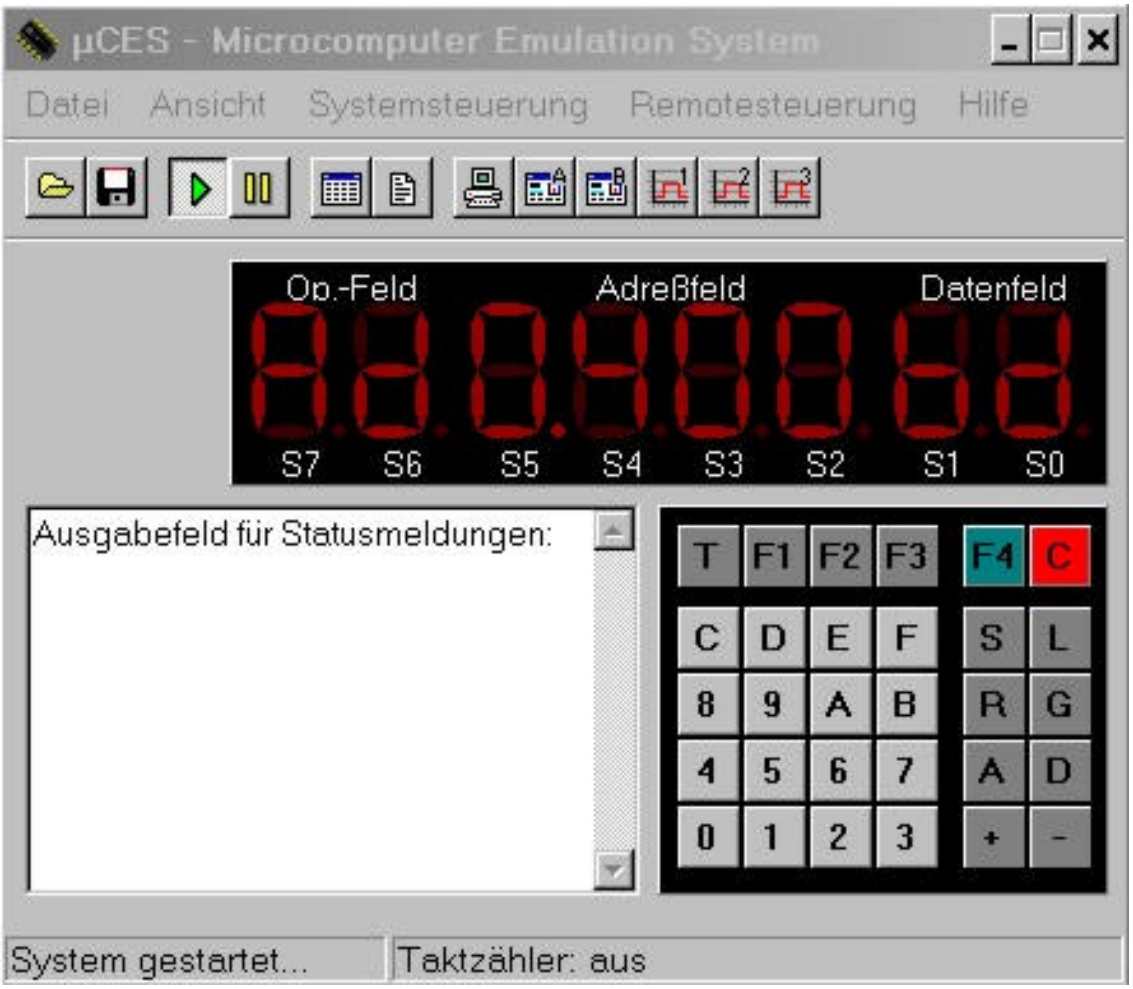
Skript zum Mikrorechner-Praktikum

Kapitel 1:

Bedienung des Praktikumsrechners und Programmierung des Prozessors

Autor: Helmut Bähring



Inhalt von Kapitel 1:

1 Ein universeller Mikroprozessor: Der Motorola MC6809

1.1 Einleitung

Der verwendete Praktikumsrechner gehört zur Klasse der "Einplatinen-Computer", die mit einer Minimalausrüstung an Komponenten versehen sind und dem Benutzer einfache Hilfsmittel zur Programmerstellung in Maschinencode bieten. Sie verfügen meist über eine Hexadezimaltastatur, eine mehrstellige 7-Segmentanzeige und ein Interface zum Anschluß an einen PC. Sie werden wegen ihrer Einfachheit und der Möglichkeit, mit ihnen Anwendungssoftware zu entwickeln, als "Entwicklungskits" bezeichnet. In der Vergangenheit war ihre Bedeutung mit der Verbreitung universeller oder spezieller Entwicklungssysteme zeitweise gesunken. Diese Systeme verfügen über eine ASCII-Tastatur, über einen Bildschirm, über Floppy-Disk- oder Harddisk-Speicher und eine große Palette von Software-Entwicklungshilfsmittel. Dazu gehören in der einfachen Ausführung (Macro-Cross-)Assembler, Linker, Lader und Editoren, sowie auch Compiler für höhere Programmiersprachen, hauptsächlich C. Viele von ihnen sind heute als PC-gestützte Geräte realisiert. Durch den Einsatz von **Simulatoren** können Programme für den Zielprozessor auf diesen Systemen ausge-testet werden. Durch den Einsatz von (Hardware-)**Emulatoren** ist es möglich, diese Programme im Zielsystem, aber durch den (bzw. einen) Prozessor des Entwicklungs-systems ausführen zu lassen. In den letzten Jahren haben die Entwicklungskits wie-der eine größere Bedeutung gewonnen, da ihr platzsparender und modularisierter Aufbau auch den kostengünstigen Einsatz in Kleinserien zuläßt.

Da den Entwurf des Praktikumsrechners galten die folgenden Hauptforderungen:

- Einsatz eines modernen, leistungsfähigen 8-bit-Prozessors, Hexadezimaltastatur und 7-Segmentanzeige,
- leichte Bedienung über einen einfachen Monitor im Festwertspeicher (EPROM), (mit Monitor wird in der Regel ein rudimentäres Betriebssystem zur Bedienung der Rechnerkomponenten bezeichnet,)
- V24-Schnittstelle zum Anschluß eines Terminals oder anderer Peripheriegeräte (Drucker etc.), dazu Terminalprogramm im Monitor,
- ein Interface zum Anschluß und zur Benutzung eines IBM-kompatiblen PCs,
- Ausbaufähigkeit des Systems auf maximal 64-kbyte-Arbeitsspeicher auf der Platine, d.h. ohne die Benutzung einer Zusatzplatine,
- gepufferter Systembus als Erweiterungsschnittstelle für den Anschluß weiterer, externer Systemeinheiten,
- 8 bit breiter Eingabe/Ausgabe-"Port" zur Verbindung mit Schaltungen auf einem Experimentierboard.
- Benutzung von Standard-Peripheriebausteinen, wie PIO (Parallel-Eingabe/Ausgabe-Baustein), UART (universeller, asynchroner, serieller Sender/Empfänger-Baustein), Timer (Zeitgeber/Zähler-Baustein).

Da in der kurzen Zeit, die für die Durchführung eines Praktikums zur Verfügung steht, einer der modernen, universellen 16/32-bit-Prozessoren auch nicht annähernd

erschöpfend behandelt werden kann, verbot sich von vorneherein deren Einsatz. Aus diesem Grund fiel die Wahl auf den Typ Motorola MC6809. Er war wohl einer der leistungsfähigste Vertreter der 8/16-bit-Klasse. Ihn zeichnen vor allem die folgenden Punkte aus, die im Abschnitt 1.3 ausführlich dargestellt werden:

- 1464 verschiedene Instruktionen,
- 10 "mächtige" Adressierungsarten, (das ist der vollständigste Satz unter allen 8-bit-Prozessoren,)
- 16-bit-Arithmetik,
- Befehl zur vorzeichenlosen 8x8-bit-Multiplikation,
- Programmzähler-relative Adressierung, wodurch eine "verschiebliche" Programmierung des Prozessors unterstützt wird, d.h. ein geschriebenes Programm kann ohne Änderung an verschiedenen Stellen im Speicher laufen *(position independent)*.
- Ansprechen von Eingabe/Ausgabe-Komponenten wie gewöhnliche Speicherzellen *(memory-mapped-I/O)*, deshalb keine besonderen Ein/Ausgabe-Befehle,
- softwaregesteuerte Unterbrechungsmöglichkeiten.

Durch die genannten Punkte ist der Prozessor sowohl für die Ausführung von Programmen, die in einer höheren Programmiersprache geschrieben wurden, wie auch für Steuerungsaufgaben bestens geeignet.

Ziel der Kapitel 1 bis 3 ist es, Sie mit der Bedienung des Praktikumsrechners vertraut zu machen und Ihnen seine Komponenten im Detail zu erklären. Dazu gehören der Prozessor selbst, das Bussystem, der Speicher sowie die Tastatur und die Anzeigeeinheit. In den Kapiteln 4 und 5 folgt dann die Beschreibung der etwas spezielleren Komponenten, des Portbausteins, des seriellen Schnittstellen-Bausteins sowie des Zeitgeber-/Zählerbausteins.

1.2 Bedienung des Praktikumsrechners

1.2.1 Funktion der Tasten

In diesem Abschnitt sollen Sie zunächst mit der Bedienung des Praktikumsrechners vertraut gemacht werden[1]. Grundlage für die Benutzung des Rechners ist die im Festwertspeicher abgelegte Systemsoftware, der sog. Monitor. Er ist insbesondere für die Ansteuerung von Tastatur und Anzeige verantwortlich. Wir werden Ihnen im folgenden die Funktion der einzelnen Tasten und der Anzeige erklären und danach die Hilfsroutinen des Monitors beschreiben, die Sie in den Selbsttest- und Einsendeaufgaben zur Lösung komplexerer Probleme einsetzen sollten. Der Aufbau des Monitors wird in Kapitel 3 detailliert dargestellt.

dargestellt, in drei Felder unterteilt.Die **Anzeige** (*Display*) des Mikrorechners besteht aus acht 7-Segmentanzeigen mit je einem Dezimalpunkt. Sie sind, wie in Bild 1.2-1 dargestellt, in drei Felder unterteilt.

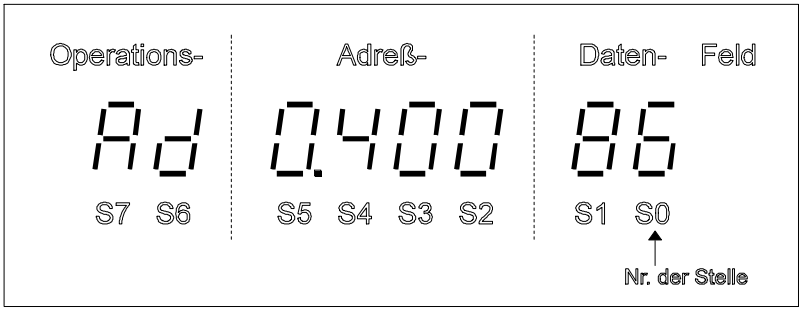


Bild 1.2-1: Die 8-stellige 7-Segmentanzeige

Das linke Feld (S7,S6) dient (bis auf wenige Ausnahmen) zur Kennzeichnung (KZ) der gerade ausgeführten Operation und wird im folgenden deshalb mit **Operationsfeld** bezeichnet.

Das mittlere Feld stellt meistens einen hexadezimalen Zahlenwert dar. Das ist in der Regel eine 4-stellige Adresse. Deshalb wird dieses Feld im folgenden vereinfacht **Adreßfeld** genannt. Zur Darstellung des Inhalts eines 8-bit-Registers werden von den vier Stellen S2-S5 dieses Feldes lediglich die beiden linken Stellen S4 und S5 benutzt.

Das rechte Feld (S1,S0) kennzeichnet innerhalb einer ausgeführten Operation die einzugebenden Parameter bzw. Register und stellt für eine angewählte Speicherzelle deren Inhalt in hexadezimaler Form dar. Es wird abkürzend mit **Datenfeld** bezeichnet.

---

[1] Es mag Ihnen merkwürdig erscheinen, daß das bereits an dieser Stelle geschieht und nicht erst nach der vollständigen Beschreibung des Systems, die in den nächsten Abschnitten folgt. Der Grund liegt darin, daß Sie so in die Lage versetzt werden, während der folgenden Beschreibung des Mikroprozessors und seiner Komponenten schon einfache Befehlsfolgen und Programme einzugeben, die die Funktion und Benutzung der einzelnen Baugruppen verdeutlichen.

Schließen Sie nun zunächst den Praktikumsrechner ans Netz an und schalten Sie ihn mit dem Netzschalter auf der Rückseite ein. Wenn das System betriebsbereit ist, wird es automatisch in einen Grundzustand versetzt (*Power on Reset*). Dabei werden insbesondere die Tabellen für die Systemsoftware, dem rudimentären Betriebssystem (Monitor), erzeugt. Anschließend erscheint in der Anzeige beispielsweise die in Bild 1.2-2 gezeigte Information.

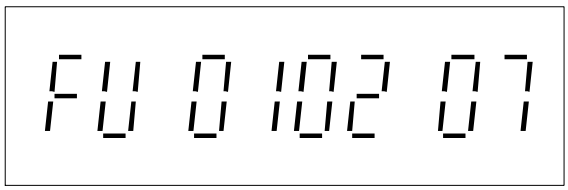


Bild 1.2-2: Die Anzeige nach dem Einschalten

Darin ist:

FU	die Abkürzung für die "FernUniversität",
0102	die Nummer des Ihnen vorliegenden Mikrorechners,
07	die Versionsnummer der im Rechner implementierten Betriebssoftware.

zeigt die Tastatur und die Einteilung der Tasten in verschiedene Bild 1.2-3 zeigt die Tastatur und die Einteilung der Tasten in verschiedene Grup-pen.

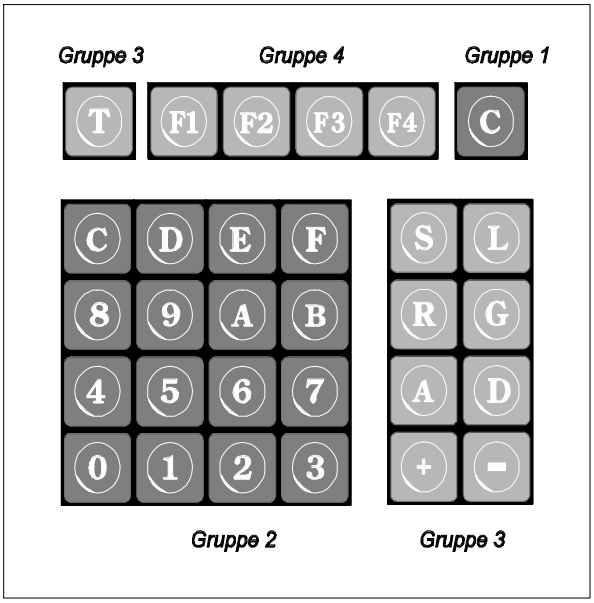


Bild 1.2-3: Die Tastatur und die verschiedenen Tastengruppen

Im folgenden werden nun die einzelnen Tastenfunktionen ausführlich erklärt. Wenn es nicht extra anders gesagt wird, stellen alle benutzten absoluten Zahlenwerte Beispiele dar[2].

[2] Die Notation $xx bzw. $xxxx bezeichnet dabei 8- bzw. 16-bit-Hexadezimalwerte.

Ein leuchtender Dezimalpunkt hinter einer Ziffer bildet den Zeiger (*Cursor*) auf die Position, an der das nächste über die Tastengruppe 2 (0,..,9,A,..,F) eingegebene Zeichen dargestellt wird. Dieser Cursor läuft im Adreßfeld mit jedem eingegebenen Zeichen um eine Position nach rechts, bis er nach der Anzeigenstelle S2 wieder in der Stelle S5 erscheint ("zyklische Verschiebung"). Im Datenfeld ändert er bei jedem eingegebenen Hexadezimalzeichen seine Position innerhalb des Feldes, was ebenfalls als zyklische Verschiebung interpretiert werden kann. Ein Überschreiben falsch eingegebener Werte ist durch nochmaliges Eingeben der Zahl möglich[3].

Praktische Übung P1.2-1:

Führen Sie alle im folgenden angegebenen Tastenbetätigungen und Beispiele mit Ihrem Praktikumsrechner textbegleitend selbst durch, um sich mit der Bedienung des Gerätes vertraut zu machen! Da die *Zero-Page*, also die Speicherzellen mit den Adressen $0000-$00FF, für die Systemvariablen des Monitors benutzt werden, sollten Sie diese Adressen nicht verwenden !

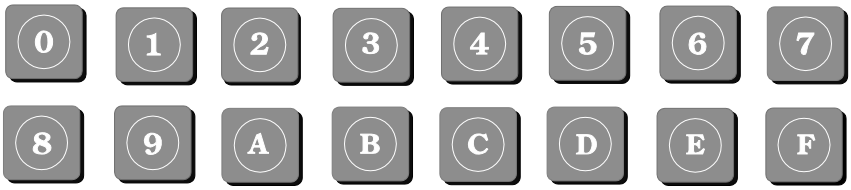
Die 1. Tastengruppe

Die 1. Tastengruppe besteht lediglich aus der `Taste C` („*Clear*“).

Mit ihr kann das System jederzeit in den Grundzustand versetzt werden. Dies ist besonders dann wichtig, wenn sich der Rechner „aufgehängt“ hat, d.h. auf andere Weise nicht mehr beeinflußt werden kann. Dabei bleiben alle eingegebenen Programme im Arbeitsspeicher erhalten. Verändert werden jedoch die internen Prozessorregister und die Systemvariablen im Speicher. Danach erzeugt das System die im Bild 1.2-2 gezeigte Meldung.

Die 2. Tastengruppe

Die 2. Tastengruppe besteht aus den 16 schwarzen Tasten



Sie dienen zur Eingabe von Adressen, Daten und Maschinenbefehlen in hexadezimaler Form, wobei stets mit dem höchstwertigen Zeichen begonnen wird.

Die 3. Tastengruppe

Die 3. Tastengruppe enthält die 9 grauen Tasten

[3] Betätigung der Funktionstaste A bzw. D bringt den Cursor zur Neueingabe an die jeweils linke Position.

Mit diesen werden fest vorgegebene Funktionen aufgerufen. (Die Tasten A und D dürfen nicht mit den Datentasten gleicher Bezeichnung verwechselt werden!)

(„*Address*", Kennzeichen in der Anzeige - KZ: Ad)
Mit dieser Taste wird der Rechner in den Modus **Adreßeingabe** umgeschaltet. In der Anzeige erscheint die Information:

In diesem Modus ist es möglich, die im Adreßfeld dargestellte Adresse zu ändern. Der Cursor zeigt dabei jeweils auf die Position, an der das näch-ste Zeichen einzugeben ist. Zu einer vollständig eingegebenen Adresse wird die zugehörende Speicherstelle angewählt und das darin enthaltene Datum im Datenfeld dargestellt.

("*Data*", KZ: dA)
Mit ihr wird das System in den Modus **Dateneingabe** umgeschaltet.

In diesem Modus ist es möglich, Daten oder Programme über die Tastatur in den Schreib/Lese-Speicher oder Daten in die Register der Eingabe/ Ausgabe-Bausteine des Systems einzugeben. In nicht belegte Adreßbereiche oder in EPROM-Bereiche ist eine Eingabe natürlich nicht möglich. Das eingegebene Datum wird in diejenige Speicherstelle übertragen, deren Adresse gerade im Adreßfeld der Anzeige steht. Die Übernahme in die Speicherzelle geschieht dabei erst, wenn beide Tetraden des Datums eingegeben wurden[4].

("*Increment*", kein KZ)
In beiden eben beschriebenen Modi (Adreß- bzw. Dateneingabe) wird durch diese Taste die dargestellte Adresse um 1 erhöht und der Inhalt der neu angewählten Speicherzelle im Datenfeld angezeigt.
Somit ist es möglich, durch mehrfaches Betätigen dieser Taste einen zusammenhängenden Speicherbereich zu betrachten. Dabei wird der Cursor auf die erste Position des jeweils angewählten Anzeigefeldes verschoben, also

- im Adreßeingabe-Modus auf die Stelle S5,
- im Dateneingabe-Modus auf die Stelle S1.

[4] Der Begriff Tetrade bezeichnet im folgenden eine 4-bit-Hexadezimalziffer.

("*Decrement*", kein KZ)
In beiden eben beschriebenen Modi wird durch diese Taste die dargestellte Adresse um 1 erniedrigt. Der Cursor wird wie bei der `Taste +` verändert.

("*Register*", KZ: rE)
Durch das Betätigen dieser Taste kann der Benutzer die Inhalte der internen Prozessorregister betrachten, wie sie bei der letzten Programmunterbrechung im Arbeitsspeicher abgelegt wurden (vgl. `Taste T` und `Taste F4`).
Durch die `Taste +` kann das nächste Register, durch die `Taste -` das vorhergehende in einer fest vorgegebenen Reihenfolge angezeigt werden. Der Inhalt der Register wird im Adreßfeld dargestellt, und zwar in den Stellen

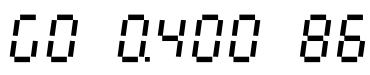
- S2-S5 bei 16-bit-Registern,
- S4,S5 bei 8-bit-Registern.

Das Datenfeld enthält jeweils die Bezeichnung des gerade angewählten Registers, wie Sie sie im Abschnitt 1.3 kennenlernen werden.

rE 0.0 A	Akkumulator A
rE 0.0 b	Akkumulator B
rE 0.000 X	Index Register X
rE 0.000 y	Index Register Y
rE 0.000 U	User Stack Pointer U
rE 2.000 SP	System Stack Pointer S
rE 0.400 Pc	Program Counter PC
rE 8.0 dP	Direct Page Register DP
rE 0.0 cc	Condition Code Register CC

Alle "Registerinhalte" können beliebig geändert werden. Dabei ist natürlich zu beachten, daß nicht die internen Register des Mikroprozessors selbst, sondern ihr "Bild" im Arbeitsspeicher geändert wird. Wird jedoch das Programm an der Unterbrechungsstelle fortgesetzt, so wird dieses Bild in die internen Register geladen und dann mit seinen Werten weiter gerechnet (vgl. die `Tasten T` und `F4`).

("*Go*", KZ: Go)
Diese Taste dient zur Vorbereitung eines Programmstarts. In der Anzeige erscheint zunächst:

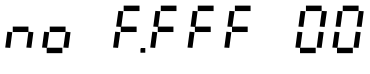


Die Startadresse $0400 ist für Benutzerprogramme vorgegeben. Beachten Sie bitte, daß die folgenden Funktionen Trace und Break nur mit dieser Startadresse fehlerfrei arbeiten.
Im Adreßfeld steht die Startadresse des Programms, wie sie zuletzt eingegeben wurde. Sie kann nun beliebig abgeändert werden. Im Datenfeld steht der Inhalt (Daten) der so adressierten Speicherstelle.
Die `Taste +` ermöglicht es, im Programm einen Unterbrechungspunkt (*Breakpoint*) zu setzen. Ihre Betätigung führt zu der Anzeige:

Vorgegeben ist die Adresse $0000, die anzeigt, daß kein Breakpoint gesetzt wird. Sie muß auch immer dann angegeben werden, wenn ein vorher gesetzter Breakpoint wieder gelöscht werden soll.
Sobald im laufenden Programm die eingegebene Breakpoint-Adresse angesprochen wird, wird die Ausführung des Programms unterbrochen und der Befehl SWI1 (s. Abschnitt 1.3.5) ausgeführt. Ist z.B. die Breakpoint-Adresse $0450 gewählt worden so erscheint in der Anzeige die Information:

(Breakpoint in $0450)

Nun ist es möglich, sich bestimmte Registerinhalte (`Taste R`) oder Speicherinhalte (`Taste A`) anzuschauen und danach das Programm an der Unterbrechungsstelle fortzusetzen (`Taste G` oder `Taste T`). Das Fortsetzen des Programmes funktioniert jedoch nur dann fehlerfrei, wenn die Programmstartadresse $0400 gewählt wurde. Breakpoint-Adressen müssen immer auf den Opcode, d.h. auf das 1. Byte eines Maschinenbefehls weisen. Sie können grundsätzlich nur im RAM-Bereich gesetzt werden. Ist ein Setzen des Breakpoints (BP) an einer bestimmten Adresse nicht möglich (z.B. im ROM-Bereich oder nicht belegten Speicherbereich), so erscheint in der Anzeige:



(kein BP möglich)

Eine Fortsetzung ist dann über die `Taste +` möglich, wonach ein neuer Breakpoint eingegeben werden kann, oder über die `Tasten G` und `T` zum Starten des Programms.

Starten des Programms

Nach der Eingabe der Programm-Startadresse und eventuell eines Breakpoints wird das Programm durch erneutes Drücken der `Taste G` gestartet.

Kurze Erklärung zur Wirkung des Breakpoints

Durch die Angabe der Breakpoint-Adresse wird der Inhalt der angesprochenen Speicherzelle in einer anderen Zelle des Schreib/Lese-Speichers abgelegt und danach durch den Befehl SWI1 (Opcode 3F, Software Interrupt 1, s. Abschnitt 1.3) ersetzt. Sobald das laufende Programm diese Adresse anspricht, wird in eine Interruptroutine verzweigt (deren Startadresse im RAM unter den Adressen $0044 und $0045 abgelegt ist). In dieser Routine wird zunächst der alte Wert der "Breakpoint-Speicherzelle" rekonstruiert. Danach werden die Inhalte der internen Register im RAM abgelegt, und der Monitor wird wieder aktiviert. Aus dem Gesagten ergibt sich, daß die Rekonstruktion der Breakpoint-Speicherzelle "von Hand" geschehen muß, wenn die entsprechende Adresse im Programmlauf nie aufgerufen wird und deshalb das Programm durch die Interruptroutine nicht unterbrochen wird. Außerdem muß bei jedem Programmstart die Breakpoint-Adresse gegebenenfalls erneut gesetzt werden.

("*Trace*", KZ: `Tr`)

(Die Startadresse des Programmes sollte $0400 sein.) Nach dem Drük-ken dieser Taste wird jeweils genau ein Befehl des Programms ausge-führt (Einzelschrittausführung, *Single Step Execution*).
Danach können die aktuellen Inhalte der internen Register betrachtet und gegebenenfalls verändert werden (`Taste R`). Ein erneutes Drücken der `Taste T` veranlaßt danach die Ausführung des nächsten Maschinenbefehls. Drücken der `Taste G` veranlaßt die Fortsetzung des Programms ab dem nächsten Maschinenbefehl. Die Darstellung im Display sieht folgendermaßen aus:

In der Anzeige erscheint die Adresse des Opcodes des nächsten auszuführenden Maschinenbefehles sowie dieser selbst, nicht jedoch die zugehörenden Operanden. Nach Sprung- und Verzweigungsbefehlen erscheint die Adresse des Befehles, zu dem gesprungen wurde. Vor der nächsten Ausführung des Trace-Befehles kann eine neue Adresse eingegeben oder die dargestellte durch die `Tasten +` und `-` geändert werden. Auf diese Weise können bestimmte Programmteile angewählt oder übersprungen werden, was besonders bei Schleifen mit vielen Durchläufen nützlich ist. Die Trace-Funktion kann auch auf Programme im ROM angewendet werden, indem die Speicherzelle $0091 auf den Wert $00 gesetzt wird. (Achtung: Sie wird nach dem Rücksetzen durch die Taste C wieder auf den Default-Wert $80 zurückgesetzt.)

("*Save*", KZ: SA)
Diese Taste dient zur Übertragung eines Programmes oder eines Datenbereiches - beides wird im folgenden kurz **Datensatz** genannt - aus dem Speicher des Praktikumsrechners zu einem über die serielle Schnittstelle angeschlossenen PC[5]. Der übertragene Datensatz wird im PC auf der Festplatte oder Diskette abgespeichert. Nach Betätigen der Taste S erscheint zunächst in der Anzeige:

SA ≙ Startadresse

Die Startadresse kann nun beliebig geändert werden. Sie gibt die Adresse des ersten abzuspeichernden Bytes im Arbeitsspeicher des Praktikumsrechners an und wird zusammen mit dem Datensatz in der Datei abgelegt. Nach Eingabe der Startadresse führt die Taste + zur Anzeige:

EA ≙ Endadresse

und erlaubt dadurch die Eingabe der Datensatz-Endadresse. Voreingestellt ist die Adresse $04FF, also eine Datensatzlänge von 256 Bytes.
Durch den nächsten Druck der Taste S wird die eigentliche Übertragung gestartet. Am Ende der Übertragung wird die Initialisierungsroutine des Praktikumsrechners aufgerufen und erzeugt die im Bild 1.2-2 gezeigte Systemmeldung. Während der Übertragung auftretende Fehler werden nicht festgestellt und gemeldet.
Sollte die Übertragung nicht erfolgreich beendet werden, müssen Sie nach entsprechender Wartezeit die Taste C ("*Clear*") drücken und ggf. den Übertragungsvorgang wiederholen.[6]

("*Load*", KZ: Lo)
Mit dieser Taste wird ein Datensatz (Programm oder Datenbereich) vom PC in den Arbeitsspeicher des Praktikumsrechners geladen. Die erste Betätigung der Taste führt zur Anzeige:

of ≙ Offset

Hier können Sie nun einen Lade-Offset angeben, der beim Einlagern der Daten zur aufgezeichneten Startadresse (s. Taste S) addiert wird. Dadurch ist eine Verschiebung der geladenen Daten gegenüber den abgespeicherten Daten im Arbeitsspeicher möglich. (Achtung: Bei absoluten Adressen in einem Programm kann das natürlich zu Fehlern führen.)
Nach dem nächsten Betätigen der Taste L erlöscht die Anzeige und der Praktikumsrechner wartet auf die Daten vom PC. Während der Über-

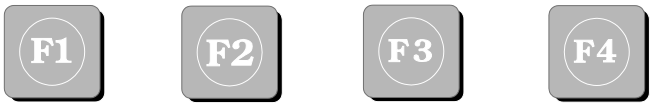
[5] Das dafür benötigte Terminalprogramm im PC wird in Kapitel 2 beschrieben.
[6] In Kapitel 2 finden Sie eine Beschreibung der technischen Grundlagen der Verbindung zwischen Praktikumsrechner und PC und der Bedienung des mitgelieferten PC-Programms. Die serielle V.24-Schnittstelle wird in Kapitel 4 genau beschrieben.

tragung wird das aktuelle empfangene Datum zur Kontrolle in den beiden rechten Anzeigestellen S1,S0 ausgegeben.
Nach erfolgreicher Beendigung des Ladevorgangs erscheint die Meldung[7]:

Lo EndE FF (Beendigung des Ladens)

Die 4. Tastengruppe

Die 4. Tastengruppe wird aus den speziellen Funktionstasten



gebildet. Die ersten drei werden beim Initialisieren des Systems mit bestimmten Funktionen vorbelegt. Jedoch kann der Benutzer jederzeit dafür sorgen, daß durch sie andere, selbstdefinierte Funktionen aufgerufen werden.

("*Insert*", KZ: In)
Diese Taste ist mit der "Einfüge-Funktion" vorbelegt. Diese erlaubt das Einfügen eines Bytes in eine bestimmte Stelle eines zusammenhängenden Datenbereiches.
Vor dem Aufruf der Insert-Funktion muß im Adreßeingabe-Modus (s. `Taste A`) die um 1 verminderte Adresse angewählt werden, in die das Byte eingeschoben werden soll. Nach dem Druck der `Taste F1` werden zunächst die nächsten 255 Bytes im Speicher um eine Zelle nach „oben“, also zu höheren Adressen hin, verschoben. Achtung: Das 256. Byte geht bei dieser Operation verloren!
In der Anzeige erscheint danach die Adresse der Einfügestelle:

In 0400 F.F (FF vom Monitor vorgegeben)

In die angezeigte Speicherstelle wird nun das neu eingegebene Datum eingetragen. Ein wiederholtes Betätigen der `Taste F1` erlaubt auf einfache Weise das Einfügen eines größeren Datenblockes.
Das folgende Bild 1.2-4 zeigt ein Beispiel für die Verwendung der Insert-Funktion. Darin werden ab der Adresse $0450 die Bytes $45 und $A7 eingefügt.

[7] In Kapitel 2 finden Sie eine Beschreibung der technischen Grundlagen der Verbindung zwischen Praktikumsrechner und PC und der Bedienung des mitgelieferten PC-Programms. Die serielle V.24-Schnittstelle wird in Kapitel 4 genau beschrieben.

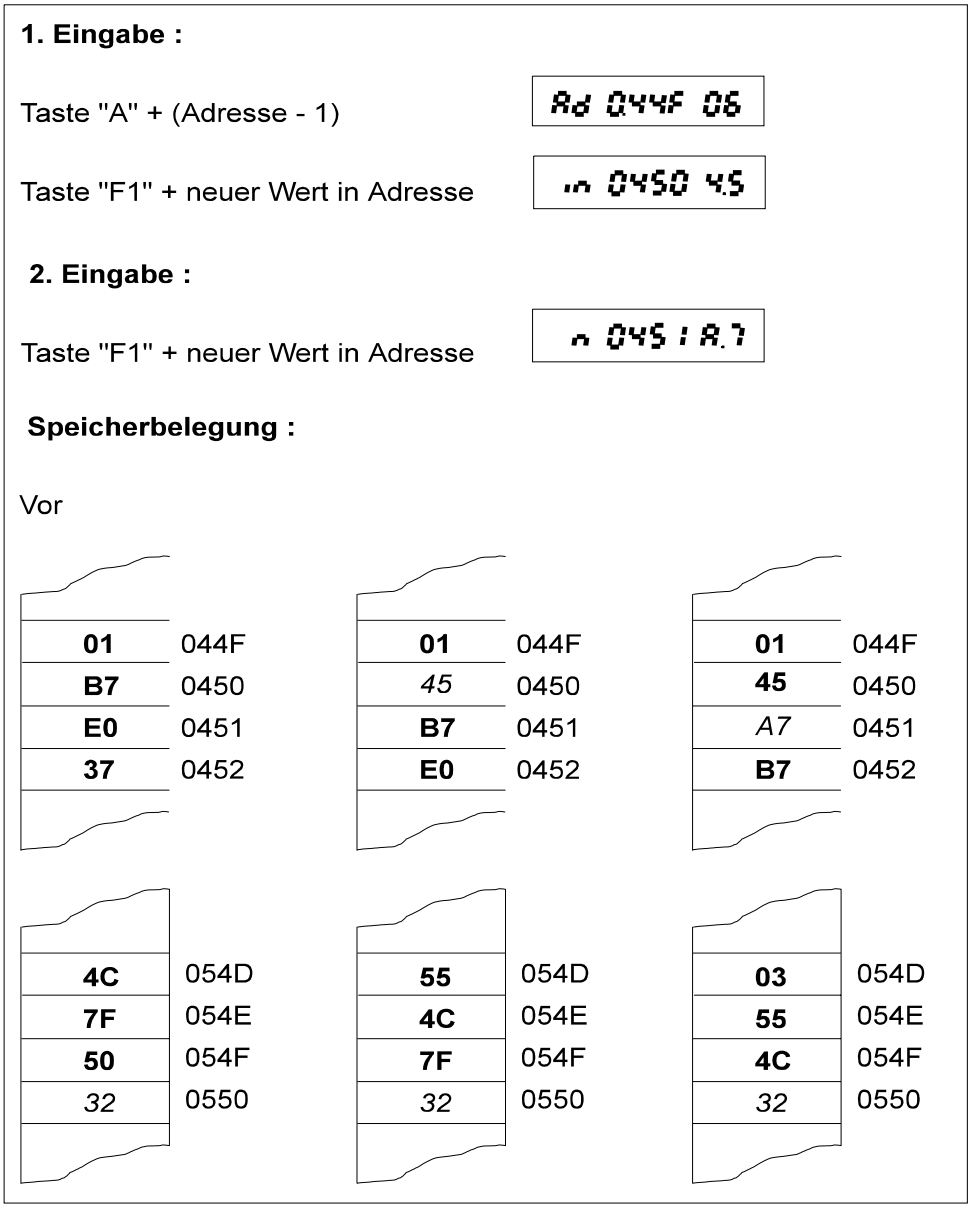


Bild 1.2-4: Beispiel für die mehrfache Ausführung der Insert-Funktion

Bedingt durch die Verschiebung des Datenblockes vergrößern sich die Sprungdistanzen zwischen Adressen vor der Einfügestelle und innerhalb des verschobenen Blockes. Daher müssen die in den entsprechenden Sprungbefehlen angegebenen Distanzen ggf. angepaßt werden.

Hinweis:
Die Startadresse der F1-Funktion wird beim Drücken der `Taste F1` aus den Speicherstellen $006C,$006D (H-Byte,L-Byte) entnommen. Diese Stellen liegen im Schreib/Lese-Speicher. Daher können sie vom Benutzer jederzeit geändert werden. So kann durch die `Taste F1` eine eigendefinierte Funktion aktiviert werden.

("*Delete*", KZ: dE)
Diese Taste ist mit der "Lösch-Funktion" vorbelegt, mit der ein einzelnes Byte aus einem zusammenhängenden Datenbereich oder einem Programm entfernt werden kann.
Dazu muß zunächst (mit der `Taste A`) die Adresse der zu löschenden Speicherzelle ins Adreßfeld geschrieben werden. Durch Drücken der `Taste F2` werden dann die 255 folgenden Bytes jeweils um eine Zelle nach unten (gegen niedrigere Adressen) verschoben. In das 256 Byte wird der hexadezimale Wert $FF nachgezogen. Mehrmalige Betätigung der `Taste F2` erlaubt so das einfache Löschen eines ganzen Speicherbereiches. Auch hier müssen Sprungdistanzen ggf. korrigiert werden.

Bild 1.2-5 zeigt ein Beispiel für die zweifache Ausführung der Delete-Funktion.

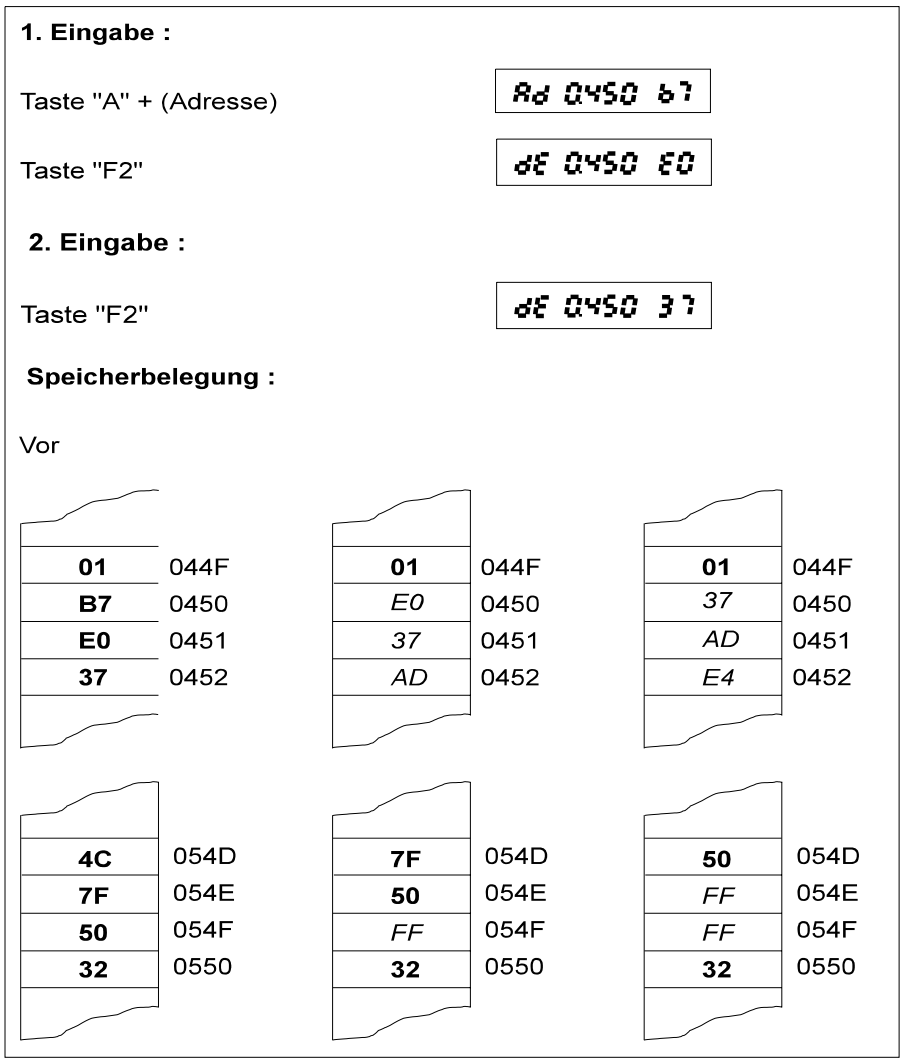


Bild 1.2-5: Beispiel für die zweifache Ausführung der Delete-Funktion

Hinweis:
Die Startadresse für die F2-Funktion wird den RAM-Speicherzellen $006F, $0070 (H-Byte,L-Byte) entnommen und kann daher vom Benutzer jederzeit geändert werden. So ist es möglich, durch die `Taste F2` eine andere Funktion aufzurufen.

("*Dump*", KZ: `dU`)
Diese Taste erlaubt in ihrer Vorbelegung ein schnelleres "Durchblättern" eines zusammenhängenden Speicherbereiches. Das Betätigen der Taste führt zu der Anzeige:

Nach Eingabe der Anfangsadresse des Bereiches, der dargestellt werden soll, muß ein weiteres Mal die `Taste F3` gedrückt werden. Danach erscheinen in der Anzeige die ersten vier Bytes des angewählten Speicherbereiches. Durch die `Taste +` können jeweils die vier folgenden Bytes, durch die `Taste -` die vier vorhergehenden Bytes dargestellt werden. Die Dump-Funktion wird durch das Betätigen einer beliebigen anderen Funktionstaste abgebrochen. Das folgende Beispiel zeigt die mehrfache Anwendung der Dump-Funktion (vgl. Bild 1.2-6).

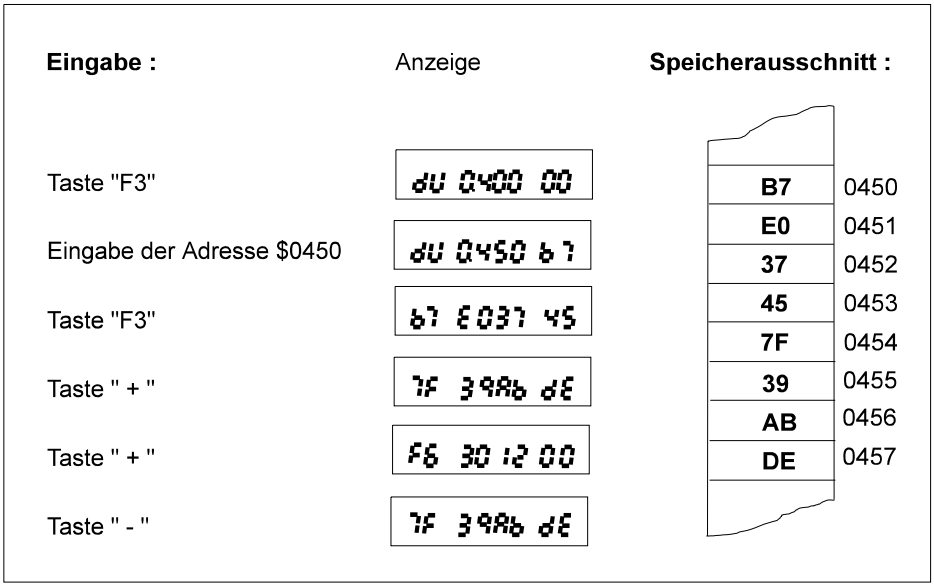


Bild 1.2-6: Mehrfache Anwendung der Dump-Funktion

Die Startadresse der F3-Funktion ist in den Speicherzellen $0072, $0073 (H-Byte,L-Byte) abgelegt und kann wiederum vom Benutzer zum Aufruf einer eigenentwickelten Funktion über die `Taste F3` modifiziert werden.

("*Break*", KZ: br)
(Die Startadresse des Programmes sollte $0400 sein.) Durch diese Taste kann ein laufendes Programm jederzeit unterbrochen werden. Voraussetzung dafür ist, daß im Bedingungsregister CC des 6809 nicht das Interruptflag (I-Flag, s. Abschnitt 1.3.2) gesetzt und im Port-Baustein 6821 der Interrupteingang CA1 (s. Kapitel 4) nicht deaktiviert wurde. Die Startadresse der Break-Routine liegt in den RAM-Speicherzellen $0036 und $0037. In der Anzeige erscheint z.B. die Information:

br 1011 F7

Das heißt, daß das Programm vor der Ausführung des Opcodes in Adresse $1011 unterbrochen wurde. Nun ist es möglich, sich die Registerinhalte mit der `Taste R` oder bestimmte Speicherplätze mit der `Taste A` anzusehen, und danach das Programm an der Unterbrechungsstelle fortzusetzen (`Taste G` oder `Taste T`).
Im Unterschied zu den drei `Tasten F1-F3` ist hier eine Änderung der Startadresse nicht möglich, da die `Taste F4` einen Hardware-Interrupt erzeugt.

Zusammenfassung

Die bisher beschriebenen Funktionen erlauben es, auf einfache Weise durch die Taste:

bestimmte Speicherstellen zu adressieren,

einen Datenwert einzugeben oder zu ändern,

ein Programm zu starten und dabei

einen Breakpoint zu setzen,

ein Programm im Einzelschritt-Modus auszutesten,

den Inhalt der internen Register anzuschauen und zu ändern,

Programme und Daten zum PC zu übertragen und dort zu speichern,

Programme und Daten vom PC zu laden,

ein Byte in einen Speicherbereich einzufügen,

ein Byte aus einem Speicherbereich zu löschen,

einen zusammenhängenden Speicherbereich anzuschauen,

ein laufendes Program zu unterbrechen.

Im nächsten Abschnitt werden Sie weitere fest implementierte Funktionen kennenlernen, die Ihnen die Arbeit mit dem Mikrorechner erleichtern werden. Sie sind im Gegensatz zu den bisher erklärten Funktionen nicht durch eine bestimmte Taste, sondern nur durch einen Unterprogrammsprung aufzurufen.

Hinweis auf eine Testmöglichkeit

Zum Abschluß dieses Abschnittes möchten wir Ihnen noch kurz erklären, wie man den Befehl SWI1 (Opcode $3F), den Sie bereits von der Beschreibung der `Taste G` kennen, einsetzen kann, um das Testen eines Programmes zu erleichtern:
Dazu können Sie an wichtigen Stellen diesen Befehl einfügen. Jedesmal, wenn im Programmlauf dieser Befehl gefunden wird, wird das Programm durch die oben beschriebene Interruptroutine unterbrochen. Sie können danach den abgespeicherten Inhalt der internen Register (`Taste R`) und bestimmter Speicherstellenanschauen (`Taste A`) und verändern (`Taste D`) und das Programm an der Unterbrechungsstelle erneut starten (`Taste G` oder `Taste T`). Erst wenn dieser Test ein fehlerfreies Programm zeigt, ersetzen Sie alle SWI1-Befehle durch den Befehl *No Operation* (NOP, Opcode $12), der lediglich ein Inkrementieren des Befehlszählers im Mikroprozessor bewirkt (s. Abschnitt 1.3).

Wichtiger Tip

Der Befehl SWI1 bietet auch eine geschickte Möglichkeit, ein Anwenderprogramm mit einem Sprung in das Monitor-Programm zu verlassen.

1.2.2 Hilfsroutinen

In diesem Abschnitt werden wir Ihnen einige Hilfsroutinen vorstellen, die Ihnen das Arbeiten mit dem Mikrorechner erleichtern sollen. Natürlich wäre es besser, wenn Sie jetzt bereits die Architektur und den Befehlssatz des Mikroprozessors kennen würden. Dennoch haben wir uns dazu entschlossen, diese Routinen bereits hier zu erklären. Auf diese Weise sind Sie in der Lage, sie bereits bei der Beschreibung des Mikroprozessors und den dabei von Ihnen begleitend durchzuführenden Übungen nutzbringend anzuwenden.

Für das Verständnis dieses Abschnittes müssen Sie nur wissen, daß der Prozessor zwei 8-bit-Akkumulatoren A und B sowie zwei 16-bit-Indexregister X,Y besitzt. Die Akkumulatoren A und B können auch zu einem 16-bit-Akkumulator D (A: H-Byte, B: L-Byte) verkettet werden. Die Register werden von den Hilfsroutinen zur Aufnahme der Eingangs- und Ausgangsparameter benutzt. Grundsätzlich sind die Routinen so geschrieben worden, daß in ihnen zunächst die Inhalte aller zusätzlich benötigten Register in den Kellerspeicher (Systemstack) gerettet werden. Vor dem Ende der Routinen werden diese Register dann wieder "restauriert", so daß Ihnen die ursprünglichen Werte danach unverändert zur Verfügung stehen. Nur die Inhalte der in den folgenden Beschreibungen explizit als Ein- oder Ausgaberegister angegebenen Register werden durch die Routinen verändert.

Nach Eingabe der jeweils benötigten Parameter in den Registern A,B,X,Y können Sie die Routinen durch einen Unterprogrammsprung (*Jump to SubRoutine* – JSR, *Branch to SubRoutine* – BSR, s. Abschnitt 1.3) ansprechen.

Mit dem Begriff Anzeigepuffer werden 8 Speicherzellen im RAM bezeichnet, die durch die entsprechenden Routinen "zusammenhängend" in die Anzeige übertragen werden. Dies ermöglicht es, vor der Darstellung in der Anzeige die Information zunächst im RAM zu "editieren".

Die Startadressen der Hilfsroutinen sind in einer Tabelle untergebracht, die im Adreßbereich $F100–$F1FF im EPROM liegt. In Tabelle 1.2-1 sind diese Startadressen mit einer Kurzbeschreibung der Routinen zusammengefaßt.

Es folgt nun eine detailliertere Beschreibung der aufgeführten Hilfsroutinen. Eine Übersicht zeigt Tabelle 1.2-1. Ein vorangesetztes $-Zeichen bezeichnet stets eine Hexadezimalzahl. In der 7-Segmentanzeige des Mikrorechners dargestellte Zeichen werden in ".." gesetzt. Verkürzend werden im folgenden auch Speicherzellen mit dem Begriff Ein- bzw. Ausgabe-"Register" bezeichnet. Durch eckige Klammern ([ ]) wird der Inhalt einer Speicherzelle angegeben, durch spitze Klammern (< >) eine durch ein Register adressierte Speicherzelle. Mit ":=" wird die Übertragung eines Wertes in eine Speicherzelle oder ein Register symbolisiert.

Tabelle 1.2-1: Übersicht über die implementierten Monitor-Routinen

Name	Funktion	Adresse
	1. Umwandlungsroutinen	
T7SG	Umwandlung der unteren Tetrade von B in den 7-Segment-Code, Ergebnis in A	F100
B7SG	Umwandlung beider Tetraden von B in den 7-Segment-Code, Ergebnis in D=A,B	F103
D7SG	Umwandlung der vier Tetraden von D in den 7-Segment-Code, Ergebnis in D,Y	F106
	2. Darstellungsroutinen	
CLRDISP	Löschen der Anzeige	F110
SHOWA	Bringt A in die Anzeige, Position in X	F113
SHOWD	Bringt D in die Anzeige, Position in X	F116
SHOWYD	Bringt Y,D in die Anzeige, Position in X	F119
SHOWT7SG	Umwandlung der unteren Tetrade von B in 7-Segment-Code, Darstellung in der Anzeige, Position in X	F11C
SHOWB7SG	Umwandlung beider Tetraden von B in den 7-Segment-Code, Darstellung in der Anzeige, Position in X	F120
SHOWD7SG	Umwandlung der vier Tetraden von D in den 7-Segment-Code, Darstellung in der Anzeige, Position in X	F123
	3. Routinen zur Bearbeitung des Anzeigepuffers Adresse des Puffers in X	
CLRDBUF	Löschen des Anzeigepuffers	F130
SHOWDBUF	Übertragen des Puffers in die Anzeige	F133
RRDBUF	Rotieren des Puffers um eine Stelle nach rechts	F136
RLDBUF	Rotieren des Puffers um eine Stelle nach links	F139
COPYDBUF	Kopieren eines 2. Puffers in den Anzeigepuffer	F13C
	4. Routinen zur Abfrage der Tastatur	
KEY	Lesen der Tastatur ohne Warten, Tastencode nach B	F140
HALTKEY	Lesen der Tastatur mit Warten, Tastencode nach B	F143
SHOWKEY	Lesen der Tastatur ohne Warten und Anzeigen, Tastencode nach B	F146
SHOWHKEY	Lesen der Tastatur mit Warten und Anzeigen, Tastencode nach B	F149
INDATA	Einlesen eines 8-bit-Datums über die Tastatur, Datum nach A, Tastencode nach B	F14C
SHOWDATA	Einlesen eines 8-bit-Datums über die Tastatur und Anzeigen, Datum nach A, Tastencode nach B	F150
INADR	Einlesen einer 16-bit-Adresse über die Tastatur, Adresse nach Y, Tastencode nach B	F153
SHOWADR	Einlesen einer 16-bit-Adresse über die Tastatur und Anzeigen, Adresse nach Y, Tastencode nach B	F156
	5. weitere Routinen	
DLY1MS	Schleife zur Zeitverzögerung, Zeitdauer in Y vorgegeben	F160
RANDOM	Pseudo-Zufallszahlengenerator, alter und neuer Wert in Y	F163
COPYXD	Verschieben von Speicherbereichen, Startadressen in X,Y, Länge in D	F166

Praktische Übung P1.2-2:

Führen Sie alle folgenden Routinen mit geeigneten Parametern selbst durch. Diese Parameter können Sie mit der Registerauswahl-Funktion der `Taste R` (s. Ab-schnitt 1.2.1) eingeben. Danach rufen Sie mit der `Taste G` (z.B. für die Routine SHOWA) das folgende kurze Programmstück auf:

Adresse	Befehl	Mnemocode	Bemerkung
0400 0401 0402	BD F1 13	JSR SHOWA	*ggf. andere Startadresse einsetzen,*

danach für:

- Hilfsroutinen, die etwas in der Anzeige darstellen

0403	13	SYNC	*Warteschleife, beenden durch „Clear“ (*Taste C*).*

- Hilfsroutinen, die ohne Anzeige arbeiten

0403	3F	SWI1	*Softwareinterrupt, Sprung in den Monitor,* Taste R: *Register überprüfen und ändern.*

1. Umwandlungsroutinen

T7SG

Startadresse: $F100
Eingaberegister: B
Ausgaberegister: A
Funktion: Die untere Tetrade von B, also die Bits B0-B3, werden in den 7-Segment-Code umgewandelt. Dieser steht danach im Akkumulator A zur Verfügung. Der Akkumulator B bleibt unverändert.
Beispiel: Eingabe: B:=$D8
Ausgabe: A:=$7F, B:=$D8

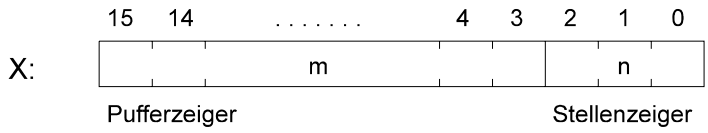
B7SG

Startadresse: $F103
Eingaberegister: B
Ausgaberegister: D (bzw. A,B)
Funktion: Beide Tetraden von B werden in den 7-Segment-Code umgewandelt. Der Code der oberen Tetrade steht danach im Akkumulator A, der der unteren Tetrade im Akkumulator B zur Verfügung.
Beispiel: Eingabe: B:=$D8
Ausgabe: A:=$5E, B:=$7F.

D7SG	*Startadresse:*	$F106
	Eingaberegister:	D
	Ausgaberegister:	D,Y
	Funktion:	Die in D (bzw. A,B) stehenden 2 Bytes, entsprechend 4 Tetraden, werden in den 7-Segment-Code umgewandelt. Nach der Ausführung stehen die beiden Codes des Bytes aus A in Y, die beiden Codes des Bytes aus B in D (vgl. B7SG)
	Beispiel:	Eingabe: D=A,B:=$D87C Ausgabe*: Y:=$5E7F, D:=$0758

2. Darstellungsroutinen

In den folgenden Routinen besitzt das X-Register eine fest vorgegebene Doppelfunktion. Dazu ist es in die beiden Bitgruppen X15,..,X3 und X2,X1,X0 unterteilt. Die Zahl n=X2,X1,X0 ($0 \le n \le 7$) dient als Zeiger auf die rechte Stelle des Anzeigefeldes, in dem eine Information geschrieben werden soll. Die Zahl m=X15,..,X3 ($m=8*i$, $0 \le i \le 2^{13}$) zeigt auf den Beginn eines 8 byte großen Pufferbereiches im Speicher. Dieser Pufferbereich wird durch <X>,...,<X+7> dargestellt. Nach der Vorgabe durch m beginnt er immer mit einer Adresse $xxx0 oder $xxx8 (x aus dem Bereich 0,..,F).[2]



Alle Darstellungsroutinen lassen den Pufferzeiger m unverändert, so daß es reicht, diesen für eine Folge von Routinen-Ausführungen ein einziges Mal zu setzen. Das X-Register mit der beschriebenen Doppelfunktion wird im folgenden mit X=(m,n) bezeichnet. Die 8 Stellen der Anzeige werden S7,...,S0 genannt. Dabei ist S0 die rechte, S7 die linke Stelle in der Anzeige (s. Abschnitt 1.2.1). Wird eine Stelle Si mit i>7 angesprochen, so hat diese Aktion keine Auswirkung auf die Anzeige.

CLRDISP	*Startadresse:*	$F110
	Eingaberegister:	keines
	Ausgaberegister:	Anzeige S7,...,S0
	Funktion:	Diese Routine löscht die gesamte Anzeige, d.h. in die entsprechenden Speicherzellen[3] wird der Wert $00 eingeschrieben.

* Die Hexadezimal-Ziffer 'C' wird im Praktikumsrechner durch ein kleines 'c' dargestellt.

[2] Im nächsten Unterabschnitt werden Routinen zur Bearbeitung des Puffers beschrieben.

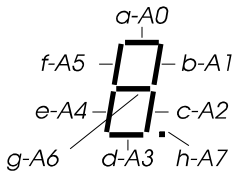
[3] In Kapitel 2 wird gezeigt, daß und wie die einzelnen Stellen der Anzeige unter festen Adressen angespro-chen werden

SHOWA *Startadresse:* $F113

Eingaberegister: A, X=(m,n)

Ausgaberegister: Anzeige S7,...,S0

Funktion: Die Nummer n=X2,X1,X0 der Anzeigestelle steht in den 3 unteren Bits von X. Das Byte in A wird in die so ausgewählte Anzeigestelle im Hexadezimalcode übertragen. Dabei gilt folgende Beziehung zwischen den Bits A0,..,A7 des Akkumulators A und den Seg-menten a,b,..,h der Anzeige:



Beispiel: Eingabe: A:=$7F, X:=0005
Ausgabe: S5:="8"

SHOWD *Startadresse:* $F116

Eingaberegister: D; X=(m,n)

Ausgaberegister: Anzeige S7,...,S0

Funktion: Die beiden Bytes in D=A,B werden hexadezimal in die Anzeigestellen Sn,Sn+1 geschrieben, wobei B nach Sn und A nach Sn+1 kommt.

Beispiel: Eingabe: D=A,B:=$5E7F, X:=$0004
Ausgabe: S5:="d", S4:="8"

SHOWYD *Startadresse:* $F119

Eingaberegister: Y; D; X=(m,n)

Ausgaberegister: Anzeige S7,...,S0

Funktion: Diese Routine stellt die vier Bytes in den Registern Y und D hexadezimal in der Anzeige dar. Das Byte aus B kommt in die Stelle Sn, das Byte aus A in die Stelle Sn+1. Das niederwertige Byte aus Y wird nach Sn+2 und das höherwertige nach Sn+3 übertragen (natürlich nur, sofern n+i<8).

Beispiel: Eingabe: Y:=$5E7F, D:=$075B, X:=$0000
Ausgabe: S3:="d", S2:="8", S1:="7", S0:="2"

SHOWT7SG	*Startadresse:*	$F11C
	Eingaberegister:	B, X=(m,n)
	Ausgaberegister:	Anzeige S7,...,S0
	Funktion:	Diese Routine verknüpft die Routinen T7SG und SHOWA. Die Akkumulatoren A,B bleiben jedoch unverändert. Die untere Tetrade von B wird zunächst in den 7-Segmentcode umgewandelt. Danach wird dieser in die Anzeigestelle Sn gebracht.
	Beispiel:	Eingabe: B:=$D8, X:=$0005 Ausgabe: S5:="8"

SHOWB7SG	*Startadresse:*	$F120
	Eingaberegister:	B, X=(m,n)
	Ausgaberegister:	Anzeige S7,...,S0
	Funktion:	Diese Routine verknüpft die Routinen B7SG und SHOWD. Beide Tetraden in B werden zunächst in den 7-Segmentcode umgewandelt. Danach werden diese in die Anzeigenstellen Sn und Sn+1 übertragen.
	Beispiel:	Eingabe: B:=$D8, X:=$0004 Ausgabe: S5:="d", S4:="8"

SHOWD7SG	*Startadresse:*	$F123
	Eingaberegister:	D (bzw. A,B) , X=(m,n)
	Ausgaberegister:	Anzeige S7,...,S0
	Funktion:	Diese Routine kombiniert die Routine D7SG und SHOWYD. Alle vier Tetraden von D werden zunächst in den 7-Segmentcode umgewandelt. Danach werden sie in die 4 Anzeigestellen Sn,Sn+1,Sn+2, Sn+3 übertragen (sofern n+i<8 !).
	Beispiel:	Eingabe: D=A,B:=$D87C, X:=$0000 Ausgabe: S3:="d",S2:="8",S1:="7",S0:="c"

3. Routinen zur Bearbeitung des Anzeigepuffers

CLRBUF

Startadresse: $F130
Eingaberegister: X
Ausgaberegister: <X>,...,<X+7>
Funktion: Das X-Register zeigt auf den Anfang eines 8-byte-Speicherbereiches (Anzeigepuffer). Dieser Bereich wird auf den Wert $00 zurückgesetzt.

Beispiel:
Eingabe: X:=$1040
Ausgabe: [$1040]:=$00,...,[$1047]:=$00

SHOWDBUF

Startadresse: $F133
Eingaberegister: X, <X>,..,<X+7>
Ausgaberegister: Anzeige S7,...,S0
Funktion: Durch das X-Register wird wieder ein 8 byte großer Anzeigepuffer <X>,..,<X+7> im Speicher selektiert. Dieser Bereich wird in die Anzeigestellen S7,..,S0 übertragen, wobei <X+i> nach Si kommt (i=0,..,7).

Beispiel:
Eingabe: X:=$1048
Ausgabe: S0:=[$1048],...,S7:=[$104F]

RRDBUF

Startadresse: $F136
Eingaberegister: X, <X>,..,<X+7>
Ausgaberegister: <X>,...,<X+7>
Funktion: Der 8-byte-Anzeigepuffer <X>,...,<X+7> wird um eine Stelle "nach rechts rotiert", d.h. der Inhalt der Speicherzelle <X+n-1> wird in die Speicherzelle <X+n> (n=1,..,6) übertragen, der Inhalt der Zelle <X+7> in die Zelle <X>.

Beispiel:
Eingabe: `X:=$1040`
Ausgabe:
```
[$1041]:=[$1040]⁴
 ..........
[$1046]:=[$1045]
[$1047]:=[$1046]
[$1040]:=[$1047]
```

Hinweis: Da die Reihenfolge der 8 Stellen in der Anzeige umgekehrt gewählt wurde, bedeutet eine Linksverschiebung im Puffer mit anschließender Darstellung eine Rechtsverschiebung in der Anzeige. Diese Richtungsumkehrung gilt entsprechend auch für die folgende Routine.

[4] Verkürzte Schreibweise: Zwischenspeicherung in weiterer Variablen nötig.

RLDBUF	*Startadresse:*	$F139
	Eingaberegister:	X, <X>,..,<X+7>
	Ausgaberegister:	<X>,...,<X+7>
	Funktion:	Der 8-byte-Anzeigepuffer <X>, ..., <X+7> wird um eine Stelle „nach links rotiert“, d.h. der Inhalt der Speicherzelle <X+n> wird in die Speicherzelle <X+n-1> (n=1,...,7) übertragen, der Inhalt der Zelle <X> in die Zelle <X+7>.
	Beispiel:	Eingabe: X:=$1040 Ausgabe: [$1040]:=[$1041] [$1041]:=[$1042] ..... [$1046]:=[$1047] [$1047]:=[$1040]

COPYDBUF	*Startadresse:*	$F13C
	Eingaberegister:	X,Y,<Y>,..,<Y+7>
	Ausgaberegister:	<X>,...,<X+7>
	Funktion:	Der durch das Y-Register adressierte 8-byte-Speicherbereich <Y>,..,<Y+7> wird in den Anzeigepuffer <X>,..,<X+7> kopiert.
	Beispiel:	Eingabe: X:=$1040, Y:=$1060 Ausgabe: [$1040]:=[$1060] [$1041]:=[$1061] ..... [$1047]:=[$1067]

4. Routinen zur Abfrage der Tastatur

Die folgenden Routinen verlangen teilweise nur eine Eingabe von der Tastatur. Daher entfällt in diesen Fällen die Angabe eines Eingaberegisters. Bei allen Routinen wird dafür gesorgt, daß das "Prellen" einer Taste nicht zu einer falschen Information führt.

KEY	*Startadresse:*	$F140
	Ausgaberegister:	B
	Funktion:	Diese Routine liest einmalig die Tastatur. Im Akkumulator B wird der Wert $FF zurückgegeben, wenn keine Taste gedrückt wird, in allen anderen Fällen enthält B den Tastencode nach der folgenden Tabelle 1.2-2.
	Beispiel:	Ausgabe: B:=$88, Taste T gedrückt, B:=$FF, keine Taste gedrückt.

Tabelle 1.2-2: Tastencodes

Datentasten				Funktionstasten			
Taste	Code	Taste	Code	Taste	Code	Taste	Code
0	$00	8	$08	+	$80	T	$88
1	$01	9	$09	-	$81	F1	$89
2	$02	A	$0A	A	$82	F2	$8A
3	$03	B	$0B	D	$83	F3	$8B
4	$04	C	$0C	R	$84		
5	$05	D	$0D	G	$85		
6	$06	E	$0E	S	$86		
7	$07	F	$0F	L	$87		

(Sie dürfen diese Tastencodes nicht mit den durch die Hardware vorgegebenen Codes der Tastatur verwechseln, wie Sie sie in Kapitel 2 kennenlernen werden. Hier handelt es sich um eine "willkürlich" gewählte Codierung der Routine KEY.)

HALTKEY *Startadresse:* $F143

Ausgaberegister: B

Funktion: Diese Routine liefert den gleichen Code wie die Routine KEY, wenn eine Taste gedrückt wird. Im Unterschied zu dieser wird nun jedoch die Tastatur zyklisch solange abgefragt, bis eine Taste gedrückt wurde oder aber die Routine unterbrochen wird (`Taste F4` oder `Taste C`).

Beispiel: Ausgabe: B:=$81 , `Taste -` gedrückt.
Hinweis: Diese Routine bietet die Möglichkeit, ein Benutzerprogramm zu "beenden", ohne daß die Anzeige verändert wird. Ist der nächste Befehl nach dem Aufruf der Routine ein SWI1, so führt jeder Tastendruck zu einem Sprung in den Monitor.

SHOWKEY *Startadresse:* $F146

Eingaberegister: X=(m,n)

Ausgaberegister: B, Anzeige S7,..,S0

Funktion: Diese Routine liest wie KEY einmal die Tastatur. Der erhaltene Tastencode nach Tabelle 1.2-2 wird jedoch in den 7-Segment-Code umgewandelt und in der Anzeigestelle Sn dargestellt. Bei den Funktionstasten wird lediglich die untere Tetrade umgewandelt und dargestellt. Zu ihrer Erkennung wird jedoch außerdem der Dezimalpunkt der Stelle Sn angesprochen. Ist keine Taste gedrückt worden, so wird wie-

der B:=$FF gesetzt und die angewählte Stelle durch Sn:=" " gelöscht.

Beispiel: Eingabe: X:=$0003
Ausgabe: B:=$04, S3:="4" , `Taste 4,`
B:=$86,S3:="6.", `Taste S,`
B:=$FF, S3:=" ", keine Taste

SHOWHKEY *Startadresse:* $F149
Eingaberegister: X=(m,n)
Ausgaberegister: B, Anzeige S7,..,S0
Funktion: Diese Routine verknüpft die Routinen HALTKEY und SHOWT7SG. Die Tastatur wird ununterbrochen solange gelesen, bis eine Taste gedrückt wird. Der Tastencode (s. Tabelle 1.2-2) wird dann in der Stelle Sn dargestellt.

Beispiel: Eingabe: X:=$0005
Ausgabe: B:=$87, S5:="7.", `Taste L`.

INDATA *Startadresse:* $F14C
Ausgaberegister: A, B
Funktion: Diese Routine liest zyklisch die Tastatur und packt die beiden jeweils zuletzt eingegebenen Hexadezimalzeichen {0,..,F} nach A. Die Routine wird durch Betätigen einer beliebigen Funktionstaste verlassen. Deren Tastencode steht danach in B zur Verfügung. Der Accu A wird mit dem Wert $00 initialisiert.

Beispiel: Initialisierung: A:=$00
`Taste 4`: A=$04
`Taste 8`: A=$48
`Taste D`: A=$8D
`Taste +`: A=$8D, B=$80

SHOWDATA *Startadresse:* $F150
Eingaberegister: X=(m,n)
Ausgaberegister: A,B, Anzeige S7,..,S0
Funktion: Diese Routine verknüpft die Routinen INDATA und SHOWB7SG, liest also ein Datum über die Tastatur und stellt es in der Anzeige dar.

Beispiel: Initialisierung: A:=$00, X:=$0000,
S1:="0", S0:="0"

Taste 4: A=$04, S1="0", S0="4"
Taste F: A=$4F, S1="4", S0="F"
Taste D: A=$FD, S1="F", S0="D"
Taste +: A=$FD, S1="F", S0="D", B=$80

INADR	*Startadresse:*	$F153
	Ausgaberegister:	Y,B
	Funktion:	Diese Routine liest zyklisch die Tastatur und packt jeweils die letzten vier eingegebenen Hexadezimalzeichen ins Y-Register. Dabei werden die Zeichen in Y tetradenweise zu den höherwertigen Bitpositionen verschoben. Die Routine wird durch das Betätigen einer beliebigen Funktionstaste verlassen. Der entsprechende Tastencode steht danach im Accu B zur Verfügung. Y wird mit dem Wert $0000 initialisiert.
	Beispiel:	Initialisierung: Y:=$0000 `Taste 4:` Y=$0004 `Taste F:` Y=$004F `Taste D:` Y=$04FD `Taste 9:` Y=$4FD9 `Taste 0:` Y=$FD90 `Taste +:` Y=$FD90, B=$80

SHOWADR	*Startadresse:*	$F156
	Eingaberegister:	X=(m,n)
	Ausgaberegister:	Y,B, Anzeige S7,...,S0
	Funktion:	Diese Routine verknüpft die Routinen INADR und SHOWD7SG, d.h. sie liest eine vierstellige Zahl (Adresse) über die Tastatur ins Y-Register und stellt sie gleichzeitig in der Anzeige dar. Y wird mit dem Wert $0000 initialisiert.
	Beispiel:	Initialisierung: Y:=$0000, X:=$0002 S5..S2:="0000" `Taste 4:` Y=$0004, S5..S2="0004" `Taste F:` Y=$004F, S5..S2="004F" `Taste D:` Y=$04FD, S5..S2="04FD" `Taste 9:` Y=$4FD9, S5..S2="4FD9" `Taste 0:` Y=$FD90, S5..S2="FD90" `Taste +:` Y=$FD90, S5..S2="FD90", B=$80

5. Weitere Hilfsroutinen

DLY1MS

Startadresse: $F160

Eingaberegister: Y

Funktion: Diese Routine erzeugt eine Zeitverzögerung bis zur Ausführung des nächsten Befehls im Benutzerprogramm. Die Verzögerungszeit ist in msec-Schritten im Y-Register zu wählen und berechnet sich nach der Formel:

$$T_d = (0.034 + Y) \text{ msec.}$$

Y=$0000 wird dabei als $10000=$2^{16}$=65536 interpretiert und ergibt somit die maximale Verzögerungszeit

$$T_{d,max} = 65.536034 \text{ sec.}$$

Beispiel: Eingabe: Y:=$03E8=1000 ergibt eine Verzögerung von ca. 1 Sekunde

RANDOM

Startadresse: $F163

Eingaberegister: Y

Ausgaberegister: Y

Funktion: Diese Routine erzeugt aus der eingegebenen Zahl im Y-Register eine 16-bit-Pseudo-Zufallszahl, die ebenfalls im Y-Register ausgegeben wird. Startet man eine Programmfolge zur Erzeugung von Zufallszahlen mit einem Wert Y>$0000 und wird das jeweils zuletzt erhaltene Ergebnis als Eingabe für den neuen Durchlauf benutzt, so werden in einer fest vorgegebenen Reihenfolge alle 65535 möglichen Zufallszahlen aus dem Bereich $0001 – $FFFF generiert. Diese Zufallszahlen sind gleichverteilt. Man spricht von Pseudo-Zufallszahlen, weil dieselbe Reihenfolge der Zahlen immer dann gezogen wird, wenn man mit demselben Startwert beginnt.

Beispiel: Eingabe: Y:=$BB8C (Startwert)
Ausgabe: Y:=$4B35 (= neue Eingabe)
Y:=$3FCA (= neue Eingabe)
Y:=$405B

COPYXD

Startadresse: $F166

Eingaberegister: X,Y, D=A,B

Funktion: Diese Routine kopiert einen zusammenhängenden Speicherbereich in einen anderen. Die Startadresse des Quellbereiches wird dazu im X-Register, die des

Zielbereiches im Y-Registers angegeben. Die Länge des Bereiches muß im D-Register übergeben werden.

Beispiel:

Eingabe: X:=$0400, Y:=$0600
D:=$00FF,

Ausgabe:
[$0600]:=[$0400]
[$0601]:=[$0401]
....
[$06FF]:=[$04FF]

1.3 Der Mikroprozessor 6809

1.3.1 Die Architektur

Der Prozessor 6809 ist ein 8-bit-Mikroprozessor aus der 68xx-Bausteinreihe von Motorola. Er ist gegenüber seinen „Vorgängern“ 6800 und 6802 wesentlich verbessert und verfügt intern zum Teil über eine 16-bit-Struktur, denn die meisten Register sind 16 bit breit. Sie ermöglichen somit 16-bit-Operationen. Diese werden jedoch in mehreren Schritten durch eine 8-bit-ALU ausgeführt. Die beiden 8-bit-Akkumulatoren (A und B) lassen sich für einige Operationen zu einem 16-bit-Akkumulator (D) zusammenschalten (A: H-Byte, B: L-Byte). Bild 1.3-1 zeigt die interne Organisation des 6809. Der 6809 ist in einem DIL-Gehäuse (*dual inline*) mit 40 Anschlüssen unter-gebracht.

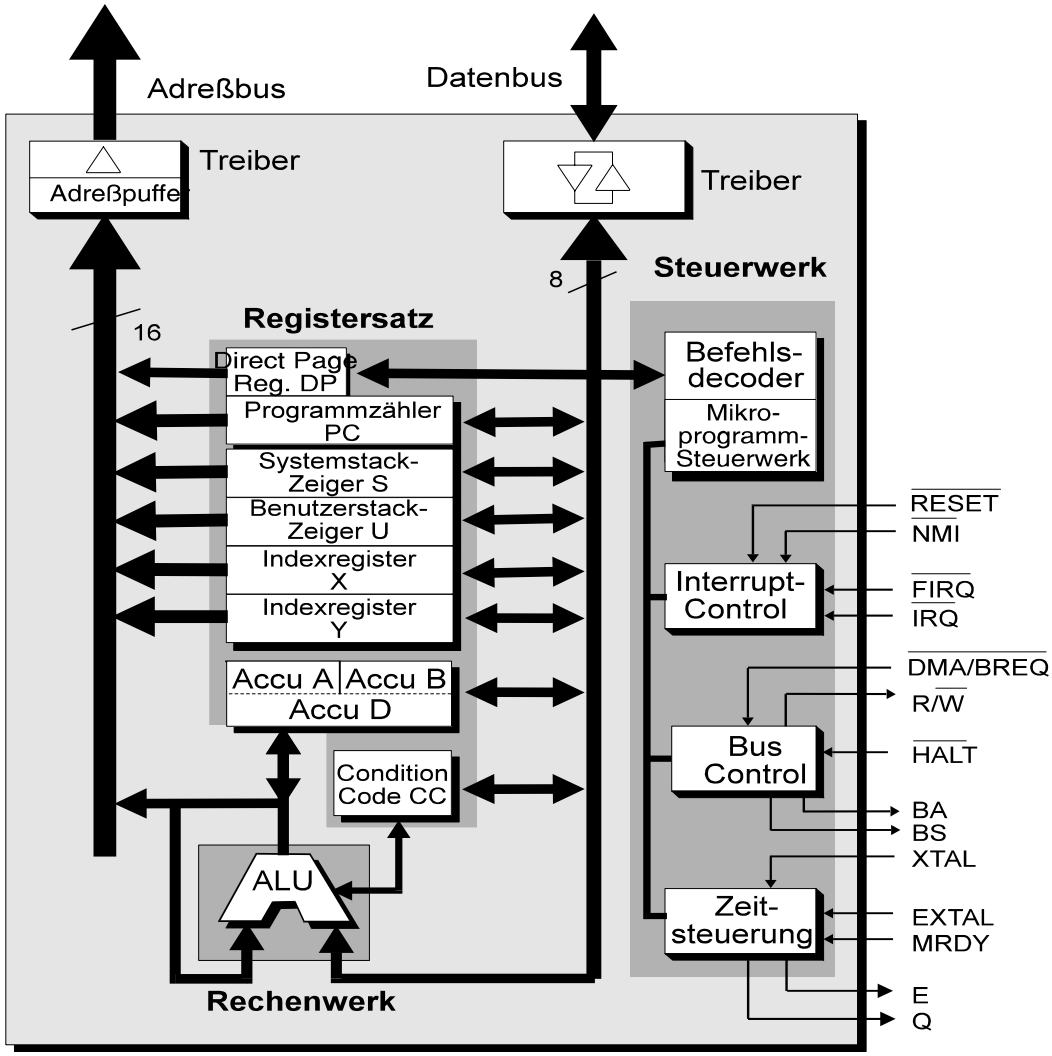


Bild 1.3-1: Interne Organisation des 6809

.

1.3.2 Steuer- und Statussignale

Die Steuer- und Statussignale des 6809 lassen sich in vier Gruppen einteilen:

- Taktsignale
- Systembus-Zugriffssteuersignale
- Speicher-Steuersignale
- Unterbrechungssignale

1. Taktsignale

Der 6809 verfügt über einen internen Taktgenerator, so daß an den Anschlüssen XTAL und EXTAL extern nur ein Quarz angeschlossen wird. Je nach Ausführung kann der 6809 mit einem 4, 6 oder 8 MHz-Quarz betrieben werden[3] . Die für den Maschinenzyklus maßgebende interne Taktfrequenz beträgt ein Viertel der Quarzfrequenz, so daß Zykluszeiten zwischen 500 ns und 1 µs erreicht werden. Aus diesem Taktsignal werden intern zwei Takte abgeleitet (vgl. die folgende Skizze in Bild 1.3-2 für eine Quarzfrequenz von 8 MHz):

- Das E-Signal: Mit der fallenden Flanke des Signals E (*Enable*, freigeben) werden die Daten vom Prozessor bzw. von den Speicher- und Peripheriebausteinen übernommen. Die Adreß- und Steuerleitungen werden in den ersten 200 ns stabil.
- Das Q-Signal: Die steigende Flanke des Taktes Q zeigt an, daß alle Ausgangssignale des Prozessors gültig sind.

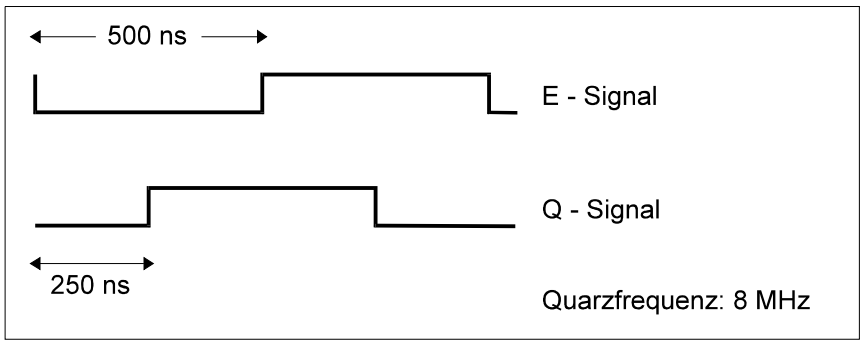


Bild 1.3-2: Die Taktsignale des 6809

2. Systembus-Zugriffssteuersignale

Die Steuersignale DMA/BREQ, BS und BA werden verwendet, um den Prozessor unter bestimmten Voraussetzungen vom Bussystem zu trennen und so einer anderen Rechnerkomponente den Zugriff zum Systembus zu gewähren, z.B. für den DMA-Betrieb (*Direct Memory Access*). Wird der CPU über die $\overline{\text{DMA / BREQ}}$-Leitung eine

[3] Verglichen mit der Taktfrequenz Ihres PCs mögen Ihnen diese Werte vielleicht sehr klein vorkommen. Diese Frequenzen sind aber für viele Steuerungsaufgaben auch heute immer noch ausreichend und im Mikrocontrollerbereich anzutreffen.

Busanforderung gemeldet, gibt diese nach Beendigung des gerade ausgeführten Befehls den Daten- und Adreßbus frei, d.h. sie schaltet ihre Adreß- und Datenbustreiber in den hochohmigen Zustand (*Tristate*, *„high impedance*“). Der 6809 kann den Bus an DMA-Komponenten jeweils nur für die Dauer von 15 Taktzyklen abgeben, da er ihn danach für die Dauer von mindestens einem Taktzyklus für das Auffrischen des Inhalts der internen Register (*Refresh*) benötigt.

Auf den Leitungen BA und BS wird der monentane Zustand angezeigt und zwar:

BA	BS	Funktion
0	0	Normaler Busbetrieb
0	1	Unterbrechung oder Rücksetzen (RESET)
1	0	SYNC-Quittung
1	1	Bustreiber hochohmig

3. Speicher-Steuersignale

MRDY und R/$\overline{W}$ sind die beiden Signale, die für den Speicherverkehr benötigt werden. Das MRDY-Signal dient Ein-/Ausgabe-Komponenten niedriger Geschwindig-keit dazu, sich mit dem 6809 zu synchronisieren. Eine logische „0“ signalisiert der CPU, daß die betreffende Einheit noch nicht bereit ist. Der 6809 wartet so lange, bis das Signal gesetzt wird. Das MRDY-Signal darf nur für maximal 10 Taktzyklen logisch „0“ sein (bis zu 10 µs). Im Gegensatz zu anderen Mikroprozessoren (z.B. den Intel-Prozessoren 80x80) kennt der 6809 kein Steuersignal (*Memory/Input-Output* - M/IO) für den Zugriff auf Peripheriebausteine. Die Register dieser Bausteine werden wie Speicherzellen adressiert.

4. Unterbrechungssignale

Dies sind die drei Hardware-Interruptsignale $\overline{NMI}$, $\overline{IRQ}$, $\overline{FIRQ}$ sowie die Signale $\overline{HALT}$, $\overline{RESET}$.

- Durch $\overline{RESET}$ wird der Prozessor in den Anfangszustand zurückgesetzt: Der Programmzähler PC wird mit der Startadresse, das *Direct Page Register* DP mit $00 und das I- und F-Flag des Statusregisters CC mit „1“ geladen, die übrigen Flags sind undefiniert. Auch die anderen Register (Akkumulatoren A,B, Indexregister X,Y, Stackregister U,S) sind nach einem RESET in einem nicht vorhersehbaren Anfangszustand. Der $\overline{IRQ}$ bzw. der $\overline{FIRQ}$ sind gesperrt. (Alle Register werden im nächsten Unterabschnitt ausführlicher beschrieben.)

  In den meisten Anwendungen wird das $\overline{RESET}$-Signal gleichzeitig den Peripheriebausteinen zugeführt, damit diese ebenfalls in einen Grundzustand versetzt werden.

Im Gegensatz zu den anderen Unterbrechungszuständen „Interrupt“ und „Halt“ wird der laufende Befehl nicht zu Ende geführt, sondern abgebrochen.

- Das $\overline{\text{HALT}}$-Signal unterbricht die Arbeit des Prozessors. Der gerade begonnene Befehl wird noch ausgeführt. Danach wartet der Prozessor, ohne seine Daten zu verlieren. Die Signale BA und BS sind logisch „1“. Sobald das $\overline{\text{HALT}}$-Signal zurückgesetzt wird, arbeitet der Prozessor weiter.

Die Hardware-Interrupts sind der nicht maskierbare Interrupt (NMI), der schnelle maskierbare Interrupt (FIRQ) und der maskierbare Interrupt (IRQ). Sie werden durch Signale an den gleichnamigen Eingängen aktiviert.

- Signal $\overline{\textbf{NMI}}$ (*Non Maskable Interrupt*)
  Liegt keine Busanforderung bzw. kein Signal am $\overline{\text{DMA / BREQ}}$-Eingang vor, so kann der NMI unverzüglich abgearbeitet werden. Im anderen Fall muß erst die Freigabe des Busses abgewartet werden. Es werden automatisch der Programmzähler PC und alle Register (Ausnahme: Stackregister U,S) auf den Systemstack geschrieben. Im Statusregister CC werden die Flags E, F und I gesetzt.

- Signal $\overline{\textbf{IRQ}}$ (*Maskable Interrupt*)
  Der Interrupt IRQ rettet alle Register, außer dem Stackregister S, auf dem Stack. Dieser Vorgang entspricht dem bei der nicht maskierbaren Unterbrechung. Das E-Flag im Statusregister wird gesetzt. Ferner wird das I-Flag gesetzt, um weitere Unter-brechungen zu verhindern.

- Signal $\overline{\textbf{FIRQ}}$ (*Fast Maskable Interrupt*)
  Die schnelle Unterbrechungsanforderung entspricht im wesentlichen dem IRQ. Es werden aber nur die Inhalte des Programmzählers PC und des Statusregisters CC im Stack gespeichert. Der FIRQ kann maskiert werden durch Manipulation des F-Flags im Statusregister. Um weitere Unterbrechungen zu verhindern, werden sowohl das F**-** wie auch das I-Flag automatisch gesetzt.

Im Vorgriff auf den nächsten Abschnitt sei hier schon einmal gesagt, daß der 6809 zusätzlich zu den Hardware-Interrupts über drei Software-Interrupts verfügt. Diese werden durch die Befehle SWI1, SWI2 und SWI3 aufgerufen und können als spezielle Unterprogramme angesehen werden. Wenn eine Software-Interruptanforderung vorliegt, werden ebenfalls alle Registerinhalte im Systemstack gerettet. SWI2 und SWI3 haben die geringste Priorität von allen Unterbrechungsanforderungen, da sie keinerlei Interrupts sperren, es wird nur das E-Flag gesetzt. Der SWI1 hingegen setzt das E-, F- und I-Flag und sperrt daher den FIRQ und den IRQ.

Die Startadressen (Interruptvektoren) für die Behandlungsroutinen der verschiedenen Unterbrechungen sind unter den höchsten Adressen im (Festwert-)Speicher des Praktikumsrechners abgelegt. Die folgende Tabelle 1.3-1 zeigt die Lage der Interruptvektoren.

Tabelle 1.3-1: Speicheradressen der Interruptvektoren

Adresse H-Byte L-Byte	**Interrupt-Vektor**	**Ausnahme-situation**	**Bemerkung**
FFFE FFFF		$\overline{\text{RESET}}$	Rücksetzen
FFFC FFFD		$\overline{\text{NMI}}$	nicht maskierbarer Interrupt
FFF6 FFF7		$\overline{\text{FIRQ}}$	schneller maskierbarer Interrupt
FFF8 FFF9		$\overline{\text{IRQ}}$	maskierbarer Interrupt
FFFA FFFB		SWI1	Software-Interrupt 1
FFF4 FFF5		SWI2	Software-Interrupt 2
FFF2 FFF3		SWI3	Software-Interrupt 3
FFF0 FFF1			(reserviert)

Praktische Übung P1.3-1:

Ergänzen Sie die obenstehende Tabelle 1.3-1 um die Interruptvektoren, die im Festwertspeicher des Praktikumsrechners programmiert sind. Schauen Sie sich dazu jeweils den Beginn der zugehörigen Interruptroutine an !

Treten mehrere Unterbrechungsanforderungen gleichzeitig auf, werden sie nach fest vorgegebenen Prioritäten abgearbeitet. Diese Prioritäten sind in der folgenden Tabelle 1.3-2 angegeben.

Tabelle 1.3-2: Interrupts des 6809, nach Prioritäten geordnet

Priorität	**Interrupt**	**gesetztes Flag im CC-Register**	**gesperrte Interrupts**
1	$\overline{\text{NMI}}$	E,F,I	$\overline{\text{IRQ}}$, $\overline{\text{FIRQ}}$
2	$\overline{\text{FIRQ}}$	-,F,I	$\overline{\text{IRQ}}$, $\overline{\text{FIRQ}}$
3	$\overline{\text{IRQ}}$	E,-,I	$\overline{\text{IRQ}}$
4	SWI1	E,F,I	$\overline{\text{IRQ}}$, $\overline{\text{FIRQ}}$
5	SWI2	E,-,-	----
6	SWI3	E,-,-	----

Eine Zusammenfassung dieses Unterabschnittes enthält das Flußdiagramm im Bild 1.3-3.

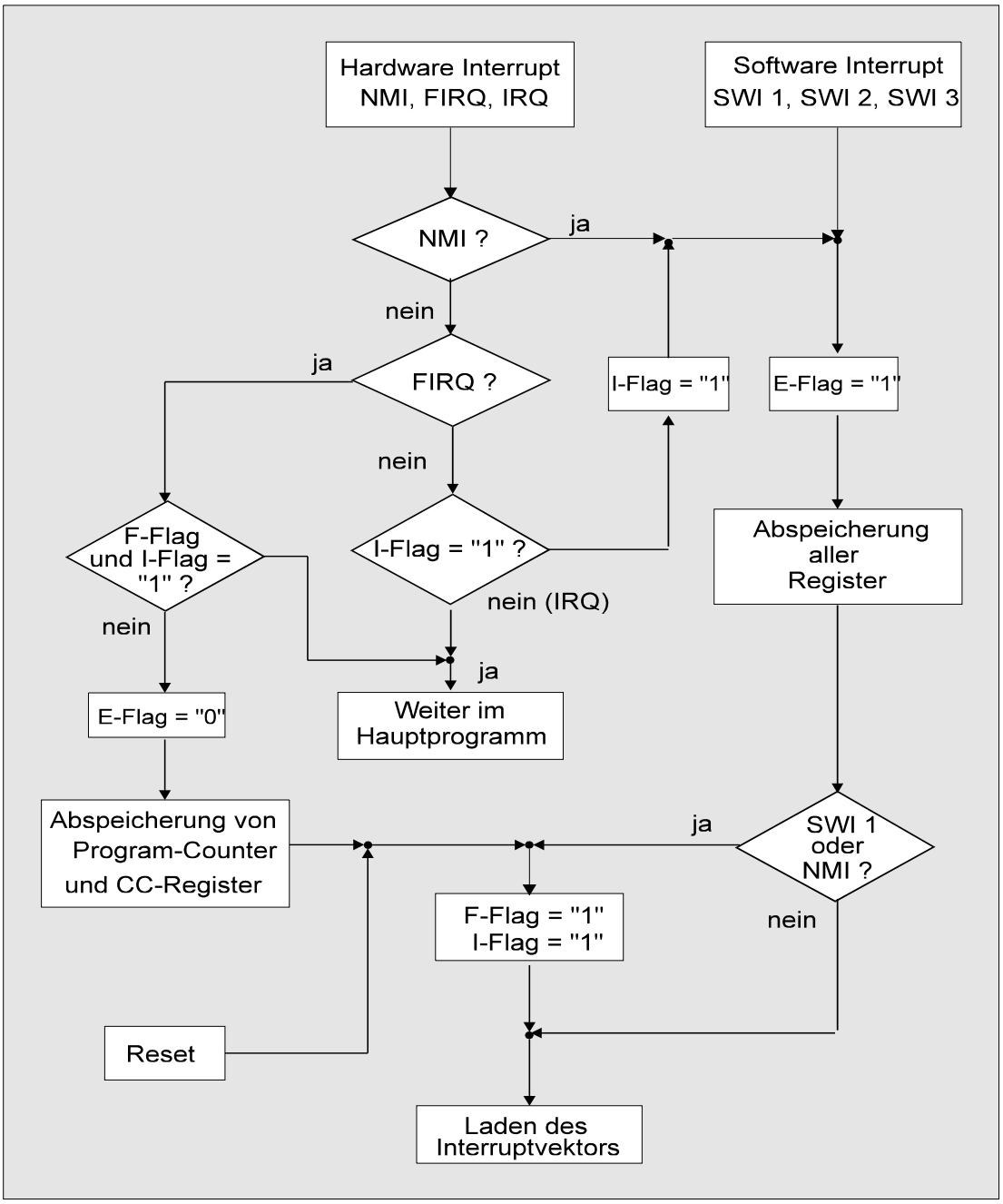


Bild 1.3-3: Flußdiagramm zur Interruptbehandlung

1.3.3 Der Registersatz

Der Registersatz des 6809 wurde bereits im Bild 1.3-1 gezeigt. Hier sollen nun die Register ausführlicher beschrieben werden. Tabelle 1.3-3 stellt noch einmal die Register, ihre abkürzende Bezeichnungen und ihre Bitlängen zusammen.

Tabelle 1.3-3: Register der CPU 6809

Bez.	Name	Funktion	Länge
A	Accumulator	8-bit-Akkumulator	8 bit
B	Accumulator	8-bit-Akkumulator	8 bit
D	Accumulator (A,B)	16-bit-Akkumulator	16 bit
PC	Program Counter	Programmzähler	16 bit
X	Index Register	Indexregister mit Auto-Inkr./Dekr.,	16 bit
Y	Index Register	auch als Universalreg. einsetzbar	16 bit
U	User Stack Pointer	Stackregister für Benutzer- oder	16 bit
S	System Stack Pointer	Systemstack, PUSH/PULL-Befehle	16 bit
DP	Direct Page Register	Seiten-Adreßregister	8 bit
CC	Condition Code Register	Statusregister	8 bit

- **Akkumulatoren A,B (D)**
  sind zwei 8-bit-Register, die bei (fast) allen arithmetischen oder logischen Operationen einen der Operanden und das Ergebnis enthalten. Sie können in einigen Befehlen zu einem 16-bit-Akkumulator D zusammengefaßt werden, wobei dann A das höherwertige und B das niederwertige Byte enthalten. Bei der Multiplikation stehen in A und B die Faktoren und in D das Ergebnis.

- **Programmzähler PC**
  Der Programmzähler enthält stets die Adresse der Speicherzelle, in der der als nächster auszuführende Befehl oder Befehlsteil steht.

- **Indexregister X,Y**
  Dies sind 16-bit-Register (2 byte), die über die Möglichkeit der automatischen Modifikation um den Wert w=1 oder w=2 verfügen:
  - Prädekrement: Vor der Ausführung der Operation wird der Registerinhalt um w erniedrigt,
  - Postinkrement : Nach der Ausführung der Operation wird der Registerinhalt um w erhöht.

Sie werden zur Berechnung und Änderung relativer Adressen verwendet (s. nächsten Abschnitt). Damit kann z.B. in einer Tabelle in aufsteigender oder absteigender Reihenfolge mit

- w=1: das nächste Datenbyte,
- w=2: das nächste 16-bit-Datenwort (oder die nächste 16-bit-Adresse)

angesprochen werden.

Durch die Befehle LEAX (*Load Effective Address X*) und LEAY (*Load Effective Address Y,* siehe Abschnitt 1.3.5) kann während der Laufzeit eines Programms die effektive Adresse eines Operanden ermittelt und ins Indexregister X bzw. Y gebracht und dort manipuliert werden. Dies erlaubt - zusammen mit der relativen Adressierung, siehe nächsten Abschnitt - das Schreiben von lageunabhängigen Programmen, die an einem beliebigen Ort im Speicher ablauffähig sind.

X und Y können aber auch als allgemeine Datenregister benutzt werden.

- **Stackregister S, U**

  Das Stackregister S dient zur Verwaltung des Systemstacks, der bei Unterprogramm-Sprüngen und Unterbrechungen von der CPU zur Speicherung der Rücksprungadressen und der internen Register verwendet wird. Über den Systemstack können Haupt- und Unterprogramme aber auch Daten austauschen. S wird durch die Speichervorgänge bei Sprüngen in bzw. Rücksprüngen aus Unterprogrammen und Interruptroutinen sowie durch spezielle Befehle (PSHS/PULS) automatisch modifiziert, und zwar - wie bei den Indexregistern beschrieben - prädekrementell und postinkrementell. Der Stack Pointer U dient zur Verwaltung eines dem Anwender zur freien Verfügung stehenden Benutzerstacks. Dieser Stack wird über die speziellen Befehle PSHU/PULU aufgebaut.

- **Seitenadreßregister DP**

  Bei der „direkten Adressierung“ (s. nächsten Abschnitt) wird die Adresse durch Konkatenation aus dem Wert des DP-Registers und der im Befehl angegebenen 8-bit-Kurzadresse gebildet, wobei der Inhalt des DP-Registers das höherwertige Byte der Adresse ist. Beim RESET wird das DP-Register auf $00 gesetzt. Mit dem DP-Register lassen sich 2^8=256 Seiten (*Pages*) ansprechen. Jede Page ist 256 Bytes lang. Damit ergibt sich für die effektive Gesamtadresse:

  $$EA = (DP)*256 + Kurzadresse.$$

  Die Vorteile der Adressierung über das DP-Register bestehen in kürzeren und schnelleren Programmen, da für das höherwertige Adreßbyte kein Speicherplatz und für sein Holen aus dem Speicher kein zusätzlicher Zyklus benötigt wird.

- **Statusregister CC**

  Im Statusregister (*Condition Code Register*) werden in der Regel Informationen über das Ergebnis arithmetischer oder logischer Operationen gespeichert. Diese Informationen werden insbesondere von Verzweigungsbefehlen ausgewertet. Im folgenden Bild 1.3-4 wird der Aufbau des Statusregisters dargestellt und die Lage

der Flags darin gezeigt. Bei ihrer Beschreibung werden wir schon im Vorgriff auf Abschnitt 1.3.5 kurz auf einige Befehle eingehen, bei denen die besprochenen Flags eine besondere Rolle spielen.

Bit	7	6	5	4	3	2	1	0
Bez.	E	F	H	I	N	Z	V	C
Flag	Entire	FIRQ-Mask	Half-Carry	IRQ-Mask	Nega-tive	Zero	Over-flow	Carry

Bild 1.3-4: Die Flags des Statusregisters

- **Carry Flag** (Bit 0)
  Dieses Flag wird gesetzt, wenn bei Operationen mit zwei Bytes bei Bit 15 und Operationen mit einem Byte bei Bit 7 ein Übertrag entsteht (z.B. bei Additionsbefeh-len). Es nimmt ferner bei Schiebe- und Rotationsbefehlen die Information der geschobenen Bits auf. Bei Schiebe- und Rotationsbefehlen nach rechts wird das niederstwertige, nach links das höchstwertige Bit aufgenommen.

  Zusätzlich gelten folgende Bedingungen:
  Der COM-Befehl setzt das C-Flag, der CLR-Befehl löscht das C-Flag. Beim MUL-Befehl wird es gesetzt, wenn Bit 7 im Akkumulator B gesetzt ist. Beim CWAI und ANDCC Befehl nimmt es den Wert der UND-Verknüpfung zwischen den Bits 0 des CC-Registers und der im Befehl angegebenen Konstanten an und beim ORCC-Befehl den Wert der entsprechenden ODER-Verknüpfung.

- **Overflow Flag** (Bit 1)
  Zur Bearbeitung von vorzeichenbehafteten Speicher- oder Registerinhalten wird die Zweierkomplement-Arithmetik angewendet. (Bit 7 gilt dabei als Vorzeichen, wobei „1“ mit „-“ und „0“ mit „+“ identifiziert werden.) Dieses Flag wird gesetzt, wenn bei den Befehlen ADD,ADC, CMP,SBC,SUB,DEG,INC und NEG ein Überlauf im Zweier-Komplement auftritt, also eine Zahl größer als 127 ($7F) oder kleiner als -128 ($80) erzeugt wird.

- **Zero Flag** (Bit 2)

Dieses Bit zeigt an, daß das Ergebnis einer vorangegangenen Operation Null war.

- **Negative Flag** (Bit 3)
  Dieses Flag nimmt den Wert des höchstwertigen Bits eines Operationsergebnisses auf. Es bezeichnet durch den Wert 1 eine negative, durch 0 eine positive Zahl.

- **Interrupt Mask** (Bit 4)
  Dieses Flag sperrt während der Abarbeitung der Interruptroutine alle weiteren IRQ-Anforderungen. Es wird gesetzt bei der Aktivierung einer der Unterbrechungsanforderungen $\overline{\text{RESET}}$, $\overline{\text{NMI}}$, SWI1, $\overline{\text{FIRQ}}$, $\overline{\text{IRQ}}$, nicht jedoch bei SWI2, SWI3.

- **Half-Carry Flag** (Bit 5)
  Dieses Flag wird bei Additionsbefehlen gesetzt, wenn ein Übertrag von Bit 3 nach Bit 4 erfolgt oder wenn bei dem Befehl SBC das Bit 7 gesetzt wird.. Es wird z.B. bei der Abarbeitung des Befehls DAA verwendet, der aus einem dualen Rechenergebnis eine BCD-Zahl erzeugt.

- **Fast Interrupt Mask** (Bit 6)
  Dieses Flag sperrt den Interrupteingang $\overline{\text{FIRQ}}$. Es wird gesetzt bei der Aktivierung einer der Unterbrechungsanforderungen $\overline{\text{RESET}}$, $\overline{\text{NMI}}$, SWI1 und $\overline{\text{FIRQ}}$, nicht jedoch bei SWI2 und SWI3.

- **Entire Flag** (Bit 7)
  Dieses Flag zeigt an, welche Register während des letzten Interrupts im Systemstack gerettet wurden. Ist Bit 7=1, wurden alle Register gerettet; ist es gleich 0, wurde nur das CC-Register und der Program Counter PC abgespeichert. Diese Information wird vom RTI-Befehl (*Return from Interrupt*) ausgewertet.

1.3.4 Die Adressierungsarten

Die Adressierungsarten des 6809 bezeichnen die verschiedenen Möglichkeiten, Operandenadressen, d.h. Befehls- und Datenadressen, aus den 16 bit langen Indexregistern X,Y, den Stackregistern U,S und dem Programmzähler PC zu bilden. Mit Hilfe des 8 bit langen Direct Page Registers DP kann für Kurzbefehle das höher-wertige Byte der Adresse fest eingestellt werden.

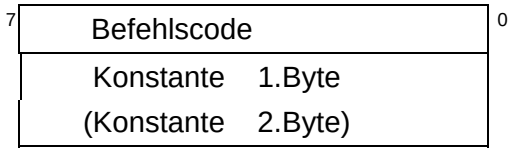
Praktische Übung P1.3-2:
Führen Sie die im folgenden als Beispiele angegebenen Befehle bzw. Befehlsfolgen im Praktikumsrechner selbst durch. Geben Sie dazu diese Befehle ab der Startadres-se $0400 ein und beenden Sie die Eingabe durch einen Sprung ins Monitorprogramm mit Hilfe des Software-Interrupts SWI1 ($3F). Schauen Sie sich mit der `Taste R` die Ergebnisse der Befehle in den Registern an und vergleichen Sie sie mit den adressierten Originalwerten im Speicher !
(Zur Initialisierung sind leider einige Register- und Speicherzellen-Zuweisungen nötig, um sinnvolle Ergebnisse zu erzielen. Diese Zuweisungen können Sie über die Tastatur vornehmen. Falls Sie auf die praktische Ausführung der Befehle verzichten, können Sie diese Zuweisungen außer acht lassen und nur die dunkel unterlegten Befehle betrachten.)

Der 6809 unterstützt die im folgenden aufgezählten Adressierungsarten.
(Dabei bedeuten immer: # = immediate ; $ = hex.)

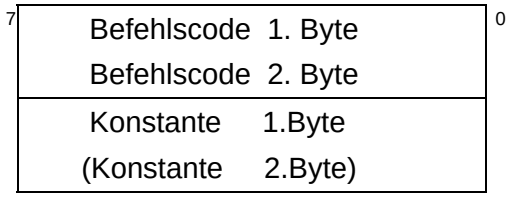
1. Unmittelbare Adressierung (*immediate addressing*)

Bei der unmittelbaren Adressierung folgt dem Befehlscode eine Konstante von 8 oder 16 bit Länge. Der Befehlscode ist meist 1 byte lang. Insbesondere für Befehle, die das Y-Register ansprechen, folgt aber ein 2. Befehlscode-Byte.

1. Variante:



2. Variante:



Beispiele:

LDA #$F0	86 F0	*Lade Akku A mit $F0*
LDX #$1234	8E 12 34	*Lade X-Reg. mit $1234*
LDY #$ABCD	10 8E AB CD	*Lade Y-Reg. mit $ABCD*
SWI1	3F	(Sprung in den Monitor)

Beim 6809 wird die Möglichkeit, ein oder mehrere Register direkt zu adressieren, zu der unmittelbaren Adressierung gezählt. Dies sind insbesondere die Stackbefehle PUSH/PULL und die Register-Transfer- und -Austausch-Befehle TFR und EXG. Diese Befehle haben folgendes Format:

Befehlscode
Postbyte

Die Bestimmung der Register wird im Postbyte vorgenommen. Dies hat den im folgenden Bild 1.3-5 dargestellten Aufbau. In der Tabelle 1.3-4 ist der Code für die Register des 6809 angegeben.

Befehle PUSH und PULL

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
PC	*	Y	X	DP	B	A	CC

PSHS/PULS: * = U PSHU/PULU: * = S

Befehle TFR und EXG

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Quellregister-Code				Zielregister-Code			

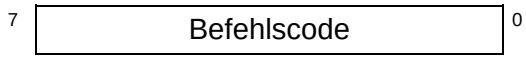
Bild 1.3-5: Der Aufbau des Postbytes für Register-Befehle

Tabelle 1.3-4: Die Codierung der 6809-Register

Register	Binärcode	Hex-Code
D	0000	0
X	0001	1
Y	0010	2
U	0011	3
S	0100	4
PC	0101	5
A	1000	8
B	1001	9
CC	1010	A
DP	1011	B

2. Implizite Adressierung (*inherent addressing*)

Diese Adressierung ist allein durch den Opcode bestimmt. Ein Operand wird nicht benötigt. Der Befehl ist 1 byte lang. (Als Ausnahmen umfassen SWI2 und SWI3 je 2 byte.) Die impliziten Befehle beziehen sich auf die Register A,B,X und auf die Sprungbefehle SWI1,SWI2,SWI3,RTI und RTS.



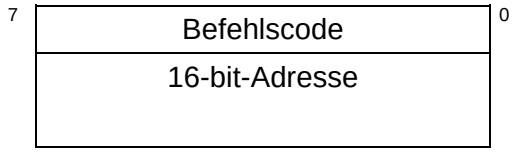
Beispiel:

DECA	4A

Erniedrige den Inhalt des Akkumulators A um 1.

3. Erweiterte Adressierung (*extended addressing*)

Bei dieser Adressierungsart folgt nach dem Opcode die vollständige Adresse. Der Befehl ist mindestens 3 Bytes lang.



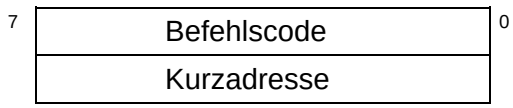
Beispiel:

LDA $1234	B6	12 34	Opcode, Adresse
SWI1	3F		(zum Monitor)

Der Akkumulator A wird aus der Speicherzelle $1234 geladen.

4. Direkte Adressierung (*direct addressing*)

Im Falle der direkten Adressierung folgt dem Befehlscode eine Adreßangabe mit 8-bit-Wortlänge. Durch diese Adressierung werden die Adressen $XX00 bis $XXFF direkt angesprochen, wobei XX den Inhalt des DP-Registers bezeichnet. Das DP-Register läßt sich mit dem Transfer-Befehl TFR bzw. Exchange-Befehl EXG laden:



Beispiele:

1. (Das DP-Register enthalte den Wert $00).

LDA <$EB	96	EB	OpCode, Kurzadresse
SWI1	3F		(zum Monitor)

Lade Akkumulator A mit dem Wert der Zelle $00EB

2.

```
LDB  #$1A          C6   1A        (DP auf $1A
TFR  B,DP          1F   9B         setzen)
LDA  <$EB          96   EB        OpCode,  Kurzadresse
.....
LDB  #0            C6   00        (Restaurieren des DP-Registers
TFR  B,DP          1F   9B         für den Monitor)
SWI1               3F             (zum Monitor)
```

Lade den Akkumulator A mit dem Wert der Speicherzelle $1AEB.

Um diesen Befehl direkt ausführen zu können, muß das DP-Register mit dem H-Byte der Adresse geladen werden. In diesem Beispiel wurde der Transfer-Befehl TFR benutzt. Der Akkumulator B wird zunächst mit dem H-Byte $1A geladen (LDB #$1A). Danach folgt der Transfer ins DP-Register (TFR B,DP). Abschließend wird das DP-Register für das Monitorprogramm restauriert.

Selbsttestaufgabe S1.3-1

Schreiben Sie eine Programmsequenz, die folgende Funktion erfüllt:
Lade Akkumulator B mit dem Wert der Zelle $1015.
Benutzen Sie die direkte Adressierung und den TFR-Befehl bzw. den EXG-Befehl.

Musterlösung zur Selbsttestaufgabe S1.3-1

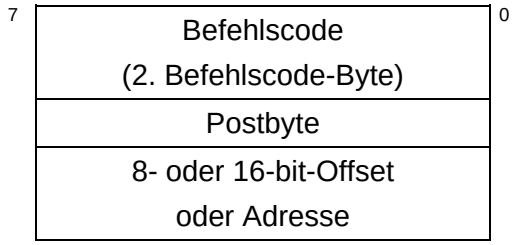
```
LDA  #$10          86   10
TFR  A,DP          1F   8B        bzw.  EXG A,DP     1E   8B
LDB  <$15          D6   15                       OpCode, Kurzadresse
LDA  #0            86   00
TFR  A,DP          1F   8B
SWI1               3F
```

5. Indizierte Adressierung (*indexed addressing*, relative Adressierung)

Die effektive Adresse wird durch die Addition des Inhalts eines der Zeigerregister X,Y,S,U mit dem Offset gebildet. Der Offset ist im Zweierkomplement angegeben. Er läßt sich in drei Wertebereiche einteilen und ist vorzeichenbehaftet:

Offset-Länge in bit	hexadezimal	dezimal
5	$10 - $0F	-16 - +15
8	$80 - $7F	-128 - +127
16	$8000 - $7FFF	-32768 - +32767

Ein Befehl besteht aus dem Opcode, dem Postbyte und (gegebenenfalls) dem Offset. (In Ausnahmefällen besteht der Opcode auch aus 2 byte).



Mit dem Postbyte wird das verwendete Pointer Register X,Y,U,S ausgewählt. Im Postbyte sind für den Offset 5 Bits reserviert, wobei das 5. Bit das Vorzeichen des Offsets angibt. Für den Bereich -16 bis +15 ist so kein zusätzliches Befehlsbyte erforderlich. Der Offset kann selbst 2 byte lang sein. Wird ein Registerinhalt als Offset verwendet, so wird auch dieser als vorzeichenbehaftete Zahl interpretiert. Im Bild 1.3-6 ist der Aufbau des Postbytes für die indizierte Adressierung dargestellt. Für die zwei Postbyte-Bits rr ist die im Bild 1.3-6 angegebene Ersetzung vorzunehmen

a) **kein 5-bit-Offset**

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	r	r	d/i	Adressierungsart			

d/i=0: direkt, d/i=1: indirekt

b) **5-bit-Offset**

0	r	r	n	n	n	n	n

nnnnn: 5-bit-Offset im Zweierkomplement

Register	rr
X	00
Y	01
U	10
S	11

Bild 1.3-6: Der Aufbau des Postbytes für die indizierte Adressierung

Die folgende Tabelle 1.3-5 zeigt die verschiedenen Möglichkeiten der indizierten Adressierung. Dabei ist für R wiederum eines der vier Index-/Stackregister X,Y,S,U einzusetzen. Die Codierung der beiden Bits rr finden Sie im obenstehenden Bild 1.3-6. (Zur Erleichterung der Programmierarbeit finden Sie im Anhang A Tabellen, in denen für alle vier Register diese Ersetzungen bereits durchgeführt wurden.)

Tabelle 1.3-5: Die verschiedenen indizierten Adressierungsarten

Lfd.	Offset	Direkt		X	+	Indirekt		X	+
Nr.		Assembler	Postbyte (Hex)	~	#	Assembler	Postbyte (Hex)	~	#
1	kein Offset	,R	1rr00100	0	0	[,R]	1rr10100	3	0
2	5-Bit*	n,R	0rrnnnnn	1	0	-	8-Bit Offset benutzen	/	/
3	8-Bit	n,R	1rr01000	1	1	[n,R]	1rr11000	4	1
4	16-Bit	n,R	1rr01001	4	2	[n,R]	1rr11001	7	2
5	A-Register	A,R	1rr00110	1	0	[A,R]	1rr10110	4	0
6	B-Register	B,R	1rr00101	1	0	[B,R]	1rr10101	4	0
7	D-Register	D,R	1rr01011	4	0	[D,R]	1rr11011	7	0
8	Inc. um1	,R+	1rr00000	2	0	nicht	erlaubt	/	/
9	Inc. um 2	,R++	1rr00001	3	0	[,R++]	1rr10001	6	0
10	Dec. um 1	,-R	1rr00010	2	0	nicht	erlaubt	/	/
11	Dec. um 2	,--R	1rr00011	3	0	[,--R]	1rr10011	6	0
12	PCR 8-Bit	n,PCR	1rr01100 1)	1	1	[n,PCR]	1rr11100 2)	4	1
13	PCR 16-Bit	n,PCR	1rr01101 1)	5	2	[n,PCR]	1rr11101 2)	8	2
14	Extended Indirekt	-	-	-	-	[n]	1rr11111 2)	5	2

* Der Offset ist im Postbyte enthalten, deshalb ist das Postbyte binär dargestellt. An der Stelle nnnnn wird der Offset im Binärcode gesetzt und dann in den Hex-Code umgeformt.
1) 1. Tetrade auch: 8, A, C, E
2) 1. Tetrade auch: 9, B, D, F

In der Tabelle wird zwischen der direkt- und der indirekt-indizierten Adressierung unterschieden.

Beispiel zur direkt-indizierten Adressierung:

	X := $1000	(Initialisierung)
LDA 10,X	A6 0A	OpCode, Postbyte
SWI1	3F	(zum Monitor)

Lade den Inhalt der Speicherzelle, deren Adresse sich aus der Summe des Inhalts des X-Registers mit dem Offset (hier dezimal 10) ergibt, in den Akkumulator A.
Der Offset wird hier im Postbyte angegeben, da er im 5-bit-Bereich liegt. (Zur Initialisierung wurde das X-Register mit $1000 geladen.)

Beispiel zur indirekt-indizierten Adressierung:

	X := $1000	(Initialisierung)
	($100A,$100B):=$0400	(Initialisierung)
LDA [10,X]	A6 98 0A	OpCode, Postbyte, Offset
SWI1	3F	(zum Monitor)

Die Summe des X-Registers mit dem Offset ergibt die Adresse einer Speicherzelle, in der das H-Byte der Adresse der in den Akkumulator A zu ladenden Speicherzelle steht. Das L-Byte der Adresse steht in der folgenden Speicherzelle. Hier kann der Offset 10 (=$A) nach Tabelle 1.3-5 nicht im Postbyte untergebracht werden. (Zur Initialisierung wurde der Wert $1000 ins X-Register und der Wert $0400 in die Speicherzellen $100A,$100B gebracht.)

Beispiel für einen 4-Byte-Befehl:

	X := $1000	(Initialisierung)
LDY 20,X	10 AE 88 14	Opcode, Postbyte, Offset
SWI1	3F	(zum Monitor)

Lade das Y-Register mit dem Inhalt der Speicherzellen, die durch X+20 und X+20+1 angegeben werden. (Zur Initialisierung wurde X auf den Wert $1000 gesetzt.)

Beispiel für einen 5-Byte-Befehl:

	X := $1000	(Initialisierung)
	Y := $1234	(Initialisierung)
STY [501,X]	10 AF 99 01 F5	Opcode, Postbyte, Offset
SWI1	3F	(zum Monitor)

Der Inhalt des Y-Registers ($1234) soll in zwei hintereinanderfolgenden Speicherzellen ($11F5,$11F6) abgelegt werden. Die Adresse der 1. Speicherzelle steht in der Speicherzelle (H-Byte), die durch die Summe aus dem X-Register und dem Offset adressiert wird, sowie in der darauf folgenden Speicherzelle (L-Byte). (Zur Initialisierung wurden die Register X,Y auf geeignete Werte gesetzt.)

Bei den bisherigen Befehlen wurde ein spezieller Offset angeben. Als nächstes werden einige Beispiele gezeigt, bei denen die Akkumulatoren die Offsets liefern. Die effektive Adresse wird wie bei den bisherigen Offset-Typen gebildet. Ein zusätzliches Byte wird nicht benötigt, da in dem Postbyte die Register ausgewählt werden.

Beispiele:

Y := $1000	B := $10	(Initialisierung)
LDA B,Y	A6 A5	Opcode, Postbyte
SWI1	3F	(zum Monitor)

Der Inhalt derjenigen Speicherzelle, deren Adresse durch die Summe aus den Inhalten von B und Y gebildet wird, soll in den Akkumulator A geladen werden.

X := $1000	D := $0500	(Initialisierung)
STD [D,X]	ED 9B	Opcode, Postbyte
SWI1	3F	(zum Monitor)

Der Inhalt des Akkumulators D wird in zwei hintereinanderliegenden Speicherzellen abgelegt. Die Adresse der 1. Zelle steht in der durch die Addition des Akkumulators D zum X-Register adressierten und der folgenden Speicherstelle. (Zur Vorberei-tung wurden die Register X, D geeignet initialisiert.)

5a. Indizierte Adressierung mit „Autoinkrement/Autodekrement“:

Autoinkrement (Postinkrement)
Nach Selektierung der Adresse des Operanden wird der Wert des benutzten Regi-sters automatisch um 1 bzw. 2 erhöht.
Beispiel:

	D := $1234	(Initialisierung)
	U := $1000	(Initialisierung)
STD ,U++	ED C1	Opcode, Postbyte
SWI1	3F	(zum Monitor)

\+ bedeutet „+1“,
++ bedeutet „+2“.

Speichere den Inhalt des Akkumulators D auf dem Speicherplatz, der durch den Stack Pointer U angegeben wird, und erhöhe danach den Inhalt des Stack Pointers U um 2.

Autodekrement (Prädekrement)
Der Inhalt des ausgewählten Registers wird zunächst um 1 bzw. 2 erniedrigt. Die erniedrigte Adresse ist dann die effektive Adresse.

Beispiel:

	X := $1000	(Initialisierung)
LDA ,-X	A6 82	Opcode, Postbyte
SWI1	3F	(zum Monitor)

\- bedeutet „-1“,
-- bedeutet „-2“.

Lade den Inhalt derjenigen Speicherzelle in den Akkumulator A, deren Adresse durch den vorher um 1 verminderten Wert des X-Registers angegeben wird.

Eine weitere Möglichkeit der indizierten Adressierung besteht in der relativen Adressenbildung mit Hilfe des Program Counters (PC-relative Adressierung, s. auch nächste Seite). Durch Addition des Program Counters mit einem vorzeichenbehafteten 8- bzw. 16-bit-Offset wird die Adresse gebildet. Hierzu werden bei einem 8-bit-Offset ein zusätzliches Byte, bei einem 16-bit-Offset zwei zusätzliche Bytes benötigt.

Beispiel:

SUBD $1000,PC	A3 8D 10 00,	Opcode, Postbyte, Offset
SWI1	3F	(zum Monitor)

Vom Inhalt des Akkumulators D wird der Inhalt derjenigen Speicherzelle subtrahiert, deren Adresse durch die Addition des Offsets zum PC gegeben ist.

Die erweiterte (*extended*) indirekte Adressierung ermöglicht eine relative Adressierung mit Hilfe von 2 zusätzlichen Bytes.

Beispiel:

	($1234,$1235):=$1000	(Initialisierung)
LDB [$1234]	E6 9F 12 34	Opcode, Postbyte, Offset
SWI1	3F	(zum Monitor)

Lade den Inhalt der Speicherzelle, deren Adresse in der Speicherzelle $1234, $1235 steht, in den Akkumulator B.

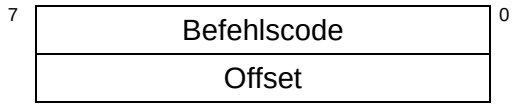
6. PC-Relative Adressierung bei Verzweigungsbefehlen

Die PC-relative Adressierung wird beim 6809 insbesondere für Verzweigungsbefehle benötigt. Die Darstellung des Offsets geschieht im Zweierkomplement, wobei das höchstwertige Bit das Vorzeichen festlegt: „0“ = „+“, „1“ = „-“. Bei einigen Prozessoren wird bei der PC-relativen Adressierung mit einem Offset von einem Byte gearbeitet, so daß Sprünge im Bereich von +127 und -128 möglich sind. Mit dem 6809 ist es jedoch möglich, zu jeder Adresse im Speicher zu verzweigen. Hierzu werden der Opcode sowie der Offset auf jeweils 2 Bytes erweitert, so daß sich insgesamt 4-byte-Befehle ergeben. Der Sprungbereich beträgt somit -32768 bis 32767.

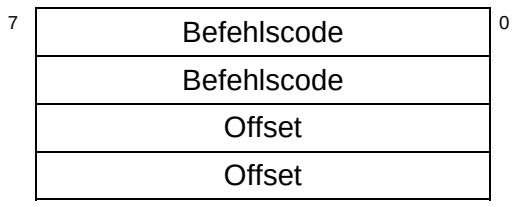
Wichtig:

Ausgangspunkt für die Berechnung des Offsets ist immer die Adresse des OpCodes des 1. Befehls nach dem Verzweigungsbefehl! Die Distanz ergibt sich als Differenz zur Adresse, unter der der OpCode des Zielbefehls steht.

Short Branch:



Long Branch:



Beispiel:

BMI -12	2B F4	Rücksprung um -12
LBMI 2020	10 2B 07 E4	Vorwärtssprung um +2020

Ist im Statusregister das N-Flag (*Negative Flag*) gesetzt, so werden diese relativen Sprünge ausgeführt. Beim BMI wird der PC um $F4 erniedrigt, beim LBMI um $07E4 erhöht.

Das folgende Bild 1.3-6 zeigt die verschiedenen Möglichkeiten der Adressenbildung im Zusammenhang. Darin bedeuten:

EA effektive Adresse, DP Direct Page Register,
A Akkumulator A, X Indexregister
und die Nummern 1- 5 bezeichnen die folgenden Befehle:

1	LDA #$30	
2	LDA <$30	: EA=256*DP+$30
3	LDA 18,X	: EA=$AF00+$12
4	LDA $8FD6	: EA=$8FD6
5	LDA [10,X]	: EA=[$AF00+$A]*256+[$AF00+$A+1]

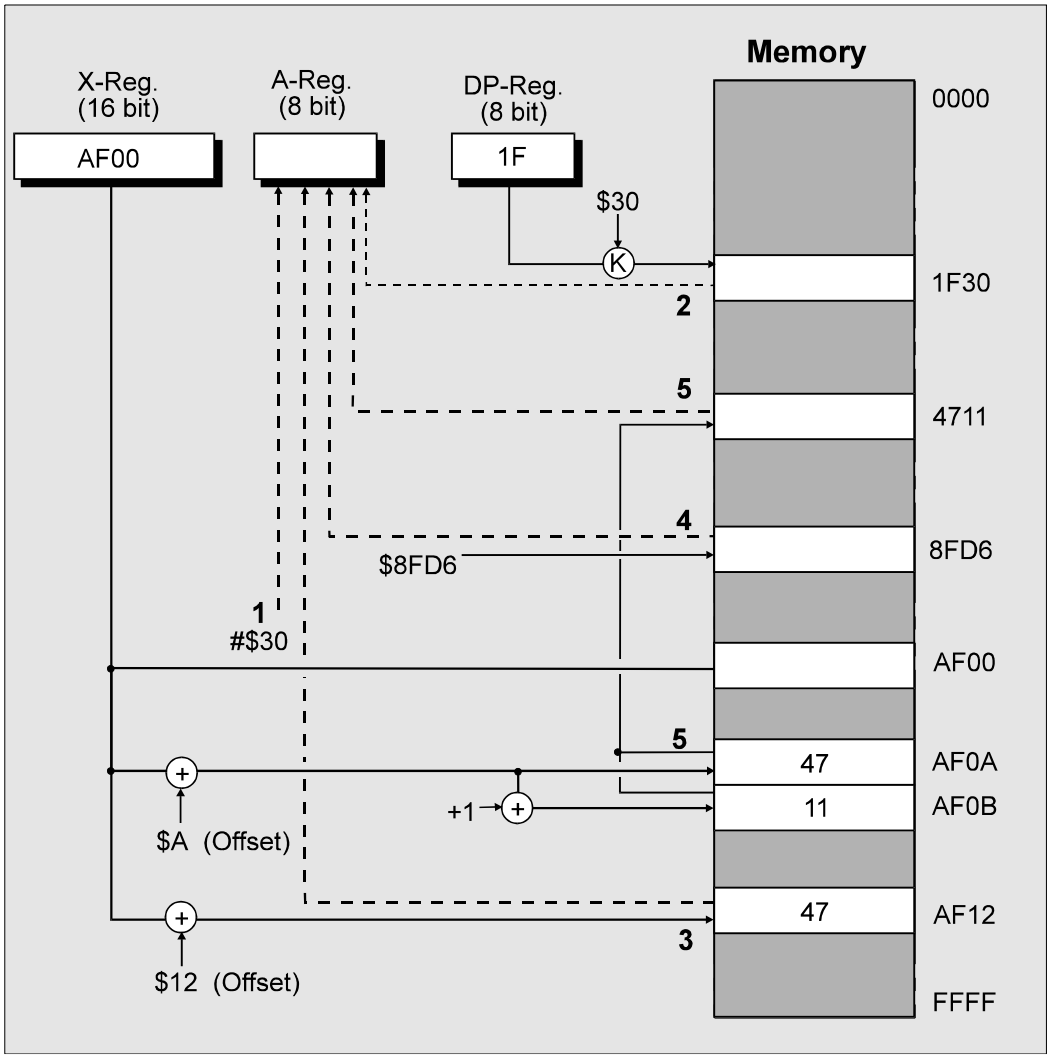


Bild 1.3-6: Gesamtübersicht zur Adressenbildung

1.3.5 Der Befehlssatz

Der Befehlssatz des 6809 wird in diesem Abschnitt nur kurz anhand einiger Beispiele beschrieben, die das Arbeiten mit den im Anhang B aufbereiteten Tabellen erleichtern sollen. Grundsätzliches über Mikroprozessor-Befehlssätze finden Sie im Band I des Buches über Mikrorechner-Technik. Die genaue Funktion der Befehle sollten Sie durch Ausprobieren und Analyiseren der Ergebnisse erkennen.

In den Tabellen im Anhang finden Sie – matrixförmig angeordnet – zeilenweise die verschiedenen Befehle, in den Spalten die unterstützten Adressierungsarten eingetragen. Die fast vollständig ausgefüllten Tabellen zeigen einen der Punkte, die den 6809 zum sicher besten 8-bit-Prozessor machen: Der Befehlssatz ist nahezu **orthogonal**.

Für jede Spalte, also jede Adressierungsart, ist in der 1. Unterspalte der OpCode eingetragen. Er ist für die meisten Befehle 1 byte lang. Hauptsächlich für Befehle, die die Register Y,U,S und z.T. auch D betreffen, ist der OpCode 2 byte lang. Das 1. Byte ist dabei stets $10 oder $11[2]. In der 2. Unterspalte, gekennzeichnet durch „~“, ist die Anzahl der Maschinenzyklen angegeben, die die Befehlsabarbeitung dauert. Die 3. Unterspalte gibt die Anzahl der Bytes an, die der Befehl im Speicher belegt. In beiden Fällen muß für die indizierte Adressierungart noch die Anzahl der Maschinenzyklen und Befehlsbytes hinzugezählt werden, die in der Tabelle 1.3-5 angegeben sind. Die vorletzte Spalte gibt für jeden Befehl eine kurze, formelmäßige Beschreibung, bei den Schiebe- und Rotationsbefehlen eine kleine Skizze. In der letzten Spalte wird angezeigt, wie jeder Befehl auf die Flags im Statusregister wirkt. Dabei wird insbesondere unterschieden, ob das Flag unverändert bleibt („•“), seinen Zustand ändert („|“) oder auf „0“ gesetzt wird. Diese Angaben müssen Sie bei den verlangten Programmen sehr sorgfältig beachten, da sonst „Seiteneffekte“ drohen[3].

Bei der folgenden Beschreibung der Befehle wird die alphabetische Reihenfolge der Befehle genommen. Ausgenommen davon sind die Branch-Befehle, die am Ende dieses Abschnittes beschrieben werden. (Durch das Zeichen "/" werden in der folgenden Beschreibung die verschiedenen Befehls- und Registervarianten voneinander getrennt.)

Praktische Übung P1.3-3:

Führen Sie die folgenden Befehle (oder wenigstens einige von Ihnen) mit Ihrem Praktikumsrechner durch. Initialisieren Sie die benutzten Register durch geeignete Parameter und benutzen Sie die Trace-Funktion (`Taste T`) zur Einzelschrittausführung der Befehle!

[2] Diese Befehle und die angesprochenen Register sind z.T. Ergänzungen, die im Vorgängerprozessor 6800 nicht vorhanden waren. Die im 6800 „freien“ OpCodes $10, $11 wurden zur Implementierung der neuen Befehle benutzt.

[3] Z.B.: INCA verändert nicht das Carry Flag! Eine Schleifenendabfrage darf sich daher nicht auf das C-Flag abstützen.

ABX (*Add Accumulator B into Index Register X*)

Der Inhalt des Akkumulators B wird vorzeichenlos zum Inhalt des X-Registers addiert. Das Ergebnis steht im X-Register. Das CC-Register wird durch diese Operation nicht beeinflußt.

```
ABX          3A
```

ADCA/ADCB (*Add with Carry into Register A/B*)

Zum Inhalt des Akkumulators A bzw. B wird das Carry Flag und der Inhalt einer Speicherzelle oder eine Konstante addiert. Das Ergebnis wird in den Akkumulator A bzw. B geschrieben.

```
ADCA   #$10     89 10
ADCB   10,X     E9 0A
```

ADDA/ADDB/ADDD (*Add Memory into Register A/B/D*)

Zum Inhalt des ausgewählten Registers wird der Inhalt einer Speicherzelle oder eine Konstante addiert. Beim 16-bit-Akkumulator D sind es zwei aufeinander folgende Speicherzellen bzw. eine 2-byte-Konstante.

```
ADDA   #$10     8B 10
ADDB   10,X     EB 0A
```

ANDA/ANDB/ANDCC (*Logical AND Memory into Register A/B/CC*)

Der Inhalt von A bzw. B wird mit dem Inhalt einer Speicherzelle oder eine Konstante (logisch) UND-verknüpft. Das Ergebnis wird in A bzw. B geschrieben. Das Carry Flag und das Halfcarry Flag werden nicht beeinflußt. Das Overflow Flag wird gelöscht.

Beim CC-Register kann eine Verknüpfung ausschließlich mit einer Konstanten erfolgen. Das Ergebnis der Operation steht im CC-Register. Dies ist die einzige Möglichkeit beim 6809, bestimmte Flags des CC-Registers zu setzen, rückzusetzen oder abzufragen. (Spezielle Flag-Befehle wie bei anderen Prozessoren gibt es hier nicht.)

```
ANDA   $EB      94 EB
ANDB   $1243    F4 12 43
ANDCC  #$AA     1C AA
```

ASL/ASLA/ASLB (*Arithmetic Shift Left*)

Jedes Bit der Speicherzelle bzw. des Akkumulators A/B wird um eine Position nach links verschoben. Bit 0 wird auf '0' gesetzt. Bit 7 wird ins Carry Flag geschoben. Das Overflow Flag erhält das Ergebnis der XOR-Verküpfung von Bit 6 und Bit 7. Das Carry Flag erhält den ursprünglichen Wert von Bit 7.

```
ASLA           48
ASLB           58
ASL    [10,Y]  68 B8 0A
```

ASR/ASRA/ASRB (*Arithmetic Shift Right*)

Jedes Bit der Speicherzelle bzw. des Akkumulators A/B wird um eine Position nach rechts verschoben. Bit 0 wird ins Carry Flag geschoben. Bit 7 bleibt unverändert:

ASRA		47
ASRB		57
ASR	$1234	77 12 34

BITA/BITB (*Bit Test*)

Der Inhalt von A/B wird mit dem Inhalt einer Speicherzelle oder eine Konstante logisch bitweise UND verknüpft, wobei die Inhalte des Akkumulators A/B und der Speicherzelle nicht verändert werden. Das Ergebnis beeinflußt nur das CC-Register. Das Overflow Flag wird gelöscht. Carry Flag und Halfcarry Flag werden nicht geändert.

BITA	#$55	85 55
BITB	$EE	D5 EE

CLR/CLRA/CLRB (*Clear*)

Der Inhalt der Speicherzelle bzw. des Akkumulators A/B wird gelöscht, d.h. auf den Wert $00 gesetzt. Das Zero Flag wird gesetzt. Negative, Overflow und Carry Flag werden gelöscht.

CLRA		4F
CLRB		5F
CLR	13124,Y	6F A9 33 44

CMPA/CMPB/CMPD/CMPS/CMPU/CMPX/CMPY (*Compare Memory*)

Der Inhalt von A/B/D/S/U/X/Y wird mit dem Inhalt einer bzw. zweier Speicherzelle(n) oder eine 1-byte- bzw. 2-byte-Konstante verglichen. Der Vergleich wird durch eine Subtraktion durchgeführt. Die Register bzw. die Speicherzelle(n) wer-den nicht verändert. Abhängig vom Ergebnis der Subtraktion werden die entsprechenden Flags im CC-Register gesetzt**.** Dieser Befehl dient in der Regel dazu, die Vorausetzungen für eine bedingte Verzweigung abzuprüfen.

CMPA	#$AA	81 AA
CMPB	$AA	D1 AA
CMPD	13124,S	10 A3 E9 33 44
CMPS	10,U	11 AC 4A
CMPU	$1234	11 B3 12 34
CMPX	[13124,U]	AC D9 33 44
CMPY	[D,U]	10 AC DB

COM/COMA/COMB (*Complement*)

Der Inhalt der Speicherzelle bzw. des Akkumulators A/B wird durch das Einerkomplement ersetzt. Das Overflow Flag wird gelöscht, und das Carry Flag wird gesetzt.

COMA		43

```
COMB            53
COM     $1234   73 12 34
```

CWAI (*Clear CC-Flags and Wait for Interrupt*)

Das CC-Register wird mit der nachfolgenden Konstanten logisch UND-verknüpft. Das Ergebnis wird in das CC-Register geschrieben. Anschließend wird das E-Flag gesetzt und auf einen Hardware-Interrupt gewartet. Bis zur Interruptauslösung wird kein weiterer Befehl abgearbeitet.

```
CWAI    3C xx   xx
                :   FF :  FIRQ und IRQ  gesperrt
                    EF :  FIRQ          gesperrt
                    BF :  IRQ           gesperrt
                    AF :  kein Interrupt gesperrt
```

DAA (*Decimal Addition Adjust*)

Da bei arithmetischen Operationen mit BCD-Zahlen unerwünschte Ziffern zwischen $A und $F entstehen können, kann mit Hilfe des DAA-Befehls eine Korrektur die-ser Zahlen vorgenommen werden, so daß im Akkumulator A wieder eine gültige BCD-Zahl steht. Beispiel:

```
LDA     #$67    86 67   BCD-Zahl 67
ADDA    #$75    8B 75   BCD-Zahl 75
                        „falsches“ Ergebnis = $DC
DAA             19      „korr.“ Ergebnis = 142
                        C-Flag = 1, Akku A = $42
```

DEC/DECA/DECB (*Decrement*)

Der Inhalt der Speicherzelle bzw. des Akkumulators A/B wird um 1 erniedrigt. Das Carry Flag und das Half-Carry Flag werden nicht beeinflußt. Das Overflow Flag wird nur gesetzt, wenn der Inhalt vorher $80 war, da der neue Wert $7F einen Vor-zeichenwechsel (zum „+“) bedeutet.

```
DECA            4A
DECB            5A
DEC     $AA     0A AA
```

EORA/EORB (*Logical EXOR Memory into Register*)

Der Inhalt des Akkumulators A/B wird mit dem Inhalt einer Speicherzelle oder eine Konstante logisch bitweise exklusiv-ODER-verknüpft. Das Ergebnis steht in Akku-mulator A/B.

```
EORA    #$FF    88 FF
EORB    $1234   F8 12 34
```

EXG (*Exchange Register*)

Der Befehl ermöglicht den Austausch zweier Registerinhalte, wobei ein Austausch nur unter Registern gleicher Länge zulässig ist. Die Register werden durch das Postbyte ausgewählt. Die Bits 0-3 enthalten die Kennung des zweiten, die Bits 4-7 die des ersten Registers nach folgender Codierung (vgl. Tabelle 1.3-4):

A	= 1000,	PC	= 0101,
B	= 1001,	U	= 0011,
D	= 0000,	S	= 0100,
X	= 0001,	CC	= 1010,
Y	= 0010,	DP	= 1011,

EXG	A,B	1E 89	($89=10001001)

INC/INCA/INCB (*Increment*)

Der Inhalt einer Speicherzelle bzw. des Akkumulators A/B wird um 1 erhöht. Das Overflow Flag wird gesetzt, wenn die Zahl vorher den Wert $7F hatte, da der erhöhte Wert $80 einen Vorzeichenwechsel (zum „-“) darstellt.

INCA		4C
INCB		5C
INC	10,X	6C 0A

JMP (*Jump*)

Der Program Counter PC wird mit der effektiven Adresse geladen, die sich aus den dem Opcode folgenden 1-3 Bytes ergibt. Dadurch wird ein unbedingter Sprung zu einer neuen Programmstelle durchgeführt.

			Sprungadresse
JMP	$50	0E 50	DP*256+$0050
JMP	$1234	7E 1234	$1234
JMP	A,X	6E 86	X+A

JSR (Jump to Subroutine)

Der JSR-Befehl erwirkt einen Sprung in ein Unterprogramm, wobei die Rücksprungadresse im Systemstack abgespeichert wird. Besonders wichtig ist dabei die Möglichkeit der indizierten Adressierung, auch der indirekten indizierten Adressierung. Hierdurch ist es insbesondere möglich, zur Laufzeit eines Programmes z.B. Tabellen von Einsprungadressen der Unterprogramme zu berechnen.

JSR	$1234	BD 12 34
JSR	A,S	AD E6
JSR	[$6F,PCR]	AD 9C 6F

LDA/LDB/LDD/LDS/LDU/LDX/LDY (*Load Register from Memory*)

Das Register A/B wird mit dem Inhalt einer Speicherzelle oder mit einer Konstanten geladen. Bei den Registern D/S/U/X/Y wird der Inhalt zweier Speicherzellen oder eine 2-byte-Konstante geladen. Das Overflow Flag wird gelöscht.

LDA	#$FF	86 FF
LDB	$AA	D6 AA
LDD	10,X	EC 0A
LDS	#$1234	10 CE 12 34
LDU	#$11	CE 11
LDX	$0A	9E 0A
LDY	[13124,X]	10 AE 99 33 44

LEAS/LEAU/LEAX/LEAY (*Load Effective Address*)

Es wird die effektive Adresse einer Speicherzelle, die durch den Offset und den Inhalt eines Registers nach der im Postbyte bestimmten Adressierungsart gebildet wird, in das angegebene Register geladen. Dieser Befehl erlaubt es, zur Laufzeit eines Programmes die Adresse eines Operanden zu bestimmen und weiter zu verarbeiten. Sehr wichtig sind auch die beiden erstgenannten Beispielbefehle zum Aufbau von Schleifenvariablen. (Aber Vorsicht: LEAX ,X+ tut nicht, was er soll !)

LEAX	1,X	30 01
LEAY	,-Y	31 A2
LEAS	,X	32 84
LEAU	B,X	33 85
LEAX	[13124,Y]	30 B9 33 44
LEAY	[D,S]	31 FB

LSL/LSLA/LSLB (*Logical Shift Left*)

Jedes Bit der Speicherzelle bzw. des Akkumulators A/B wird um eine Position nach links verschoben. Bit 0 wird auf '0' gesetzt, und Bit 7 wird ins Carry Flag geschoben. Das Overflow Flag nimmt den Wert der EXOR-Verknüpfung von Bit 7 und Bit 6 an.

LSLA		48
LSLB		58
LSL	$12	08 12

LSR/LSRA/LSRB (*Logical Shift Right*)

Jedes Bit der Speicherzelle bzw. des Akkumulators A/B wird um eine Position nach rechts verschoben. Bit 7 wird auf '0' gesetzt, und Bit 0 wird ins Carry Flag übertragen.

LSRA		44
LSRB		54
LSR	20,X	64 88 14

MUL (*Multiply*)

Der Inhalt von Akkumulator A wird mit dem Inhalt von Akkumulator B multipliziert. Das Ergebnis wird in D geschrieben (A höherwertiges, B niederwertiges Byte). Das Carry Flag erhält den Inhalt des Bits 7 von B nach Beendigung der Operation.

MUL		3D

NEG/NEGA/NEGB (*Negate*)

Der Inhalt der Speicherzelle bzw. des Akkumulators A/B wird durch sein Zweierkomplement ersetzt. Das Overflow Flag wird nur gesetzt, wenn der Inhalt vor Ausführung $80 war. Das Carry Flag wird nur dann nicht gesetzt, wenn der Inhalt vor der Ausführung $00 war.

NEGA		40
NEGB		50
NEG	$AA	00 AA

NOP (*No Operation*)

Der Program Counter wird um 1 erhöht. Alle Bedingungs-Flags bleiben unverändert.[4]

NOP		12

ORA/ORB/ORCC (*Logical OR into Register*)

Der Inhalt des Registers A/B wird mit dem Inhalt einer Speicherzelle oder eine Konstante, der Inhalt des CC-Registers nur mit einer Konstanten logisch ODER-verknüpft. Die Verknüpfung wird bitweise vorgenommen. Das Ergebnis steht im angegebenen Register. Das Overflow Flag wird gelöscht.

ORA	$AA	9A AA
ORB	$1234	FA 12 34
ORCC	#$11	1A 11

PSHS/PSHU (*Push Register on Stack*)

PULS/PULU (*Pull Register from Stack*)

Alle im Postbyte angegebenen Register werden im Stack gespeichert bzw. aus dem Stack gelesen. Beim Abspeichern wird zunächst das betreffene Stackregister S oder U um 1 erniedrigt (prädekrement). Mit dieser Adresse beginnt die Abspeicherung in der Reihenfolge, daß zunächst das höchste Bit (Bit 7) abgefragt wird und das zugehörige Register ggf. in den Stack gebracht wird.

Das Einlesen vom Stack erfolgt in umgekehrter Reihenfolge, d.h. es wird zunächst das Bit 0 geprüft, das entsprechende Register ggf. geladen und das Stackregister erhöht (postdekrement). Danach wird mit den anderen Registern in aufsteigender Bitfolge des Postbytes weitergemacht.

[4] Da der Befehl Speicherplatz und Ausführungszeit benötigt, ist er ein einfaches Mittel, Platz für später einzufügende Befehle frei zu halten oder Zeitverzögerungen geringer Dauer zu erzeugen.

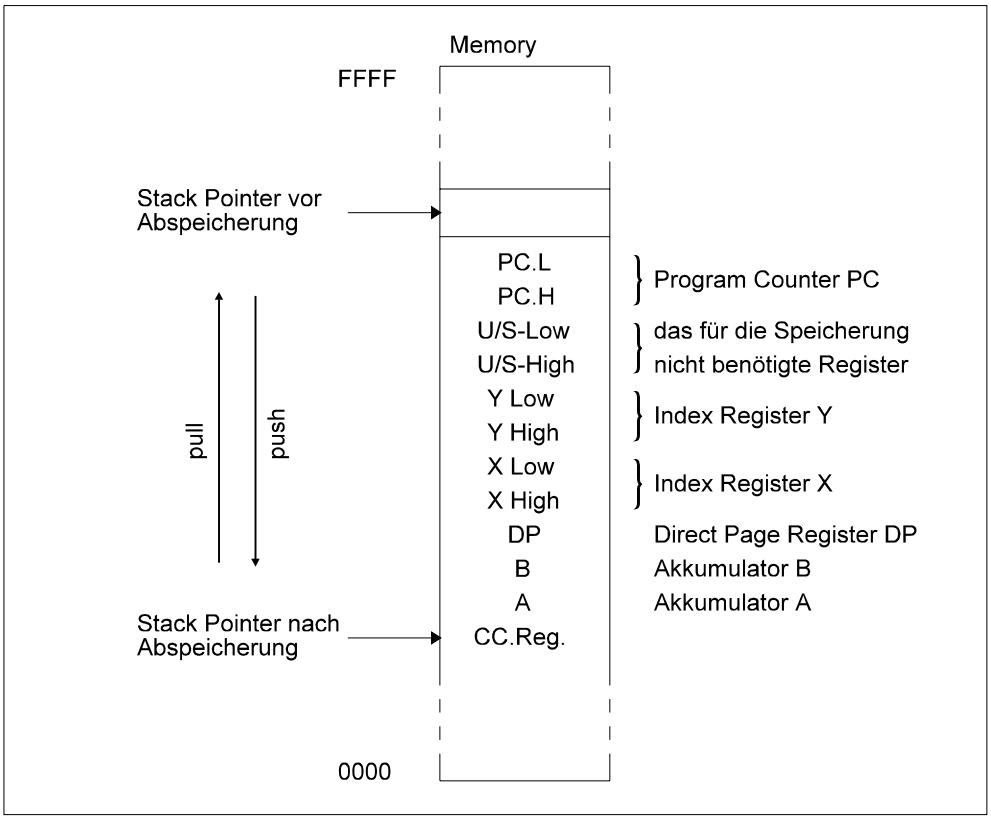
Für beide Stacks U bzw. S gibt es ein eigenes Paar von Befehlen. Das dem Bit 6 im Postbyte zugeordnete Register ist jeweils das Stackregister des durch den Befehl nicht angesprochenen Stacks.

Push/Pull Postbyte:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
PC	*	Y	X	DP	B	A	CC

PSHS/PULS: *=U PSHU/PULU: *=S

PSHU	#$88	36 88	(10001000)
PULU	#$88	37 88	(10001000)
PSHS	#$FF	34 FF	(11111111)
PULS	#$FF	35 FF	(11111111)



ROL/ROLA/ROLB (*Rotate Left*)

Jedes Bit der Speicherzelle bzw. des Akkumulators A/B wird um eine Position nach links verschoben. Bit 7 wird ins Carry Flag, das Carry Flag in Bit 0 geschoben. Das Overflow Flag erhält die EXOR-Verknüpfung der ursprünglichen Werte von Bit 7 und Bit 6.

ROLA		49
ROLB		59
ROL	$1F	09 1F

ROR/RORA/RORB (*Rotate Right*)

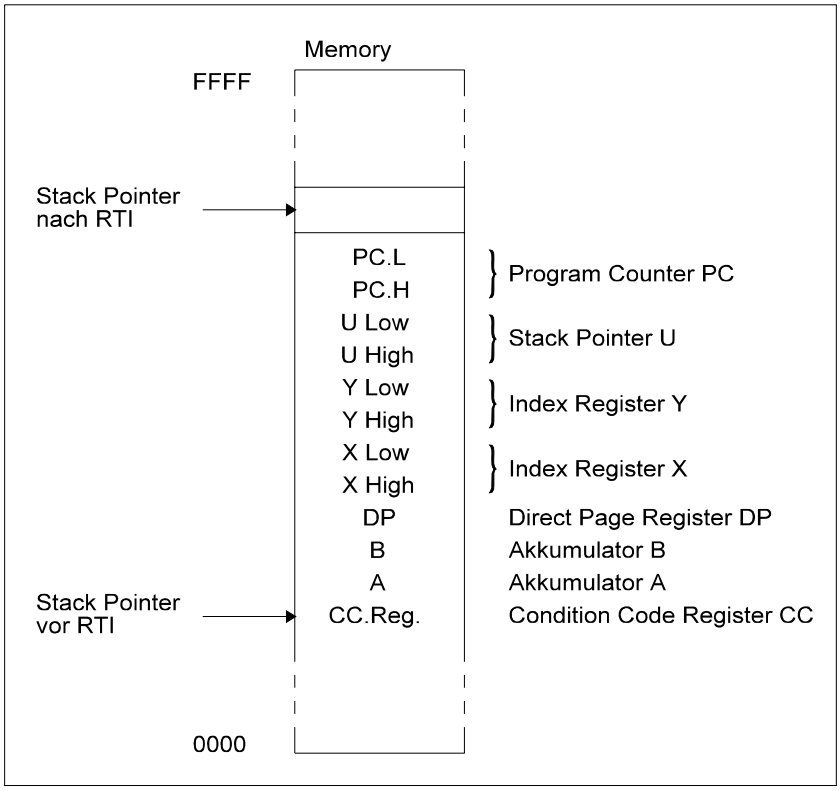
Jedes Bit der Speicherzelle bzw. des Akkumulators A/B wird um eine Position nach rechts verschoben. Bit 0 wird ins Carry Flag, das Carry Flag ins Bit 7 geschoben. Das Overflow Flag wird nicht beeinflußt.

RORA		46
RARB		56
ROR	10,X	66 0A

RTI (*Return from Interrupt*)

Mit diesem Befehl wird das Interruptprogramm beendet und die Register werden vom Systemstack eingelesen. Ist das E-Bit im CC-Register gesetzt, werden alle Register restauriert. Im anderen Falle werden nur der Program Counter und das CC-Register geladen.

RTI		3B



RTS (*Return from Subroutine*)

Dieser Befehl beendet das aufgerufene Unterprogramm. Die Rücksprungadresse wird aus dem Systemstack gelesen, und das aufrufende Programm wird an dieser Adresse fortgeführt.[5]

RTS		39

[5] Die Rücksprungadresse wurde vorher beim Unterprogramm-Aufruf durch den Befehl JSR oder BSR auf den Stack geschrieben.

SBCA/SBCB (*Subtract with Borrow*)

Vom Inhalt des Akkumulators A/B wird das Carry Flag und der Inhalt einer Speicherzelle oder eine Konstante subtrahiert.

SBCA	#$1F	82 1F
SBCB	[13124,X]	E2 99 33 44

SEX (*Sign Extend*)

Ein im Akkumulator B stehender, 8 bit langer Zweierkomplementwert wird unter Beachtung des Vorzeichens in einen 16 bit langen Wert im Akkumulator D umgewandelt. Ist Bit 7 von B gesetzt, wird in A der Wert $FF, im anderen Fall $00 geschrieben.

SEX		1D

STA/STB/STD/STS/STU/STX/STY (*Store in Memory*)

Der Inhalt des Registers wird in die angegebene Speicherzelle, bei 16-bit-Registern auch in die nachfolgende Zelle geschrieben. Das Overflow Flag wird gelöscht.

STA	$1F	97 1F
STB	,X	E7 84
STD	$201F	FD 20 1F
STS	[D,X]	10 EF 9B
STU	,-X	EF 82
STX	$A00A	BF A0 0A
STY	20,X	10 AF 88 14

SUBA/SUBB/SUBD (*Subtract Memory from Register*)

Vom Inhalt des Akkumulators A/B/D wird der Inhalt einer Speicherzelle oder eine Konstante subtrahiert; beim Akkumulator D der Inhalt zweier aufeinander folgender Speicherzellen bzw. eine 2-byte-Konstante.

SUBA	#$AA	80 AA
SUBB	$EB	D0 EB
SUBD	$1234	B3 12 34

SWI1/SWI2/SWI3 (*Software Interrupt*)

Bei diesen Befehlen werden zunächst alle Registerinhalte der CPU in den Systemstack abgespeichert und das E-Flag im CC-Register gesetzt[6]. Der SWI1 sperrt zusätzlich den $\overline{\text{FIRQ}}$ und $\overline{\text{IRQ}}$ durch Setzen des I- und F-Flags im CC-Register. Anschließend wird die Startadresse des Unterbrechungsprogrammes geladen. (Siehe hierzu in Tabelle 1.3-1.)

SWI1		3F
SWI2		10 3F
SWI3		11 3F

[6] vgl. Bild zum Befehl RTI.

SYNC (*Synchronize to External Event*)

Der Befehl stoppt die Befehlsabarbeitung, d.h. der Prozessor hält an und wartet auf einen Hardware-Interrupt. Ist der betreffene Interrupt gesperrt (durch Setzen des entsprechenden Flags im CC-Register) oder das Interrupt-Signal kürzer als 3 Taktzyklen, wird die nach dem SYNC-Befehl folgende Sequenz abgearbeitet. Der Interrupt muß also, damit er abgearbeitet werden kann, mindestens 3 Taktzyklen lang anliegen und darf nicht maskiert sein. Dieser Befehl bietet die Möglichkeit der Synchronisation zwischen der Software und der Hardware.

SYNC		13

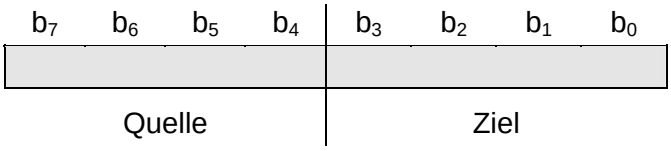
Beispiel:

```
LOOP:   SYNC
        Hardware- Interrupt vom Gerät
        Load      - Daten vom Gerät
        Store     - Daten zum Gerät
        Abfrage:  Datenübertragung beendet?
        wenn nicht: Branch to LOOP
```

TFR (*Transfer Register to Register*)

Mit diesem Befehl können Daten vom einem Register in ein anderes übertragen werden. Die Quelle ist das Register, in dem die Daten stehen. Das Ziel ist das Register, das die Daten aufnehmen soll. Quelle und Ziel werden im Postbyte angeben. Die Codierung der Register wurde bereits in Tabelle 1.3-4 angegeben.

Transfer-Postbyte:



Anmerkung: Es können nur 8-bit-Registerinhalte in 8-bit-Register und 16-bit-Registerinhalte in 16-bit-Register übertragen werden !

TFR	X,Y	1F 12

TST/TSTA/TSTB (*Test*)

Dieser Befehl testet den Inhalt einer Speicherzelle bzw. des Akkumulators A/B auf den Wert $00, positive oder negative Werte. Das Overflow Flag wird gelöscht und das Zero bzw. Negative Flag im CC-Register entsprechend dem Ergebnis gesetzt.

TSTA		4D
TSTB		5D
TST	$1F	0D 1F

Branch Befehle (Verzweigungsbefehle bzw. Sprungbefehle)

Ein Branch-Befehl (Verzweigungsbefehl) wird in der Regel dann ausgeführt, wenn der Test der Flags im Condition Code Register CC die für eine bestimmte Verzweigung erforderliche Bedingung erfüllt. Ausnahmen sind hier BRA (Branch Always), BRN (Branch Never) und BSR (Branch to Subroutine). Die Zieladressen der Verzweigungsbefehle werden immer Programmzähler-relativ berechnet, d.h. sie ergeben sich wie folgt:

PC := PC + Offset

Für alle Branch-Befehle existiert die Möglichkeit, einen 1-byte-Offset oder einen 2-byte-Offset anzugeben. Im 2. Fall kann der gesamte Adreßraum von 64kbyte übersprungen werden. Der OpCode ist dabei ebenfalls 2 byte lang.

BRA/LBRA (*Branch Always*)

Es wird unbedingt zu der Adresse gesprungen, die sich durch Addition des Program Counters mit dem Offset ergibt. Der Unterschied zum JMP-Befehl besteht darin, daß hier keine absolute Sprungziel-Adresse, sondern ein Offset angegeben wird. (Dadurch ist wieder eine lageunabhängige Programmierung möglich.)

BRA	$AA	20 AA
LBRA	$1234	16 12 34

BRN/LBRN (*Branch Never*)

Es wird kein Sprung ausgeführt. Diese Befehle werden als NOP-Befehle eingesetzt und ermöglichen eine Verlängerung des Programmzweiges um 3 bzw. 5 CPU-Zyklen (Dauer des NOP-Befehls: 1 CPU-Zyklus). Sie können aber auch zu Testzwecken zunächst im Programm untergebracht werden und dann durch andere Branch-Befehle überschrieben werden.

BRN	$11	21 11
LBRN	$1234	10 21 12 34

BSR/LBSR (*Branch to Subroutine*)

Vor Abarbeitung des Befehls wird der Inhalt des Program Counters (Adresse des 1. OpCodes nach dem BSR-Befehl) mit Hilfe des Stackregisters S auf den Systemstack geschrieben. Diese Adresse wird vom Rücksprungbefehl RTS benötigt! Die Unterprogrammadresse berechnet sich aus dem Inhalt des Program Counters und dem Offset.

BSR	$AB	8D AB
LBSR	$1234	17 12 34

A. Einfache Verzweigungen

Wird durch einen Verzweigungsbefehl nur eine Bedingung (Flag) im Condition Code Register CC abgefragt bzw. getestet, spricht man von einer einfachen bedingten Verzweigung. Abgefragt werden das N-, Z-, V- oder C-Flag des CC-Registers.

BMI/LBMI (*Branch on Minus*)

Es wird ein relativer Sprung ausgeführt, wenn das N-Flag im CC-Register gesetzt ist (N="1"). Die Sprungadresse ergibt sich durch Addition des Program Counters mit dem Offset.

BMI	$10	2B 10
LBMI	$ABCD	10 2B AB CD

BPL/LBPL (*Branch on Plus*)

Es wird ein relativer Sprung ausgeführt, wenn das N-Flag im CC-Register gelöscht ist (N="0").

BPL	$FF	2A FF
LBPL	$5566	10 2A 55 66

BEQ/LBEQ (*Branch on Equal*)

Es wird ein relativer Sprung ausgeführt, wenn im CC-Register das Z-Flag gleich "1" ist, d.h. das Resultat einer Operation gleich Null war.

BEQ	$01	27 01
LBEQ	$1122	10 27 11 22

BNE/LBNE (*Branch on NOT Equal*)

Es wird ein relativer Sprung ausgeführt, wenn das Z-Flag den Wert "0" hat, d.h. das Ergebnis einer Operation war größer oder kleiner als Null.

BNE	$21	26 21
LBNE	$0102	10 26 01 02

BVS/LBVS (*Branch on Overflow Set*)

Es wird ein relativer Sprung ausgeführt, wenn das Overflow Flag im CC-Register gesetzt wurde (V=1). Bei der Zweierkomplement-Arithmetik wird beim Überschreiten des Zahlenbereiches (negativ: $80-$FF, positiv: $00-$7F) in negativer oder in positiver Richtung das Overflow Flag gesetzt.

BVS	$10	29 10
LBVS	$3001	10 29 30 01

BVC/LBVC (*Branch on Overflow Clear*)

Es wird ein relativer Sprung ausgeführt, wenn die Bedingung V="0" erfüllt ist, d.h. das Overflow Flag gelöscht ist. Dies ist der Fall, wenn bei der Zweierkomplement-Arithmetik der Zahlenbereich nicht überschritten wird.

BVC	$A0	28 A0

LBVC	$A0A1	10 28 A0 A1

BCS/LBCS (*Branch on Carry Set*)

Es wird ein relativer Sprung ausgeführt, wenn das Carry Flag im CC-Register gesetzt ist (C="1"), d.h. z.B., daß ein Übertrag bei einer Operation entstanden ist.

BCS	$BB	25 BB
LBCS	$1234	10 25 12 34

BCC/LBCC (*Branch on Carry Clear*)

Es wird ein relativer Sprung ausgeführt, wenn das Carry Flag den Wert "0" hat. (Achtung: Folgende Befehle invertieren den Wert des Carry Flags: SUB, SBC, CMP.)

BCC	$EF	24 EF
LBCC	$ABCD	10 24 AB CD

B. Verzweigungen mit verknüpften Bedingungen

Bei den nachfolgenden Branch-Befehlen werden die Sprungentscheidungen aus logischen Verknüpfungen verschiedener Flags des CC-Registers abgeleitet. Die Verknüpfungen bekommen ihre besondere Bedeutung im Zusammenhang mit dem CMP-Befehl. Bei den ersten vier Verzweigungen werden die Operanden dieses Befehls als vorzeichenbehaftete Zahlen im Zweierkomplement, bei den letzten vier als vorzeichenlose positive Zahlen betrachtet. Mit v bzw. (xor) bezeichnen wir darin stets die logische ODER-Verknüpfung bzw. die Antivalenz. M bzw. R stehen für die 8- bzw-16-bit-Operanden von CMP im Speicher oder einem der Register.

a) Vorzeichenbehaftete Operanden

M,R ℳ{-128,..,127} bzw. M,R ℳ {-32768,..,32767}.

BLT/LBLT (*Branch on Less Than*)

Es wird ein relativer Sprung ausgeführt, wenn die Bedingung N(xor)V=1 gilt, d.h. das N-Bit oder das V-Bit ist gesetzt aber nicht beide gleichzeitig. Dies ist der Fall für: R < M.

BLT	$BB	2D BB
LBLT	$CDEF	10 2D CD EF

BLE/LBLE (*Branch on Less than or Equal to*)

Es wird ein relativer Sprung ausgeführt, wenn die Bedingung Zv(N(xor)V)=1 erfüllt ist. Dies ist dann der Fall, wenn entweder das N- oder das V-Flag, nicht jedoch beide, oder das Z-Flag gesetzt ist. Dem entspricht: **R ≤ M**.

BLE	$45	2F 45
LBLE	$1011	10 2F 10 11

BGT/LBGT (*Branch on Greater than*)

Es wird ein relativer Sprung ausgeführt, wenn die Bedingung Zv(N(xor)V)=0 erfüllt ist. In diesem Fall ist das Ergebnis der Operation größer Null (Z=0) und die XOR-Verknüpfung des N- und V-Flags hat den Wert "0", d.h. das N- und das V-Flag sind beide gesetzt oder gelöscht. Hier ist: R > M.

BGT	$11	2E 11
LBGT	$1234	10 2E 12 34

BGE/LBGE (*Branch on Greater than or Equal to*)

Es wird ein relativer Sprung ausgeführt, wenn die Bedingung N(xor)V=0 gilt, d.h. wenn das N-Flag und das V-Flag beide gesetzt oder gelöscht sind. Es gilt: R ≥ M.

BGE	$20	2C 20
LBLE	$1221	10 2C 12 21

b) Vorzeichenlose Operanden

M,R ∈ {0,..,255} bzw. M,R ∈ {0,..,65535}

BLO/LBLO (*Branch on LOwer*)

Es wird ein relativer Sprung ausgeführt, wenn im CC-Register das Carry Flag gesetzt ist (C="1"). Dieser Befehl ist identisch mit dem Befehl BCS (Branch on Carry Set). Es gilt: R < M.

BLO	$77	25 77
LBLO	$7788	10 25 77 88

BLS/LBLS (*Branch on Lower or Same*)

Die Bedingung für einen relativen Sprung (CvZ)=1 ist erfüllt, wenn im Condition-Code-Register das C-Flag oder das Z-Flag oder beide gesetzt sind. Es ist: R ≤ M.

BLS	$12	23 12
LBLS	$1000	10 23 10 00

BHI/LBHI (*Branch on Higher*)

Ein relativer Sprung wird ausgeführt, wenn die Bedingung (ZvC)=0 erfüllt ist, d.h. das Zero Flag und das Carry Flag des Statuswortes müssen beide den Wert "0" haben. Das gilt für: R > M.

BHI	$A1	22 A1
LBHI	$10A1	10 22 10 A1

BHS/LBHS (*Branch on Higher or Same*)

Es wird ein relativer Sprung ausgeführt, wenn das Carry Flag im CC-Register den Wert "0" hat. Dieser Befehl ist identisch mit dem Befehl BCC *(Branch on Carry Clear*). Für M und R gilt nun: R ≥ M**.**

BHS	$55	24 55
LBHS	$4455	10 24 44 55

1.4 Anleitung zur Lösung der Übungsaufgaben

In diesem Abschnitt wollen wir Ihnen anhand einer Beispielaufgabe aufzeigen, wie Sie die Lösung der Übungsaufgaben systematisch und erfolgversprechend durchführen können. Die folgende Musterlösung zeigt Ihnen dazu die einzelnen Schritte einer Problemlösung auf. Die Aufgabe lautet:

Aufgabe:

Schreiben Sie ein Programm, das der Reihe nach

1. die Anzeige löscht,
2. die Kennung “dA“ ins Operationsfeld der Anzeige schreibt und danach drei Bytes über die Tastatur einliest und im Adreß- und Datenfeld anzeigt,
3. die Anzeige löscht,
4. die Kennung “AA“ ins Operationsfeld schreibt und eine (Anfangs-)Adresse einliest und im Adreßfeld darstellt,
5. die Kennung “EA“ ins Operationsfeld schreibt und eine (End-)Adresse einliest und im Adreßfeld darstellt,
6. den durch die Anfangs- und Endadresse angegebenen Adreßbereich im Speicher nach dem Auftreten von drei direkt hintereinander folgenden Bytes durchsucht, die die mit den unter 2. angegebenen übereinstimmen; dabei muß auch die Reihenfolge dieselbe sein,
7. werden die drei eingegebenen Bytes gefunden, wird im Operationsfeld die Kennung “F0“ und im Adreßfeld die Adresse ihres Fundortes ausgegeben,
8. werden die Bytes nicht gefunden, wird die Kennung “00“ im Operationsfeld und die Endadresse (siehe 5.) im Adreßfeld angezeigt.

Lösung:

Das Flußdiagramm in Bild 1.4-1 stellt den Aufgabentext ohne Verfeinerungen dar. Lediglich die Adreßausgaben von Punkt 7. und 8. wurden zusammengeführt. Die Aufgabenstellung dieses ersten Diagramms ohne inhaltliche Änderung empfiehlt sich, weil die Darstellung übersichtlicher als der Text ist. Dadurch werden Mißverständnisse geklärt bzw. vermieden und ähnliche oder gleiche Funktionsblöcke sind leichter erkennbar. In diesem Fall kommt der Ablauf “Anzeige löschen, Kennung ‘XX’ ins Operationsfeld schreiben“ fünfmal vor. Die Programmlänge und damit die Schreib- und „Eintipp“-Arbeit läßt sich wesentlich verringern, wenn dieser Ablauf als Unter-programm geschrieben wird, das an den entsprechenden Stellen im Hauptprogramm aufgerufen wird. Die Kennung muß als Parameter übergeben werden, am einfachsten in einem CPU-Register.

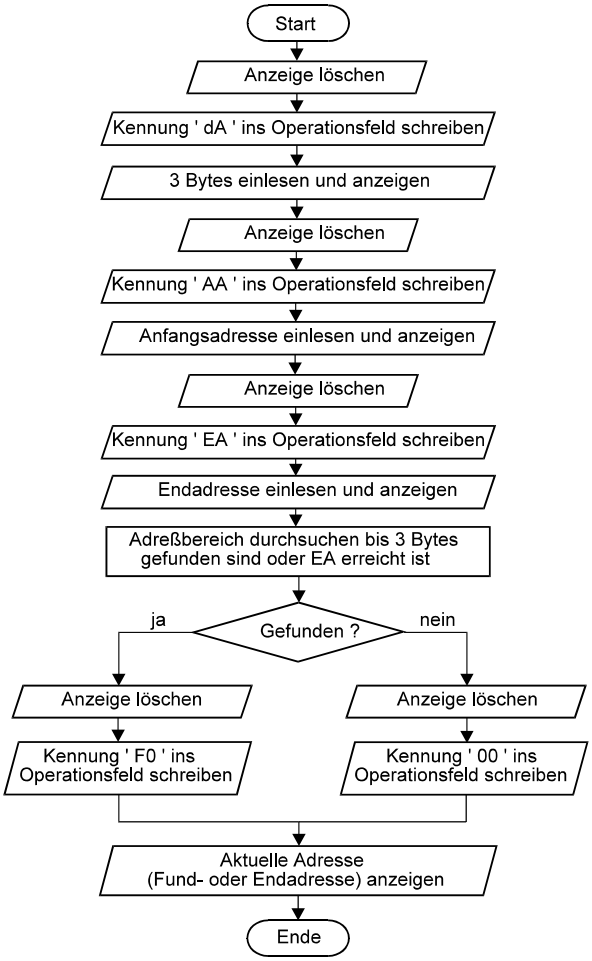


Bild 1.4-1 : Grobes Flußdiagramm nach Aufgabentext

Bei der folgenden Verfeinerung des Programmablaufs ist zu beachten, welche Aufgaben von schon vorhandenen Hilfsroutinen (siehe Abschnitt 1.2.2) übernommen werden können: "Anzeige löschen“ entspricht “CLRDISP“, “Kennung ‘X’ ins Operationsfeld schreiben“ wird von “SHOWB7SG“ geleistet, wenn ‘XX’ in B steht und das X-Register den Wert $0006 enthält. Beim Aufruf des oben erwähnten Unterprogramms sollte die Kennung also gleich im B-Register übergeben werden. Eingabe und Anzeige eines bzw. zweier Bytes wird von “SHOWDATA“ bzw. “SHOWADR“ ausgeführt . Die Anzeige der aktuellen Adresse wird mit “SHOWD7SG“ gelöst.

Ohne Hilfsroutinen muß nur die Aufgabe “Adreßbereich durchsuchen“ gelöst werden. Hierzu legen wir zunächst ein allgemeines Flußdiagramm an (s. Bild 1.4-2).

Die Entscheidung „Gefunden?“ liegt außerhalb der Suchroutine, da dies zunächst übersichtlicher ist. Selbstverständlich muß schon während des Suchens entschieden werden, ob die Bytefolge gefunden wurde. An diese Entscheidung wird nun direkt angeknüpft und in Abhängigkeit vom Ergebnis die Kennung auf “F0“ oder “00“ gesetzt. Da die aktuelle Adresse nicht gleich der Endadresse ist, wenn die Suche erfolglos abgebrochen wird, muß in diesem Fall noch die aktuelle Adresse auf den Wert der Endadresse gesetzt werden, damit bei der abschließenden Adreßausgabe immer die aktuelle Adresse ausgegeben werden kann (s. Punkt 8. der Aufgabenstellung).

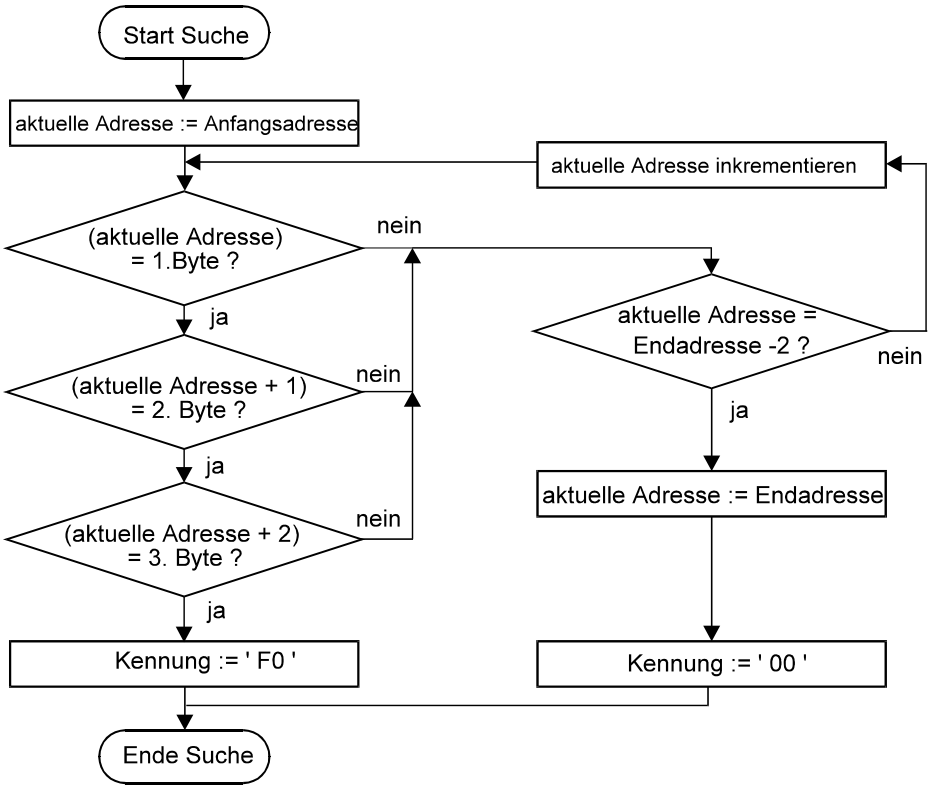


Bild 1.4-2 : Verfeinerung der Suchroutine

Bevor wir die Diagramme unter Berücksichtigung der CPU-Architektur verfeinern, müssen wir entscheiden, wie die Variablen abgelegt und die CPU-Register benutzt werden:

Da Vergleiche nur zwischen Registerinhalt und Speicherinhalt stattfinden können, sollten die drei zu suchenden Bytes in Registern gehalten werden, z.B. in A und U. Die aktuelle Adresse kann im X-Register stehen. Die Endadresse-2 muß für den Vergleich im Speicher stehen. Die Endadresse kann im Y-Register abgelegt werden. PC, DP und S werden für den Programmablauf gebraucht. B steht zur freien Verfügung bereit. Die Anfangsadresse muß im Speicher abgelegt werden, weil in der CPU kein 16-bit-Register frei ist.

Wenn wir das Flußdiagramm in Bild 1.4-2 an diese Festlegungen anpassen, erhalten wir das Flußdiagramm in Bild 1.4-3. Ein zusätzlicher Unterschied zu Bild 1.4-2 besteht in der Übergabe der Fundortadresse: Das X-Register wird vom Unterprogramm KENNUNG verwendet, das Y-Register wird dagegen nicht benötigt. Deshalb wird die auszugebende Adresse im Y-Register an das Hauptprogramm übergeben. Im rechten Zweig des Flußdiagramms in Bild 1.4-3 wird das Y-Register nicht modifiziert, da es die Endadresse enthält. Im linken Zweig wird die Endadresse in Y mit der aktuellen Adresse überschrieben.

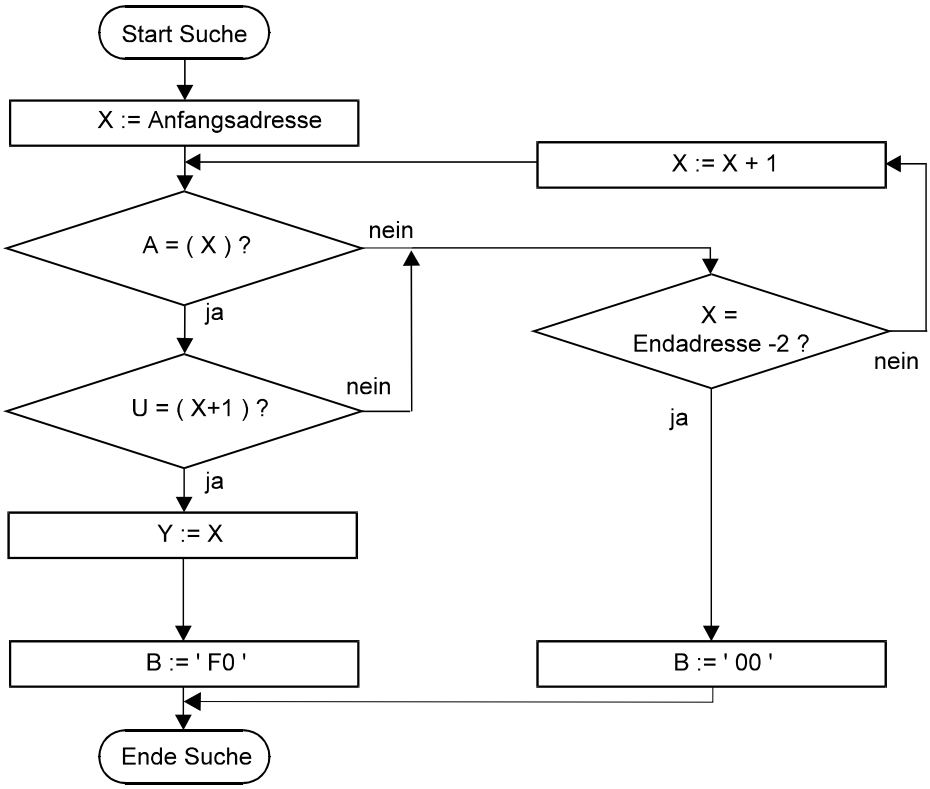


Bild 1.4-3: Suchroutine, an Prozessorarchitektur angepaßt

Für das Unterprogramm “KENNUNG“ erhalten wir:

```
CLRDISP aufrufen
X := #$0006
SHOWB7SG aufrufen
Rückkehr zum Hauptprogramm
```

Beachten Sie, daß in diesem Unterprogramm keine Registerinhalte auf den Stack „gerettet“ werden. Zumindest der Inhalt des X-Registers wird innerhalb des Unterprogramms verändert. In unserem kleinen Musterprogramm ergeben sich daraus keine Probleme . In komplexeren Progammen ist es jedoch ratsam, zu Beginn jedes Unterprogramms alle Registerinhalte, die im Verlauf der Unterroutine verändert werden, auf dem Stack abzulegen und vor der Rückkehr zum aufrufenden Programm wieder in die Register zu holen.

Das Hauptprogramm enthält nun keine Verzweigungen mehr. Wir verzichten daher auf ein Flußdiagramm. Die Bezeichner “ERSTBYTE“, “ZWEITBYTE“, “DRITTBYTE“, “ANFADR“ und “LETZTADR“ entsprechen je einer bzw. zwei konsekutiven Speicheradressen. Nach Anpassung an die Prozessorarchitektur erhalten wir:

```
B := “DA“                ( * Kennung *)
KENNUNG aufrufen
X := #$0004
SHOWDATA aufrufen        ( * erstes Byte lesen * )
(ERSTBYTE) := A
X := #$0002
SHOWDATA aufrufen        ( *zweites Byte lesen * )
(ZWEITBYTE) :=A
X := #$0000
SHOWDATA aufrufen        ( *drittes Byte lesen * )
(DRITTBYTE ) := A
U:= (ZWEITBYTE) und (DRITTBYTE)
B := “AA“                ( * Kennung * )
KENNUNG aufrufen
X := #$0002
SHOWADR aufrufen         ( * Anfangsadresse lesen * )
(ANFADR) := Y
B := “EA“                ( * Kennung * )
KENNUNG aufrufen
X := #$0002
SHOWADR aufrufen         ( * Endadresse lesen * )
D := Y
D:= D-2
(LETZTADR) := D          ( * LETZTADR := Endadresse -2 * )
A := (ERSTBYTE)
SUCHE
KENNUNG aufrufen         ( * “F0“ oder “00“ ausgeben * )
D := Y
X := #$0002
SHOWD7SG aufrufen        ( * Fund- oder Endadresse ausgeben * )
Endlosschleife
```

Zur Kontrolle sollte man sich noch eine Tabelle (s. Tabelle 1.4-1) anlegen, in der die Verwendung der CPU-Register gelistet ist. Man kann mit Hilfe der Tabelle unbeabsichtigte Mehrfachbenutzungen erkennen.

Der nächste Schritt ist nun die Übertragung der einzelnen Operationen in Assemblercode. Hierbei dürfte es im allgemeinen keine Schwierigkeiten geben, da die Abbildung fast überall „isomorph“ ist, wenn die Verfeinerung ausführlich genug war. Statt numerischer Sprungadressen werden Marken (*Labels*) verwendet und die Variablennamen für Speicherplätze werden zunächst beibehalten. Das einzige Problem stellt in unserem Beispiel die Anweisung “X := X + 1“ dar. Der INC-Befehl ist nur auf die Register A und B und den Speicher anwendbar. Das X-Register muß daher z.B. mit der Anweisung “LDB ,X+“ inkrementiert werden.

Tabelle 1.4-1: Registerbelegungen

Reg.	Kennung	Eingabe	Suchroutine	Ausgabe
A	---	Eingabebyte	1. Byte	---
B	Kennung	verändert	verändert	Kennung
X	Anzeigestelle	Anzeigestelle	aktuelle Adresse	Anzeigestelle
Y		Eingabeadresse	Endadresse	Adresse
S	Stackzeiger, für Betriebssystem und UP “KENNUNG“ nötig			
U	---	2. und 3. Byte	2. und 3. Byte	---
DP	für Betriebssystemroutinen unverändert gelassen			

Abschließend werden die mnemonischen Assembleranweisungen in Maschinencode übersetzt und gleichzeitig die dadurch belegten Speicheradressen hingeschrieben. Für die „Übersetzung“ eines Assemblerbefehls in den Maschinencode müssen Sie zunächst in den Tabellen des Anhangs B die Zeile suchen, die durch den Mnemocode des Assemblerbefehls bezeichnet ist. Danach entnehmen Sie den gesuchten OpCode der Spalte, die durch die gewünschte Adressierungsart gekennzeichnet ist. Hier finden Sie auch die Gesamtzahl der Bytes, die der Befehl umfaßt. Es empfiehlt sich, Programmblöcke, die nur über Programmverzweigungen oder Unterprogrammaufrufe erreichbar sind, nicht direkt an das übrige Programm anschließen zu lassen, sondern einige Bytes Freiraum zu schaffen. Auf diese Weise müssen Sie nicht das gesamte Programm verschieben und eventuell viele Sprungadressen ändern, wenn Sie nachträglich einige Programmbytes einfügen[2].

Für Variablennamen, die Speicherzellen bezeichnen, werden jetzt die Speicheradressen eingesetzt. Wenn alle Labeladressen bekannt sind, können Sie die Sprungziele der absoluten Sprünge einsetzen und die Sprungweiten der relativen Sprünge berechnen. Beachten Sie, daß die Sprungweite “0“ keinem Sprung entspricht, d.h. es wird die Anweisung nach dem Sprungbefehl abgearbeitet.

```
ADR.   OPCODE      LABEL   MNEMON.          BEMERKUNGEN

0400   C6 DA               LDB  #$DA
0402   BD 05 00            JSR  KENNUNG     “DA“ IM OP.FELD
0405   8E 00 04            LDX  #0004
0408   BD F1 50            JSR  SHOWDATA    EINGABE 1.BYTE
040B   B7 06 00            STA  ERSTBYTE
040E   8E 00 02            LDX  #0002
0411   BD F1 50            JSR  SHOWDATA    EINGABE 2.BYTE
0414   B7 06 01            STA  ZWEITBYTE
0417   8E 00 00            LDX  #0000
041A   BD F1 50            JSR  SHOWDATA    EINGABE 3.BYTE
041D   B7 06 02            STA  DRITTBYTE
0420   FE 06 01            LDU  ZWEITBYTE   U := 2.+3.Byte
```

[2] Falls Sie dazu die Einfügefunktion der Taste F1 nutzen wollen, sollten Sie den Zwischenraum zwischen den Programmteilen relativ groß wählen, da bei Betätigen der Taste F1 die folgenden 256 Bytes verschoben werden. Achtung: Das 256. Byte geht verloren.

```
0423    C6 AA          LDB  #$AA
0425    BD 05 00       JSR  KENNUNG      "AA" IM OP.FELD
0428    8E 00 02       LDX  #0002
042B    BD F1 56       JSR  SHOWADR      EINGABE ANF.ADR.
042E    10 BF 06 03    STY  ANFADR
0432    C6 EA          LDB  #$EA
0434    BD 05 00       JSR  KENNUNG      "EA" IM OP.FELD
0437    8E 00 02       LDX  #0002
043A    BD F1 56       JSR  SHOWADR      EINGABE ENDADR
043D    1F 20          TFR  Y,D          D := ENDADR.
043F    83 00 02       SUBD #$0002       LETZTADR :=
0442    FD 06 05       STD  LETZTADR     ENDADR - 2
0445    BE 06 03       LDX  ANFADR
0448    B6 06 00       LDA  ERSTBYTE
044B    A1 84     VERGLCH: CMPA ,X       A = (X) ?
044D    26 11          BNE  COMPADR
044F    11 A3 01       CMPU 1,X          U = (X+1) ?
0452    26 0C          BNE  COMPADR
0454    1F 12          TFR  X,Y          Y := AKT.ADR.
0456    C6 F0          LDB  #$F0         KENNUNG := "F0"
0458    20 16          BRA  ERGEBNIS

0460    5F        COMPADR: CLRB          KENNUNG := "00"
0461    BC 06 05       CMPX LETZTADR     AKT.ADR.=
0464    27 0A          BEQ ERGEBNIS      LETZTADR ?
0466    E6 80          LDB  ,X+          X := X + 1
0468    20 E7          BRA  VERGLCH

0470    BD 05 00  ERGEBNIS: JSR  KENNUNG  "F0" ODER "00"
0473    1F 20          TFR  Y,D          AKT.ADR  BZW.
0475    8E 00 02       LDX  #$0002       ENDADRESSE
0478    BD F1 23       JSR  SHOWD7SG     ANZEIGEN
047B    20 FE     ENDE:    BRA  ENDE

0500    BD F1 10  KENNUNG: JSR  CLRDISP
0503    8E 00 06       LDX  #$0006       AUF STELLE 6 U. 7
0506    BD F1 20       JSR  SHOWB7SG     B ANZEIGEN
0509    39             RTS
```

Variablennamen:	ERSTBYTE	=	$0600
	ZWEITBYTE	=	$0601
	DRITTBYTE	=	$0602
	ANFADR	=	$0603
	LETZTADR	=	$0605

Der oben beschriebene Lösungsweg erscheint vielleicht etwas aufwendig. Wenn er auch in dieser oder ähnlicher Form immer beschritten wird, so werden häufig nicht alle Zwischenschritte zu Papier gebracht. Ihre Einsendeaufgaben brauchen auch nicht alle Zwischenergebnisse zu enthalten, sondern könnten sich in diesem Fall auf die Flußdiagrammme der Bilder 1.4-1 und 1.4-2 und das kommentierte Assembler- und Maschinenprogramm beschränken. Wir empfehlen Ihnen aber (vor allem, wenn Sie keine Erfahrung mit Maschinenprogrammierung besitzen), alle hier beschriebenen Schritte explizit abzuarbeiten. Sie strukturieren Ihre Programme dadurch automatisch und die Problemlösung und Fehlersuche gestaltet sich für Sie leichter.

Praktische Übung 1.4-1:

Geben Sie das obenstehende Programm in den Praktikumsrechner ein und führen Sie es mit verschiedenen zu suchenden Bytefolgen aus.
Machen Sie sich anhand dieses Programms mit den Testmöglichkeiten (*debugging*) des Monitorprogramms „Einzelschrittausführung“ (Taste T) und „Setzen eines Breakpoints“ (Tasten G, +) vertraut. Nutzen Sie diese Möglichkeiten insbesondere zur Überprüfung der relativen Verzweigungsbefehle.

Anhang A:

Tabelle A-1: Code für die indizierte Adressierung mit dem X-Register

Lfd. Nr.	Offset	Direkt		X ~	+ #	Indirekt		X ~	+ #
		Assembler	Postbyte (Hex)			Assembler	Postbyte (Hex)		
1	kein Offset	,X	84	0	0	[,X]	94	3	0
2	5-Bit*	n,X	000nnnnn	1	0	-	8-Bit Offset benutzen	-	-
3	8-Bit	n,X	88	1	1	[n,X]	98	4	1
4	16-Bit	n,X	89	4	2	[n,X]	99	7	2
5	A-Register	A,X	86	1	0	[A,X]	96	4	0
6	B-Register	B,X	85	1	0	[B,X]	95	4	0
7	D-Register	D,X	8B	4	0	[D,X]	9B	7	0
8	Inc. um1	,X+	80	2	0	nicht	erlaubt	-	-
9	Inc. um 2	,X++	81	3	0	[,X++]	91	6	0
10	Dec. um 1	,-X	82	2	0	nicht	erlaubt	-	-
11	Dec. um 2	,--X	83	3	0	[,--X]	93	6	0
12	PCR 8-Bit	n,PCR	8C [1]	1	1	[n,PCR]	9C [2]	4	1
13	PCR 16-Bit	n,PCR	8D [1]	5	2	[n,PCR]	9D [2]	8	2
14	Extended Indirekt	-	-	-	-	[n]	9F [2]	5	2

* Der Offset ist im Postbyte enthalten, deshalb ist das Postbyte binär dargestellt. An dieer Stelle nnnnn wird der Offset im Binärcode gesetzt und dann in den Hex-Code umgeformt.
1) 1. Tetrade auch: 8, A, C, E
2) 1. Tetrade auch: 9, B, D, F

Tabelle A-2: Code für die indizierte Adressierung mit dem Y-Register

Lfd. Nr.	Offset	Direkt		X ~	+ #	Indirekt		X ~	+ #
		Assembler	Postbyte (Hex)			Assembler	Postbyte (Hex)		
1	kein Offset	,Y	A4	0	0	[,Y]	B4	3	0
2	5-Bit*	n,Y	001nnnnn	1	0	-	8-Bit Offset benutzen	-	-
3	8-Bit	n,Y	A8	1	1	[n,Y]	B8	4	1
4	16-Bit	n,Y	A9	4	2	[n,Y]	B9	7	2
5	A-Register	A,Y	A6	1	0	[A,Y]	B6	4	0
6	B-Register	B,Y	A5	1	0	[B,Y]	B5	4	0
7	D-Register	D,Y	AB	4	0	[D,Y]	BB	7	0
8	Inc. um1	,Y+	A0	2	0	nicht	erlaubt	-	-
9	Inc. um 2	,Y++	A1	3	0	[,Y++]	B1	6	0
10	Dec. um 1	,-Y	A2	2	0	nicht	erlaubt	-	-
11	Dec. um 2	,--Y	A3	3	0	[,--Y]	B3	6	0
12	PCR 8-Bit	n,PCR	AC [1]	1	1	[n,PCR]	BC [2]	4	1
13	PCR 16-Bit	n,PCR	AD [1]	5	2	[n,PCR]	BD [2]	8	2
14	Extended Indirekt	-	-	-	-	[n]	BF [2]	5	2

* Der Offset ist im Postbyte enthalten, deshalb ist das Postbyte binär dargestellt. An dieer Stelle nnnnn wird der Offset im Binärcode gesetzt und dann in den Hex-Code umgeformt.
1) 1. Tetrade auch: 8, A, C, E
2) 1. Tetrade auch: 9, B, D, F

Tabelle A-3: Code für die indizierte Adressierung mit dem S-Register

Lfd. Nr.	Offset	Direkt		X ~	+ #	Indirekt		X ~	+ #
		Assembler	Postbyte (Hex)			Assembler	Postbyte (Hex)		
1	kein Offset	,S	E4	0	0	[,S]	F4	3	0
2	5-Bit*	n,S	010nnnnn	1	0	-	8-Bit Offset benutzen	-	-
3	8-Bit	n,S	E8	1	1	[n,S]	F8	4	1
4	16-Bit	n,S	E9	4	2	[n,S]	F9	7	2
5	A-Register	A,S	E6	1	0	[A,S]	F6	4	0
6	B-Register	B,S	E5	1	0	[B,S]	F5	4	0
7	D-Register	D,S	EB	4	0	[D,S]	FB	7	0
8	Inc. um1	,S+	E0	2	0	nicht	erlaubt	-	-
9	Inc. um 2	,S++	E1	3	0	[,S++]	F1	6	0
10	Dec. um 1	,-S	E2	2	0	nicht	erlaubt	-	-
11	Dec. um 2	,--S	E3	3	0	[,--S]	F3	6	0
12	PCR 8-Bit	n,PCR	EC [1)]	1	1	[n,PCR]	FC [2)]	4	1
13	PCR 16-Bit	n,PCR	ED [1)]	5	2	[n,PCR]	FD [2)]	8	2
14	Extended Indirekt	-	-	-	-	[n]	FF [2)]	5	2

* Der Offset ist im Postbyte enthalten, deshalb ist das Postbyte binär dargestellt. An dier Stelle nnnnn wird der Offset im Binärcode gesetzt und dann in den Hex-Code umgeformt.
1) 1. Tetrade auch: 8, A, C, E
2) 1. Tetrade auch: 9, B, D, F

Tabelle A-4: Code für die indizierte Adressierung mit dem U-Register

Lfd. Nr.	Offset	Direkt		X ~	+ #	Indirekt		X ~	+ #
		Assembler	Postbyte (Hex)			Assembler	Postbyte (Hex)		
1	kein Offset	,U	C4	0	0	[,U]	D4	3	0
2	5-Bit*	n,U	011nnnnn	1	0	-	8-Bit Offset benutzen	-	-
3	8-Bit	n,U	C8	1	1	[n,U]	D8	4	1
4	16-Bit	n,U	C9	4	2	[n,U]	D9	7	2
5	A-Register	A,U	C6	1	0	[A,U]	D6	4	0
6	B-Register	B,U	C5	1	0	[B,U]	D5	4	0
7	D-Register	D,U	CB	4	0	[D,U]	DB	7	0
8	Inc. um1	,U+	C0	2	0	nicht	erlaubt	-	-
9	Inc. um 2	,U++	C1	3	0	[,U++]	D1	6	0
10	Dec. um 1	,-U	C2	2	0	nicht	erlaubt	-	-
11	Dec. um 2	,--U	C3	3	0	[,--U]	D3	6	0
12	PCR 8-Bit	n,PCR	CC [1)]	1	1	[n,PCR]	DC [2)]	4	1
13	PCR 16-Bit	n,PCR	CD [1)]	5	2	[n,PCR]	DD [2)]	8	2
14	Extended Indirekt	-	-	-	-	[n]	DF [2)]	5	2

* Der Offset ist im Postbyte enthalten, deshalb ist das Postbyte binär dargestellt. An dier Stelle nnnnn wird der Offset im Binärcode gesetzt und dann in den Hex-Code umgeformt.
1) 1. Tetrade auch: 8, A, C, E
2) 1. Tetrade auch: 9, B, D, F

1. **Befehle PUSH und PULL**

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
PC	*	Y	X	DP	B	A	CC

PSHS/PULS: * = U PSHU/PULU: * = S

2. **Befehle TFR und EXG**

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Quellregister-Code				Zielregister-Code			

Code siehe Tabelle A-5

.

3. **indizierte Adressierung**

a) kein 5-bit-Offset

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	r	r	d/i	Adressierungsart			

d/i=0: direkt, d/i=1: indirekt, rr siehe Tabelle A-5

b) 5-bit-Offset

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	r	r	n	n	n	n	n

nnnnn: 5-bit-Offset im Zweierkomplement, rr siehe Tabelle A-5

Bild A-1: Der Aufbau des Postbytes

Tabelle A-5: Codierung der Register für die Befehle TFR und EXG

Register	Binärcode	Hex-Code	rr-Code
D	0000	0	
X	0001	1	00
Y	0010	2	01
U	0011	3	10
S	0100	4	11
PC	0101	5	
A	1000	8	
B	1001	9	
CC	1010	A	
DP	1011	B	

Anhang B:

Tabelle B-1: Befehle in alphabetischer Reihenfolge - Teil I

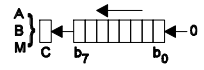
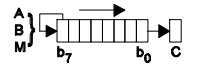
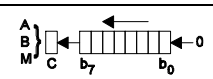
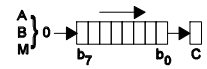
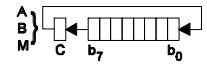
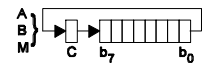
		Adressing Modes																			
		Immediate			Direct			Indexed [1]			Extended			Inherent				5	3	2	1	0
Instruction	Forms	Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#	Description	H	N	Z	V	C
ABX														3A	3	1	B + X → X (Unsigned)	•	•	•	•	•
ADC	ADCA	89	2	2	99	4	2	A9	4+	2+	B9	5	3				A + M + C → A	↕	↕	↕	↕	↕
	ADCB	C9	2	2	D9	4	2	E9	4+	2+	F9	5	3				B + M + C → B	↕	↕	↕	↕	↕
ADD	ADDA	8B	2	2	9B	4	2	AB	4+	2+	BB	5	3				A + M → A	↕	↕	↕	↕	↕
	ADDB	CB	2	2	DB	4	2	EB	4+	2+	FB	5	3				B + M → B	↕	↕	↕	↕	↕
	ADDD	C3	4	3	D3	6	2	E3	6+	2+	F3	7	3				D + M : M + 1 → D	•	↕	↕	↕	↕
AND	ANDA	84	2	2	94	4	2	A4	4+	2+	B4	5	3				A ∧ M → A	•	↕	↕	0	•
	ANDB	C4	2	2	D4	4	2	E4	4+	2+	F4	5	3				B ∧ M → B	•	↕	↕	0	•
	ANDCC	1C	3	2													CC ∧ IMM → CC					7
ASL	ASLA													48	2	1		8	↕	↕	↕	↕
	ASLB													58	2	1		8	↕	↕	↕	↕
	ASL				08	6	2	68	6+	2+	78	7	3					8	↕	↕	↕	↕
ASR	ASRA													47	2	1		8	↕	↕	•	↕
	ASRB													57	2	1		8	↕	↕	•	↕
	ASR				07	6	2	67	6+	2+	77	7	3					8	↕	↕	•	↕
BIT	BITA	85	2	2	95	4	2	A5	4+	2+	B5	5	3				Bit Test A (M ∧ A)	•	↕	↕	0	•
	BITB	C5	2	2	D5	4	2	E5	4+	2+	F5	5	3				Bit Test B (M ∧ B)	•	↕	↕	0	•
CLR	CLRA													4F	2	1	0 → A	•	0	1	0	0
	CLRB													5F	2	1	0 → B	•	0	1	0	0
	CLR				0F	6	2	6F	6+	2+	7F	7	3				0 → M	•	0	1	0	0
CMP	CMPA	81	2	2	91	4	2	A1	4+	2+	B1	5	3				Compare M from A	8	↕	↕	↕	↕
	CMPB	C1	2	2	D1	4	2	E1	4+	2+	F1	5	3				Compare M from B	8	↕	↕	↕	↕
	CMPD	10	5	4	10	7	3	10	7+	3+	10	8	4				Compare M : M + 1 from D	•	↕	↕	↕	↕
		83			93			A3			B3											
	CMPS	11	5	4	11	7	3	11	7+	3+	11	8	4				Compare M : M + 1 from S	•	↕	↕	↕	↕
		8C			9C			AC			BC											
	CMPU	11	5	4	11	7	3	11	7+	3+	11	8	4				Compare M : M + 1 from U	•	↕	↕	↕	↕
		83			93			A3			B3											
	CMPX	8C	4	3	9C	6	2	AC	6+	2+	BC	7	3				Compare M : M + 1 from X	•	↕	↕	↕	↕
	CMPY	10	5	4	10	7	3	10	7+	3+	10	8	4				Compare M : M + 1 from Y	•	↕	↕	↕	↕
		8C			9C			AC			BC											
COM	COMA													43	2	1	¬ A → A	•	↕	↕	0	↕
	COMB													53	2	1	¬ B → B	•	↕	↕	0	↕
	COM				03	6	2	63	6+	2+	73	7	3				¬ M → M	•	↕	↕	0	↕
CWAI		3C	≥20	2													CC ∧ IMM → CC Wait for Interrupt					7
DAA														19	2	1	Decimal Adjust A	•	↕	↕	0	↕
DEC	DECA													4A	2	1	A - 1 → A	•	↕	↕	↕	•
	DECB													5A	2	1	B - 1 → B	•	↕	↕	↕	•
	DEC				0A	6	2	6A	6+	2+	7A	7	3				M - 1 → M	•	↕	↕	↕	•
EOR	EORA	88	2	2	98	4	2	A8	4+	2+	B8	5	3				A ⩒ M → A	•	↕	↕	0	•
	EORB	C8	2	2	D8	4	2	E8	4+	2+	F8	5	3				B ⩒ M → B	•	↕	↕	0	•
EXG	R1, R2													1E	8	2	R1 ↔ R2[2]	•	•	•	•	•
INC	INCA													4C	2	1	A + 1 → A	•	↕	↕	↕	•
	INCB													5C	2	1	B + 1 → B	•	↕	↕	↕	•
	INC				0C	6	2	6C	6+	2+	7C	7	3				M + 1 → M	•	↕	↕	↕	•
JMP					0E	3	2	6E	3+	2+	7E	4	3				EA[3] → PC	•	•	•	•	•
JSR					9D	7	2	AD	7+	2+	BD	8	3				Jump to Subroutine	•	•	•	•	•
LD	LDA	86	2	2	96	4	2	A6	4+	2+	B6	5	3				M → A	•	↕	↕	0	•
	LDB	C6	2	2	D6	4	2	E6	4+	2+	F6	5	3				M → B	•	↕	↕	0	•
	LDD	CC	3	3	DC	5	2	EC	5+	2+	FC	6	3				M : M + 1 → D	•	↕	↕	0	•
	LDS	10	4	4	10	6	3	10	6+	3+	10	7	4				M : M + 1 → S	•	↕	↕	0	•
		CE			DE			EE			FE											
	LDU	CE	3	3	DE	5	2	EE	5+	2+	FE	6	3				M : M + 1 → U	•	↕	↕	0	•
	LDX	8E	3	3	9E	5	2	AE	5+	2+	BE	6	3				M : M + 1 → X	•	↕	↕	0	•
	LDY	10	4	4	10	6	3	10	6+	3+	10	7	4				M : M + 1 → Y	•	↕	↕	0	•
		8E			9E			AE			BE											

Tabelle B-2: Befehle in alphabetischer Reihenfolge - Teil II

		Adressing Modes																5	3	2	1	0
		Immediate			Direct			Indexed [1]			Extended			Inherent								
Instruction	Forms	Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#	Description	H	N	Z	V	C
LEA	LEAS							32	4+	2+							EA[3] → S	•	•	•	•	•
	LEAU							33	4+	2+							EA[3] → U	•	•	•	•	•
	LEAX							30	4+	2+							EA[3] → X	•	•	|	•	•
	LEAY							31	4+	2+							EA[3] → Y	•	•	|	•	•
LSL	LSLA													48	2	1		•	|	|	|	|
	LSLB													58	2	1		•	|	|	|	|
	LSL				08	6	2	68	6+	2+	78	7	3					•	|	|	|	|
LSR	LSRA													44	2	1		•	0	|	•	|
	LSRB													54	2	1		•	0	|	•	|
	LSR				04	6	2	64	6+	2+	74	7	3					•	0	|	•	|
MUL														3D	11	1	A × B → D (Unsigned)	•	•	|	•	9
NEG	NEGA													40	2	1	¬ A + 1 → A	8	|	|	|	|
	NEGB													50	2	1	¬ B + 1 → B	8	|	|	|	|
	NEG				00	6	2	60	6+	2+	70	7	3				¬ M + 1 → M	8	|	|	|	|
NOP														12	2	1	No Operation	•	•	•	•	•
OR	ORA	8A	2	2	9A	4	2	AA	4+	2+	BA	5	3				A ∨ M → A	•	|	|	0	•
	ORB	CA	2	2	DA	4	2	EA	4+	2+	FA	5	3				B ∨ M → B	•	|	|	0	•
	ORCC	1A	3	2													CC ∨ IMM → CC				7	
PSH	PSHS[10]	34	5+[4]	2													Push Registers on S Stack	•	•	•	•	•
	PSHU[10]	36	5+[4]	2													Push Registers on U Stack	•	•	•	•	•
PUL	PULS[10]	35	5+[4]	2													Pull Registers from S Stack	•	•	•	•	•
	PULU[10]	37	5+[4]	2													Pull Registers from U Stack	•	•	•	•	•
ROL	ROLA													49	2	1		•	|	|	|	|
	ROLB													59	2	1		•	|	|	|	|
	ROL				09	6	2	69	6+	2+	79	7	3					•	|	|	|	|
ROR	RORA													46	2	1		•	|	|	•	|
	RORB													56	2	1		•	|	|	•	|
	ROR				06	6	2	66	6+	2+	76	7	3					•	|	|	•	|
RTI														3B	6 - 15	1	Return from Interrupt					7
RTS														39	5	1	Return from Subroutine	•	•	•	•	•
SBC	SBCA	82	2	2	92	4	2	A2	4+	2+	B2	5	3				A - M - C → A	8	|	|	|	|
	SBCB	C2	2	2	D2	4	2	E2	4+	2+	F2	5	3				B - M - C → B	8	|	|	|	|
SEX														1D	2	1	Sign Extend B into A	•	|	|	0	•
ST	STA				97	4	2	A7	4+	2+	B7	5	3				A → M	•	|	|	0	•
	STB				D7	4	2	E7	4+	2+	F7	5	3				B → M	•	|	|	0	•
	STD				DD	5	2	ED	5+	2+	FD	6	3				D → M : M + 1	•	|	|	0	•
	STS				10	6	3	10	6+	3+	10	7	4				S → M : M + 1	•	|	|	0	•
					DF			EF			FF											
	STU				DF	5	2	EF	5+	2+	FF	6	3				U → M : M + 1	•	|	|	0	•
	STX				9F	5	2	AF	5+	2+	BF	6	3				X → M : M + 1	•	|	|	0	•
	STY				10	6	3	10	6+	3+	10	7	4				Y → M : M + 1	•	|	|	0	•
					9F			AF			BF											
SUB	SUBA	80	2	2	90	4	2	A0	4+	2+	B0	5	3				A - M → A	8	|	|	|	|
	SUBB	C0	2	2	D0	4	2	E0	4+	2+	F0	5	3				B - M → B	8	|	|	|	|
	SUBD	83	4	3	93	6	2	A3	6+	2+	B3	7	3				D - M : M + 1 → D	•	|	|	|	|
SWI	SWI[6]													3F	19	1	Software Interrupt 1	•	•	•	•	•
														10	20	2	Software Interrupt 2	•	•	•	•	•
	SWI2[6]													3F								
														11	20	1	Software Interrupt 3	•	•	•	•	•
	SWI3[6]													3F								
SYNC														13	≥4	1	Synchronize to Interrupt	•	•	•	•	•
TFR	R1, R2													1F	6	2	R1 → R2[2]	•	•	•	•	•
TST	TSTA													4D	2	1	Test A	•	|	|	0	•
	TSTB													5D	2	1	Test B	•	|	|	0	•
	TST				0D	6	2	6D	6+	2+	7D	7	3				Test M	•	|	|	0	•

LEGEND:					
OP	Operation Code (Hexadecimal)	¬ M	Complement of M	|	Test and set if true, cleared otherwise
~	Number of CPU Cycles	→	Transfer Into	•	Not Affected
#	Number of Program Bytes	H	Half-carry (from bit 3)	CC	Condition Code Register
+	Arithmetic Plus	N	Negative (sign bit)	:	Concatenation
-	Arithmetic Minus	Z	Zero result	∨	Logical or
×	Multiply	V	Overflow, 2's complement	∧	Logical and
		C	Carry from ALU	⊻	Logical Exclusive or

NOTES:

1. This column gives a base cycle and byte count. To obtain total count, add the values obtained from the INDEXED ADDRESSING MODE table (s. Tabellen A-1 bis A-4).
2. R1 and R2 may be any pair of 8 bit or any pair of 16 bit registers.
   The 8 bit registers are A, B, CC, DP.
   The 16 bit registers are X, Y, U, S, D, PC.
3. EA is the effective address.
4. The PSH and PUL instructions require 5 cycles plus 1 cycle for each **byte** pushed or pulled.
5. 5(6) means 5 cycles if branch not taken, 6 cycles if taken (Branch instructions).
6. SWI1 sets I and F bits. SWI2 and SWI3 do not affect I and F.
7. Conditions Codes set as a direct result of the instruction.
8. Value of half-carry flag is undefined.
9. Special case - Carry set if b_7 is SET.
10. (Bestimmung des Postbytes nach Bild A-1)

Tabelle B-3: Branch Befehle

Instruction	Forms	Addressing Mode Relative OP	~	#	Description	H	N	Z	V	C	CMPR *)	Vorzeichen
BCC	BCC	24	3	2	Branch C = 0	•	•	•	•	•		
	LBCC	10	5(6)	4	Long Branch C = 0	•	•	•	•	•		
		24										
BCS	BCS	25	3	2	Branch C = 1	•	•	•	•	•		
	LBCS	10	5(6)	4	Long Branch C = 1	•	•	•	•	•		
		25										
BEQ	BEQ	27	3	2	Branch Z = 1	•	•	•	•	•		
	LBEQ	10	5(6)	4	Long Branch Z = 1	•	•	•	•	•		
		27										
BGE	BGE	2C	3	2	Branch ≥ Zero	•	•	•	•	•	R ≥ M	Vorzeichen-
	LBGE	10	5(6)	4	Long Branch ≥ Zero	•	•	•	•	•		behaftet
		2C										
BGT	BGT	2E	3	2	Branch > Zero	•	•	•	•	•	R > M	Vorzeichen-
	LBGT	10	5(6)	4	Long Branch > Zero	•	•	•	•	•		behaftet
		2E										
BHI	BHI	22	3	2	Branch Higher	•	•	•	•	•	R > M	Vorzeichenlos
	LBHI	10	5(6)	4	Long Branch Higher	•	•	•	•	•		
		22										
BHS	BHS	24	3	2	Branch Higher or Same	•	•	•	•	•	R ≥ M	Vorzeichenlos
	LBHS	10	5(6)	4	Long Branch Higer or Same	•	•	•	•	•		
		24										
BLE	BLE	2F	3	2	Branch ≤ Zero	•	•	•	•	•	R ≤ M	Vorzeichen-
	LBLE	10	5(6)	4	Long Branch ≤ Zero	•	•	•	•	•		behaftet
		2F										
BLO	BLO	25	3	2	Branch Lower	•	•	•	•	•	R < M	Vorzeichenlos
	LBLO	10	5(6)	4	Long Branch Lower	•	•	•	•	•		
		25										
BLS	BLS	23	3	2	Branch Lower or Same	•	•	•	•	•	R ≤ M	Vorzeichenlos
	LBLS	10	5(6)	4	Long Branch Lower or Same	•	•	•	•	•		
		23										
BLT	BLT	2D	3	2	Branch < Zero	•	•	•	•	•	R < M	Vorzeichen-
	LBLT	10	5(6)	4	Long Branch < Zero	•	•	•	•	•		behaftet
		2D										
BMI	BMI	2B	3	2	Branch Minus (N=1)	•	•	•	•	•		
	LBMI	10	5(6)	4	Long Branch Minus	•	•	•	•	•		
		2B										
BNE	BNE	26	3	2	Branch on Z=0	•	•	•	•	•		
	LBNE	10	5(6)	4	Long Branch on Z=0	•	•	•	•	•		
		26										
BPL	BPL	2A	2	2	Branch Plus (N=0)	•	•	•	•	•		
	LBPL	10	5(6)	4	Long Branch Plus	•	•	•	•	•		
		2A										
BRA	BRA	20	3	2	Branch Always	•	•	•	•	•		
	LBRA	16	5	3	Long Branch Always	•	•	•	•	•		
BRN	BRN	21	3	2	Branch Never	•	•	•	•	•		
	LBRN	10	5	4	Long Branch Never	•	•	•	•	•		
		21										
BSR	BSR	8D	7	2	Branch to Subroutine	•	•	•	•	•		
	LBSR	17	9	3	Long Branch to Subroutine	•	•	•	•	•		
BVC	BVC	28	3	2	Branch V = 0	•	•	•	•	•		
	LBVC	10	5(6)	4	Long Branch V = 0	•	•	•	•	•		
		28										
BVS	BVS	29	3	2	Branch V = 1	•	•	•	•	•		
	LBVS	10	5(6)	4	Long Branch V = 1	•	•	•	•	•		
		29										

*) R: A, B, D, S, U, X, Y
M: Wert einer bzw. zweier Speicherzellen

Anang C:

Tabelle C-1: Befehlscode in hexadezimaler Reihenfolge - Teil I

OP	Mnemonic	Mode	~	#	OP	Mnemonic	Mode	~	#	OP	Mnemonic	Mode	~	#
00	NEG	Direct	6	2	30	LEAX	Indexed	4+	2+	60	NEG	Indexed	6+	2+
01	•	-	-	-	31	LEAY	Indexed	4+	2+	61	•	-	-	-
02	•	-	-	-	32	LEAS	Indexed	4+	2+	62	•	-	-	-
03	COM	Direct	6	2	33	LEAU	Indexed	4+	2+	63	COM	Indexed	6+	2+
04	LSR	Direct	6	2	34	PSHS	Immed.	5+	2	64	LSR	Indexed	6+	2+
05	•	-	-	-	35	PULS	Immed.	5+	2	65	•	-	-	-
06	ROR	Direct	6	2	36	PSHU	Immed.	5+	2	66	ROR	Indexed	6+	2+
07	ASR	Direct	6	2	37	PULU	Immed.	5+	2	67	ASR	Indexed	6+	2+
08	ASL, LSL	Direct	6	2	38	•	-	-	-	68	ASL, LSL	Indexed	6+	2+
09	ROL	Direct	6	2	39	RTS	Inherent	5	1	69	ROL	Indexed	6+	2+
0A	DEC	Direct	6	2	3A	ABX	Inherent	3	1	6A	DEC	Indexed	6+	2+
0B	•	-	-	-	3B	RTI	Inherent	6/15	1	6B	•	-	-	-
0C	INC	Direct	6	2	3C	CWAI	Inherent	≥20	2	6C	INC	Indexed	6+	2+
0D	TST	Direct	6	2	3D	MUL	Inherent	11	1	6D	TST	Indexed	6+	2+
0E	JMP	Direct	3	2	3E	•	-	-	-	6E	JMP	Indexed	3+	2+
0F	CLR	Direct	6	2	3F	SWI	Inherent	19	1	6F	CLR	Indexed	6+	2+
10	Page 2	-	-	-	40	NEGA	Inherent	2	1	70	NEG	Extended	7	3
11	Page 3	-	-	-	41	•	-	-	-	71	•	-	-	-
12	NOP	Inherent	2	1	42	•	-	-	-	72	•	-	-	-
13	SYNC	Inherent	≥4	1	43	COMA	Inherent	2	1	73	COM	Extended	7	3
14	•	-	-	-	44	LSRA	Inherent	2	1	74	LSR	Extended	7	3
15	•	-	-	-	45	•	-	-	-	75	•	-	-	-
16	LBRA	Relative	5	3	46	RORA	Inherent	2	1	76	ROR	Extended	7	3
17	LBSR	Relative	9	3	47	ASRA	Inherent	2	1	77	ASR	Extended	7	3
18	•	-	-	-	48	ASLA, LSLA	Inherent	2	1	78	ASL, LSL	Extended	7	3
19	DAA	Inherent	2	1	49	ROLA	Inherent	2	1	79	ROL	Extended	7	3
1A	ORCC	Immed.	3	2	4A	DECA	Inherent	2	1	7A	DEC	Extended	7	3
1B	•	-	-	-	4B	•	-	-	-	7B	•	-	-	-
1C	ANDCC	Immed.	3	2	4C	INCA	Inherent	2	1	7C	INC	Extended	7	3
1D	SEX	Inherent	2	1	4D	TSTA	Inherent	2	1	7D	TST	Extended	7	3
1E	EXG	Immed.	8	2	4E	•	-	-	-	7E	JMP	Extended	4	3
1F	TFR	Immed.	6	2	4F	CLRA	Inherent	2	1	7F	CLR	Extended	7	3
20	BRA	Relative	3	2	50	NEGB	Inherent	2	1	80	SUBA	Immed.	2	2
21	BRN	Relative	3	2	51	•	Inherent	-	-	81	CMPA	Immed.	2	2
22	BHI	Relative	3	2	52	•	Inherent	-	-	82	SBCA	Immed.	2	2
23	BLS	Relative	3	2	53	COMB	Inherent	2	1	83	SUBD	Immed.	4	3
24	BHS, BCC	Relative	3	2	54	LSRB	Inherent	2	1	84	ANDA	Immed.	2	2
25	BLO, BCS	Relative	3	2	55	•	Inherent	-	-	85	BITA	Immed.	2	2
26	BNE	Relative	3	2	56	RORB	Inherent	2	1	86	LDA	Immed.	2	2
27	BEQ	Relative	3	2	57	ASRB	Inherent	2	1	87	•	-	-	-
28	BVC	Relative	3	2	58	ASLB, LSLB	Inherent	2	1	88	EORA	Immed.	2	2
29	BVS	Relative	3	2	59	ROLB	Inherent	2	1	89	ADCA	Immed.	2	2
2A	BPL	Relative	3	2	5A	DECB	Inherent	2	1	8A	ORA	Immed.	2	2
2B	BMI	Relative	3	2	5B	•	Inherent	-	-	8B	ADDA	Immed.	2	2
2C	BGE	Relative	3	2	5C	INCB	Inherent	2	1	8C	CMPX	Immed.	4	3
2D	BLT	Relative	3	2	5D	TSTB	Inherent	2	1	8D	BSR	Relative	7	2
2E	BGT	Relative	3	2	5E	•	Inherent	-	-	8E	LDX	Immed.	3	3
2F	BLE	Relative	3	2	5F	CLRB	Inherent	2	1	8F	•	-	-	-

LEGEND

~ Number of CPU cycles (less possible push pull or indexed-mode cycles)

Number of program bytes

• Denotes unused opcode

Tabelle C-2: Befehlscode in hexadezimaler Reihenfolge - Teil II

OP	Mnemonic	Mode	~	#
90	SUBA	Direct	4	2
91	CMPA	Direct	4	2
92	SBCA	Direct	4	2
93	SUBD	Direct	6	2
94	ANDA	Direct	4	2
95	BITA	Direct	4	2
96	LDA	Direct	4	2
97	STA	Direct	4	2
98	EORA	Direct	4	2
99	ADCA	Direct	4	2
9A	ORA	Direct	4	2
9B	ADDA	Direct	4	2
9C	CMPX	Direct	6	2
9D	JSR	Direct	7	2
9E	LDX	Direct	5	2
9F	STX	Direct	5	2
A0	SUBA	Inexed	4+	2+
A1	CMPA	Inexed	4+	2+
A2	SBCA	Inexed	4+	2+
A3	SUBD	Inexed	6+	2+
A4	ANDA	Inexed	4+	2+
A5	BITA	Inexed	4+	2+
A6	LDA	Inexed	4+	2+
A7	STA	Inexed	4+	2+
A8	EORA	Inexed	4+	2+
A9	ADCA	Inexed	4+	2+
AA	ORA	Inexed	4+	2+
AB	ADDA	Inexed	4+	2+
AC	CMPX	Inexed	6+	2+
AD	JSR	Inexed	7+	2+
AE	LDX	Inexed	5+	2+
AF	STX	Inexed	5+	2+
B0	SUBA	Extended	5	3
B1	CMPA	Extended	5	3
B2	SBCA	Extended	5	3
B3	SUBD	Extended	7	3
B4	ANDA	Extended	5	3
B5	BITA	Extended	5	3
B6	LDA	Extended	5	3
B7	STA	Extended	5	3
B8	EORA	Extended	5	3
B9	ADCA	Extended	5	3
BA	ORA	Extended	5	3
BB	ADDA	Extended	5	3
BC	CMPX	Extended	7	3
BD	JSR	Extended	8	3
BE	LDX	Extended	6	3
BF	STX	Extended	6	3

NOTE: All unused opcodes are both undefined and illegal

OP	Mnemonic	Mode	~	#
C0	SUBB	Immed	2	2
C1	CMPB	Immed	2	2
C2	SBCB	Immed	2	2
C3	ADDD	Immed	4	3
C4	ANDB	Immed	2	2
C5	BITB	Immed	2	2
C6	LDB	Immed	2	2
C7	•	-	-	-
C8	EORB	Immed	2	2
C9	ADCB	Immed	2	2
CA	ORB	Immed	2	2
CB	ADDB	Immed	2	2
CC	LDD	Immed	3	3
CD	•	-	-	-
CE	LDU	Immed	3	3
CF	•	-	-	-
D0	SUBB	Direct	4	2
D1	CMPB	Direct	4	2
D2	SBCB	Direct	4	2
D3	ADDD	Direct	6	2
D4	ANDB	Direct	4	2
D5	BITB	Direct	4	2
D6	LDB	Direct	4	2
D7	STB	Direct	4	2
D8	EORB	Direct	4	2
D9	ADCB	Direct	4	2
DA	ORB	Direct	4	2
DB	ADDB	Direct	4	2
DC	LDD	Direct	5	2
DD	STD	Direct	5	2
DE	LDU	Direct	5	2
DF	STU	Direct	5	2
E0	SUBB	Inexed	4+	2+
E1	CMPB	Inexed	4+	2+
E2	SBCB	Inexed	4+	2+
E3	ADDD	Inexed	6+	2+
E4	ANDB	Inexed	4+	2+
E5	BITB	Inexed	4+	2+
E6	LDB	Inexed	4+	2+
E7	STB	Inexed	4+	2+
E8	EORB	Inexed	4+	2+
E9	ADCB	Inexed	4+	2+
EA	ORB	Inexed	4+	2+
EB	ADDB	Inexed	4+	2+
EC	LDD	Inexed	5+	2+
ED	STD	Inexed	5+	2+
EE	LDU	Inexed	5+	2+
EF	STU	Inexed	5+	2+
FO	SUBB	Extended	5	3
F1	CMPB	Extended	5	3
F2	SBCB	Extended	5	3
F3	ADDD	Extended	7	3
F4	ANDB	Extended	5	3
F5	BITB	Extended	5	3
F6	LDB	Extended	5	3
F7	STB	Extended	5	3
F8	EORB	Extended	5	3
F9	ADCB	Extended	5	3
FA	ORB	Extended	5	3
FB	ADDB	Extended	5	3
FC	LDD	Extended	6	3
FD	STD	Extended	6	3
FE	LDU	Extended	6	3
FF	STU	Extended	6	3

Page 2 and 3 Machine Codes

OP	Mnemonic	Mode	~	#
1021	LBRN	Relative	5	4
1022	LBHI	Relative	5(6)	4
1023	LBLS	Relative	5(6)	4
1024	LBHS,LBCC	Relative	5(6)	4
1025	LBCS,LBLO	Relative	5(6)	4
1026	LBNE	Relative	5(6)	4
1027	LBEQ	Relative	5(6)	4
1028	LBVC	Relative	5(6)	4
1029	LBVS	Relative	5(6)	4
102A	LBPL	Relative	5(6)	4
102B	LBMI	Relative	5(6)	4
102C	LBGE	Relative	5(6)	4
102D	LBLT	Relative	5(6)	4
102E	LBGT	Relative	5(6)	4
102F	LBLE	Relative	5(6)	4
103F	SWI2	Inherent	20	2
1083	CMPD	Immed	5	4
108C	CMPY	Immed	5	4
108E	LDY	Immed	4	4
1093	CMPD	Direct	7	3
109C	CMPY	Direct	7	3
109E	LDY	Direct	6	3
109F	STY	Direct	6	3
10A3	CMPD	Indexed	7+	3+
10AC	CMPY	Indexed	7+	3+
10AE	LDY	Indexed	6+	3+
10AF	STY	Indexed	6+	3+
10B3	CMPD	Extended	8	4
10BC	CMPY	Extended	8	4
10BE	LDY	Extended	7	4
10BF	STY	Extended	7	4
10CE	LDS	Immed	4	4
10DE	LDS	Direct	6	3
10DF	STS	Direct	6	3
10EE	LDS	Indexed	6+	3+
10EF	STS	Indexed	6+	3+
10FE	LDS	Extended	7	4
10FF	STS	Extended	7	4
113F	SWI3	Inherent	20	2
1183	CMPU	Immed	5	4
118C	CMPS	Immed	5	4
1193	CMPU	Direct	7	3
119C	CMPS	Direct	7	3
11A3	CMPU	Indexed	7+	3+
11AC	CMPS	Indexed	7+	3+
11B3	CMPU	Extended	8	4
11BC	CMPU	Extended	8	4

LEGEND

~ Number of CPU cycles (less possible push pull or indexed-mode cycles)

Number of program bytes

• Denotes unused opcode

Tabelle D-1: Übersicht über die implementierten Monitorroutinen

Name	Funktion	Adresse
	1. Umwandlungsroutinen	
T7SG	Umwandlung der unteren Tetrade von B in den 7-Segment-Code, Ergebnis in A	F100
B7SG	Umwandlung beider Tetrade von B in den 7-Segment-Code, Ergebnis in D=A,B	F103
D7SG	Umwandlung der vier Tetrade von D in den 7-Segment-Code, Ergebnis in D,Y	F106
	2. Darstellungsroutinen	
CLRDISP	Löschen der Anzeige	F110
SHOWA	Bringt A in die Anzeige, Position in X	F113
SHOWD	Bringt D in die Anzeige, Position in X	F116
SHOWYD	Bringt Y,D in die Anzeige, Position in X	F119
SHOWT7SG	Umwandlung der unteren Tetrade von B in 7-Segment-Code, Darstellung in der Anzeige, Position in X	F11C
SHOWB7SG	Umwandlung beider Tetraden von B in den 7-Segment-Code, Darstellung in der Anzeige, Position in X	F120
SHOWD7SG	Umwandlung der vier Tetraden von D in den 7-Segment-Code, Darstellung in der Anzeige, Position in X	F123
	3. Routinen zur Bearbeitung des Anzeige-Puffers Adresse des Puffers in X	
CLRDBUF	Löschen des Anzeigepuffers	F130
SHOWDBUF	Übertragen des Puffers in die Anzeige	F133
RRDBUF	Rotieren des Puffers um eine Stelle nach rechts	F136
RLDBUF	Rotieren des Puffers um eine Stelle nach links	F139
COPYDBUF	Kopieren eines 2. Puffers in den Anzeigepuffer	F13C
	4. Routinen zur Abfrage der Tastatur	
KEY	Lesen der Tastatur ohne Warten, Tastencode nach B	F140
HALTKEY	Lesen der Tastatur mit Warten, Tastencode nach B	F143
SHOWKEY	Lesen der Tastatur ohne Warten und Anzeigen, Tastencode nach B	F146
SHOWHKEY	Lesen der Tastatur mit Warten und Anzeigen, Tastencode nach B	F149
INDATA	Einlesen eines 8-bit-Datums über die Tastatur, Datum nach A, Tastencode nach B	F14C
SHOWDATA	Einlesen eines 8-bit-Datums über die Tastatur und Anzeigen, Datum nach A, Tastencode nach B	F150
INADR	Einlesen einer 16-bit-Adresse über die Tastatur, Adresse nach Y, Tastencode nach B	F153
SHOWADR	Einlesen einer 16-bit-Adresse über die Tastatur und Anzeigen, Adresse nach Y, Tastencode nach B	F156
	5. weitere Routinen	
DLY1MS	Schleife zur Zeitverzögerung, Zeitdauer in Y vorgegeben	F160
RANDOM	Pseudo-Zufallszahlengenerator, alter und neuer Wert in Y	F163
COPYXD	Verschieben von Speicherbereichen, Startadressen in X,Y, Länge in D	F166