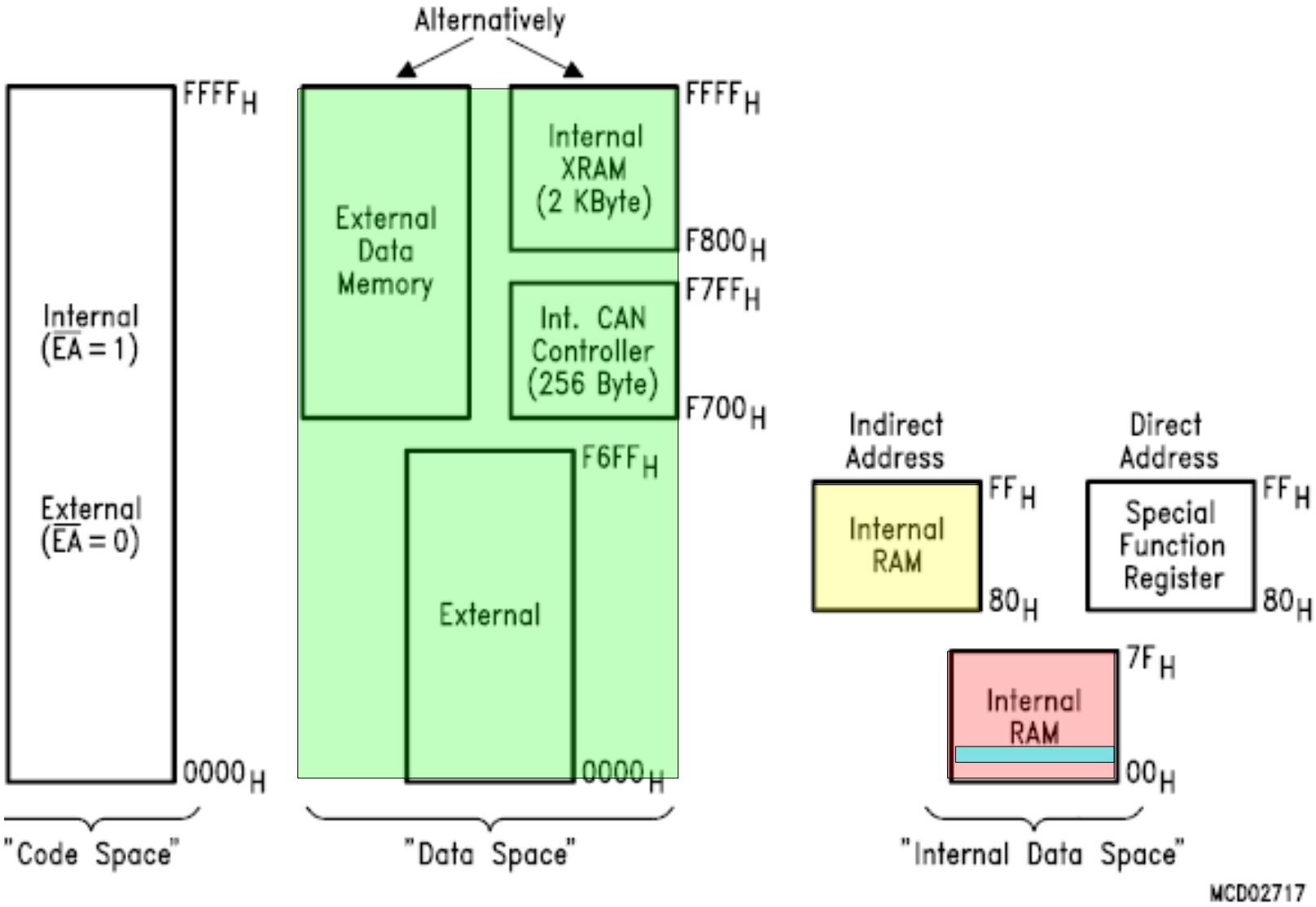
```
Adresse2   DS     1
           :
           MOV    Adresse2,#0x01
```

- In den Fällen 1. und 2. setzt der Assembler die absolute Adresse (0x33) in den Maschinenbefehl ein. Der Maschinenbefehl selbst befindet sich in einem Programmsegment.
  - Der EQU Pseudobefehl kann dabei an einer beliebigen Stelle im Quellcode stehen.
  - Segmente für den Datenspeicher sind bei dieser Adressierungsmethode nicht nötig.
- Die Platzreservierung wie im Fall 3. muss in einem Datenspeicher-Segment stehen, z.B.:

```
            DSEG   AT   0x40      ;Datensegment im direkt adressierbaren
                                   ;internen 8051 Datenspeicher
Adresse2:   DS     1               ;reserviert 1 Byte
```

Strukturierung der Datenspeicher Forts.)

- Auch für die Datenspeicherspezifikationen gibt es absolute Segmente und relokative Segmente.



- Syntax für einen absoluten und einen relokativen Segment-Pseudobefehl für den direkt adressierbaren internen Datenspeicher :

absolutes Segment			relokatives Segment		
	DSEG	AT 0x40	MyData1	SEGMENT	DATA
Byte1:	DS	1		RSEG	MyData1
			Byte2:	DS	1

- In einem **8051** System gibt es **mehrere Datenspeichertypen**. Für jeden Datenspeichertyp können sowohl absolute als auch relokative Segmente spezifiziert werden :
  - Der direkt adressierbare interne Datenspeicher (Adresse 0x00 bis 0x7F)
    - absolutes Segment : DSEG AT Startadresse
    - relokatives Segment : SegmentName SEGMENT DATA
  - Der nur indirekt adressierbare interne Datenspeicher (Adresse 0x80 bis 0xFF)
    - absolutes Segment : ISEG AT Startadresse
    - relokatives Segment : SegmentName SEGMENT IDATA
  - Der externe Datenspeicher (XRAM)
    - absolutes Segment : XSEG AT Startadresse
    - relokatives Segment : SegmentName SEGMENT XDATA
- Die Symbole im SFR Bereich sind in der Entwicklungsumgebung standardmäßig über EQU Pseudobefehle bereits definiert und können direkt in Assemblerbefehlen verwendet werden. Z.B.: MOV A,PSW

Bitsegmente

- Ein weiterer 8051 Datenspeichertyp, den man mit Segmenten strukturieren kann, ist der Bitspeicherbereich, der im internen Datenspeicher bei den Adressen 0x20 bis 0x2F angesiedelt ist.
  - Die Bitadressen dieses Bereichs sind 0x00 bis 0x7F (16 Bytes * 8 Bits = 128 Bits)
- Syntax für einen absoluten und einen relokativen Segment-Pseudobefehl für den Bitspeicherbereich :

absolutes Segment

```
        BSEG    AT    0x61
Bit11:  DBIT    1
```

relokatives Segment

```
MyBits12    SEGMENT    BIT
            RSEG       MyBits12
Bit1bis4:   DBIT       4
```

  - Die Spezifikation im DBIT Pseudobefehl bezeichnet die Anzahl der zu reservierenden **Bits**.
  - Die Adressangabe im BSEG Pseudobefehl (0x61) und die Werte der Symbole (Bit11, Bit1bis4) sind Bitadressen.
- Dem EQU Pseudobefehl, der ein Symbol für ein Byte festlegt, entspricht der **BIT** Pseudobefehl für die Zuweisung von Symbolen zu Bits.
  - Z.B.:

```
Byte1    EQU    0x33      ;weist Byte1 den konstanten Wert (Adresse ?) 0x33 zu

Bit7     BIT    46        ;weist Bit7 die Bitadresse 46 zu
Bit03    BIT    0x20.3    ;Bitadresse ist Bit3 in Byte 20, Bit23 ist also gleich Bitadresse 3
```

- Die direkt adressierbaren Bits im SFR Bereich sind in der Entwicklungsumgebung bereits spezifiziert und können direkt in Bitbefehlen verwendet werden.
  - Z.B.: `JNB **CY**,SprungZiel ;Springe, wenn Carry Flag (= Bit CY) = 0`

Relokation von Daten

- Wie die oben diskutierte Relokation von Codemodulen ist in gleicher Weise eine Relokation für Datenspeicherspezifikationen erforderlich.
  - Zu jedem Codemodul gibt es üblicherweise Variablen, die über Symbole angesprochen werden.
    - Für jedes Variablensymbol werden mit dem **Pseudobefehl DS** ein oder mehrere Bytes im Adressraum des Datenspeichers reserviert.
    - Mit DBIT werden Bits im Bitbereich reserviert.
    - D.h. jedem Variablensymbol wird eine eigene Datenspeicheradresse zugeordnet.
    - Für ein Variablensymbol in einem Assemblerbefehl fügt der Assembler die entsprechende Adresse in den Maschinenbefehl ein.
  - Die Speicherbereiche für Variablen in den relokativen Segmenten werden ebenfalls ab Adresse '0' angelegt.
  - Im Locate/Linkvorgang müssen diese Speicherbereiche ebenfalls hinter einander angeordnet werden.
    - Dadurch ändern sich Speicheradressen von Variablen.
    - In allen Befehlen müssen die Referenzen zu diesen Adressen angepasst werden.

Relokation beim Programmladen

- Eine weiterer Grund für Relokation wurde beim Thema Sprungbefehle schon erwähnt :
  - Beim Laden eines Programms stimmt die beim Linken festgelegte Anfangsadresse des Programms nicht mit der Ladeadresse überein.
  - Es wird beim Programmladen dynamisch eine Relokation durchgeführt.

Typische Verwendung von absoluten Segmenten

- **Nach RESET** führt der Prozessor den ersten Befehl an Adresse 0x0000 im Programmspeicher aus.
  - Ein absolutes Codesegment bei Adresse 0 (CSEG AT 0) erlaubt die Spezifikation der ersten Befehle.
- **Interrupt Service Routinen** beginnen an festen Adressen.
- Beispiel :

```
CSEG    AT   0x0000    ;absolutes Codesegment bei Adresse 0
LJMP    InitCode       ;zum Initialisierungscode (vorzugsweise in einem relokativen Segment)
CSEG    AT   0x0013    ;absolutes Codesegment bei Adresse 0x0013
                       ;0x0013 ist die verdrahtete Startadresse im Fall von Timer 0 Interrupts
LJMP    Timer0ISR      ;zur Timer 0 Interrupt Service Routine
```

- Alternative :

```
CSEG    AT   0x0000    ;absolutes Codesegment bei Adresse 0
LJMP    InitCode       ;zum Initialisierungscode
ORG     0x0013         ;der nächste Befehl wird an Adresse 0x0013 platziert
LJMP    Timer0ISR      ;zur Timer 0 Interrupt Service Routine
```

  - Der ORG Pseudobefehl blockiert den dazwischen liegenden Speicherbereich für den Linker/Locator.
  - Der ORG Befehl ist in absoluten und relokativen Segmenten einsetzbar.
  - Mit dem ORG Befehl kann man das Ende eines Segments festlegen und den Platz darin blockieren.

```
z.B.:    ORG    0x4000    ;blockiert den Platz bis einschließlich 0x3FFF
```

- Ein absolutes Datensegment (z.B. XSEG AT 0x6000) macht Sinn, wenn ein externer Speicherbaustein installiert ist, der ab einer bestimmten Adresse ansprechbar ist.

Inhalte eines Codesegments

- In einem (absoluten oder relokativen) **Codesegment** befinden sich :
  - Assemblerbefehle (z.B. MOV, ADD, ….)
  - Definitionen von Byte-Konstanten
    - Pseudobefehl : ***Konstantenname: DB Byte-Spezifikation(en)***
      - Beispiele :

```
Byte1:    DB   34                  ; ein Byte mit dem Wert 34 = 0x22
W3By:     DB   0x00, 0x01, 0x02    ; drei Bytes
String1:  DB   'Dies ist ein Text', 0   ; nullterminierter String, 18 Bytes
```

  - Definitionen von Wort-Konstanten - (Wort = 2 Bytes = 16 Bits)
    - Pseudobefehl : ***Konstantenname: DW Wort-Spezifikation(en)***
      - Beispiele :

```
Worte2:   DW   2000, 0x1234
Strg1Adr: DW   String1          ;Programmspeicheradresse von String1 (s.o.)
```

  - Beispiel für ein Codesegment

```
          CSEG    AT   0        ;absolutes Codesegment bei Adresse 0
          LJMP    InitCode     ;Sprung zum Initialisierungscode
Worte2:   DW      2000, 0x1234
          DB      'Nachricht 1',0   ;sog. 0-terminierter String
InitCode:
          CLR     A
          …...
```

Inhalte eines Datensegments

- In einem (absoluten oder relokativen) **Datensegment** für den internen und externen Datenspeicher (DATA, IDATA und XDATA) befinden sich Deklarationen von Variablen
  - Pseudobefehl : ***Variablenname: DS Anzahl Bytes***
    - Beispiele :

```
                    ……..        ; Annahme : Speicher reserviert bis Adresse 21
LoopCnt:    DS  1   ; ein Byte bei Adresse 22
Buffer:     DS  10  ; 10 Bytes ab Adresse 23
                    ; dem Symbol Buffer ist die Adresse 23 zugeordnet
```

Inhalte eines Bitsegments

- In einem (absoluten oder relokativen) **Bitsegment** befinden sich Deklarationen von Bitvariablen.
  - Pseudobefehl : ***Variablenname: DBIT Anzahl Bits***
    - Beispiele :

```
                    ……..        ; Annahme : Speicher reserviert bis zur Bitadresse 21
OnOff:      DBIT    1   ; ein Bit bei Bitadresse 22
FiveBits:   DBIT    5   ; 5 Bits bei Bitadressen 23 bis 27
```

Konstanten und Adressen

- Zuweisung eines konstanten Werts : Pseudobefehl **EQU**

	Symbol	**EQU**	**konstanter Wert**
z.B.:	Anzahl	EQU	37

  - Erzeugt keinen Maschinenbefehl, sondern lediglich einen Eintrag in der Symboltabelle.
  - Beim Assemblieren wird das Symbol durch den im EQU Pseudobefehl spezifiziert Wert ersetzt.

- Zuweisung einer **konstanten Adresse** : Pseudobefehle DATA, IDATA, XDATA, und BIT
  - weisen einem Symbol einen absoluten Adresswert im entsprechenden Adressraum zu.

	Symbol	**Adresszuweisung**	**konstanter Wert**	
z.B.:	LoopCnt	DATA	0x12	
	Liste	XDATA	0x3000	
	Bit12	BIT	0x0C	
	Switch	BIT	0x25.1	;Bit 1 in Byte 0x25

- Sowohl der EQU- als auch die Adresszuweisungs-Pseudobefehle können an beliebigen Stellen in der Quelldatei stehen. Denn sie sind vollkommen unabhängig von den Spezifikationen des Segments, in dem sie ggf. stehen.

Schlüsselwörter

Segmenttyp	**absolut**	**relokativ**	**Speicherreserv.**	**Adresszuweisung**
Programmspeicher	CSEG AT	SEGMENT CODE	DB, DW (*)	CODE
Direkt adressierbarer Datenspeicher	DSEG AT	SEGMENT DATA	DS	DATA
Indirekt adressierbarer Datenspeicher	ISEG AT	SEGMENT IDATA	DS	IDATA
Externer Datenspeicher	XSEG AT	SEGMENT XDATA	DS	XDATA
Bitspeicher	BSEG AT	SEGMENT BIT	DBIT	BIT

(*) : DB und DW reservieren Speicherplatz und definieren den Inhalt.

Modulübergreifende Referenzen in Assemblercode

- In einer Quellcodedatei können Segmente verschiedener Typen vorkommen.
- In einer Quellcodedatei können auch mehrere Segmente des selben Typs vorkommen.
- Ein Segment beginnt mit einem der Segmentbefehle RSEG, CSEG, DSEG, ISEG, XSEG oder BSEG.
- Ein Segment endet mit dem nächsten Segmentbefehl bzw. dem Ende der Quelldatei.
- Alle Symbole in einer Quellcodedatei sind in allen Segmenten der Quellcodedatei direkt ansprechbar.
  - Eine Speicherreferenz in einem Codesegment zu einer Variablen in einem Datensegment in der selben Quellcodedatei ist gültig – globale Referenz.
  - Ein Sprung von einem Codesegment zu einem Symbol in einem anderen Codesegment der selben Quellcodedatei ist ebenfalls gültig.
- Eine Referenz zu einem Symbol in einer anderen Quellcodedatei muss explizit spezifiziert werden :
  - In der Quellcodedatei, in der das Symbol definiert ist, muss das Symbol **veröffentlicht** werden mit dem Pseudobefehl

```
                PUBLIC    Symbol
        z.B.    PUBLIC    Wert1, Zaehler, Summe      ;mehrere Symbole sind zulässig
```

  - In der Quellcodedatei, in der eine Referenz des Symbols verwendet wird, muss angegeben werden, dass das Symbol in einer anderen Quellcodedatei definiert ist. Dazu dient der Pseudobefehl :

```
                EXTRN     Segmenttyp (Symbol)
        z.B.    EXTRN     CODE(InitCode)          ;Symbol 'InitCode' in einem Codesegment
                EXTRN     DATA(Summe)             ;Symbol 'Summe' in einem Segment des direkt
                                                  ;adressierbaren internen Datenspeichers
```

- Als Schlüsselworte für den Segmenttyp werden die Pseudocodes für die Adresszuweisung verwendet.

Programmstruktur des 8051 Laborprojekts

Interrupt Vektor Code

```
0000    CSEG    AT    0x000
0000    LJMP    Init
0003    LJMP    ExtInt0ISR
0006

000B    CSEG    AT    0x000B
000B    LJMP    Timer0ISR
000E

0013    CSEG    AT    0x0013
0013    LJMP    ExtInt1ISR
0016

0053    CSEG    AT    0x0053
0053    LJMP    ExtInt3ISR
0056
        :
        :
        :
        :
```

Datenspeichersegmente

```
InitCode   SEGMENT   CODE
           RSEG      InitCode

Init:                          LJMP  Main
```

```
MainCode   SEGMENT   CODE
           RSEG      MainCode

                Endlosschleife

Main:                          LJMP  Main
```

```
ISRCode    SEGMENT   CODE
           RSEG      ISRCode

ExtInt0ISR:                    RETI

Timer0ISR:                     RETI
        :
        :
        :
        :
```

8051 Assemblerprogrammierung

Im MCT Labor Projekt im nächsten Semester sollen Sie mit Hilfe der Entwicklungsumgebung der Firma KEIL ein Programm entwickeln, das dann auf einem 8051 Prozessor läuft. Aus diesem Grund wählen wir diese Entwicklungsumgebung für die detailierte Diskussion zum Thema Assemblerprogrammierung.

KEIL Entwicklungs-Umgebung

- Um ein neues Programm zu entwickeln, muss in der KEIL Entwicklungs-Umgebung (KEIL EU) zunächst ein **Projekt** angelegt werden.
  - Dabei erzeugt die KEIL EU in einem vom Benutzer festgelegten Ordner, dem **Projektordner**, einige PC-Dateien.
    - Diese Dateien enthalten die Verwaltungsdaten für die ganze Entwicklungszeit des Programms.
- Die eigentlichen Codespezifikationen speichern die Programmierer in anderen PC-Dateien, den **Quellcode**-Dateien, ab.
  - Assembler-Quellcode in Dateien mit dem Dateityp .a51.
  - C-Quellcode in Dateien mit dem Dateityp .c
- Es wird empfohlen, die Dateien mit den Codespezifikationen ebenfalls im Projektordner zu halten.
- Die KEIL EU hält in einer Projektdatei eine Liste aller zum Programm gehörenden Quellcodedateien.
  - Der Programmierer muss alle zu verwendenden Quellcodedateien explizit spezifizieren (Funktion : Add File to Source Group)
- Beim Assemblieren und Linken erzeugt die KEIL EU eine ganze Reihe weiterer Dateien im Projektordner, z.B. :
  - Objektdateien (.obj),
  - Listingdateien (.lst).
- Nachdem ein Programm erfolgreich übersetzt (assembliert und gelinkt) wurde, kann es direkt im **Simulator**, der auch in der KEIL EU verfügbar ist, debugged werden.

Übungsaufgabe AP1 :

1. Legen Sie ein Segment für das XRAM an, und reservieren Sie an der absoluten Adresse 0x4000 einen 10 Byte langen Puffer mit dem Namen 'Buffer1' !

2. Legen Sie ein relokatives Codesegment an, in dem der Befehl MOV DPTR,#Buffer1 vorkommt ('Buffer1' soll den Puffer in Aufgabe 1 ansprechen) !

3. Machen Sie den konstanten Text 'Dies ist Text' als null-terminierten String über das Symbol Text1 ansprechbar !

4. Was ändert sich, wenn man annimmt, dass das Segment für das XRAM in Aufgabe 1 und das Codesegment in Aufgabe 2 in unterschiedlichen Quelldateien liegen ?

Assemblerdirektiven

- Assemblerdirektiven sind Anweisungen, die der Assembler im Assembliervorgang berücksichtigen soll.
- Beispiele :

```
$TT('Titelzeile')          ;'Titelzeile' wird auf jeder Seite der LIST-Datei eingefügt.

$NOLIST
    :
    :                      ;diese Zeilen werden in der LIST-Datei unterdrückt
    :
$LIST

$NOMOD51                   ;deaktiviert die eingebauten Symboldefinitionen für SFRs

$INCLUDE (reg515c.inc)     ;fügt die Zeilen der Include-Datei in den Modultext ein.
                           ;reg515c.inc enthält Symboldefinitionen speziell für den 80C515C Prozessortyp

END                        ;identifiziert das Ende der Spezifikationen in einer Quelldatei

.....                      ;weitere Assemblerdirektiven siehe KEIL Hilfeseiten
```

Programmumwandlung

- Assemblier-Vorgang (Module werden separat assembliert bzw. compiliert).
  - Input : Assembler Quelldatei (xxx.a51)
  - Output 1 : Objektdatei (xxx.obj)
    - Maschinencode (Binärcode)
    - Symboltabelle der verschiebbaren Adressen (z.B.Sprungziele, Datenspeicheradressen)
    - PUBLIC und EXTRN Spezifikationen
  - Output 2 : List-Datei (xxx.lst)
    - Quell- und Objektcode
    - Crossreferenz (optional)
- Locate/Link-Vorgang
  - Relokation : Die Segmente jedes Segmenttyps werden im entsprechenden Adressraum hinter einander platziert.
  - Input : alle Objektdateien
  - Output 1 : Objektdatei für das Programm (Projektname (ohne Dateityp))
    - Maschinencode (Binärcode)
    - Symboltabelle der verschiebbaren Adressen (nicht für 8051 Programme)
  - Output 2 : Link List Datei (Projektname.m51)
    - Anfangsadressen und Längen der Segmente

Testverfahren

Hardwaretests

- **Prozessortests** :
  Auf Grund des hohen Integrationsgrades heutiger Technologie (über eine Milliarde Schaltkreise pro Chip) sind nur noch wenige Signale für Messungen erreichbar.
  - Simulation
    - Hardwareimplementierung muss auf ein detailiertes Modell abgebildet werden
    - Genaue Abbildung des Zeitverhaltens wichtig.
    - Simulation ganzer Programmroutinen wünschenswert
      - Problem : Expansionsfaktor
  - Evolutionäre Entwicklung
    - Wiedervervendung funktionierender Chipfunktionen
- Peripherietests
  - Hilfsmittel : Oszilloskop, Logic Analyzer

Softwaretests

- **Monitor** (z.B. KEIL Laborkoffer)
  - Zweck : Debug jeglichen Codes, auch Fehlersuche in Produktionssystemen.
  - zusätzliche Hardwarefunktionen und zusätzlichen Monitorcode im ROM
  - Interface für das Testpersonal (z.B. Tastatur/Display, PC Anschluss)
  - Testen auf Maschinencode-Basis; keine symbolischen Adressen
  - Testfunktionen
    - Anzeige/Änderung von Speicherinhalten
    - Programmausführung starten/stoppen
    - Einzelschritt
    - Programmausführung stoppen bei Erreichen einer Vergleichsadresse
    - Unterbrechungspunkte (sog. **Break Points**) setzen (nur für Code in einem RAM)

- **Debugger** (z.B. DEBUG.EXE Programm unter DOS)
  - Zweck : Debug von neuen Programmen
  - reine Softwarelösung
  - erfordert eine komplette Infrastruktur (Hardware + Betriebssystem)
  - Funktionen ähnlich wie Monitor
  - Mehr Komfort beim Testen
    - disassemblierter Code
    - symbolische Adressen

- **Simulator** (z.B. KEIL Simulator)
  - Zweck : Debug von neuen Programmen
  - Testen findet nicht auf der Zielhardware statt
    - Zielhardware ist im Simulatorprogramm modelliert
    - Testen auf einem mächtigen System führt zu einem erträglichen Expansionsfaktor
    - Testprogramme sind quasi Daten für das Modellprogramm
    - Modellierung von Peripherie mehr oder weniger rudimentär
  - Komfortabelste Testumgebung
  - kein Echtzeitverhalten

- **Emulator**
  - Zweck : Debug und Test von Hardware und/oder Software
  - Anstelle des Prozessors wird eine Emulatorhardware an das System angeschlossen.
  - Die Emulatorhardware bietet Zugriff auf viele Signale, die im regulären Prozessorbaustein nicht erreichbar sind.
  - Varianten von Emulatorhardware
    - universelle Emulatoren, in denen durch entsprechende Konfiguration unterschiedliche Zielbausteine modelliert werden können.
    - spezielle Emulatoren, die genau einen Zielbaustein ersetzen.
  - zum Teil sehr mächtige Testunterstützung
    - Traces
    - erlaubt die Simulation von Hardwarefehlern
    - ...

Lösung zu Übungsaufgabe

Übungsaufgabe AP1 :

```
1.              XSEG    0x4000          ;XRAM Segment bei Adresse 0x4000
        Buffer1:DS      10              ;reserviere 10 Bytes

2.      MyCode1 SEGMENT     CODE
                RSEG    MyCode1
                …....
                MOV     DPTR,#Buffer1   ;Lade Adresse in DPTR
                …....
3.      Text1:  DB      'Dies ist Text',0   ;Konstante Daten im Code-Segment

4.
                PUBLIC  Buffer1
                XSEG  AT  0x4000        ;XRAM Segment bei Adresse 0x4000
        Buffer1:DS      10              ;reserviere 10 Bytes
        ------------------------------------------------------------------------------------------
                EXTRN   XDATA(Buffer1)
        MyCode1 SEGMENT     CODE
                RSEG    MyCode1
                …....
                MOV     DPTR,#Buffer1   ;Lade Adresse in DPTR
```