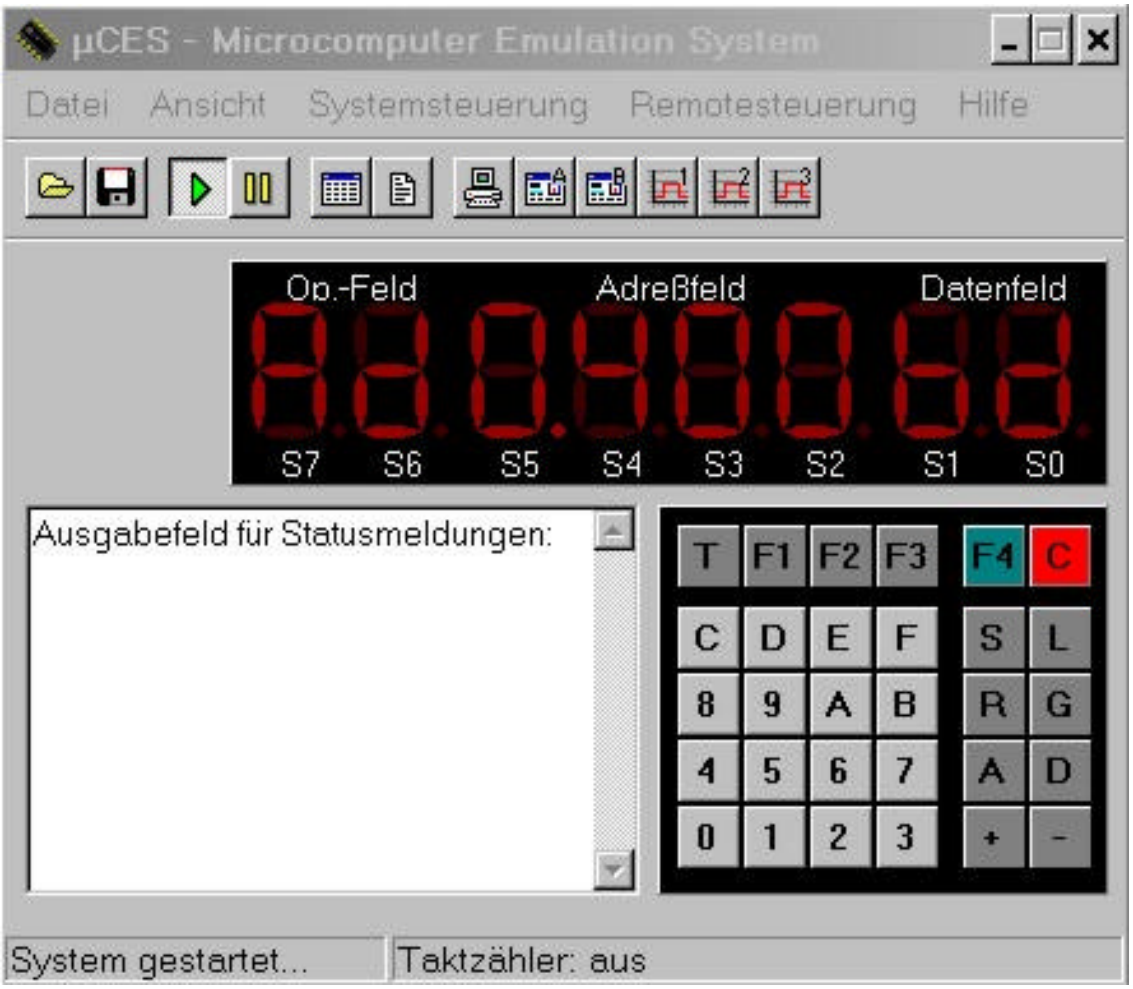
Skript zum Mikrorechner-Praktikum

Kapitel 2:

Assemblerprogrammierung und Software-Entwicklungswerkzeuge

Autoren: Holger Küfner und Helmut Bähring



Inhalt des Kapitels 2:

2 Einführung in die Assemblerprogrammierung und Software-Entwicklungswerkzeuge

2.1 Einleitung

Programmiersprachen können in der Informatik in zwei Klassen eingeteilt werden[1]. Bei maschinenorientierten Sprachen (Assemblersprachen) bestehen die Programme aus prozessorspezifischen Befehlen. Dies bedeutet, das Programm läßt sich nur für einen speziellen Prozessortyp, höchstens für eine ganze Prozessorfamilie, verwenden. Im Gegensatz hierzu wird von "höheren" oder "problemorientierten" Programmiersprachen gesprochen, wenn die Programme aus Anweisungen bestehen, die unabhängig von einer bestehenden Prozessorfamilie definiert worden sind. Diese Sprachen sind hardwareunabhängig und orientieren sich an den zu bearbeitenden Problemfeldern. Vergleicht man diese beiden Klassen von Programmiersprachen, so zeigen sich folgende Unterschiede:

- Maschinenorientierte Programmiersprachen beinhalten einen speziellen auf den Prozessor oder die Prozessorfamilie zugeschnittenen Befehlsvorrat. Eine Portierung des Programms auf eine andere Prozessorfamilie oder auf einen Prozessor eines anderen Herstellers erfordert in der Regel ein völliges Neuschreiben des Programms. Hochsprachenprogramme hingegen können wegen ihres normierten Sprachumfanges leicht auf andere Systeme portiert werden.
- Programme, die in einer Assemblersprache geschrieben worden sind, sind schwerer zu verstehen, da der zugrundeliegende Algorithmus in einer "maschinengerechteren", hardwarenahen Beschreibung vorliegt (Laden von Operanden in Prozessorregister, Stackoperationen, usw). Dagegen lassen sich in Hochsprachen Algorithmen abstrakter formulieren und einfacher modellieren (imperative, funktionale, prädikative Sprachen).
- Um optimal in einer maschinenorientierten Sprache programmieren zu können, ist ein fundiertes Wissen über den logischen Aufbau des zugrundeliegenden Prozessors nötig. Bei höheren Programmiersprachen ist dies nicht notwendig, da diese hardwareunabhängig sind.
- Maschienenorientierte Programme können effizienter gestaltet werden, da sie die speziellen, leistungssteigernden Strukturen und Befehle des Prozessors optimal nutzen können. Nicht alle Aufgabenstellungen sind in einer höheren Programmiersprache effizient lösbar, dies gilt besonders für systemnahe und zeitkritische Abläufe (Betriebssysteme, Automatisierungs- und Industrieanwendungen).
- Hochsprachen stellen Kontrollkonstrukte für die Programmierung zur Verfügung. In maschinenorientierten Sprachen müssen diese nachgebildet werden.
- Hochsprachenprogramme werden durch Compiler und Interpreter auf das Zielsystem übersetzt, maschinenorientierte Programme durch einen Assembler.

[1] Sieht man von der Maschinensprache ab, in der die Programme als reine Bitmuster vorliegen.

Beiden Klassen ist gemeinsam, daß ein Programmierer, der Algorithmen in einer maschinenorientierten oder in einer höheren Programmiersprache exakt ausdrücken will, diese Sprache syntaktisch und semantisch beherrschen muß. Syntax ist hierbei die "Lehre vom Satzbau". Ein Programm muß, im Sinne der verwendeten Programmiersprache, grammatikalisch richtig geschrieben sein, damit der Übersetzer (Compiler, Assembler) das Programm lesen und korrekt auf das Zielsystem übersetzen kann. Syntaktische Fehler werden vom Übersetzer erkannt und durch Fehlermeldungen angezeigt. Semantik ist die "Lehre von der inhaltlichen Bedeutung" einer Sprache. Das heißt, alle Befehle, Anweisungen und Ausdrücke einer Sprache muß der Programmierer in ihren Aktionen, Auswirkungen und Einflußnahmen auf andere Teile der Sprache richtig verstanden haben, damit er ein korrekt arbeitendes Programm schreiben kann. Semantische Fehler werden von einem Übersetzerprogramm nicht erkannt.

Die Entwicklung von Programmen stellt, in der Sprache der Softwaretechnik (*Software-Engineering*), einen kreativen und ingenieurmäßig ablaufenden Prozeß unter Anwendung von "Prinzipien, Methoden und Techniken" für die Implementierung von Programmen und Programmsystemen dar. Ein Programm soll benutzerfreundlich, zuverlässig, wartbar, anpaßbar sowie erweiterbar sein. Der Lebenszyklus von Software wird meistens nach Bild 2.1-1 beschrieben.

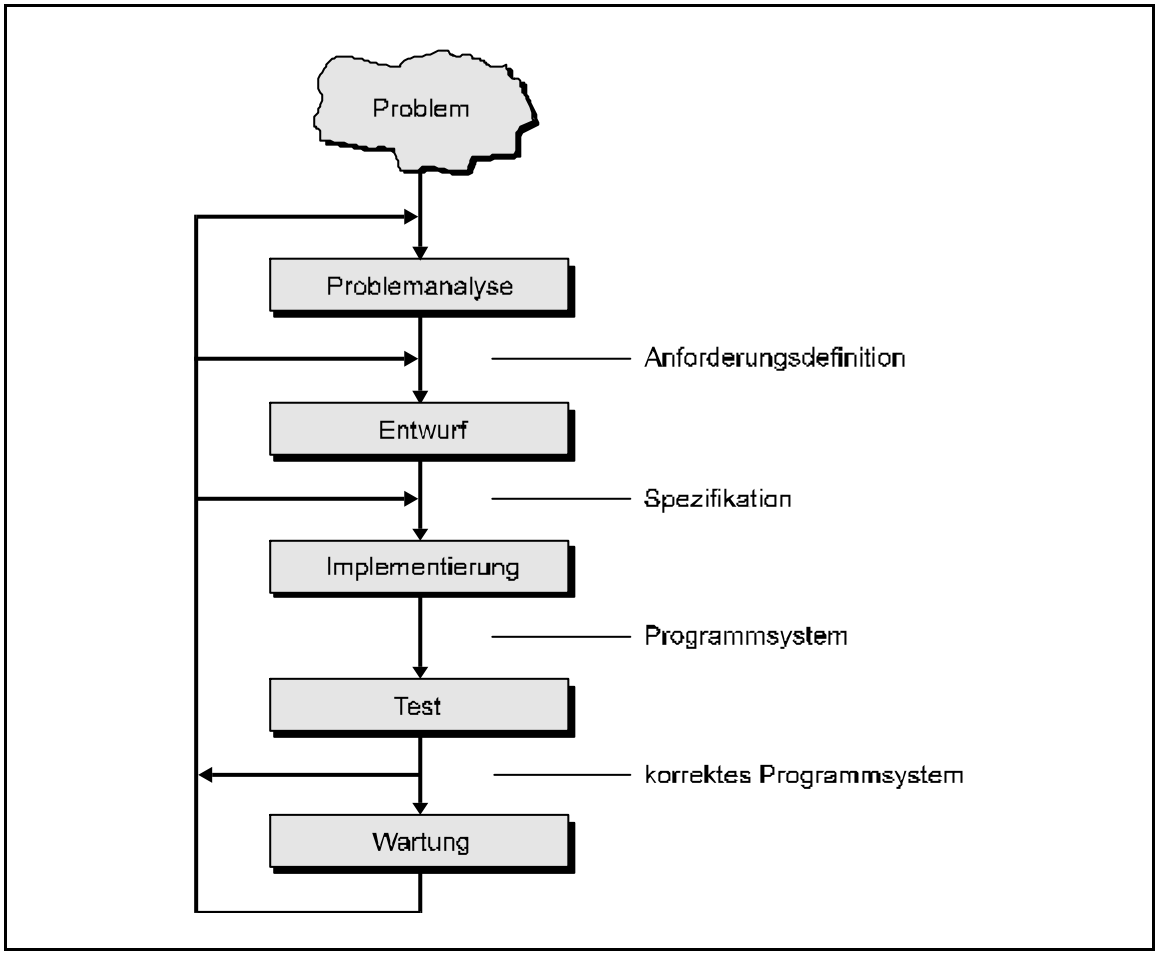


Bild 2.1-1: Software-Lebenszyklus

Am Anfang der Entwicklung steht das "Problem", das vom Programmentwickler analysiert und in seiner Komplexität vollständig erkannt werden muß. Hieraus ergibt sich der erste Entwurf einer algorithmischen Lösung des Problems, die nach Anpassungen an die verwendete Programmiersprache in eine Implementierung mündet. Nach ausführlichen Tests und Dokumentation der Software wird das Programmsystem dem Anwender als korrekt arbeitendes Programm übergeben. Da die heutige Lebensdauer eines größeren Programmpaketes ca. 10 - 20 Jahre beträgt, wird das Paket nach Fertigstellung gepflegt, wiederholt erweitert und an die Wünsche der Anwender angepaßt.

Eine Verfeinerung des Abschnittes "Implementierung" von Bild 2.1-1 stellt das folgende Bild dar. Es zeigt einen allgemeinen Software-Entwicklungszyklus, in dem neben Assemblerprogrammen, C-Programme mit ihren Include-Dateien (H-Dateien) sowie Bibliotheken Verwendung finden. Hilfsdateien, wie z.B. "*Make-Files*", wurden nicht in das Bild aufgenommen.

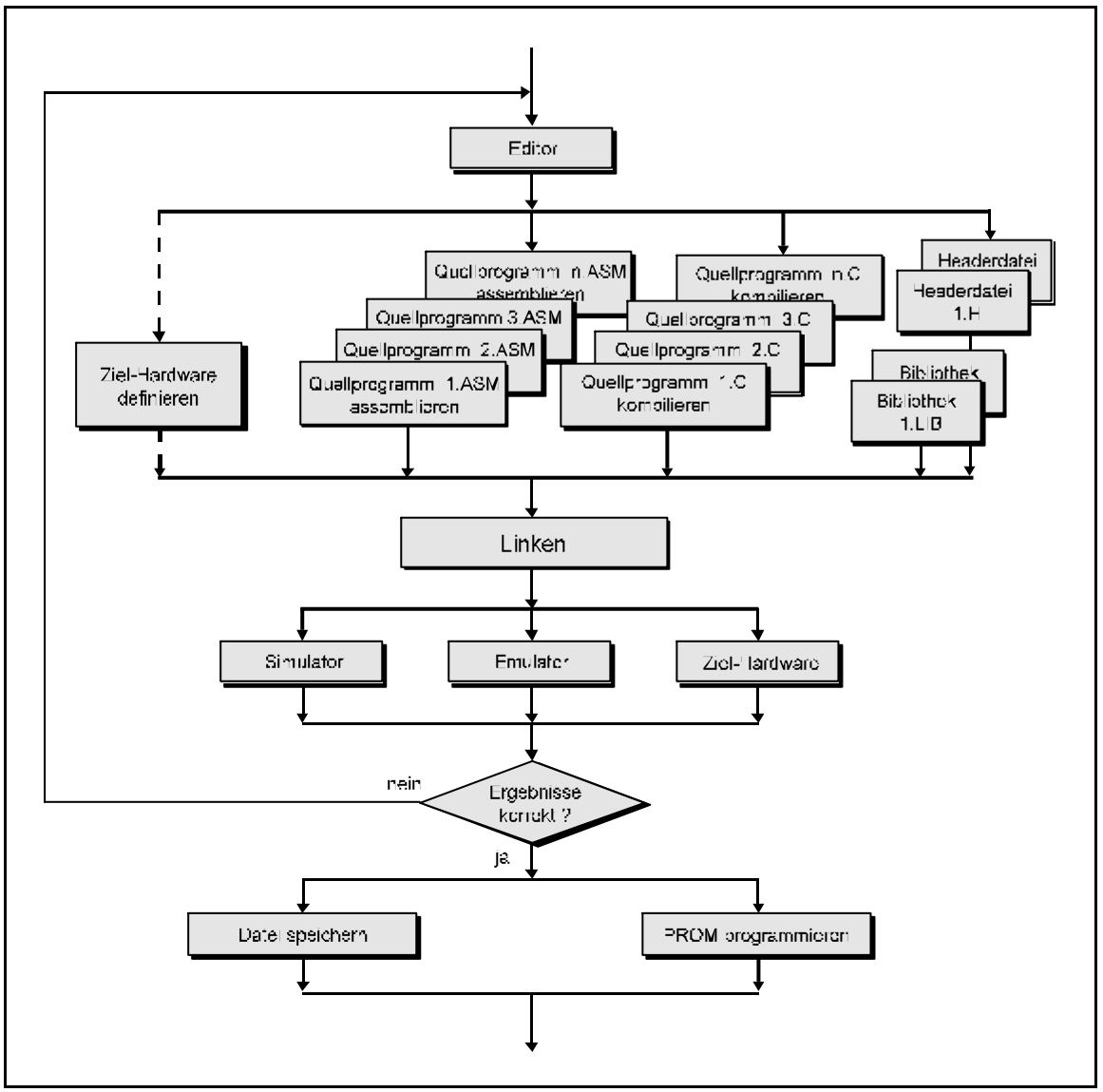


Bild 2.1-2: Software-Entwicklungszyklus

Bei komfortablen Entwicklungsumgebungen (insbesondere bei Umgebungen, die Digitale Signalprozessoren unterstützen) können verschiedene Zielarchitekturen definiert werden. Dies muß vor dem Übersetzen und Linken der Programme erfolgen,

werden. Dies muß vor dem Übersetzen und Linken der Programme erfolgen, damit die erzeugten Maschinenprogramme auf der gewünschten Architektur korrekt ablaufen. Nach Erstellung der Quellprogramme werden Assemblerprogramme assembliert und Hochsprachen-Programme kompiliert (Unterabschnitte 2.2.1 und 2.2.3).
Der Linker bindet alle Teilprogramme in der richtigen "Art und Weise" (Unterabschnitt 2.2.2) zu einem ausführbaren Programm zusammen. Der erzeugte Maschinencode kann dann auf einem Simulator, einem Emulator oder direkt auf der Zielarchitektur ausgetestet werden. Werden semantische Fehler festgestellt (syntaktische Fehler sollten bei der Übersetzung moniert worden sein), so ist das fehlerhafte Quellprogramm zu korrigieren und neu zu übersetzen. Nach einem erneuten Linken ist das Programm auf den behobenen Fehler und etwaige Seiteneffekte zu überprüfen. Wird das Programm als korrekt angenommen, kann es abgespeichert werden, oder in ein EPROM programmiert werden. Zum Beispiel befinden sich die Monitorroutinen des Praktikumsrechners in einem EPROM, das in den Adressbereich $E000 - $FFFF des Rechners eingeblendet wird.

Im folgenden Abschnitt wird eine kurze Übersicht über die in der Industrie verwendeten Entwicklungstools zur Erstellung von Software-Paketen gegeben. Auf die Beschreibung von Hilfsprogrammen, wie Editoren, *Make*-, *Touch*- und *Grep*-Programmen, wurde verzichtet. Abschnitt 2.3 führt in die Assemblerprogrammierung ein, während Abschnitt 2.4 die im Praktikum für den 6809-Prozessor verwendeten Entwicklungstools beschreibt. Abschnitt 2.5 diskutiert schließlich einige praktische Beispiele zur Assemblerprogrammierung.

Literaturverzeichnis

und Empfehlungen für weiterführende Literatur zu Themen diesem Kapitel

[Ana95] Analog Devices: *ADSP-21000 Family Assembler Tools & Simulator Manual*, Analog Devices, Inc., Massachusetts, 1995

[Bae02] H. Bähring: *Mikrorechner-Technik*, 2 Bände, 3. Auflage, Springer-Verlag, Berlin, 2002

[Fae94] H. Färber, *Prozeßrechentechnik*, Springer-Verlag, Berlin, 1994

[Fur91] B. Furth, et al.: *Real-time UNIX-Systems: Design and Application Guide*, Kluwer Academic Publishers, 1991

[Goo89] A. J. van de Goor: *Computer Architecture and Design*, Addision Wesley, Reading, Massachusetts, 1989

[Mot81] Motorola: *8-Bit Microprocessor & Periphal Data* Datenbuch, Motorola Inc., 1981

[Tan95] A. S. Tanenbaum: *Moderne Betriebssysteme*, 2. Auflage, Hanser-Verlag, München. 1995

2.2 Entwicklungswerkzeuge

2.2.1 Assembler, Cross-Assembler, Macro-Assembler, Disassembler

1. Assembler

Der Begriff "Assembler" ist im technischen Sprachgebrauch zweideutig. Zum einen wird mit "Assembler" eine maschinenorientierte Programmiersprache bezeichnet. Zum anderen wird dem Begriff "Assembler" das Übersetzungsprogramm, das ein Programm, welches in der Syntax einer maschinenorientierten Programmiersprache verfaßt ist, in ein Maschinen- (Prozessor-) lesbares Programm übersetzt, zugeordnet. Synonym werden diese Übersetzungsprogramme auch "Assemblierer" genannt.

Eine Assemblersprache ist, wie bereits erwähnt, eine symbolische, maschinen- (prozessor-) orientierte Programmiersprache. Für jede Prozessorfamilie existieren spezielle, auf den Befehlssatz des Prozessors oder der Prozessorfamilie zugeschnittene Assemblersprachen. Die Syntax der Assemblersprachen kann sich von Prozessorhersteller zu Prozessorhersteller, aber auch von Assembler zu Assembler für den gleichen Prozessor ändern. Instruktionen werden in Assemblersprachen in Form mnemonischer Befehle, kurz Mnemonics (Mnemotechnik: Kunst, das Einprägen von Gedächtnisstoff durch besondere Lernhilfen zu erleichtern) notiert. Jedem Operationscode (OP-Code) auf Maschinensprachebene wird eine an die Funktion des Befehls erinnernde Abkürzung zugewiesen. Beipielsweise steht das Mnemonic "STA" für "STORE A", d.h. speichere den Inhalt des Registers A in die Speicherzelle, die nach dem Mnemonic notiert ist. Nach dem Mnemonic folgen kein, ein, zwei oder drei Operanden, die die Adressierungsart sowie Quelle und Ziel des Befehls angeben.

Anhand der Mnemonics setzt das Übersetzungsprogramm, der Assembler, den Programmtext Zeile für Zeile in entsprechende Maschineninstruktionen um. Bei der Umsetzung der Befehle handelt es sich um eine 1:1-Umsetzung, d.h. der Assembler generiert selbständig keinen neuen zusätzlichen Maschinencode. Der erzeugte Code ist eine Folge von Bitmustern, die der jeweilige Prozessor (für den das Programm geschrieben wurde) direkt lesen, d.h. dekodieren, kann. Ein weiterer Bestandteil eines Assemblerprogramms sind sogenannte Assembler-Direktiven, die zur Steuerung des Übersetzungsvorganges sowie des Assemblers benötigt werden. Die Mächtigkeit der Assembler-Direktiven ist von Assembler zu Assembler verschieden. Sie kann sehr rudimentär ausgelegt sein, aber auch z. B. bis zu einem *Conditional-Assembly* durch die Verwendung von speziellen "IF-THEN-ELSE"-Konstrukten reichen.

2. Cross-Assembler

Cross-Assembler, wie z.B. der Motorola Assembler "AS9", erlauben es, Maschinenprogramme, welche auf einem Computer (*host*) geschrieben wurden, für einen vom Host-Prozessor typverschiedenen Prozessor (*target*) zu generieren. Diese ausführbaren Maschinenprogramme können dann auf die Zielarchitektur geladen (*download*) und gestartet werden. Das Übertragen der erzeugten Programme erfolgt überwiegend über die serielle Schnittstelle oder via Netzwerk. Beispielsweise stellt im Falle des Cross-Assemblers "AS9" Ihr PC den *Host* dar, während der Praktikumsrechner das *Target* repräsentiert.

3. Makro-Assembler

Macro-Assembler ermöglichen es, eine bestimmte Folge von Assemblerbefehlen mit einem einzigen, vom Programmierer zu definierenden "Macro-Befehl" aufzurufen. Die Zuordnung einer Befehlssequenz zu einem definierten Namen wird als Definition des Macros bezeichnet. Diese "benutzerdefinierten" Befehle sind eine gewisse Art von Unterprogrammen, die während der Übersetzung anstelle des aufrufenden Befehls als substituierendes Codefragment an dessen Stelle im Maschinenprogramm eingesetzt werden.

Macros können in der Regel in Bibliotheken zusammengefaßt werden und über eine "INCLUDE"-Direktive im Quellprogramm dem Assembler bekannt gemacht werden. Macro-Bibliotheken sind hierbei Dateien, in denen einzelne Makroroutinen abgespeichert werden. Im Gegensatz zu Programmbibliotheken in höheren Programmiersprachen werden Makrobibliotheken nicht übersetzt (assembliert). Der Assembler fügt ab der Stelle, an der die Direktive "INCLUDE" auftritt, die Bibliothek in das Originalprogramm ein. Bei Auftreten eines Macros durchsucht der Assembler die eingefügten Routinen und setzt das Codefragment des Makros anstelle des Aufrufes in das Programm ein. Dieses (unter Umständen mehrmalige) Einsetzen von Code in das Programm unterscheidet ein Makro von einem Unterprogramm.

4. Algebraische Assembler

Algebraische Assembler werden im Bereich der Digitalen Signalprozessoren verwendet. Diese Assembler unterstützen die hohe Parallelität, die Digitale Signalprozessoren auszeichnet. Sie sind in der Regel Macro-Assembler mit sehr mächtigen Assembler-Direktiven. Ihre Notation ist in den meisten Fällen der Sprache C angelehnt. Als Beispiel verdeutliche folgende Sequenz die benutzte Notation:

```
F13=F0*F4,  F9=F9+F12,  F0=DM(I0,M1),  F5=PM(I9,M8);
F12=F0*F5,  F8=F8-F13,  F4=PM(I8,M8);
```

Jede Anweisung in einer Zeile wird im DSP zur gleichen Zeit parallel zu den anderen Anweisungen einer Zeile ausgeführt! Die einzelnen Anweisungen werden durch Kommata voneinander getrennt. Die Zeile muß mit einem Strichpunkt abgeschlossen werden.

5. Disassembler

Ein Disassembler ist ein Programm, welches ein Maschinenprogramm in ein Assemblerprogramm (Quellcodeprogramm) zurück übersetzt. Im Idealfall erhält man als disassembliertes Programm das korrekte Assemblerprogramm, reduziert auf die Mnemonics und den zugehörigen Operanden der unterstützten Assemblersprache. Als Marken und Symbole sind bei disassemblierten Programmen in der Regel die hexadezimalen Werte der Marken und Symbole des Maschinenprogramms eingesetzt und müssen in der Regel "von Hand" nachgearbeitet werden. Der OP-Code des disassemblierten Programms wird in den meisten Fällen zur Orientierung mit ausgegeben.

2.2.2 Binder, Linker, Lader, Boot-Lader, Boot-PROM

1. Binder, Linker

Die Begriffe "Binder" und "Linker" werden in der Informatik synonym benutzt. Da ein komplexeres Programm in der Regel aus mehreren Einzelprogrammen und bereits übersetzten Standardroutinen besteht, müssen die einzelnen Maschinenprogramme zu einem lauffähigen Programm zusammengesetzt werden. Diese Aufgabe übernehmen sogenannte Binder (Linker). Die zu bindenden Programme liegen in der Regel als Objekt-Dateien vor. Objekt-Dateien sind hierbei Maschinenprogramme, in denen einzelne Adressen durch den Assembler noch nicht aufgelöst (zugeordnet) werden konnten, da die zugehörige Referenz in anderen Programmen oder Bibliotheken liegt. Der Linker löst diese Zuordnungsprobleme auf, und verschiebt gegebenenfalls bei Doppelbelegung von Speicheradressen einzelne Code-/Datenbereiche.

2. Lader

Der Lader ist eine Routine des Betriebssystems oder des Monitors. Er hat die Aufgabe, das auszuführende Maschinenprogramm in den Speicher des Rechners zu "laden". Hierzu erhält das Programm vom Betriebssystem die Speicheradresse mitgeteilt, ab der das Programm abgelegt werden soll. Der Lader übernimmt hierbei auch etwaige Adreßumrechnungen zwischen den im Maschinenprogramm vorkommenden absoluten Adressen und den für den Speicherbereich gültigen Adressen.

3. Boot-Lader

Ein Boot-Lader (Ur-Lader) ist ein Hilfsprogramm des Entwicklungssystems, mit dem ein entwickeltes Programm ohne Betriebssystem oder Monitor auf einem Target ablauffähig ist. Dieses Hilfsprogramm installiert und startet das eigentliche Programm. Bei Assembler-Programmen kann diese Aufgabe, abhängig von dem Prozessortyp und dem Programmodus, auf ein Minimum reduziert werden oder völlig fehlen. Bei einem Hochsprachenprogramm, das in der Regel eine gewisse Betriebssystemunterstützung erwartet, hat der Boot-Lader diese Unterstützung zu leisten. Beispielsweise müssen verschiedene Controller und der Stack initialisiert werden, Speicherverwaltungstabellen angelegt und mit Zugriffspointern versehen werden, Interruptvektortabellen mit den zugehörigen Routinen zur Verfügung gestellt werden sowie unter Umständen zusätzliche Schnittstellen, wie Netzwerkkarten, initialisiert werden.

4. Boot-PROM

Ein Boot-PROM ist ein EPROM, in dem ein *Startup*-Code, Monitorroutinen oder sogar ein vollständiges Betriebssystem gespeichert ist. Beispielsweise enthält Ihr PC ein EPROM, in dem die grundlegenden Routinen zum Initialisieren des PCs untergebracht sind. Nach dem Einschalten des PCs werden diese Routinen (BIOS: Basic Input Output System) ausgeführt. Hierdurch werden alle Controller und weiteren Elemente des System korrekt initialisiert und können für das weitere "Hochfahren" des Systems genutzt werden. Hierzu wird nach Abarbeitung der BIOS-Routinen auf den Boot-Sektor Ihrer Festplatte (oder Diskette) verzweigt, um das eigentliche Betriebssystem in den Hauptspeicher zu laden und zu starten.

2.2.3 Compiler, Cross-Compiler, Interpreter, Debugger

1. Compiler, Cross-Compiler

Ein Compiler übersetzt Programme, welche in einer höheren Programmiersprache geschrieben worden sind, in Maschinenprogramme, die unter einem bestimmten Betriebssystem auf bestimmten Prozessoren ablauffähig sind. Das ablaufende Programm benutzt verschiedene Dienste des Betriebssystems, die dem Compiler bekannt sein müssen. Hochsprachenprogramme bestehen in der Regel aus mehreren Teilprogrammen, die nach erfolgter Übersetzung zusammen gebunden werden müssen. Da Hochsprachenprogramme komfortable Kontrollstrukturen, wie z.B. Schleifenkonstrukte, unterstützen, kann der Compiler diese Strukturen nicht zeilenweise auf Maschinensprachebene übersetzten. Er berücksichtigt vielmehr die "Umgebung" einer Zeile bei der Übersetzung mit, um ein sinnvolles Maschinencodefragment zu erzeugen. Hierdurch kann der Compiler z. B. auch erkennen, daß Zeilen eines Quellcodeprogrammes nie ausgeführt werden (*dead code*), und damit auch nicht in Maschinencode übersetzt werden müssen.

Cross-Compiler werden bei der Programmierung von Mikrocontrollern eingesetzt. Das Quellprogramm wird auf einem leistungsstarken Rechensystem, z.B. einem PC, für eine bestimmte Zielarchitektur entwickelt und übersetzt. Wie bei Cross-Assemblern kann das Programm danach zum Testen auf die Zielarchitektur geladen werden.

2. Interpreter

Ein Interpreter ist ein Programm, welches ein Quellcodeprogramm Zeile für Zeile einliest und unmittelbar nach der syntaktischen Analyse ausführt. Eine Übersetzung und Abspeicherung in ein vollständig vorliegendes Maschinenprogramm findet hierbei nicht statt. Der Interpreter bearbeitet das Quellprogramm sequentiell, analysiert jede Instruktion und führt diese sofort aus. Syntaktische Fehler werden hierbei erst bei der Befehlsinterpretation entdeckt. Der Vorteil von Interpretern liegt in einer schnelleren Testmöglichkeit von Programmen, da der eigentliche Compilierungsvorgang entfällt. Nachteilig wirkt sich die wesentlich längere Rechenzeit bei der Ausführung von Programmen aus, da die Befehlszeilen interpretiert und Variablenwerte über Tabellen ausgetauscht werden müssen.

3. Debugger

Mittels eines Debuggers können semantische Fehler zur Laufzeit auf Assembler- oder Hochsprachenebene gezielt untersucht werden. Dazu wird das zu untersuchende Programm mit zusätzlichem "Debug-Code" compiliert und unter der Kontrolle des Debuggers gestartet. Zur Fehlersuche bieten Debugger mehrere Kontrolloptionen. Beispielsweise kann ein Programm "befehlsweise" ausgeführt werden (*Trace*), in dem entsprechend dem Programmfluß ein Befehl ausgeführt wird und dann das Programm angehalten wird. Danach ist es möglich, Register-, Variablen- oder Speicherinhalte zu kontrollieren oder für den nächsten *Trace*-Schritt zu verändern. Ein weitere Möglichkeit, in einen Haltepunkt zu gelangen, ist die Verwendung von *Breakpoints*. Hierbei wird nach Definition eines *Breakpoints* für einen Befehl in einem bestimmten Programmsegment, das Programm gestartet und wie zur normalen Laufzeit bedient. Läuft das Programm während seiner Abarbeitung auf einen *Breakpoint*, so wird das Programm an dieser

Stelle unterbrochen und man hat die Möglichkeit, wie im *Trace*-Modus, Variablenwerte zu kontrollieren. Ein nachfolgender Trace oder das Setzen eines anderen Breakpoints ist in der Regel möglich. Mit diesen Optionen ist es möglich, gezielt das Verhalten von als fehlerhaft vermuteten Routinen zu untersuchen, Parameterübergaben zwischen Unterprogrammen zu kontrollieren, Feldüberläufe oder falsche Wertzuweisungen zu entdecken.

2.2.4 Simulatoren, Emulatoren

Ein Simulatorprogramm bildet das Verhalten eines Systems auf der Basis von Modellen dieser Zielarchitektur nach. Die Abarbeitung eines Programms der Zielarchitektur erfolgt in der Art eines Interpreters. In der Regel wird bei Simulatoren nur der Programmfluß nachgebildet. Das zeitliche und physikalische Verhalten der Zielarchitektur wird dabei nicht berücksichtigt.

Ein Emulator bildet dagegen die Eigenschaften einer Zielarchitektur derart nach, daß ablauffähige Programme dieser Zielarchitektur auf einem Host-System mittels Emulator direkt abgearbeitet werden können. Eine Interpretation findet nicht statt. Stattdessen werden die benutzten Hardwarekomponenten (Prozessoren, Speicher, Controller, ...) vom Emulator derart nachgebildet, daß sie die Maschineninstruktionen der Zielarchitektur direkt verarbeiten und in entsprechende Aktionen umsetzen können. Das zeitliche und/oder physikalische Verhalten der einzelnen Komponenten kann hierbei berücksichtigt werden. Im wesentlichen existieren heutzutage zwei Ausprägungen von Emulatoren: hardwareorientierte Emulatoren und softwareorientierte Emulatoren.

Unter hardwareorientierten Emulatoren versteht man im allgemeinen komplexere FPGA[2]- oder Prozessor-gesteuerte Geräte, die in der Zielarchitektur anstelle des Prozessors / Controllers über einen Stecker mit Kabel eingesteckt werden. Diese Emulatoren bilden das zeitliche Verhalten der emulierten Komponenten in idealer Weise nach, ohne das Zielsystem zeitlich zusätzlich zu belasten. Es sind allerdings für diese Emulatoren hohe Investitionskosten notwendig. Softwareorientierte Emulatoren bilden die Hardware des Zielsystems nur durch die Anwendung von Software nach. Die Kosten für diese Emulatoren sind geringer. Es ist allerdings eine sehr genaue zeitliche Abbildung des Verhaltens, insbesondere des Echtzeitverhaltens, im Vergleich zu den hardwareorientierten Emulatoren nicht möglich.

2.2.5 Entwicklungsumgebungen

Bis Anfang der neunziger Jahre bestand die klassische Entwicklungsumgebung aus einem Editor, einem Compiler, einem Assembler und einem Debugger. Bei Mikrocontrollern wurde für eingebettete Anwendungen (*embedded control*) zusätzlich ein hardwareorientierter Emulator (*In-Curcuit* Emulator) und ein Hostsystem mit Cross-Compiler verwendet. Der Host war in der Regel eine UNIX-Workstation. Echtzeitbetriebssysteme wurden selten eingesetzt. Heutzutage führt der Trend zu immer weniger Hardware und immer mehr Software, um die für ein System immer kürzer werdenden Entwicklungszyklen (*time-to-market*) auffangen zu können. In Zeiten harter Konkurrenz

[2] FPGA: *Field Programmable Gate Array*

können Applikationen durch Simulatoren und Software-Emulatoren bereits entwickelt werden, obwohl die darunter liegende Hardware noch nicht verfügbar ist.

Eine heutige Entwicklungsumgebung präsentiert sich dem Entwickler mit einer graphischen Benutzeroberfläche, in der alle Entwicklungswerkzeuge integriert sind. Als Host-System wird eine Workstation oder ein PC benutzt. Echtzeitbetriebssysteme sind bei *embedded control*-Anwendungen die Regel. An modernen Entwicklungswerkzeugen werden neben C-Compilern, C++-Compilern, Assemblern, mächtigen Bibliotheken, Linkern und Debuggern auch Simulatoren und softwareorientierte Emulatoren angeboten. Die Umgebungen werden durch Dienstleistungsprogramme, wie Projektmanager, Browser, Versionsverwaltung, Revisionskontrolle und leistungsfähige Online-Hilfe ergänzt. Für das Hardware-Debugging wird ein universeller Logikanalysator eingesetzt. Nur in speziellen Problemfällen bzw. zum abschließenden Test des Projektes wird auf hardwareorientierte Emulatoren und Hardware-Debugger zurückgegriffen.

2.3 Einführung in die Assemblerprogrammierung

2.3.1 Grundlegendes zur Assemblerprogrammierung

Maschinenorientierte Sprachen versus problemorientierte Sprachen

Assemblersprachen stehen in unmittelbarem Zusammenhang mit der zugrundeliegenden Maschinensprache und sind somit stark prozessorabhängig. Bei der Entwicklung von Assemblerprogrammen ist es daher nicht möglich, sich ausschließlich auf die Umsetzung des Algorithmus in das Programm zu konzentrieren, so wie man es von höheren Programmiersprachen gewöhnt ist. Assemblersprachen orientieren sich in ihrer Semantik an den implementierten Befehlen des Prozessors und nicht an einer problemorientierten und den Programmierer unterstützenden Darstellung der zu modellierenden Algorithmen. Dieser Umstand behindert zum einen eine zügige Umsetzung des verwendeten Algorithmus in das Assemblerprogramm, eröffnet zum anderen aber die Möglichkeit einer effizienteren Lösung des Problems als in einer Hochsprache. Nach einer Studie von Motorola treten Maschinenbefehle in 8-bit Prozessoren in Standard-Anwendungen mit folgender Häufigkeit auf:

Transferbefehle	39%
Verzweigungsbefehle	17%
Unterprogrammaufrufe	13%
Arithmetische Befehle	3%
Vergleichsbefehle	6%
restliche Befehle	22%

Der hohe Prozentsatz an Transferbefehlen belegt, daß allein eine geschickte Programmierung der Ein- und Ausgabe auf Assemblerebene einen Geschwindigkeitsgewinn erbringen kann.

Anforderungen an Assemblierer

Assemblierer müssen aus der Sicht des Entwicklers ein Mindestmaß an Anforderungen erfüllen, damit ein produktives Programmieren garantiert ist:

> Unterstützung des vollständigen Befehlssatzes und der Adressierungsarten der unterstützten Prozessorfamilie.
> Symbolische Darstellung der Prozessorbefehle.
> Verwendung von symbolischen Namen für Konstanten, Datenbereiche und Sprungziele.
> Unterstützung von Assembler-Direktiven zur Definition von Konstanten und Zeichenfolgen (*Strings*), zur Initialisierung von Speicherbereichen und zur Manipulation des Programmzählers.
> Berechnung von arithmetischen Ausdrücken und Zuweisung an Konstanten und Datenbereiche.
> Möglichkeiten zur Kommentierung und Strukturierung des Programms.
> Möglichkeiten zur Ausgabe von Symbol- und *Cross-Reference*-Tabellen.
> Bei der Erkennung von syntaktischen Fehlern Anzeigen des Fehlertypes und der Programmzeile, in der Fehler aufgetreten sind.

Bei der Entwicklung größerer Programmroutinen können zusätzlich folgende Optionen nützlich sein:

> Bereitstellung von Assembler-Direktiven zur Makrodefinition.
> Möglichkeit der bedingten Assemblierung. Abhängig von "Konstanten" werden nur bestimmte Programmteile während eines Übersetzungslaufes assembliert.
> Modulare Programmentwicklung (Import und Export von Symbolen).
> Deklaration von Datentypen (*byte, word, double word*), Datenstrukturen (*records, arrays*) und Speichersegmenten.

2.3.2 Funktionsweise von Assemblierern

Um die grundlegenden Anforderungen, die an einen Assembler gestellt werden, zu erfüllen, benutzt ein Standard-Assembler als Datenstrukturen mindestens zwei Tabellen und einen Zähler.

Informationen über den zugrundeliegenden Befehlssatz sind in der sogenannten Zuordnungstabelle (Befehlstabelle) statisch gespeichert. Sie ermöglicht eine Abbildung zwischen der symbolischen Darstellung der Prozessorbefehle (und ihren Adressierungsarten) und den zugehörigen binären Bitmustern der Maschineninstruktionen. Darüber hinaus ist in dieser Tabelle für jeden Befehl die Anzahl der im Speicher zu reservierenden Bytes und die Anzahl der benötigten Prozessortaktzyklen abgelegt. Die Zuordnungstabelle entspricht in ihrem grundlegendem Aufbau den ersten Feldern der Tabellen B-1, B-2 und B-3 des Anhangs B des ersten Kapitels.

Zur Verwaltung der vom Programmierer definierten Symbole baut der Assembler während des Übersetzungsvorganges die Symboltabelle auf. In dieser Tabelle werden alle Symbole mit ihren Werten (Konstanten, Sprungziele) abgelegt. Auf Anforderung durch den Benutzer wird diese Tabelle vom Assembler ausgegeben.

Zur Vergabe von eineindeutigen Speicheradressen für die einzelnen Befehle des zu generierenden Maschinenprogramms ist ein Zähler, der Adreßzähler, notwendig. Er wird im Laufe der Abarbeitung um die jeweils benötigte Anzahl von Bytes (Zuordnungstabelle) für den aktuell zu verarbeitenden Befehl inkrementiert. Der Adreßzähler modelliert den Programmzähler des Prozessors.

Neben den obigen Datenstrukturen ist für einen Assembler im wesentlichen nur noch ein *Parser* (*to parse*: Satz grammatikalisch zergliedern), ein Modul zur Fehlerbehandlung (*exception handling*), ein Modul zur Ein-/Ausgabebehandlung und ein Modul zur Steuerung des Übersetzungsvorganges notwendig. Ein *Parser* ist eine Programmroutine, die eine eingelesene Programmzeile des Quellprogrammes (*String*) syntaktisch korrekt nach *token* (hier: Symbole, Mnemonics, Operanden) durchsucht und die Befehlszeile entsprechend der Syntax in Felder zerschneidet (softwaretechnisch wird ein Ableitungsbaum mittels einer kontextfreien Grammatik erzeugt).

Der Übersetzungsvorgang ist ein mehrmaliges sequentielles Lesen und Verarbeiten des Quellprogramms. In der Regel werden für die Generierung eines Maschinenprogramms zwei Lesezyklen (*two pass assembler*) benötigt, um alle Referenzen auflösen zu können. Bei Assemblern mit sehr mächtigen Assembler-Direktiven ist noch ein dritter Durchlauf nötig. Das Quellprogramm wird von dem Assembler Zeile für Zeile abgearbeitet. Als Ergebnis ist das erzeugte Maschinenprogramm eine 1:1-Übersetzung des eingegebenen Quellprogramms.

Bei einem *two pass assembler* wird im ersten Durchlauf die Symboltabelle aufgebaut und versucht, allen Symbolen während dieses Durchlaufes ihre zugewiesenen bzw. zu berechnenden Werte zuzuordnen. Im zweiten Durchlauf werden Assembler-Direktiven nochmals berechnet und das Maschinenprogramm generiert. Im allgemeinen ist es nicht möglich, die Übersetzung in einem Durchlauf durchzuführen, da Vorwärts-Referenzen (*forward reference*) nicht sofort aufgelöst werden können. Beispielsweise hat das folgende Codefragment eine Vorwärts-Referenz, d.h. der Marke "NEXT" wird erst in einer der folgenden Programmzeilen ein Wert (Adresse) zugewiesen.

```
        CMPA    #$0100
        BEQ     NEXT
        ...     ...
NEXT:   EXG     X,Y
```

Da die Zuweisung später erfolgt, kann im ersten Durchlauf dem Befehl "BEQ" nicht sofort die relative/absolute Adresse des Sprungziels als Operand mitgegeben werden. Aus diesem Grund wird im ersten Durchlauf eine vollständige Symboltabelle aufgebaut, durch die im zweiten Durchlauf die Generierung des Maschinenprogrammes ermöglicht wird. Wird ein Sprungziel zuerst definiert und später darauf Bezug genommen (*backward reference*), so kann eine Zuordnung sofort erfolgen. Für Interessierte ist im folgenden der genaue Programmablauf einer Assemblierung beschrieben.

Pass 1

Zu Beginn des Übersetzungsvorganges wird der Adreßzähler mit Null initialisiert. Nach Einlesen der ersten Programmzeile und fehlerfreiem Zerlegen in einzelne *token*, wird überprüft, ob das Programmende erreicht ist und gegebenenfalls zu *pass 2* verzweigt. Ist dies nicht der Fall, so wird eine eventuell vorliegende Assembler-Direktive verarbeitet und danach ein eventuell vorhandendes Symbol (Sprungziel) in die Symboltabelle (bei *backward reference* mit Speicheradresse entsprechend dem Inhalt des Adreßzählers) aufgenommen. Danach wird der Adreßzähler entsprechend der benötigten Bytes des vorliegenden Befehls erhöht und die nächste Programmzeile eingelesen. Wird ein syntaktischer Fehler bei der Zerlegung erkannt bzw. wird ein Symbol im Quellcodeprogramm zweimal definiert, so wird eine Fehlermeldung für diese Programmzeile ausgegeben und der Übersetzungsvorgang gestoppt.

Pass 2

Zu Beginn des zweiten Durchlaufs wird die Symboltabelle überprüft und bei nicht- oder falschdefinierten Symbolen eine Fehlermeldung ausgegeben. Danach wird die erste Programmzeile eingelesen und zerteilt. Im Anschluß wird überprüft, ob das Programmende erreicht ist. Ist dies der Fall, so wird das generierte Maschinenprogramm abgespeichert, gewünschte Tabellen für Benutzer ausgegeben und das Programm beendet. Ist das Programmende nicht erreicht, so wird der Operand einer eventuell vorhandenen Assembler-Direktive berechnet und der Adreßzähler gegebenenfalls erhöht und die nächste Programmzeile eingelesen. Bei Vorliegen eines Befehls wird der Operand (mit Hilfe der Symboltabelle) berechnet und der Maschinencode für den Befehl erzeugt. Danach wird der Adreßzähler entsprechend der benötigten Bytes des verarbeiteten Befehls erhöht und die nächste Programmzeile eingelesen.

Praktische Übung P2.3-1:

Entwerfen Sie je ein Flußdiagramm für den ersten Durchlauf sowie für den zweiten Durchlauf des Übersetzungsvorganges bei einem *two pass assembler*.

2.3.3 Syntax von Assemblerprogrammen

Assemblerprogramme bestehen aus einer Folge von Befehlszeilen. Jede Befehlszeile besteht aus einer Sequenz von ASCII-Zeichen, die in der Regel mit einem Zeilenende-Zeichen (*line feed*) abgeschlossen ist. Der gültige Zeichensatz von Assemblern unterscheidet sich in den meisten Fällen nur geringfügig, allerdings kann sich die Semantik bei der Anwendung der Sonderzeichen stark unterscheiden.

Befehlszeilen von Assemblerprogrammen sind überwiegend in die Felder "Marke" (*Label*), "Operator", "Operand" und "Kommentar" nach folgendem Schema aufgeteilt:

Marke	Operator	Operand	Kommentar

Die ersten drei Felder werden meistens durch mindestens ein Leerzeichen voneinander getrennt. Das Kommentarfeld wird in der Regel durch einen führenden Strichpunkt eingeleitet. Das Operatorfeld muß, falls das Markenfeld leer ist, durch mindestens ein Leerzeichen oder ein Tabulator-Zeichen vom Zeilenanfang getrennt sein.

Im Markenfeld werden Symbole als Sprungziele oder als Konstanten definiert. Diese Symbole erhalten den aktuellen Inhalt des Adreßzählers oder eine Konstante als Wert zugewiesen. Die Definition dieser Symbole ist nur einmal möglich, d.h. der Wert bleibt während der ganzen Laufzeit des Programmes konstant. Werden Symbole als Sprungziele definiert, so fordern manche Assembler, die Symbole mit einem Doppelpunkt abzuschließen.

Im Operatorfeld stehen die Mnemonics (symbolische Darstellung der Prozessorbefehle) oder Assembler-Direktiven. In einigen Syntaxbeschreibungen von Assemblern wird die Unterscheidung zwischen Mnemonics und Assembler-Direktiven durch einen vorgestellten Punkt (.) bei den Direktiven vereinfacht.

Das Operandenfeld hinter einem Prozessorbefehl kann, in Abhängigkeit von der verwendeten Adressierungsart, leer sein oder bis zu drei Operanden enthalten. Wird die implizite Adressierung verwendet, bleibt das Operandenfeld leer. Unterstützt der Prozessor (und damit der Assembler) Drei-Adreß-Befehle, so können drei Operanden, durch Kommata getrennt, im Operandenfeld stehen. Bei einer Assembler-Direktive im Operatorfeld kann die Anzahl der Operanden stark varieren.
Ein Operand kann eine Konstante, ein Symbol oder ein Ausdruck sein. Ein Ausdruck kann bei den meisten Assemblern aus mehreren Symbolen und Konstanten bestehen, die sich über eine Rechenvorschrift zu einem konstanten Wert zusammenfassen lassen. Beispielsweise wäre ein gültiger Ausdruck "$F0 & WIDTH + 25 ", bei dem das Symbol "WIDTH" eine Konstante (Symbol) darstellt.

Das Kommentarfeld dient der Dokumentierung und der Strukturierung des Programms. Es wird in der Regel durch einen führenden Strichpunkt (;) oder ein Doppelkreuz (#) eingeleitet. Der Kommentar kann auch alleine in der Programmzeile stehen und am Anfang der Zeile beginnen.

Zur Steuerung des Übersetzungsvorganges werden Assembler-Direktiven benutzt. Die beiden wichtigsten Direktiven, "ORG" und "EQU", sind bei fast allen Assemblern mit einer identischen Syntax und Semantik zu finden.

Mittels der Direktive "ORG" kann auf den Adreßzähler des Assemblers direkter Einfluß genommen werden. Beispielsweise weist die Programmzeile " ORG $0400" dem Adreßzähler die hexadezimale Adresse "0400" zu. Das Dollarzeichen kennzeichnet bei den meisten Assemblern die nachfolgende Zahl als hexadezimal.

Durch die Direktive "EQU" wird einem Symbol im Markenfeld ein Wert, im Sinne einer Konstantendefinition, zugewiesen. Dieser Wert ist für das Symbol während der ganzen Laufzeit des Programms konstant. Zum Beispiel wird durch die Programmzeile "CLRDSP EQU $F110" dem Symbol "CLRDSP" der hexadezimale Wert "F110" zugewiesen, der der Einsprungadresse der Monitorroutine "CLRDSP" zum Löschen der Anzeige des Praktikumsrechners entspricht.

Zum Abschluß werden zwei Codefragmente als Beispiele für verschiedene Darstellungsweisen von Assemblerprogrammen vorgestellt. Das erste Beispiel zeigt einige Zeilen eines Assemblerprogramms für einen MIPS-Prozessor, der Drei-Adreß-Befehle unterstützt. Das zweite Beispiel zeigt die Syntax des im Praktikum verwendeten 6809-Assemblers "AS9".

Tabelle 2.3-1: Codefragment eines MIPS-Assemblers

```
          .text
          .globl  main

main:     addu   $sp,$sp,32     # Restore last Stack frame
          sw     $ra,20($sp)    # Save return address
          sw     $fp,16($sp)    # Save old frame pointer
          move   $a1,$v0        # Move result to $a1
```

Tabelle 2.3-2: Codefragment des 6809-Praktikumsassemblers "AS9"

```
          ORG   $0400       ; Beginn Programmbereich
          LDA   #$3E        ;CB2 als Steuerleitung (High),
          STA   CRB         ;Flanke an CB1, disable
NEWLINE   CLR   DRB         ;Port B loeschen
          ...   ...
          BRA   NEWLINE     ;Ruecksprung neue Eingabe
DLY1MS    EQU   $F160
HALTKEY   EQU   $F143
```

Als Beispiel für eine vollständige Syntaxbeschreibung eines Assemblers möge die Beschreibung des 6809-Assemblers, in Unterabschnitt 2.4.1, dienen.

2.3.4 Kontrollstrukturen in Assemblerprogrammen

Programmiererinnen und Programmierer, die es gewohnt sind, in einer problemorientierten Sprache zu denken, fällt der Übergang zu einer maschinenorientierten Programmiersprache meistens schwer, da die gewohnten Kontrollstrukturen nicht unterstützt werden. Um den Übergang zu erleichtern, werden in dieser Einführung in die Assemblerprogrammierung problemorientierte Kontrollstrukturen, wie sie aus Pascal, Modula oder C bekannt sind, den entsprechenden äquivalenten Assemblerroutinen gegenübergestellt. Anhand der Beispiele sollen Sie erkennen, welche Befehle an welchen Stellen des Assemblerprogramms notwendig sind (dunkel unterlegt), um den gewünschten Programmfluß zu erreichen.

In einigen Unterabschnitten finden Sie Hinweise und Beispiele für eine sinnvolle, prozessor-angepaßte Verwendung von Registern und Befehlen. Die Beispiele sind in der Ihnen bekannten Syntax des 6809-Prozessors notiert. Der Befehlssatz und die Adressierungsarten des 6809 sind Kapitel 1 als Anhang beigefügt und sollten von Ihnen bei der Durcharbeitung der folgenden Beispiele genutzt werden.

Praktische Übung P2.3-2:

Führen Sie die folgenden Beispiele der folgenden Unterabschnitte mit Ihrem Praktikumsrechner durch. Benutzen Sie die *Trace*-Funktion (Taste F1 des Praktikumsrechners) zur Einzelschrittausführung der Befehle.

Zur Erleichterung sind bei den Beispielen, entspr. Kapitel 1, neben den Mnemonics und den Operanden auch der OP-Code des 6809-Prozessors leicht unterlegt mit aufgeführt.

1. Wertzuweisungen

Wertzuweisungen an Konstanten werden in der Programmiersprache Pascal zu Beginn des Programms durch

```
const MAXDR    10;
```

deklariert. Hierbei ist der an die symbolische Konstante "MAXDR" zugewiesene Wert "10" während des gesamten Programmlaufes nicht mehr änderbar. Auf Assemblerebene steht die Direktive EQU für eine Konstantendeklaration zur Verfügung. Entsprechend dem Hochsprachenkonstrukt wird einer symbolischen Konstanten ebenfalls ein Wert zugewiesen, der nicht mehr veränderbar ist.

```
MAXDR    EQU    10
```

Auf Assemblerebene kann der Wert der Konstanten entgegen den normalen Konstantendeklarationen von Hochsprachen ein Datum oder eine Adresse darstellen.

Wertzuweisungen an Variablen werden auf Assemblerebene neben üblichen Operationsbefehlen durch Transfer- und Speicherbefehle durchgeführt. Variablenwerte werden in Prozessorregistern oder im Speicher gehalten. Zur Modifikation werden diese in ein Prozessorregister geladen oder, falls schon in einem Register vorhanden, sofort verändert. Werden Variablenwerte bei der aktuellen Programmbearbeitung nicht benötigt,

können diese in nicht benutzten Registern des Prozessors (schneller Zugriff) oder auf dem Stack zwischengespeichert werden. Ein Zurückschreiben des Variablenwertes in den Speicher ist natürlich auch möglich.

Die folgenden Beispiele verwenden die Konstanten "MAXDR" und "MINTMP" sowie die Variablen "Druck" und "Temp"[3]. In den Codefragmenten der Assemblerbeispiele wird die Variable "Druck" im Register A, die Variable "Temp" im Register B gehalten.

2. Einfache Verzweigungen

In der Sprache Pascal werden Verzweigungen durch die Konstrukte

```
IF  Bedingung  THEN  Anweisung  und durch
IF  Bedingung  THEN  Anweisung1  ELSE  Anweisung2
```

dargestellt. Der Programmfluß wird durch die Abfrage einer Bedingung in seinem sequentiellen Ablauf unterbrochen und in alternative Programmabläufe aufgespaltet. Ist die Bedingung im ersten Konstrukt erfüllt, so wird die Anweisung ausgeführt, ansonsten nicht. Im zweiten Konstrukt wird bei einer erfüllten Bedingung Anweisung1 ausgeführt, ansonsten Anweisung2. Das Programm wird in beiden Fällen bei einer nach dem IF-Konstrukt folgenden Anweisung fortgesetzt.

Eine Bedingung in Pascal ist dann erfüllt, wenn die Boolesche Bedingung *true* ist. Auf Prozessorebene wird ein Verzweigungsbefehl ausgeführt, wenn die zu dem Befehl korrespondierenden Flags im CC-Register (Condional Code Register) gesetzt sind. Eine Bedingung kann in höheren Programmiersprachen eine "einfache" Vergleichsoperation, wie z. B. "A < B" sein, oder auch mehrere verknüpfte Bedingungen, wie z. B. "(A < B) AND (C > D)", enthalten. Auf Prozessorebene stehen für "einfache" Vergleichsoperationen entsprechende Befehle zur Verfügung. Komplexere Bedingungen müssen aufgelöst werden (Unterabschnitt 3).

In Assemblersprachen wird die Auswahl von Alternativen durch "Überspringen" der nicht zu durchlaufenden Alternativen gelöst.

Beispielsweise ist zu der PASCAL-Struktur

```
const  MAXDR = 10;
...
IF Druck < MAXDR THEN Druck := Druck + 1;
...
```

das Assemblercodefragment

```
                    ...
81 0a               CMPA   #MAXDR  ;Vergleich auf max Druck
24 01               BHS    NEXT    ;Wenn >=, springen
4c                  INCA           ;Variable Druck erhoehen
                NEXT:  ...         ;
                                   ;
                MAX    EQU    10   ;maximaler Druck = 10
```

[3] Es hat sich als gute Programmierpraxis erwiesen, Konstantennamen nur in Großbuchstaben zu definieren und Variablennamen dagegen mit Kleinbuchstaben zu versehen.

äquivalent. Soll der "INCA"-Befehl nicht ausgeführt werden, so wird er mittels des Befehls "BHS" übersprungen. Zu beachten ist, daß die Bedingung im Assemblercode gegenüber der Pascal-Struktur negiert ist.
Da in der Struktur

```
const  MAXDR = 10;
...
IF Druck < MAXDR  THEN Druck := Druck + 1
                  ELSE Druck := Druck - 2;
...
```

eine Auswahl zwischen zwei Alternativen zu treffen ist, sind auf Assemblerebene ein bedingter und ein unbedingter Sprung notwendig.

```
                        ...
81 0a                   CMPA   #MAXDR ;Vergleich auf max Druck
24 03                   BHS    ELSE   ;Wenn >=, springen
4c                      INCA          ;Druck um Eins erhoehen
20 02                   BRA    NEXT   ;Alternative ueberspringen
80 02             ELSE: SUBA   #02    ;ELSE-Zweig: Druck - 2
                  NEXT: ...           ;naechste Anweisung

                  MAXDR EQU    10     ;maximaler Druck = 10
```

Durch den unbedingten Sprung "BRA" (*branch always*) wird nach Abarbeitung der ersten Alternative die zweite Alternative übersprungen, um damit zu der nächsten Anweisung nach den Alternativen zu gelangen.

Hinweis:
Bevor ein Verzweigungsbefehl verwendet werden kann, ist ein Vergleichsbefehl, z. B. "CMPA", zu benutzen, um die einzelnen Flags im CC-Register für den Vergleichsbefehl vorzubereiten. Es kann nur dann auf den Vergleichsbefehl verzichtet werden, wenn der vorhergehende Befehl die Flags für den nachfolgenden Verzweigungsbefehl korrekt setzt. Wird dagegen zwischen Vergleichsbefehl und Verzweigungsbefehl ein Befehl gelegt, der die Flags für den Verzweigungsbefehl anderweitig ändert, ergeben sich falsche Verzweigungsentscheidungen!

Aus den Assembler-Befehlstabellen (siehe Anhang von Kapitel 1) können Sie ersehen, welche Befehle welche Flags des CC-Registers setzen und entsprechend entscheiden, ob ein Vergleichsbefehl notwendig ist oder nicht.

3. Verzweigungen mit mehreren Bedingungen

Für den Vergleich zwischen Operanden stehen auf Prozessorebene in der Regel nur Vergleichsbefehle mit zwei Operanden zur Verfügung. Hat die für eine Verzweigung zu verwendende Bedingung mehr als zwei Operanden, z. B. "(A < B) AND (C > D)", so ist sie in mehrere Einzelvergleiche aufzulösen. Für die Zusammenfassung der Einzelvergleiche zu einem Booleschen Gesamtergebnis sind mehrere Lösungen möglich.

Denkbar ist der Einzelvergleich mit jeweiligem nachfolgendem Abspeichern des Vergleichsergebnisses. Die Ergebnisse werden danach wieder paarweise verglichen, um

in mehreren Schritten zu einem Gesamtergebnis zu gelangen und um die verlangte Aktion auszuführen oder nicht.

Eine anderer Lösungsweg nutzt für die UND-Verknüpfung zwischen Einzelvergleichen, die implizite UND-Verknüpfung der einzelnen Befehle in einem sequentiellen Programmfluß aus. Bei einer ODER-Verknüpfung zwischen Einzelvergleichen wird der Umstand ausgenutzt, daß wenn der erste Vergleich wahr ist, der zweite (oder weitere) Vergleich nicht mehr überprüft werden muß. Die beiden folgenden Beispiele zeigen jeweils einen Implementierungsvorschlag. In den Assemblerbeispielen wurde für die Variable "N" das A-Register, für die Variable "X" das B-Register verwendet. Die Konstanten "MAX" und "TEMP" wurden über die Assembler-Direktive "EQU" eingebunden.

Für die PASCAL-Struktur

```
const  MAXDR  = 10;
const  MINTMP = 90;
...
IF  (Druck  <  MAXDR)  AND  (Temp  >
MINTMP)
   THEN  Druck := Druck + 1;
...
```

ergibt sich als Assemblerimplementierung:

```
                        ...
81 0a                   CMPA  #MAXDR  ;Vergleich auf max Druck
24 05                   BHS   NEXT    ;Wenn >=, springen
c1 51                   CMPB  #MINTMP ;Vergleich auf min Temp.
23 01                   BLS   NEXT    ;Wenn <=, springen
4c                      INCA          ;Druck inkrementieren
                NEXT:   ...           ;

                MAXDR   EQU   10      ;maximaler Druck = 10
                MINTMP  EQU   90      ;minimale Temperatur = 90
```

Nur wenn beide Sprungbedingungen nicht erfüllt sind, also die dazu komplementären Bedingungen des PASCAL-Fragments erfüllt sind, wird die geforderte Aktion ausgeführt. Ist die erste Sprungbedingung erfüllt (bzw. wird der komplementäre erste Vergleich in PASCAL-Notation nicht erfüllt), wird sofort mit der Ausführung der folgenden Anweisung nach der Verzweigung begonnen.

Bei einer ODER-Verknüpfung

```
const  MAX  = 10;
const  TEMP = 90;
...
IF (Druck < MAXDR) OR (Temp > MINTMP)
   THEN  Druck := Druck + 1;
...
```

ergibt sich als Assemblerimplementierung:

```
81 0a          CMPA  #MAXDR  ;Vergleich auf max Druck
24 05          BLO   WORK    ;Wenn kleiner, addieren
c1 51          CMPB  #MINTMP ;Vergleich auf min Temp.
23 01          BLS   NEXT    ;Wenn <=, springen
4c       WORK: INCA          ;Druck inkrementieren
         NEXT: ...           ;

               MAXDR  EQU   10       ;maximaler Druck = 10
               MINTMP EQU   90       ;minimale Temperatur = 90
```

Ist die erste Bedingung erfüllt, kann sofort zu der auszuführenden Anweisung gesprungen werden. Ist die erste Bedingung nicht erfüllt, aber die zweite Bedingung, so wird ebenfalls die Aweisung ausgeführt. Werden beide Bedingungen nicht erfüllt, so wird durch die zweite Verzweigung die auszuführende Anweisung übersprungen und mit der nachfolgenden Anweisung begonnen.

Müssen in einer Verzweigung mehrere ELSE-Zweige berücksichtigt werden, so ist ein Vergleich mit mehreren Bedingungen nicht möglich. Eine Verschachtelung des "IF-THEN-ELSE"-Konstrukts ist die Folge. Zur Verdeutlichung möge das folgende Beispiel dienen.

```
const  MAXDR  = 10;
const  MINTMP = 90;
...
IF Druck < MAXDR
   THEN  IF Temp > MINTMP  THEN  Druck := Druck + 1
                           ELSE  Temp  := Temp + 4
   ELSE  Temp := Temp + 25;
...
```

Aus der PASCAL-Struktur ergibt sich als beispielhafte Implementierung:

```
               ...
81 0a          CMPA  #MAXDR  ;Vergleich auf max Druck
24 0b          BHS   ELSE1   ;Wenn >=, nach ELSE1
c1 5a          CMPB  #MINTMP ;Vergleich auf min Temp.
23 03          BLS   ELSE2   ;Wenn <=, nach ELSE1
4c             INCA          ;2. Ebene, THEN-Zweig
20 06          BRA   NEXT    ;zur naechsten Anweisung
cb 04    ELSE2: ADDB  #04     ;2. Ebene, ELSE-Zweig
20 02          BRA   NEXT    ;zur naechsten Anweisung
8b 19    ELSE1: ADDA  #25     ;1. Ebene, ELSE-Zweig
         NEXT:  ...           ;

               MAXDR  EQU   10       ;maximaler Druck = 10
               MINTMP EQU   90       ;minimale Temperatur = 90
```

Es sind in der Implementierung neben den beiden bereits bekannten bedingten Sprüngen zwei unbedingte Sprünge notwendig, um die einzelnen Anweisungen voneinander zu trennen.

4. Mehrfachauswahl

Immer wieder stellt sich das Problem, eine Auswahl aus mehreren Möglichkeiten zu treffen.

Beispielsweise könnte in Abhängigkeit von einer Tastatureingabe bei Drücken der Taste

'S' : das Programm zu beenden sein, bei
'R' : eine Addition danach eine neue Eingabe durchzuführen sein, bei
'T' : sofort eine weitere Eingabe möglich sein, bei
'L' : die Verzweigung in ein Unterprogramm mit nachfolgender neuer Eingabe möglich sein.

Trifft keine der Möglichkeiten zu, könnte die Eingabe zu wiederholen sein.

In PASCAL würde diese Abfrage u.a. mit einer CASE-Struktur modelliert werden. In einem 6809-Assemblerprogramm des Praktikumsrechners würde obige Abfrage z.B. wie folgt implementiert[4]:

```
...                 ...
bd f1 43    NEW:    JSR    HALTKEY  ;Tastatureingabe abwarten
c1 86               CMPB   #$86     ;Vergleich auf Taste "S"
27 ..               BEQ    ENDE     ;wenn ja, Programmende
c1 84               CMPB   #$84     ;Vergleich auf Taste "R"
27 0d               BEQ    ADD      ;wenn ja, Addition
c1 88               CMPB   #$88     ;Vergleich auf Taste "T"
27 f1               BEQ    NEW      ;wenn ja, neue Eingabe
c1 87               CMPB   #$87     ;Vergleich auf Taste "L"
26 ed               BNE    NEW      ;wenn nein, neue Eingabe
bd .. ..            JSR    UPTST    ;Unterpr. UPTST aufrufen
20 e8               BRA    NEW      ;Mehrfachauswahl Ende
8b 03       ADD:    ADDA   #$03     ; ....
20 e4               BRA    NEW      ;zurueck zur Eingabe
...                 ...

...                 ...
...         ENDE:   ...              ;Programmende

f1 43       HALTKEY  EQU   $F143     ;Einsprungadresse UP
```

Hinweis (für Fortgeschrittene):
Bei einer Mehrfachauswahl können unter Umständen durch geschickte Auswahl und Kombination einzelner Abfragen Vergleichsbefehle eingespart werden. Soll z.B. bei ei-

[4] Das Unterprogramm "HALTKEY" liefert im Register B den Code der gedrückten Taste.

ner positiven Zahl nach Adresse $0600 gesprungen werden und bei einer negativen Zahl nach $0700 verzweigt werden, so können die beiden Verzweigungsbefehle "BPL" and "BMI" direkt hintereinander liegen. Der Befehl vor den beiden Vergleichsbefehlen muß natürlich die Flags des CC-Registers korrekt setzen.

5. Zyklische Programme

In einem Programm werden in der Regel mehrere Schleifen benötigt. Abhängig von der Anzahl der minimalen Durchläufe durch eine Schleife werden verschiedene Schleifenkonstrukte benutzt.

Bei der WHILE-Schleife steht der Test für die Endebedingung der Schleife vor dem Schleifenkörper, in dem alle Anweisungen, die in einer Schleife durchgeführt werden sollen, stehen. Somit ist es bei der WHILE-Schleife möglich, daß der Schleifenkörper bei Aufruf der Schleife nicht ein einziges mal durchlaufen wird, da die Endebedingung bereits erfüllt ist. Entsprechend ist bei einer Assemblerimplementierung die Abfrage der Endebedingung vor den Schleifenkörper zu stellen. Bei einer REPEAT-Schleife liegt die Endebedingung hinter dem Schleifenkörper, so daß die Schleife auf jeden Fall einmal durchlaufen wird.

Innerhalb des Schleifenkörpers einer WHILE-Schleife als auch einer REPEAT-Schleife muß es eine Anweisung geben, die Einfluß auf den Wahrheitswert der Endebedingung der Schleife nimmt, damit die Schleife terminiert.

In den folgenden beiden Beispielen werden Implementierungen der WHILE- und der REPEAT-Schleife vorgestellt. Die FOR-Schleife wird nicht diskutiert, da sie einen Spezialfall der WHILE-Schleife darstellt.

Als jeweilige Anwendung wird die n-fache Aufsummierung, mit

$X = 1 + 2 + 3 + ... + N$ und $N = 10$,

gewählt. Für X ergibt sich der Wert 55.

In Pascal-Notation ergibt sich für die WHILE-Schleife:

```
const N = 10;
var   I: char;
var   X: integer;

X := 0;
I := 1;
WHILE I <= N DO
  BEGIN
    X := X + I;
    I := I + 1
  END ;
```

Bei der Umsetzung der WHILE-Schleife in ein Assemblerprogramm für den 6809-Prozessor sollten die Befehle und die möglichen Adressierungsarten der Prozessors berücksichtigt werden.

Möchte man für die Addition die Register des Prozessors ausnützen, ohne über den Speicher des Rechners arbeiten zu müssen, so bietet sich der "ABX"-Befehl an, der

eine Addition zwischen dem B-Register und dem X-Register ermöglicht. Da der Befehl das Ergebnis der Addition (Variable X) im X-Register ablegt, ist entsprechend der Aufgabenstellung das B-Register für die Zählvariable "I" zu benutzen. Dies "harmoniert" mit dem "INCB"-Befehl, so daß die Variable "I" in einem Taktzyklus inkrementiert werden kann. Die Konstante "N" wird über die Assembler-Direktive "EQU" in das Programm eingebunden. Damit Sie das Programm auf Ihrem Praktikumsrechner vollständig austesten können, sollte es ab der Speicheradresse $0400 beginnen.

```
                        ORG    $0400    ;Beginn Programmbereich
8e 00 00                LDX    #$0000   ;X mit 0 initialisieren
c6 01                   LDB    #$01     ;B := 1
c1 0a           LOOP:   CMPB   #N       ;Zaehlende erreicht ?
22 04                   BHI    END      ;Wenn ja, nach END
3a                      ABX             ;Nein, Addition
5c                      INCB            ;Zählvariable erhöhen
20 f8                   BRA    LOOP     ;zurueck zum Zaehlanfang
3f              END:    SWI1            ;Programmende

                N       EQU    10       ;Endebedingung = 10
```

Die Abfrage für das Beenden der Schleife liegt bei der WHILE-Schleife vor dem Schleifenkörper, der in einem Assemblerprogramm zwischen dem bedingten Sprungbefehl und der zugehörigen Sprungadresse "geklammert" ist. Am Ende des Schleifenkörpers ist bei der WHILE-Schleife mit einem unbedingten Sprung zum Schleifenanfang zurückzukehren.

Die REPEAT-Schleife in Pascal-Notation lautet:

```
const N = 10;
var   I: char;
var   X: integer;

X := 0;
I := 1;
REPEAT
   X := X + I;
   I := I + 1
UNTIL I = N;
```

Als Assemblerprogramm ergibt sich bei den gleichen Registerbelegungen wie für die WHILE-Schleife:

```
                        ORG    $0400    ;Beginn Programmbereich
8e 00 00                LDX    #$0000   ;X mit 0 initialisieren
c6 01                   LDB    #$01     ;B := 1
3a              LOOP:   ABX             ;Addition
5c                      INCB            ;Zählvariable erhöhen
c1 0a                   CMPB   #N       ;Zaehlende erreicht ?
23 fa                   BLS    LOOP     ;Wenn nein, weitere ADD
3f                      SWI1            ;Programmende


                N       EQU    10       ;Endebedingung = 10
```

Bei jeder Schleife ist die Reihenfolge der Anordnung der Zählvariable zu anderen Anweisungen des Schleifenkörpers wichtig, wenn diese Anweisungen die Zählvariable verwenden. Beispielsweise müßte in der REPEAT-Schleife das B-Register mit dem Wert Null initialisiert werden, wenn zuerst der "INCB"-Befehl und danach der "ABX"-Befehl im Schleifenkörper folgen würde. Die Möglichkeit solcher Variationen kann ausgenutzt werden, um einfachere Abfragebedingungen für die Beendigung der Schleife zu erreichen.

6. Vorzeichenlose und vorzeichenbehaftete Verzweigungsbefehle

Vorsicht ist bei der Benutzung von vorzeichenlosen und vorzeichenbehafteten Verzweigungsbefehlen geboten. Abhängig vom Datentyp der Variablen, die für den Vergleichstest für die Verzweigungsbedingung herangezogen wird, ist der passende Verzweigungsbefehl auszuwählen.
Da die Wertigkeit von vorzeichenlosen und vorzeichenbehafteten Zahlen verschieden ist, wie Tabelle 2.3-1 für 3-bit Zahlen zeigt, führt ein falsch gewählter Verzweigungsbefehl zu einem nicht korrekt ablaufenden Programmfluß.

Tabelle 2.3-1: Vorzeichenbehaftete und vorzeichenlose 3-bit Zahlen

vorzeichenbehaftete Zahlen		**vorzeichenlose Zahlen**	
positivste Zahl	011	111	größte Zahl
	010	110	
	001	101	
	000	100	
	111	011	
	110	010	
	101	001	
negativste Zahl	100	000	kleinste Zahl

Beispielsweise sei die Variable A = 111, die Variable B = 011. Sind die beiden Zahlen vorzeichenlos, so ist A > B. Sind die beiden Zahlen vorzeichenbehaftet, so gilt A < B.

7. Zusammenfassung

Mittels bedingten und unbedingten Verzweigungsbefehlen ist es möglich, alle Kontrollstrukturen einer problemorientierten Programmiersprache auf Assemblerebene abzubilden. Ein Assemblerbefehl, der die Flags des CC-Registers korrekt setzt, wie z.B. "CMPA", und der Verzweigungsbefehl bilden in der Regel eine Einheit.

Bei der Implementierung einer Schleife ist auf eine einfache Abfrage des Schleifenendekriteriums zu achten. Gegebenenfalls ist die Initialisierung der in der Endebedingung abgefragten Variablen zu verändern. Damit die Schleife terminiert, muß diese Variable im Sinne der Endebedingung verändert werden.

Programmierfehler können durch falsche Verwendung von vorzeichenlosen und vorzeichenbehafteten Verzweigungsbefehlen entstehen.

2.3.5 Unterprogramme und Stackoperationen

Unterprogramme sind auf Assemblerebene, wie auch in jeder höheren Programmiersprache, ein Mittel zur Strukturierung und Vereinfachung von Programmen. Zum Verständnis komplizierter Vorgänge und Berechnungen ist es meistens nicht notwendig zu wissen, wie die einzelnen Teilergebnisse berechnet wurden, wichtig ist nur die Grobstruktur. Genau dies wird durch Unterprogrammaufrufe erfüllt. Indem die Berechnungen der Teilergebnisse in Unterprogramme ausgelagert werden, wird das "aufrufende" Programm kleiner, übersichtlicher und durchschaubarer.

Ein Assemblerunterprogramm ist ähnlich strukturiert wie ein Unterprogramm einer höheren Programmiersprache. Es ist in sich abgeschlossen, d.h. es besitzt einen definierten Anfang und ein definiertes Programmende. Einem Unterprogramm können Parameter (Variablenwerte) zur Berechnung übergeben werden. Ergebnisse können von dem Unterprogramm an das "aufrufende" Programm zurückgegeben werden. Ein Unterprogramm kann wiederholt aufgerufen werden und es kann sich, je nach Konstruktion des Unterprogramms, selbst aufrufen. Unterprogramme können andere Unterprogramme aufrufen (Verschachtelung). Nachteilig wirkt sich bei der Benutzung von Unterprogrammen die etwas längere Ausführungszeit aus, da zusätzliche Befehle zur Steuerung des Programmflußes verwendet werden müssen. Zur Zwischenspeicherung von Registerinhalten wird in der Regel der Stack verwendet.

1. Kontrollfluß

Der Aufruf eines Unterprogramms auf Assemblerebene erfolgt durch spezielle Prozessorbefehle (*subroutine calls*). Gelangt eine Routine im Laufe ihrer Abarbeitung auf einen Unterprogrammaufruf, so wird in der Regel der Inhalt des Programmzählers, der die Speicheradresse des dem Aufruf nachfolgenden Befehls enthält, auf dem Stack abgelegt. Die Abspeicherung des Programmzählers auf dem Stack wird bei fast allen mo-

dernen Prozessoren automatisch durch den *subroutine call* erzwungen[5]. Danach wird an die Speicheradresse verzweigt, die als Operand dem Befehl zum Aufruf des Unterprogramms folgt. Nach Abarbeitung des Unterprogramms wird wiederum mittels eines speziellen Prozessorbefehls ("RTS": *return from subroutine*) zu dem aufrufenden Programm zurückgekehrt. Hierzu wird durch den "RTS"-Befehl die Speicheradresse, die beim Sprung in das Unterprogramm auf den Stack gelegt wurde, geholt und in den Programmzähler übertragen. Der Programmfluß wird im "aufrufenden" Programm mit dem Befehl nach dem Unterprogrammaufruf fortgesetzt.

Eine Verschachtelung von Unterprogrammen ist möglich (ein Unterprogramm ruft ein weiteres Unterprogramm auf), da der Stack nach dem LIFO-Prinzip (last in, first out) arbeitet. Hierdurch können die einzelnen Rückkehradressen der "aufrufenden" Programme ordnungsgemäß und in der richtigen Reihenfolge vom Stack geholt werden.

Die Übergabe des Kontrollflußes zwischen mehreren Aufrufen von Unterprogrammen zeigt beispielhaft Bild 2.3-1. Der Kontrollfluß wird vom "aufrufenden" Programm an das Unterprogramm abgegeben. Nach Beendigung des Unterprogramms wird die Kontrolle an das "aufrufende" Programm wieder zurückgegeben. Wird im Unterprogramm eine weitere Unterroutine aufgerufen, so wird die Kontrolle entsprechend weitergegeben.

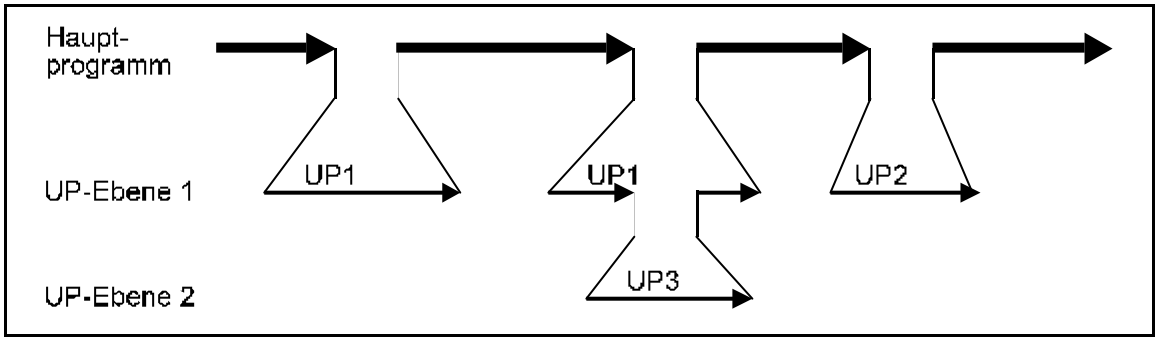


Bild 2.3-1: Kontrollflußübergabe bei Unterprogrammen

2. Sichern von Registerinhalten

Zur korrekten Abarbeitung des aufgerufenen Unterprogramms werden Prozessorregister benötigt. Bei kleinen Programmen kann unter Umständen die Benutzung der Register zwischen Hauptprogramm und Unterprogramm so geschickt aufgeteilt werden, daß keine Registerinhalte während des Unterprogrammaufrufes zwischengespeichert werden müssen. Bei komplexeren Programmen ist dies kaum möglich, so daß der Registersatz des Prozessors teilweise oder ganz im Speicher abgelegt werden muß. Als Zwischenspeicher wird in der Regel der Systemstack benutzt. Die Zwischenspeicherung der Registerinhalte kann zu zwei Zeitpunkten stattfinden:

?? Speicherung vor dem Sprung ins Unterprogramm,
?? Speicherung nach dem Sprung ins Unterprogramm.

[5] Weitere Methoden sind denkbar, bei denen das "aufrufende" Programm selbst dafür sorgt, daß der Programmzähler gerettet wird und das "aufgerufene" Programm nach Abarbeitung des Unterprogramms die Rückkehradresse in den Programmzähler überträgt.

Welche Variante gewählt wird, ist von Fall zu Fall zu entscheiden und liegt im Ermessen des Programmierers. Bei beiden Varianten ist zu beachten, daß die einzelnen Registerinhalte in umgekehrter Reihenfolge vom Stack geholt werden müssen, wie sie auf den Stack gelegt worden sind. Das heißt, die Register, die zuletzt auf den Stack geschrieben wurden, müssen als erste wieder gelesen werden.

3. Parameterübergabe

Die Art der Übergabe von Parametern (Daten, Adressen, Adreßzeiger) an das aufgerufene Unterprogramm hängt von der Anzahl der übergebenen Parameter, der zur Verfügung stehenden "Übergabezeit" und bei größeren Programmen (z.B. von Compilern erzeugte Programme) vom Typ der Parameter ab.

Für die Übergabe werden verschiedene Methoden benutzt, von denen einige ausgewählte im folgenden kurz vorgestellt werden.

> Übergabe über Prozessorregister:
Es wird versucht, zur Übergabe überwiegend Register des Prozessors zu benutzen, da dies die schnellste Übergabemöglichkeit darstellt. Es müssen keine Daten transferiert oder ausgelagert werden. Da Prozessoren nur eine beschränkte Anzahl von Registern zur Verfügung stellen, ist diese Übergabeart nur für sehr kleine Datenmengen praktikabel. Probleme bzgl. der Lageunabhängigkeit von Unterprogrammen treten nicht auf.

> Übergabe über einen Speicherbereich:
Muß eine größere Anzahl von Daten an das Unterprogramm übergeben werden, bietet es sich an, einen Speicherbereich für die Übergabe zu reservieren und die Daten darin abzulegen. Das Unterprogramm erhält in der Regel die Anfangsadresse (Adreßzeiger) des Datenbereiches über ein Indexregister des Prozessors oder über den Stack zugewiesen. Diese Methode wird in der Regel in größeren Programmen verwendet, um Variable zu übergeben, die vom Unterprogramm verändert zurückgegeben werden (*call by reference*).

> Übergabe über den Systemstack:
Bei der Übergabe einer "mittleren" Anzahl von Daten oder Adressen ist der Stack eine einfache Möglichkeit, Daten zwischen "aufrufendem" Programm und Unterprogramm zu transferieren. Diese Übergabeart wird in größeren Programmen bei Daten gewählt, die aus der Sicht des "aufrufenden" Programmes als Kopie von Variablenwerten an das Unterprogramm übergeben werden (*call by value*).

> Übergabe über den User-Stack:
Unterstützt der verwendete Prozessor einen User-Stack, wie z. B. der 6809-Prozessor, so entfällt eine Adreßberechnung für die übergebenen Daten. Da System- und Anwenderdaten getrennt sind, machen sich Programmierfehler nicht sofort im gesamten System bemerkbar.

4. Lageunabhängigkeit von Programmen und Unterprogrammen

Kann ein Maschinenprogramm an einer beliebigen Stelle des Speichers geladen und ausgeführt werden, so wird diese Eigenschaft als "Lageunabhängigkeit" bezeichnet. Hierfür müssen alle Sprungziele und Speicheradressen von (Unter-) Programmen und

Datenbereichen relativ zur Speicheradresse des ersten Maschinenbefehls des Programms adressiert werden. Lageunabhängige Programme enthalten somit keine absoluten Speicheradressen.

Lageunabhängige Programme werden durch ausschließliche Benutzung von Befehlen mit relativer Adressierung erreicht. Beispielsweise sind bei der Programmierung von Unterprogrammen des 6809-Prozessors statt des "JSR"-Befehls die Befehle "BSR" und "LBSR" zu verwenden. Für die relative Adressierung von Datenbereichen sind bei dem 6809-Prozessor von Motorola die "LEA "-Befehle zu benutzen, z.B. "LEAX OFFSET,PCR".

Hinweis:
Vertiefende Übungsbeispiele zur Programmierung von Unterprogrammen finden Sie in Abschnitt 2.5 dieses Kapitels.

5. Rekursive Programmierung

Ein rekursiver Programmablauf liegt vor, wenn sich Programmroutinen wechselseitig oder selbst aufrufen. Als Abbruchbedingung muß eine nichtrekursive Abbruchbedingung in den Programmroutinen enthalten sein. Das heißt, es ist ein Zustand (eine Variable) so zu verändern, daß nach endlich vielen Schritten ein eindeutig definierter Endzustand erreicht ist.

Rekursion eignet sich besonders dann, wenn das zugrunde liegende Problem oder die zu behandelnden Daten rekursiv definiert sind. Bei der rekursiven Programmierung wird im allgemeinen Programmcode eingespart und die Lesbarkeit von Routinen unter Umständen erhöht. Rekursive Lösungen benötigen allerdings mehr Speicher und eine längere Programmlaufzeit, da bei jedem Aufruf der rekursiven Routine Daten auf den Stack abgelegt werden müssen. Rekursive Strukturen lassen sich in Schleifenkonstrukte überführen.

Tabelle 2.3-3 zeigt als Beispiel für die Programmierung von Unterprogrammen und speziell für die rekursive Programmierung die rekursive Implementierung des Beispiels von Unterabschnitt 2.3.4 zur n-fachen Aufsummierung von Zahlen, mit

$X = 1 + 2 + 3 + ... + N$ und $N = 10$.

Für X ergibt sich der Wert 55.

Das Beispiel ist für eine Rekursion nicht charakteristisch, zeigt aber die wichtigsten Techniken. Die Zeilen $0400 - $0408 stellen das Hauptprogramm, die Zeilen $0409 - $0416 das Unterprogramm dar. Im B-Register wird die Zählvariable Rekursionsebene für Rekursionsebene dekrementiert und der Inhalt auf den Stack abgelegt. Die Zeilen $040C und $040E realisieren die Abbruchbedingung der Rekursion. In Zeile $0413 wird das B-Register restauriert und in der nächsten Zeile zu dem Inhalt des X-Registers addiert. Das X-Register stellt somit eine Art globale Variable dar. Nach dem Durchlauf durch alle zehn Rekursionsebenen terminiert das Programm in der Hauptroutine in Zeile $0408.

Der Vergleichsbefehl in Zeile $040C ist nicht unbedingt notwendig, da durch den DECB-Befehl bereits das Zero-Flag im CC-Register korreht gesetzt wird. Der Übersichtlichkeit halber wurde der Befehl in das Listing mit aufgenommen.

Tabelle 2.3-3: Rekursives Beispielprogramm zur n-fachen Aufsummierung

```
0400                        ORG    $0400    ;Beginn Programmbereich
0400  8E 00 00               LDX    #$0000   ;X initialisieren
0403  C6 0A                  LDB    #10      ;B auf 10 Durchläufe
0405  BD 04 09               JSR    UPRADD   :rekursives UP aufrufen
0408  3F                    SWI1             ;Programmende
0409  34 04      UPRADD:    PSHS   B         ;Inhalt von B auf Stack
040B  5A                    DECB             ;Zähler dekrementieren
040C  C1 00                  CMPB   #0       ;Vergleich auf Rek-Ende
040E  27 03                  BEQ    NEXT     ;wenn ja, UP überspr.
0410  BD 04 09               JSR    UPRADD   ;rekursiver UP-Aufruf
0413  35 04      NEXT:      PULS   B         ;Zähler vom Stack holen
0415  3A                    ABX              ;X := X + B
0416  39                    RTS              ;Ende Unterprogramm
```

2.3.6 Programmdokumentation

Assemblerprogramme können sehr kompakt und undurchschaubar geschrieben werden, so daß nur eine gute Strukturierung und Kommentar-Disziplin zu lesbaren und wartbaren Programmen führt.

Verschiedene Programmier-Teams haben Richtlinien und Regeln für eine gute Dokumentierung von Programmen herausgegeben. Die Praxis zeigt, daß nur durch konsequentes Anwenden dieser Hilfen ein Programm auch nach einem halben Jahr noch verständlich bleibt. Im folgenden einige Auszüge:

?? Programm- und Prozedurköpfe sind mit einer globalen Beschreibung der Routine zu versehen. Eine Beschreibung der Schnittstellen, insbesondere der Input- und Output-Parameter, hat zu erfolgen. Die Köpfe sind durch die Verwendung von Sonderzeichen besonders kenntlich zu machen.

?? Eine sinnvolle Kommentierung des Programms hängt vom Kommentarinhalt und der Kommentarformulierung ab.

?? Bei der Kommentierung soll die logische, funktionelle Bedeutung einer Instruktion im Kontext des (Teil-) Algorithmus beschrieben werden. Eine kontextfreie Beschreibung der Befehlswirkung soll nicht stattfinden.

?? Die Namensvergabe soll sinnvoll und überlegt sein. Die Aussagekraft von Symbolen unterstützt wesentlich die Lesbarkeit und das Verständnis.

?? Symbol-, Variablen- und Programmnamen sind kurz zu halten. Zwischen Aussagekraft und Länge muß ein Kompromiß gefunden werden.

?? Eine in der Assemblerpraxis geläufige Länge umfaßt acht bis zu 12 (16) Zeichen. Die Möglichkeit, Vokale innerhalb der Wörter zu streichen, ohne die Aussagekraft des Namens zu verringern, sollte genutzt werden.

?? Symbolische Adressierung ist im gesamten Programm anzuwenden.

?? Die Möglichkeit der Konstantendefinition ist zu nutzen.

Beispielsweise könnte ein Programmkopf folgende Struktur besitzen:

```
;************************************************************
; <Name des Programms>
; Funktion:  ..............
; Input:  Par 1:  ............
;         Par 2:  ............
;         Par n:  ............
; Output: Par 1:  ............
;         Par 2:  ............
;
; Aufger.von:  ........    Globale Var.: ..................
;
; Autor:  ............    Version:  ......  Datum:........
;Entwicklungswerkzeuge zum Praktikumsrechner
```

2.3.7 Der 6809-Cross-Assembler "AS9"

Der mit dem Praktikumsrechner auf Diskette mitgelieferte Assembler[6] war in der Grundversion der freiverfügbare Cross-Assembler "AS9" der Firma Motorola. Der Assembler wurde für das Praktikum um einige Optionen erweitert und verändert. Er ist auf einem IBM kompatiblen PC mit DOS 3.1 oder höher lauffähig und erzeugt als Cross-Assembler Maschinencode für den Motorola 6809-Prozessor.

Der Assembler ist ein *two pass assembler*, d.h. der Übersetzungvorgang wird in zwei Durchläufen ausgeführt. Im ersten Durchlauf wird das Quellprogramm gelesen und eine Symboltabelle aufgebaut. Im zweiten Durchlauf wird das Maschinenprogramm mit Hilfe der Symboltabelle erzeugt. Zusätzlich kann auf Anforderung (Unterabschnitte 2 und 5) eine Datei mit dem Programmlisting und der Symboltabelle generiert werden. Die Symboltabelle des "AS9" kann maximal 2000 Symbole mit acht oder weniger Zeichen verwalten. Werden mehr als acht Zeichen pro Symbol verwendet (bis zu maximal 16 Zeichen), so schrumpft entsprechend die Anzahl der maximal verwaltbaren Symbole. Ein Linker wird nicht benutzt. Das generierte Maschinenprogramm ist sofort ablauffähig.

Jede Befehlszeile des Quellprogramms wird vom Assembler vollständig bearbeitet, bevor der nächste Befehl des Quelltextes eingelesen wird. Bei jeder Befehlszeile überprüft der Assembler die Marke, den mnemonischen Operationscode und das Operandenfeld. Der mnemonische Operationscode wird mit der Zuordnungstabelle verglichen und bei Übereinstimmung der entsprechende Maschinencode in das Maschinenprogramm eingesetzt. Bei Nichtübereinstimmung wird eine Fehlermeldung generiert. Wird eine Assembler-Direktive (z.B. ORG) gefunden, wird die geforderte Aktion durch den Assembler durchgeführt. Jeden Syntaxfehler, der bei der Überprüfung entdeckt wird,

[6] Bei der Wahl des Assemblers wurde versucht, einen freiverfügbaren 6809-Assembler zu finden, der zum einen einfach zu bedienen ist, zum anderen aber auch einen gewissen Komfort bietet.

zeigt der Assembler durch eine eingeschobene Zeile vor der fehlerhaften Befehlszeile im Programmlisting an. Wird kein Programmlisting generiert, wird eine Fehlermeldung am Bildschirm ausgegeben, um auf den Fehler und die nicht erfolgreiche Assemblierung hinzuweisen.

1. Syntax eines Quellcodeprogramms

Assemblerprogramme bestehen aus einer Folge von Befehlszeilen. Jede Befehlszeile besteht aus einer Sequenz von ASCII-Zeichen, die mit einem Zeilenende-Zeichen (*line feed*) abgeschlossen ist. Gültige Zeichen sind beim "AS9" eine Teilmenge des ASCII-Zeichensatzes und zwar

- die Buchstaben — A..Z, a..z,
- die Ziffern — 0..9,
- die arithmetischen Operatoren — +, -, *, /, % (Divisionsrest),
- die logischen Operatoren — &, | , ^ (exklusiv-oder) und
- die Sonderzeichen — [,], \, _, $, #, @, . (Punkt), : (Doppelpunkt), ; (Strichpunkt), ' (Hochkomma), , (Komma), Leerzeichen oder Tabulator (*white space*).

Als gültige Zeichen für Symbole (Namen) dürfen nur Buchstaben, Ziffern und die Sonderzeichen $, . (Punkt) und _ (Unterstrich) verwendet werden. Hierbei dürfen Symbole maximal 16 Zeichen lang sein und das erste Zeichen muß ein Buchstabe, ein Unterstrich oder ein Punkt sein. Alle Zeichen eines Symbols werden berücksichtigt, wobei zwischen Groß- und Kleinschreibung unterschieden wird.

Jede Befehlszeile eines Quellcodeprogramms für den Motorola "AS9" besteht aus den vier Feldern:

MARKE	OPERATOR	OPERAND	KOMMENTAR

Innerhalb einer Befehlszeile müssen die Felder "MARKE", "OPERATOR" und "OPERAND" durch mindestens ein Leerzeichen voneinander getrennt sein. Das Feld "KOMMENTAR" wird durch einen Strichpunkt vom Operanden-Feld getrennt. Ist die Befehlszeile länger als es der verwendete Texteditor zuläßt, so kann die Befehlszeile durch das Zeichen "\" in die nächste Zeile verlängert werden und zwar bis zu maximal 256 Zeichen.

Marken-Feld

Ein Stern (*) oder ein Strichpunkt als erstes Zeichen in der ersten Spalte des Marken-Feldes definiert die ganze Befehlszeile als Kommentar. Eine so gekennzeichnete Befehlszeile wird vom Assembler vollständig ignoriert. Ein Leerzeichen (oder Tabulator) als erstes Zeichen zeigt dem Assembler an, daß das Marken-Feld leer ist, d.h. die Befehlszeile ist kein Kommentar und hat keine Marke.

Ist das Zeichen in der ersten Spalte der Zeile ein gültiges Symbolzeichen, so kennzeichnet dies die Definition einer Marke (Symbol). Eine Marke ist eine Folge von Zeichen, der durch die Definition ein Wert zugewiesen wird. Die Definition einer Marke muß eindeutig sein, d.h. die Marke darf in einem weiteren Markenfeld einer anderen Befehlszeile nicht mehr enthalten sein (in einem Operandenfeld kann die Marke mehrfach

stehen). Mehrfachdefinitionen einer Marke werden durch den Assembler erkannt und als Fehler ausgegeben. Mit der Ausnahme von einigen Assembler-Direktiven (z.B. EQU) wird einer Marke das erste Byte der Speicheradresse der gerade assemblierten Befehlszeile zugeordnet. Die Definition von Marken kann optional mit einem Doppelpunkt abgeschlossen werden. Zum Beispiel sind die folgenden beiden Codefragmente identisch:

```
Start:  STA          Start   STA
        BNE  Start           BNE  Start
```

Eine Marke kann auch alleine in einer Befehlszeile stehen und markiert dann die Speicheradresse des in der folgenden Zeile stehenden Maschinenbefehls.

Operator-Feld

Das Operator-Feld muß einen gültigen mnemonischen Operationscode oder eine Assembler-Direktive enthalten. Mnemonics entsprechen direkt den zugehörigen Maschinenbefehlen und enthalten z.T. die notwendigen Registernamen, z.B. STA, CLRB. Hierbei verdeutlichen "ST" und "CLR" den Typ des Befehls und "A" und "B" sind die Register, die der Befehl benutzen soll (implizite Adressierung). Assembler-Direktiven sind spezielle Befehle, die den Übersetzungsvorgang des Assemblers steuern. Großbuchstaben im Operator-Feld werden vom Assembler zu Kleinbuchstaben konvertiert und danach auf ein gültiges Befehlswort überprüft. Das bedeutet, die Mnemonics "LDU", "LdU", "ldu" werden vom Assembler als ein und dasselbe Mnemonic interpretiert.

Operanden-Feld

Die Bedeutung des Ausdruckes im Operanden-Feld ist vom Inhalt des Operator-Feldes abhängig. Das Feld kann ein Symbol, einen Ausdruck oder ein Symbol und einen Ausdruck enthalten. Hierbei kann ein Ausdruck ein Operand, eine Speicheradresse, ein Symbol, eine Marke oder ein Register sein. Durch die Anwendung von Kommata und eckigen Klammern wird die Adressierungsart, mit der der Befehl arbeiten soll, gekennzeichnet. In Tabelle 2.4-1 werden die einzelnen Operandenformate den zugehörigen Adressierungsarten des 6809 zugeordnet.

Tabelle 2.4-1: Operandenformat

Operandenformat *)	Adressierungsart
kein Ausdruck	implizit
Ausdruck	direkt, erweitert oder relativ
#Ausdruck	unmittelbar
<Ausdruck	kennzeichnet 8-bit-Offset
>Ausdruck	kennzeichnet 16-bit-Offset
[Ausdruck]	indirekt
Ausdruck,R	indiziert
<Ausdruck,R	indiziert, erzwingt 8-bit-Offset
>Ausdruck,R	indiziert, erzwingt 16-bit-Offset
[Ausdruck,R]	indirekt-indiziert

<[Ausdruck,R]	indirekt-indiziert, erzwingt 8-bit-Offset
>[Ausdruck,R]	indirekt-indiziert, erzwingt 16-bit-Offset
,Q+	Autoinkrement um 1
,Q++	Autoinkrement um 2
[,Q++]	Autoinkrement um 2, indirekt
,-Q	Autodekrement um 1
,--Q	Autodekrement um 2
[,--Q]	Autodekrement um 2, indirekt
W_1, [W_2, ..., W_n]	unmittelbar

*) R ist eines der Register PC, S, U, X oder Y. Q ist eines der Register S, U, X oder Y und W_i (i=1 bis n) ist eines der Symbole A, B, CC, D, DP, PC, S, U, X oder Y.

Ein Ausdruck ist eine Kombination von Symbolen, Konstanten, algebraischen und logischen Operatoren. Mit ihm kann der Wert eines Operanden oder die vom Befehl zu benutzende Adresse spezifiziert werden.

Steht zu Beginn des Operanden das Zeichen "<", so erzeugt der Assembler Maschinencode für einen 8-bit-Offset bei der Adressierung. Bei dem Zeichen ">" wird Code für eine Adressierung mit 16-bit-Offset generiert.

Operatoren

Die Notation der Operatoren entspricht den Operatoren in der Programmiersprache C:

- \+ Addition
- \- Subtraktion
- * Multiplikation
- / Division
- % Divisionsrest
- & bitweise UND-Verknüpfung
- | bitweise ODER-Verknüpfung
- ^ bitweise Exklusiv-Oder-Verknüpfung

Ausdrücke

Ausdrücke werden von links nach rechts berechnet. Klammern in Ausdrücken werden als syntaktische Fehler gewertet. Arithmetische Operationen werden im Zweierkomplement mit Integer-Genauigkeit (16-bit bei IBM kompatiblen PCs) berechnet. Besteht ein Ausdruck nur aus dem Zeichen "*", so repräsentiert der Ausdruck den aktuellen Wert des Programmzählers, z.B. bedeutet der Befehl "NEWPC EQU *", daß dem Symbol "NEWPC" der aktuelle Wert des Programmzählers zugeordnet wird.

Symbole

Jedes Symbol repräsentiert einen 16-bit-Integerwert. Symbole können innerhalb eines Ausdruckes als Repräsentant ihres Integerwertes benutzt werden. Der Assembler ersetzt bei der Berechnung des Ausdruckes das Symbol durch den zugehörigen Integerwert.

Konstanten

Konstanten repräsentieren Zahlenwerte, die während der Programmausführung nicht verändert werden können. Sie werden als 16-bit-Integerwert gespeichert. Sie können durch das Voranstellen eines Sonderzeichens in fünf verschiedenen Formaten notiert werden:

$	Hex
%	Binär
@	Oktal
'	ASCII-Zeichen
	Dezimalzahl

Wird kein Sonderzeichen verwendet, so wird die Konstante als Dezimalzahl interpretiert. Der Assembler konvertiert jedes vorgegebene Format in binären Maschinencode und stellt die Werte im Programmlisting als Hex-Zahlen dar. Im folgenden einige für den Assembler gültige und ungültige Zahlenformate:

Tabelle 2.4-2: Gültige und ungültige Zahlenformate

	Gültiges Format	**Ungültiges Format**	**Begründung**
Dezimal	12	123456	Bereichsüberschreitung
	12345	12.3	ungültiges Zeichen
Hexadezimal	$12	ABCD	kein "$"-Zeichen
	$ABCD	$G2A	ungültiges Zeichen
	$001F	$2F018	zu viele Ziffern
Binärformat	%00101	1010101	kein "%"-Zeichen
	%1	%10001100010101011	zu viele Ziffern
	%10100	%210101	ungültiges Zeichen
Oktalformat	@17634	@2317234	zu viele Ziffern
	@377	@277272	Bereichsüberschreitung
	@177600	@23914	ungültiges Zeichen
ASCII-Zeichen	'*	'XXY	zu viele Zeichen

Im Falle von mehr als einem ASCII-Zeichen verarbeitet der Assembler das erste Zeichen und ignoriert die restlichen Zeichen. Bei Bereichsüberschreitung oder zu vielen Ziffern werden die überzähligen Ziffern ebenfalls ignoriert. Eine Fehlermeldung wird nicht generiert.

Kommentar-Feld

Das letzte Feld in der Befehlszeile eines "AS9"-Quellprogramms ist das Kommentarfeld. Es ist optional und wird ohne Veränderungen in das Programmlisting übernommen. Im Kommentarfeld kann jedes druckbare ASCII-Zeichen stehen. Ein Kommentar wird durch einen Strichpunkt eingeleitet. Steht der Kommentar allein in der Befehlszeile, so kann der Kommentar auch durch einen Stern (*) in der ersten Spalte der Zeile gekennzeichnet werden.

2. Erstellen und Assemblieren eines "AS9"-Quellprogramms

Für die Erstellung eines Assembler-Quellprogramms können Sie jeden Texteditor verwenden, der es erlaubt, Ihr Programm als ASCII-Datei abzuspeichern. Hierbei werden nur der reine Text, Leerzeichen und Zeilenendezeichen in die Datei übernommen. Weitere Steuerzeichen, wie z.B. zur Formatierung des Textes oder zur Druckersteuerung werden in diesem Textformat nicht abgespeichert. Wird das Assembler-Quellprogramm mit Steuer-/Formatierungszeichen des Editors abgespeichert, so führen diese bei der Assemblierung zu Fehlern.

Das Quellprogramm ist in der Syntax des "AS9"-Assemblers zu erstellen. Hierbei ist insbesondere auf die Benutzung von Leerzeichen zur Trennung der einzelnen Felder zu achten, bzw. daß dem Kommentarfeld einen Strichpunkt voranzustellen ist. Tabelle 2.4-3 enthält ein für den "AS9" syntaktisch korrektes Programm.

Der Assembler wird von der DOS-Kommandozeile aus mit dem Aufruf

AS9 Datei1 (Datei2 . . .) (- Option1 Option2 . . .)

gestartet. Die Inhalte der Klammern geben hierbei Optionen an, die bei Bedarf eingegeben werden können. Datei1, Datei2 usw. sind die Namen der Quellprogramme, die assembliert werden sollen. Durch die Verwendung mehrerer Quelldateien kann eine "Modularisierung" des Assemblerprogramms erreicht werden. Hierbei hat der Programmierer allerdings für eine korrekte Speicherabbildung der einzelnen Teilprogramme zu sorgen. Bei Mehrfachbelegungen von Speicheradressen durch Assemblerbefehle erzeugt der Assembler Fehlermeldungen.

Nach einem Minuszeichen (durch Leerzeichen vom letzten Dateinamen getrennt und vor der ersten Option) können optional verschiedene Steuerbefehle für die Generierung zusätzlicher Ausgaben angegeben werden. Es stehen folgende zusätzliche Ausgabemöglichkeiten zur Verfügung:

L	Ausgabe des Programmlistings
NOL	kein Programmlisting (Voreinstellung)
CRE	Ausgabe der Cross-Reference-Tabelle
S	Ausgabe der Symboltabelle
C	Ausgabe der Anzahl der Zyklen
NOC	keine Ausgabe der Zyklenanzahl (Voreinstellung)
HEX	Ausgabedatei im Intel Hex-Format (Voreinstellung)
MOT	Ausgabedatei im Motorola S-Format

Die Optionen zur Ausgabesteuerung des Assemblers (nicht HEX und MOT) können auch direkt im Quellprogramm stehen und werden durch die Verwendung der Assembler-Direktive "OPT" gekennzeichnet (Unterabschnitt 4). Diese überschreiben die eventuell auf Kommandozeilenebene eingegebenen Optionen.

Die Dateinamen müssen in der Kommandozeile vollständig, d.h. Dateiname mit Endung, angegeben werden. Die Endung ist (fast) frei wählbar und wird vom Assembler nicht überprüft. Es ist allerdings gute Programmierpraxis, wenn Sie die Endung Ihrer

Quelldateien mit ".ASM" (für Assembler) benennen[7]. Der Assembler benennt die erzeugte Datei (Maschinenprogramm) nach dem Dateinamen der ersten angegebenen Quelldatei mit der Endung ".HEX". Wird z.B. das Quellprogramm "FIRSTPRG.ASM" assembliert, so erzeugt der Assembler als Ergebnis die Datei "FIRSTPRG.HEX". Die Endung ".HEX" deutet daraufhin, daß das generierte Maschinenprogramm im Intel HEX-Dateiformat abgespeichert wurde. Wird die Datei im Motorola S-Format mit der Option "-MOT" generiert, so erhält sie die Endung ".S19". Die Endungen ".HEX" bzw. ".S19" sind somit für die erzeugten Dateien mit den 6809-Maschinenprogrammen reserviert und sollten nicht anderweitig verwendet werden. Hat z.B. die Quelldatei die Endung ".S19", so wird diese Datei durch das erzeugte Maschinenprogramm überschrieben!

Werden Optionen in der Kommandozeile oder im Quellprogramm mit angegeben, so werden die zusätzlich erzeugten Listings auf die Standardausgabe gelenkt. Dies ist unter DOS in der Regel der Bildschirm. Wollen Sie diese Listings in einer Datei speichern, so erweitern Sie die Kommandozeile um folgendes Konstrukt:

AS9 Datei1 (Datei2 . . .) (- Option1 Option2 . . .) > Liste

Alle zusätzlichen Informationen (auch Fehler), die ohne das Konstrukt über den Bildschirm ausgegeben werden, werden durch "> Liste" in die Datei "Liste" umgelenkt. Der Dateiname "Liste" ist dabei frei wählbar. Es hat sich jedoch bewährt, als Dateinamen für eine Liste den Dateinamen des Quellprogramms mit der Endung ".LST" zu verwenden. Die Listenformate der möglichen Optionen werden in Unterabschnitt 5 besprochen.

Tabelle 2.4-3: Das Programm "P24-1.ASM"

```
         ORG  $0400     ;Beginn des Programmbereiches

         JSR  CLRDISP   ;Anzeige loeschen
         LDY  #$0600    ;Datenzeiger laden
         CLR  0,Y       ;Datenbereich mit Null initialisieren
         CLR  1,Y       ;
START:   LDX  #$0000    ;X loeschen
         JSR  HALTKEY   ;Zeichen von Tastatur lesen
         CMPB #$86      ;Vergleich, auf Ende der Eingabe (Taste "S")
         BEQ  ENDE      ;
         CMPB #$09      ;Test der Eingabe auf gueltige Ziffer (0..9)
         BHI  START     ;bei ungueltiger Ziffer zurueck
         ADDB 0,Y       ;Addition der Eingabe mit LSB der Summe
         TFR  B,A       ;
         ANDA #$F0      ;Test auf Ueberlauf durch vorherige Addition
         BNE  BCDADD    ;wenn ja, Korrektur
         TFR  B,A       ;
         CMPA #$09      ;Test auf gueltige BCD-Ziffer
         BLS  CARRY2    ;wenn nein, Korrektur
BCDADD:  ADDB #$06      ;Korrektur, als Ergebnis gueltige BCD-Ziffer
CARRY2:  TFR  B,A       ;
         ANDA #$F0      ;
```

[7] Für das Praktikum verwenden Sie bitte nur die Endung ".ASM", da alle Tools des Praktikums diese Endung erwarten.

```
         BEQ  SHOW      ;Test auf Ueberlauf vorh. Addition/Korrektur
         LDA  #$01      ;
         ADDA 1,Y       ;naechsthoeherwertige Ziffer um Eins erhoehen
         STA  1,Y       ;naechsthoeherwertige Ziffer abspeichern
SHOW:    ANDB #$0F      ;eventl. Uebertrag der korrig. Ziffer loeschen
         STB  0,Y       ;korrigierte Ziffer abspeichern
         JSR  SHOWT7SG  ;korrigierte Ziffer in Anzeige
         LDB  1,Y       ;naechsthoeherwertige Ziffer laden
         LEAX 1,X       ;Anzeigestelle nach links verschieben, X:=X+1
         JSR  SHOWT7SG  ;naechsthoeherwertige Ziffer in Anzeige
         BRA  START     ;zurueck zur naechsten Eingabe
ENDE:    SWI1

CLRDISP   EQU   $F110  ;Loeschen der Anzeige, In:-, Out:-
SHOWT7SG  EQU   $F11C  ;unteres Nibble von B in Anzeige, Position in X
HALTKEY   EQU   $F143  ;Lesen der Tastatur mit Warten, In:-, Out:B
```

Hinweis:

Zur Veranschaulichung kann das assemblierte Programm auch direkt in den Praktikumsrechner geladen werden. Abschnitt 2.4.5 erläutert hierzu verschiedene Möglichkeiten.

Praktische Übung P2.4-1:

Assemblieren Sie das Programm "P24-1.ASM" durch Aufruf des Assemblers mit der Kommandozeile "AS9 P24-1.ASM". Rufen Sie dazu im Start-Menue unter „Alle Programme“ in der Programmgruppe „Zubehör“ die Eingabeaufforderung auf.

Assemblieren Sie das Programm "P24-1.ASM" unter der Verwendung verschiedener Optionen, z.B. mit der Kommandozeile "AS9 P24-1.ASM - L > P24-1.LST" und betrachten Sie die erzeugte Datei "P24-1.LST" mit Hilfe eines Texteditors. (Sollten Sie keinen Texteditor zur Verfügung haben, lesen Sie Abschnitt 2.4.4). Welche Aufgabe erfüllt das Programm? Wie reagiert das Programm, wenn der Wert "99" überschritten wird? Entwerfen Sie ein Flußdiagramm!

Fehlermeldungen

Ist das assemblierte Quellprogramm syntaktisch nicht korrekt, so wird vom "AS9" vor der syntaktisch falschen Befehlszeile eine Fehlermeldung in das Listing eingefügt. Es werden zwei Arten von Fehlermeldungen ausgegeben:

Befehlszeile: Beschreibung des Fehlers

Befehlszeile: Warning --- Beschreibung des Fehlers.

Fehler im ersten Durchlauf des Assemblers führen bei "normalen" Fehlern zu keinem Abbruch der Assemblierung, d.h. der zweite Durchlauf wird durchgeführt und bei Angabe der entsprechenden Optionen ein Listing erzeugt. Dies ist eine Erweiterung des Motorola Assemblers, der bei Fehlern im ersten *pass* die Assemblierung abgebrochen hat.

Um ein Erkennen dieser Fehler in der Programmumgebung zu ermöglichen, wurde diese Erweiterung eingebaut. Allerdings können nachfolgende vom Assembler erkannte Fehler, Folgefehler des ersten Fehlers sein. Diese sollten nur unter starkem Vorbehalt (oder gar nicht) betrachtet werden, um Rückschlüsse auf den ersten Fehler zu ziehen. Nach Korrektur des ersten Fehlers ist eine nochmalige Assemblierung des Programms durchzuführen und bei Auftauchen weiterer Fehler sind diese, Fehler für Fehler, zu korrigieren. Bedingt durch die fortgesetzte Assemblierung des ersten pass im Fehlerfall wird auch eine Datei mit Maschinencode erzeugt. Diese ist natürlich nicht korrekt und sollte auch nicht zu Testzwecken benutzt werden. Erst im fehlerfreien Fall entsteht ein korrektes Maschinenprogramm.
Warnungen (Fehlertyp 2) führen ebenfalls zu keinem Abbruch der Assemblierung, weisen aber auf ein mögliches Zuordnungsproblem zwischen Symbolen und Adressen hin. Diese Warnungen sollten ernst genommen werden. Einige Fehler werden vom Assembler als "*fatal error*" klassifiziert und führen zu einem sofortigen Abbruch der Assemblierung. Bei diesen Fehlern kann der Assembler im allgemeinen keine temporären Dateien erzeugen und im Verzeichnis des Assemblers speichern (z.B. kein freier Speicher), oder diese Dateien wurden während der Assemblierung beschädigt.

Werden mehrere Dateien gleichzeitig assembliert, so erweitert sich die Fehlermeldung zu:

Dateiname, Befehlszeile: Beschreibung des Fehlers.

Am Ende des Listings wird die Gesamtanzahl der gefundenen Fehler ausgegeben.

Praktische Übung P2.4-2:
Assemblieren Sie das Programm "P24-2.ASM" durch Aufruf des Assemblers mit der Kommandozeile "AS9 P24-2.ASM -L > P24-2.LST". Korrigieren Sie die gefundenen Fehler und assemblieren Sie neu. Welcher logischer Fehler ist weiterhin in dem Programm, und wie verändert er die Funktion des Programms?

4. Assembler-Direktiven

Assembler-Direktiven sind Befehle, mit deren Hilfe der Assemblierungsvorgang und die Art der Assemblierung gesteuert werden kann. Die Direktiven des "AS9" sind im Gegensatz zu anderen Assemblern rudimentär, erfüllen jedoch die für das Praktikum gestellten Anforderungen. Für Ihre ersten Assemblerprogramme reicht es aus, wenn Ihnen die Direktiven "ORG" und "EQU" geläufig sind. Alle anderen Direktiven werden in der Regel erst bei größeren Programmen benutzt. Die Direktiven sind im folgenden alphabetisch aufgeführt. Sind Teile des Befehls in runde Klammern gesetzt, so sind diese optional. Der Begriff "Ausdruck" kann ein Zeichen, eine numerische Konstante, ein Symbol oder einen durch einen arithmetischen Ausdruck berechenbaren Wert repräsentieren. Ist es im folgenden nicht anders erwähnt, wird bei Direktiven der benutzten Marke der aktuelle Inhalt des Programmzählers zugeordnet.

BSZ

(Marke) BSZ Ausdruck (Kommentar)

Mit der Direktive BSZ (*Block Storage of Zeros*) kann eine bestimmte Anzahl von Bytes allokiert werden. Jedes Byte des Blockes wird mit Null initialisiert. Die Anzahl der zu allokierenden Bytes wird durch "Ausdruck" bestimmt. Sollte "Ausdruck" ein undefiniertes Symbol oder ein Symbol enthalten, dem erst später im Programm ein Wert zugewiesen wird (*forward reference*), wird ein Fehler (*Phasing Error*) generiert.

EQU

Marke EQU Ausdruck (Kommentar)

Mittels der Direktive EQU (*EQUate Symbol to a value*) wird einer Marke der Wert von "Ausdruck" zugewiesen. Dieser Wert ist *nicht* notwendigerweise der aktuelle Wert des Programmzählers. Der Marke kann innerhalb des Programms kein neuer Wert zugewiesen werden. Sollte "Ausdruck" ein undefiniertes Symbol oder ein Symbol enthalten, dem erst später ein Wert zugewiesen wird (*forward reference*), wird ein Fehler generiert.

FCB

(Marke) FCB Ausdruck (,Ausdruck, ..., Ausdruck) (Kommentar)

Der Direktive FCB (*Form Constant Byte*) können ein oder mehrere durch Kommata getrennte Ausdrücke folgen. Der Wert eines jeden Ausdruckes wird auf acht Bit begrenzt und wird in je einem Byte im Speicher ab dem aktuellen Wert des Programmzählers abgelegt. Bei mehreren Ausdrücken werden die einzelnen Bytes hintereinander abgespeichert.

FCC

(Marke) FCC Ausdruck Begrenzer String Begrenzer (Kommentar)

Mit der Direktive FCC (*Form Constant Character string*) kann ein String (eine Folge von Zeichen) im Speicher abgelegt werden. Das erste Zeichen wird unter der Speicheradresse abgespeichert, auf die der aktuelle Inhalt des Programmzählers zeigt. Die folgenden Zeichen werden unter den nachfolgenden Speicheradressen abgelegt. Der (optionalen) Marke wird die Speicheradresse des ersten Zeichens zugewiesen. Der String kann jedes druckbare Zeichen enthalten und wird in der Befehlszeile durch zwei identische Begrenzer, die ebenfalls jedes druckbare Zeichen sein können, gekennzeichnet. Das erste Zeichen nach der Direktive FCC, das kein Leerzeichen ist, wird als Begrenzer interpretiert. Zum Beispiel wird mit der Befehlszeile

```
MESSAGE1  FCC  ,Input correct,
```

der String "Input correct" der Marke "MESSAGE1" zugewiesen.

FDB

(Marke) FDB Ausdruck (,Ausdruck, ...,Ausdruck) (Kommentar)

Der Direktive FDB (*Form Double Byte constant*) können ein oder mehrere durch Kommata getrennte Ausdrücke folgen. Der Wert eines jeden 16-bit-Ausdruckes wird in zwei nacheinander folgenden Bytes gespeichert. Bei mehreren Ausdrücken werden die 16-bit in je zwei hintereinander folgenden Bytes abgespeichert (Big Endian Format: High Byte unter der niedrigeren Speicheradresse, Low Byte unter der höheren).

FILL

(Marke) FILL Ausdruck, Ausdruck

Mittels der Direktive FILL (*FILL memory*) kann der Assembler veranlaßt werden, Code für die Initialisierung eines Speicherbereiches mit einem konstanten Wert zu generieren. Der erste Ausdruck repräsentiert die Konstante (0 - 255), mit der der Speicherbereich aufgefüllt werden soll. Der zweite Ausdruck gibt die Anzahl der zu initialisierenden Bytes innerhalb des Speichers an.

OPT

OPT option (,option, ..., option) (Kommentar)

Die Direktive OPT (*assembler output OPTions*) wird benutzt, um neben dem zu erzeugenden Maschinenprogramm zusätzliche Ausgabe-Listings zu generieren. Die Optionen sind mit den Optionen, die auf der Kommandozeilenebene eingegeben werden können, identisch. Allerdings überschreiben die Optionen des Quellprogramms die Optionen der Kommandozeilenebene. Folgende Optionen stehen zur Verfügung:

L	Ausgabe des Programmlistings ab der aktuellen Befehlszeile
NOL	Keine Ausgabe des Programmlistings (Voreinstellung). Kann mit der Option "L" dazu verwendet werden, nur Teile des Programmlistings auszugeben.
CRE	Gibt die für das assemblierte Programm gültige Cross-Reference-Tabelle am Ende des Programmlistings aus. Wird diese Option im Quelltext verwendet, so muß die Option vor dem ersten Symbol des Programmtextes stehen.
S	Gibt am Ende des Programm-Listings die Symboltabelle aus.
C	Gibt für jeden Befehl die benötigten Prozessortakte des Befehls an. Die Taktanzahl erscheint im Programmlisting nach dem Maschinencode und vor dem Quellcode.
NOC	Schaltet die Option "C" aus (Voreinstellung).

Die Formate der einzelnen Listings werden in Unterabschnitt 5 besprochen.

ORG

ORG Ausdruck (Kommentar)

Die Direktive ORG (*set program counter to ORiGin*) setzt den Programmzähler auf den Wert, der von "Ausdruck" repräsentiert wird. Die folgenden Befehlszeilen werden vom Assembler an die entsprechend nachfolgenden Speicheradressen verschoben und assembliert. Ist in einem Quellprogramm keine "ORG"-Direktive gesetzt, so wird der Programmzähler mit Null initialisiert, und das Maschinenprogramm beginnt entsprechend dem Programmzähler an der Speicheradresse $0000. Undefinierte Ausdrücke werden vom Assembler als Fehler erkannt.

PAGE

PAGE (Kommentar)

Mit der Direktive PAGE (*top of PAGE*) kann der Assembler veranlaßt werden, das zu erzeugende Programmlisting ab der Direktive "PAGE" auf einer neuen Seite beginnen zu lassen. Wird kein Programmlisting erzeugt, so hat PAGE keine Auswirkungen.

RMB

(Marke) RMB Ausdruck (Kommentar)

Die Direktive RMB (*Reserve Memory Bytes*) reserviert einen Speicherbereich in der Größe der Anzahl der Bytes, die "Ausdruck" repräsentiert. Der Speicherbereich wird im Gegensatz zu der Direktive "FILL" nicht mit einem konstanten Ausdruck vorinitialisiert. Diese Direktive wird in der Regel für die Reservierung von Speicher für den späteren Gebrauch von Tabellen benutzt. Wird in der Befehlszeile eine "Marke" verwendet, so erhält die "Marke" die Adresse des ersten Byte des reservierten Speicherbereiches.

ZMB

(Marke) ZMB Ausdruck (Kommentar)

Die Direktive ZMB (*Zero Memory Bytes*) entspricht der Direktive BSZ und wird aus Kompatibilitätsgründen zu anderen Assemblern mitgeführt.

5. Datei-Formate

Mit den Assembler-Optionen "-L", "-S" , "-C" und "-CRE" kann der Assembler veranlaßt werden, neben dem eigentlichen Maschinencode zusätzliche Informationen in ein Listing auszugeben.
Im folgenden werden die einzelnen Ausgabeformate besprochen.

Programmlisting

Das Assembler-Programmlisting hat das folgende Zeilenformat:

Zeilennummer	Speicheradresse	Maschinencode	([#Zyklen])	Quellcode

Die Zeilennummer ist eine vierstellige fortlaufende Nummer, die als Verweis in der Cross-Reference-Tabelle verwendet wird. Die Speicheradresse ist eine vierstellige Hexadezimalzahl, die die Adresse des ersten Bytes des jeweiligen Maschinenbefehles im Speicher des Rechners angibt. Der Maschinencode ist der vom Assembler erzeugte Code für den 6809-Prozessor, entsprechend dem im Quellcode stehenden Assemblerbefehl. Der Maschinencode wird hexadezimal angegeben und kann mehrere Bytes betragen. Ist im Quellcode oder auf Kommandozeilenebene die Option "-C" gewählt worden, so erscheint im Programmlisting nach dem Maschinencode die Anzahl der benötigten Taktzyklen des Prozessors für den Maschinenbefehl in eckigen Klammern. Danach wird das Quellcodeprogramm mit Marken-, Operator-, Operanden- und Kommentarfeld ausgegeben.

Symboltabelle

Die Symboltabelle hat das folgende Zeilenformat:

Symbol	Symbolwert

Alle Symbole, die im Markenfeld des Quellprogramms definiert wurden, werden in dieser Tabelle aufgelistet. Im ersten Feld steht der Name des jeweiligen Symbols, im zweiten Feld der Wert des Symbols in hexadezimaler Form. Der Wert des Symbols ist bei einer Marke die Speicheradresse, für die die Marke definiert wurde oder bei einer Konstantendefinition der Wert der Konstanten.

Cross-Reference-Tabelle

Die Cross-Reference-Tabelle hat das folgende Zeilenformat:

Symbol Symbolwert * Zeilennummer der Def. Zeilennummer ...

Die Cross-Reference-Tabelle ist eine Erweiterung der Symboltabelle. In den ersten beiden Spalten stimmt sie mit der Symboltabelle überein. Im dritten Feld wird nach einem Stern (*) die Zeilennumer des Quellprogramms aufgelistet, in dem das Symbol definiert wurde. Danach werden alle Zeilennummern aufgelistet, in denen das Symbol (Marke) verwendet wird.

Dateiformate

Wie bereits erwähnt, speichert der Assembler "AS9" das erzeugte Maschinenprogramm im Intel HEX-Format oder im Motorola S-Dateiformat ab. Das generierte Programm (im Intel HEX-Format) kann direkt in den Praktikumsrechner geladen werden (Abschnitt 2.4.4). Im folgenden werden die beiden Datenformate für Interessierte kurz vorgestellt.

Motorola S-Dateiformat

Dieses Format wurde von Motorola entwickelt, um Programm- und Objektdateien in einer druckbaren Form darstellen zu können. Hierdurch können die Dateien mit jedem Texteditor betrachtet und ohne Probleme zwischen verschiedenen Rechnern via Modemkabel ausgetauscht werden. Das S-Format beinhaltet neben den Daten und einigen Steuerbytes ein Prüfsummen-Feld, mit dem die Integrität der Datei überprüft werden kann.

Eine S-Datei kann aus mehreren S-Datensätzen gleichen oder verschiedenen Typs zusammengesetzt sein. Ein S-Datensatz besteht aus mehreren Feldern fester und variabler Länge, die den Typ des S-Datensatzes, die Anzahl der Datenbytes, die Speicheradresse, die Daten und das Prüfsummenfeld beinhalten. Um die binären Daten als druckbare ASCII-Zeichen darstellen zu können, werden sie codiert im S-Datensatz dargestellt. Jedes Byte mit "binärem Inhalt" wird in zwei Bytes, welche das "binäre Byte" im ASCII-Code darstellen, aufgeteilt. Das erste ASCII-Byte repräsentiert die oberen vier Bits der zu codierenden acht Bits, das zweite ASCII-Byte die unteren vier Bits. Ein S-Datensatz besteht aus fünf Feldern mit folgendem Format:

Tabelle 2.4-4: Motorola S-Format

Feldtyp	Bytes	Inhalt
Datensatztyp	2	Typ des S-Datensatzes: S1, S9, ...
Satzlänge	2	Anzahl der Zeichenpaare (hexadezimal) im Datensatz - ohne die Felder "Datensatztyp" und "Satzlänge"
Speicheradresse	4,6 oder 8	Speicheradresse, ab der die nachfolgenden Daten abgespeichert werden sollen, beginnend mit dem ersten Zeichenpaar des Datenfeldes
Daten	0-2n	0 - n codierte Bytes; ausführbarer Maschinencode, Daten oder Informationen
Checksumme	2	Die 8-bit-Summe der Werte, die durch die Zeichenpaare der Felder "Satzlänge", "Speicheradresse" und "Daten" repräsentiert werden; im Einerkomplement

Jeder S-Datensatz kann mit einem der Steuerzeichen CR (*carriage return* - Wagenrücklauf), LF (*line feed* - Zeilenende) oder NUL (*null character* - Nullzeichen) abgeschlossen werden.

Motorola definierte für verschiedene Anwendungen acht S-Datensatzformate, von denen der " Assembler AS9" zwei Typen benutzt:

S1: Der Datensatz beinhaltet Maschinencode oder Daten. Die Speicheradresse ist 4 Zeichen lang.

S9: Der Datensatz "S9" wird als letzter Datensatz zur Terminierung einer Übertragung an das Ende einer S-Datei angehängt. Das Feld "Datensatztyp" hat den Wert "S9", das Feld "Satzlänge" den Wert "03", das "Speicheradressfeld" den Wert "0000", das "Datenfeld" ist leer, und das "Prüfsummenfeld" hat den Wert "FC".

Aus der Verwendung der beiden Datensatztypen "S1" und "S9" leitet sich die Endung "S19" der generierten "AS9" Maschinencodedateien ab.

Intel Hex-Dateiformat

Das Intel Hex-Format unterscheidet sich vom Motorola S-Format im wesentlichendurch die Anordnung der einzelnen Datenfelder eines Datensatzes, durch die Firmen-Kennung sowie durch das Prüfsummenfeld. Wie eine Motorola S-Datei besteht eine Intel Hex-Datei ebenfalls aus mehreren Datensätzen. Die Codierung des Datenfeldes entspricht dem Motorola-Verfahren, indem ein Byte in zwei ASCII-Zeichen aufgesplittet wird. Tabelle 2.4-5 zeigt das Format der Datensätze.

Tabelle 2.4-5: Intel Hex-Format

Feldtyp	Anzahl Zeichen	Inhalt
Kennung	1	Intel-Kennung: ":"
Satzlänge	2	Anzahl der Zeichenpaare (hexadezimal) im Datenfeld (ohne alle anderen Felder)
Speicheradresse	4	Speicheradresse, ab welcher der nachfolgende Inhalt des Datenfeldes abgespeichert werden soll, beginnend mit dem ersten Zeichenpaar des Datenfeldes
Datensatztyp	2	Typ des Intel-Datensatzes: 00, 01, ...
Daten	0-2n	0 - n codierte Bytes ausführbarer Maschinencode, Daten oder Informationen
Checksumme	2	Die 8-bit-Summe der Werte, die durch die Zeichenpaare der Felder "Satzlänge", "Speicheradresse", "Datensatztyp" und "Daten" repräsentiert werden; im Zweierkomplement

Der Datensatztyp "00" entspricht dem Motorola Typ "S1", der Intel Typ "01" entspricht dem Motorola Typ "S9". Demnach besteht der Intel Typ "01" aus der Zeichenfolge ":00000001FF". Das Feld "Speicheradresse" enthält wie bei Motorola den Wert "0000".

2.3.8 Der 6809-Disassembler "DS9"

Der im Praktikum verwendete 6809-Disassembler "DS9" "rück-übersetzt" Maschinenprogramme, die im Intel-Hex Format gespeichert sind, in Assemblerprogramme, mit der Ihnen vom "AS9" bekannten Syntax. Sie finden den Disassembler im Unterverzeichnis "ASSM6809" des Praktikumverzeichnisses "FU_MRP".

Auf DOS-Ebene wird der Disassembler durch das Kommando "DS9" gestartet. Der Bildschirm wird gelöscht und es erscheint die Meldung "6809 Disassembler Vers. #". In der folgenden Bildschirmzeile wird nach der zu disassemblierenden Datei gefragt. Nach Eingabe, eventuell mit Pfadangabe und optional ohne Endung ".HEX", wird die Datei eingelesen und disassembliert. Das Programm zeigt dies durch die Meldungen "Reading Input file:", "Processing File:" und "OK" an. Wird die einzulesende Datei nicht gefunden, so bricht das Programm ohne Fehlermeldung ab. Als Ergebnis der Disassemblierung wird eine Datei erzeugt, die das eingelesene Programm in Maschinencode und in Assemblersprache auflistet. Als Dateiname wird der Name der eingelesenen Datei mit der Endung ".LST" verwendet. Die Eingabe eines anderen Dateinamens als Ausgabedatei ist nicht möglich.

Der "DS9" unterstützt auch den Kommandozeilenmodus. Beispielsweise wird durch die Eingabe "DS9 BCDADD" auf DOS-Ebene die Datei "BCDADD.HEX" eingelesen und disassembliert. Als Ausgabedatei wird die Datei "BCDADD.LST" erzeugt.

Praktische Übung P2.4-3:

Disassemblieren Sie das Programm "P1.HEX" durch Aufruf des Disassemblers mit der Kommandozeile "DS9 P1". Kopieren Sie bitte vorher das bereits vorhandene Assemblerlisting "P1.LST" nach "P1old.LST" und vergleichen Sie nach der Disassemblierung die beiden Listings. Welche Unterschiede bestehen zwischen den beiden Listings?

Ein Disassembler kann in der Regel nicht zwischen Programmcode und Daten unterscheiden[8]. Von komfortablen Disassemblern werden einige Hilfestellungen zur Erkennung von Unterprogrammroutinen und der automatischen Generierung von Marken (*Labels*) gegeben. Dennoch ist immer ein "Nach-Disassemblieren" per Hand notwendig, um Marken und Unterprogrammen aussagekräftige Namen zu geben, Schnittstellen zwischen Unterprogrammroutinen zu erkennen und um festzustellen, ob es sich bei dem vorliegenden Speicherbereich um Programmcode, einen Stack oder einfach Daten handelt.

Disassemblierung in Verbindung mit einem Debugger ist auch heute noch in der Zeit der objektorientierten und logischen Programmiersprachen, der Client/Server-Architekturen und der graphischen Benutzeroberflächen, unter Umständen das letzte Hilfsmittel, einen Fehler in einem Programm zu finden. Programmcode, der auf Hochsprachenebene in mehreren Durchläufen debuggt wurde, - und für richtig befunden wurde -, kann durch optimierende Compiler oder auch durch schlichtweg falsch übersetzende Compiler und Interpreter zu unverständlichen Fehlern führen. In diesen Fällen hilft nur ein Debuggen und Disassemblieren auf Maschinensprachebene.

[8] Außer Programmcode und Daten werden in verschiedenen Speichersegmenten gehalten.

2.3.9 Die 6809-Entwicklungsumgebung "IDE9"

1. Allgemeine Informationen

Die Entwicklungsumgebung "IDE9" *(Integrated Development Environment*) ist auf einem IBM-kompatiblen Rechner unter DOS ablauffähig und präsentiert sich mit einer SAA-Benutzeroberfläche. Eine Bedienung mit der Maus ist möglich. Die Entwicklungsumgebung besitzt eine Dateiverwaltung, einen vollständigen Texteditor, Menüpunkte zum Assemblieren und Disassemblieren von erstellten oder geladenen 6809-Programmen sowie verschiedene Möglichkeiten zur Datenübertragung zwischen Praktikumsrechner und PC.

Nach Aufruf des Programms auf DOS-Kommandozeilenebene mit "IDE9" erscheint ein Fenster, in dem Sie den seriellen Port des PCs für die Übertragung zwischen Praktikumsrechner und PC wählen können (Voreinstellung "COM2", falls vorhanden). Nach Betätigen der Return-Taste schließt das Fenster und Sie befinden sich im Hauptfenster der Entwicklungsumgebung.

In der obersten Zeile des Fensters befindet sich die Menüleiste, mit der Sie die verschiedenen Untermenüs zur Versionskennung (?), zur Dateiverwaltung (*File*), zum Texteditor (*Edit*), zur Datenübertragung (*Transmit*), zum Assemblieren und Disassemblieren von 6809-Programmen (*Projekt*) oder zur Fensterverwaltung (*Window*) erreichen können. Die einzelnen Untermenüs erreichen Sie entweder mit einem Mausklick, oder über die F10-Taste mit nachfolgender Auswahl durch die Cursortasten, oder über die Tastenkombination "<ALT> + rot hervorgehobener Buchstabe".

In der untersten Zeile des Hauptfensters befindet sich eine Statuszeile, in der die wichtigsten HotKeys aufgelistet sind. Ganz rechts in der Zeile wird der noch zur freien Verfügung stehende Arbeitsspeicher angezeigt.

Für die einzelnen Untermenüs erscheint jeweils ein weiteres Fenster, in dem die zu dem Unterpunkt passenden Funktionen ausgewählt werden können. Bei einigen Unterpunkten öffnen sich Dialogboxen, um zwischen verschiedenen Einstellungen auszuwählen oder auch Parameter einzugeben. In einer Dialogbox kann mittels der TAB-Taste durch die verschiedenen Funktionsgruppen durchgeschaltet werden (z.B. Dateimaske, Dateiauswahl, Buttons, usw.). Ein Dialog wird in einer Box mit der Return-Taste abgeschlossen oder mit der ESC-Taste abgebrochen. Einige Funktionen können in den Boxen auch direkt durch die HotKeys "<ALT> + hervorgehobener Buchstabe" angesprochen werden.

Die Dateiverwaltung erlaubt neben dem Laden und Abspeichern von Dateien auch das Speichern einer Datei unter einem anderen Namen sowie einen Verzeichnis- oder Laufwerkswechsel. Der Texteditor entspricht in der Handhabung dem üblichen Standard und besitzt neben Undo-, Cut-, Copy-, Paste- und Delete-Funktionen auch die Möglichkeit, eine Textstelle zu suchen bzw. zu suchen und zu ersetzen.

Im folgenden wird die Menüstruktur der Entwicklungsumgebung aufgelistet, anschließend ein Assemblierungsvorgang erläutert und danach die Datenübertragungsmöglichkeiten zwischen PC und Praktikumsrechner besprochen.

2. Menüstruktur

Da die Menüstruktur im wesentlichen selbsterklärend ist, werden die einzelnen Menüunterpunkte nur kurz beschrieben. Sollte Ihnen ein Programmpunkt unklar sein, so probieren Sie die Funktion einfach aus.

FILE

- New	erzeugt eine neue zu editierende Textdatei "Untitled".
- Open (F3)	öffnet eine existierende Textdatei.
- Save (F2)	speichert die aktive Datei.
- Save as	speichert die aktive Datei unter einem anderen Namen.
- Format	öffnet eine Dialogbox zur Festlegung des Dateiformates.
- Change dir	wechselt in ein anderes Verzeichnis oder Laufwerk.
- Exit (Alt-X)	beendet das Programm.

EDIT

(Textbereiche werden mit der Shift-Cursortaste oder der Maus markiert.)

- Undo (Alt-U)	macht die letzte Aktion im Editor rückgängig.
- Cut (Shift-Entf)	schneidet den markierten Bereich aus und legt ihn im Clipboard ab.
- Copy (Strg-Einf)	kopiert den markierten Bereich ins Clipboard.
- Paste (Shift-Einf)	fügt den Inhalt des Clipboards ein.
- Show Clipboard	zeigt den Inhalt des Clipboards an.
- Delete (Strg-Entf)	löscht den markierten Bereich.
- Find	öffnet eine Dialogbox für die Textsuche.
- Replace	öffnet eine Dialogbox für den Textaustausch.
- F&R next (Strg-L)	wiederholt die letzte Find- oder Replace-Aktion.

TRANSMIT

- Send (Alt-S)	sendet die aktive Datei, falls das eingestellte Dateiformat mit der Dateiendung übereinstimmt. Öffnet sonst eine Dialogbox zur Dateiauswahl.
- Receive (Alt-R)	empfängt eine Datei im eingestellten Dateiformat.
- Port	öffnet eine Dialogbox zur Auswahl des Kommunikationsports.
- Baud	öffnet eine Dialogbox zur Einstellung der Baudrate.

PROJECT

- Assemble (F9)	assembliert die aktive Datei, falls die Dateiendung "ASM" lautet. Öffnet sonst ein Dateiauswahlfenster. Treten bei der Assemblierung Fehler auf, oder ist einer der Assembler-Parameter gesetzt, wird ein Listing erzeugt und angezeigt.
- Parameter	öffnet eine Dialogbox zur Einstellung der Assemblerparameter.
- Disassemble (Shift F9)	ruft den Disassembler auf, falls die Dateiendung der aktiven Datei ".HEX" lautet. Öffnet sonst ein Dateiauswahlfenster zur Auswahl der zu disassemblierenden Datei.

WINDOW

- List listet alle geöffneten Fenster der Entwicklungsumgebung auf.
- Size/Move (Strg-F5) bietet die Möglichkeit, Fensterposition und -größe per Tastatur zu verändern. Die Cursortasten verändern die Position und die <Shift-Taste> + Cursortasten verändern die Größe des Fensters.
- Zoom (F5) zoomt ein Fenster auf Maximalgröße und wieder zurück.
- Tile ordnet alle Fenster nicht überlappend an.
- Cascade ordnet alle Fenster überlappend an.
- Next (F6) schaltet auf das nächste Fenster.
- Previous (Shift-F6) schaltet auf das vorherige Fenster.
- Close (Alt-F3) schließt das aktive Fenster.

3. Assemblierung und Disassemblierung

Editieren Sie in Ihrem aktiven Fenster eine Datei mit der Endung "ASM", so können Sie diese sofort assemblieren, in dem Sie entweder die Funktionstaste F9 betätigen oder im Untermenü "*PROJECT*" die Funktion "*ASSEMBLE*" anwählen. Sollten Sie keine Datei mit der Endung "ASM" bearbeiten, öffnet sich bei anwählen der Funktion "*ASSEMBLE*" (oder der Taste F9) ein Dateiauswahlfenster. Nach dem Laden der ausgewählten Datei (mit Endung "ASM") wird diese durch den Assembler "AS9" in ein Maschinenprogramm übersetzt. Das Maschinenprogramm erhält das Dateiformat, das im Unterpunkt "*FORMAT*" vereinbart wurde (Voreinstellung "HEX").

Treten bei der Übersetzung Fehler auf, oder sind Optionen für den Assembler im Unterpunkt "*PARAMETER*" gesetzt worden, so wird ein Listing mit dem Dateinamen der assemblierten Datei mit der Endung "LST" erzeugt und nach der Übersetzung in einem neuen Fenster angezeigt. Vor dem Assemblieren werden alle Dateien, die den gleichen Dateinamen wie die zu übersetzende Datei und eine der Endungen "LST", "HEX" oder "S19" besitzen auf Festplatte zurückgespeichert und die entsprechenden Fenster geschlossen.

Durch die Tastenkombination <Shift-F9> oder durch die Funktion "*DISASSEMBLE*" im Untermenü "*PROJECT*" kann eine Datei mit der Endung "HEX" disassembliert werden. Hat die gerade aktive Datei die Endung "HEX", so wird diese disassembliert und das Ergebnis in einem Fenster angezeigt. Ansonsten öffnet sich ein Dateiauswahlfenster, durch das die gewünschte Datei selektiert werden kann.

Alle Fenster, die den gleichen Namen wie die zu disassemblierende Datei, aber die Endung "LST" haben, werden geschlossen. Der Disassembler erzeugt eine Datei mit dem Namen der zu disassemblierenden Datei und der Endung "LST", die in einem neuen Textfenster angezeigt wird. Achtung: eine "alte" Datei mit dem Dateinamen der aktiven Datei und der Endung "LST", die zum Beispiel vorher durch den Assembler erzeugt wurde, wird überschrieben. Sollte diese Datei nach einer Disassemblierung noch zur Verfügung stehen, können Sie der "alten" Datei durch die Funktion "*SAVE AS*" im Untermenü "*FILE*" einen neuen Dateinamen geben.

Praktische Übung P2.4-4:
Assemblieren Sie das Programm "P24-1.ASM" durch Aufruf des Assemblers in der Entwicklungsumgebung "IDE9" mit verschiedenen Assembleroptionen. Lesen Sie das erzeugte Listing P24-1.LST und laden Sie das erzeugte Maschinenprogramm P24-1.HEX in ein Textfenster. Wählen Sie im Untermenü "*FORMAT*" das S19-Dateiformat und assemblieren Sie erneut. Laden Sie die Datei P24-1.S19 in ein Textfenster und vergleichen Sie das S19-Dateiformat mit dem Intel HEX-Format.

2.3.10 Der 6809-Simulator[9] "EM9"

Der im Praktikum für den 6809-Prozessor entwickelte Simulator wurde unter der graphischen Benutzeroberfläche "GEM" programmiert. Der Simulator bildet das Verhalten jeder Komponente des Praktikumsrechners auf dem PC nach. Damit alle Funktionen des Simulators verfügbar sind, ist ein freier Hauptspeicher von ca. 540 kByte unterhalb der 1MB-Speichergrenze notwendig. Der Simulator ist auch im DOS-Fenster von Windows ablauffähig. Sie starten den Simulator, indem Sie in Ihrem jeweiligen Unterverzeichnis auf Kommandozeilenebene "EM9" eingeben.

Der 6809-Simulator erlaubt das Laden und Speichern von Programmen auf Festplatte. Eine Datenübertragung zum bzw. vom Praktikumsrechner über die serielle Schnittstelle des PCs ist ebenfalls möglich. Die Tastatur und die Anzeige des Praktikumsrechners werden in einem Fenster "wie gewohnt" dargestellt. Der Simulator benutzt für die Nachbildung des Verhaltens des Praktikumsrechners exakt die gleichen Monitorroutinen, die auch im EPROM Ihres Praktikumsrechners vorhanden sind. Der Speicherbereich des Praktikumsrechners kann in einem Editor-Fenster angezeigt und editiert werden. Mittels eines integrierten Disassemblers kann der aktuelle Speicherinhalt als Assemblerprogramm dargestellt werden. Die Registerwerte des Prozessors können in einem gesonderten Fenster angezeigt werden. Weitere Fenster simulieren das Verhalten des Timers, der ACIA und der parallelen Schnittstelle.

Die folgende Beschreibung erläutert die für die Anwendung des Simulator-Programms wesentlichen Bedienschritte. Weitergehende Anleitungen zu den einzelnen Funktionen finden Sie als "Online"-Hilfe im Simulator unter dem Menüpunkt "Anleitung" bzw. durch Anklicken von Fragezeichen (?) mit der linken und gleichzeitigem Drücken der rechten Maustaste. Beschreibungen zur Bedienung des Simulators für die Rechnerkomponenten Timer, parallele Schnittstelle und V.24-Schnittstelle finden Sie in den entsprechenden Kapitel 4, 5.

[9] synonym auch mit Emulator bezeichnet.

1. Allgemeine Informationen zur graphischen Oberfläche

Nach dem Aufruf des Simulators wird Ihnen zunächst Hilfe zur Bedienung des Programms angeboten. Falls Sie diese akzeptieren, erscheint das im Bild 2.4-1 dargestellte Fenster. In diesem Bild finden Sie die Bezeichnungen der einzelnen Komponenten, die zur Bedienung erforderlich sind. Die Bezeichnungen werden in englischer und deutscher Sprache vorgestellt, um Ihnen eine möglichst universell anwendbare Einführung in die Begrifflichkeit dieses Themas zu bieten.

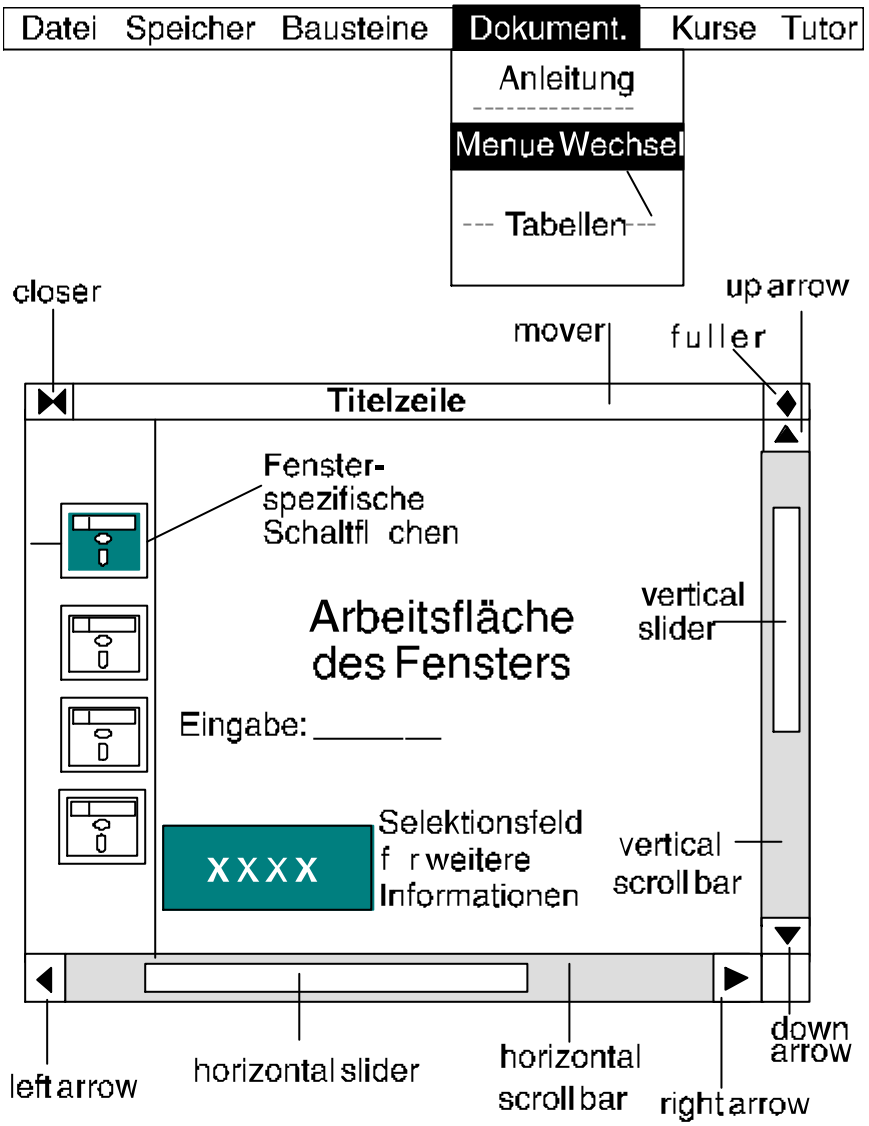


Bild 2.4-1: Aufbau eines Bildschirmfensters

Um einen im Fenster dargestellten Text komplett lesen zu können, ist es u.U. erforderlich, entweder den senkrechten Schieber (*vertical slider*) mit der linken Maustaste anzuklicken, festzuhalten und innerhalb des Verschiebebalkens (*vertical scrollbar*) nach unten zu verschieben, so daß ein nachfolgender Textausschnitt dargestellt wird, oder Sie klicken den nach unten gerichteten Pfeil (*down arrow*) mit der linken Maustaste an, um den Text nach unten zu rollen.

Es ist möglich, während der Benutzung des Simulators auf den im Bild 2.4-1 dargestellten Informationstext über den Menüpunkt **Datei**(Anleitung) **Programm**[10] zuzugreifen, indem Sie mit dem Mauszeiger auf den Menüeintrag "Datei" fahren und somit die zu diesem Eintrag gehörenden Unterpunkte anwählbar machen. Jetzt können Sie den Eintrag "(Anleitung) Programm" anwählen.

Verlassen können Sie diese Information durch Anklicken des Schließers (*closer*), der sich in der linken oberen Ecke eines Fensters befindet, mittels der linken Maustaste.

Gliederung dieses Unterabschnittes:

I. Benutzung der Objekte, die zur Bedienung der Fenster und das Öffnen und Schließen derselben zur Verfügung stehen.

II. Fensterbedienung, Eingabe von Daten in aktiven Fenstern, Anwendung der Iconen

III. Die zwei Menü-Ebenen des Simulators: Erläuterungen zum Haupt- und zum Dokumentationsmenü

IV. Bedeutung der Statusanzeige: Die Statusanzeige ist die helle Zeile am unteren Bildrand, die während der Simulatorbenutzung Informationen zum aktuellen Prozessorstatus enthält.

I. Fenster-Rahmen-Objekte

I.1 Schieber (*slider*)

Es existieren sowohl senkrechte als auch waagrechte Schieber (*vertical / horizontal slider*). Diese ermöglichen in dem Fall, in dem die Bildinformation in der Waag- oder Senkrechten das Fensterformat übersteigt, den Überblick über die gesamte Information des angesprochenen Themas. Hierbei stellt der weiße Bereich innerhalb des weißgrauen Balkens die relative Größe des Textes oder Bildausschnittes dar, der sich zur Zeit auf dem Bildschirm befindet. Wie oben bereits kurz erwähnt, ist dieser weiße Abschnitt innerhalb des Balkens durch Anklicken mit der linken Maustaste, Festhalten der Taste und gleichzeitiges Verschieben des Abschnittes nach oben oder unten zu bewegen. Mit dieser Aktion wird auch der Text- bzw. Bildinhalt in die gleiche Richtung verschoben.

Es ist auch möglich, an einer geeigneten Stelle des grauen Balkens zu klicken, um einen gewünschten Bereich in einem Text anzuwählen.

I.2 Pfeile (*arrows*)

Durch Anklicken der Pfeile mit der linken Maustaste, die nach links, rechts, oben bzw. unten weisen (also *left-, right-, up-* oder *down-arrow*), setzt sich der Fensterinhalt in die angewählte Richtung in Bewegung. Dies ermöglicht gegenüber der Benutzung der Verschieber (*slider*) eine kontinuierliche Bewegung in dem Text oder Bild in kleinen Schritten.

I.3 Fenster-Füllraute (*fuller*)

Als *fuller* bezeichnet man die Raute, die in der rechten oberen Ecke des Fensters zu erkennen ist. Durch Anklicken des *fullers* mit der linken Maustaste wird der Bildinhalt der linken Bildhälfte (hier: Graphik) vergrößert, so daß er den kompletten Bildschirm ausfüllt.

[10] Menü- und Untermenüpunkte werden im weiteren in der Form ....\....\.... angegeben.

Wenn Sie den alten Zustand wiederherstellen möchten, müssen Sie nur den *fuller* erneut anklicken.

I.4 Schließ-Knopf (*closer*)

In der linken oberen Ecke eines jeden Fensters befindet sich der sogenannte Schließer (*closer*). Durch Anklicken dieses Symbols mittels linker Maustaste wird, wie oben bereits erwähnt, das aktuelle Fenster geschlossen.

I.5 Verschiebebalken (*mover*)

Das aktuelle Fenster läßt sich komplett verschieben, indem man die Titelleiste mit der linken Maustaste anklickt, die Taste gedrückt hält und nun das Fenster in die gewünschte Position zieht. Hat man diese erreicht, so kann man durch Freigeben der linken Maustaste das Fenster an dieser Position fixieren.

II. Benutzung der Ikonen, Fensterverwaltung und Dateneingabe

II.1 Fensterbedienung

Unter der benutzten Oberfläche ist es möglich, mehrere Fenster gleichzeitig geöffnet zu haben. Hat man z.B. die Anleitung zum Zeitgeber-/Zählerbaustein (*Timer*) und den Timer selber als Fenster nacheinander geöffnet, so trägt das jeweils aktive Fenster eine schwarze Kennung in der Titelleiste, und die inaktiven Fenster tragen eine hellgraue Kennung. Durch einfaches Anklicken eines inaktiven Fensters mit der linken Maustaste wird dieses aktiviert und, sofern es vorher von anderen Fenstern verdeckt war, in seiner ganzen Größe in den Vordergrund gerückt. Ist in dem geschilderten Fall z.B. die Anleitung zur Benutzung des Timers aktiv gewesen, so kommt durch einfaches Klicken mit der linken Maustaste auf die Fläche des Praktikumsrechners dieser wieder in den Vordergrund und ist zu verwenden, ohne daß ein anderes Fenster geschlossen werden müßte.

II.2 Dateneingabe in aktiven Fenstern

In einigen Fenstern (z.B. "Bausteine \ ACIA einschalten" oder "Datei \ Speicherbereich laden") ist eine Dateneingabe über die Tastatur in das aktive Fenster möglich. Die aktuelle Position des Cursors wird über einen senkrechten Balken oder durch das Zeichen " | " angezeigt. In den Fenstern, die erscheinen, sobald die Menüpunkte "Speicherbereich laden" oder "... speichern" angewählt werden, kann man die Zeile, die man verändern, löschen oder ergänzen möchte, durch Anklicken mit der linken Maustaste aktivieren.

II.3 Handhabung der Ikonen

Ihnen werden während der Benutzung des Emulationsprogramms verschiedene Arten von Ikonen begegnen:

a.) Diskettensymbole als Schaltflächen zum Laden bzw. Löschen von Dateien auf Diskette oder Festplatte

b.) In den Dokumentationen[11] werden grüne Schaltflächen mit weißer Schrift dazu benutzt, die Bereiche darzustellen, die durch Anwählen mit der linken Maustaste weitergehende Informationen zu dem dargestellten Thema liefern.

[11] In der Praktikumsversion nicht implementiert.

c.) Die bei der Bedienung des Simulators auftretenden Fragezeichen sind wie folgt zu bedienen:

- ?? Die großen Fragezeichen dienen zur Anwahl eines Hilfe-Status. Das Anklicken dieser Quadrate hat zur Folge, daß das Hilfesystem zu dem entsprechenden Thema entweder an- oder ausgeschaltet wird.
- ?? Der aktive oder ruhende Hilfe-Status ist daran zu erkennen, daß in der betreffenden Darstellung kleine Fragezeichen unmittelbar neben den Objekten erscheinen, zu denen ein Hilfetext nach Anwahl des Fragezeichens erscheint. Ein kleines Fragezeichen wird angewählt, indem der Mauszeiger auf das Fragezeichen geführt, die rechte Maustaste gedrückt und gehalten und dann die linke Maustaste gedrückt wird.

Tiefergehende Informationen zu diesem Thema können Sie durch die Anwahl der Symbole des in Bild 2.4-1 dargestellten Fensters erhalten.

III. Die zwei Menü-Ebenen des Simulators

Wie diese Überschrift schon andeutet, gibt es während der Benutzung des Emulators die Möglichkeit, zwischen zwei Menüebenen zu wechseln. Dies geschieht über folgende Menüeinträge:

- ✍✍Befindet man sich in dem Menü, dessen Eintrag in der linken oberen Ecke "Datei" lautet (im folgenden als Hauptmenü bezeichnet), so kann man die zweite Menüzeile mit dem Eintrag "System" in der linken oberen Ecke (im folgenden als Dokumentationsmenü bezeichnet) über den Menüpunkt "Dokument. \ Menue-Wechsel" oder über die Tastenkombination Ctrl+M (bzw. Strg+M oder ^M) erreichen.
- ✍✍Befindet man sich im Dokumentationsmenü[12], so ist ein Sprung in das Hauptmenü über den Menüeintrag "System \ Menue-Wechsel" sowie ebenfalls mit der Tastenkombination Ctrl+M (bzw. Strg+M oder ^M) möglich.

Um die zwei Menüebenen kurz zu skizzieren, seien deren wesentliche Inhalte genannt:

- ✍✍Das Hauptmenü beinhaltet vorwiegend Funktionen und Hilfsmittel, um den Simulator als solchen zu benutzen. Hierzu gehören u. a. die Darstellungsmöglichkeiten der Schnittstellen und des Timers, der Registerinhalte und Speicherbereiche sowie die Organisation der Datenein- und Datenausgabe.
- ✍✍Das zweite Dokumentationsmenü liefert über seine verschiedenen Einträge wesentliche Informationen zum System, zur CPU, zu weiterer Hardware und zum Monitor.

Zu den meisten Menü-Oberpunkten existiert der Unterpunkt "Anleitung". Unter dem Oberpunkt "Datei" gibt es sogar drei Anleitungen: "Programm", also dieser Text, den Sie gerade lesen, "Emulator" und "Mikrorechner". Andere Anleitungen, in denen wesentliche Eigenschaften einzelner Teile des Simulators vorgestellt werden, sind als Menüunterpunkte zu den einzelnen Themen aufgeführt.

IV. Die Statusanzeige

Am unteren Bildrand der normalen Benutzeroberfläche befindet sich die sogenannte Statusanzeige. Sie enthält verschiedene Informationen zu wichtigen Vorgängen im Programm. So wird zum Beispiel beim Ablauf eines Programms, welches nicht definierte

[12] In der Praktikumsversion nicht implementiert.

Befehle enthält, die Ursache für die unter diesen Umständen auftretenden Programmstörungen als "*illegal instruction*" in der Statuszeile gemeldet.

In der Statusanzeige wird auch zur Anzeige gebracht, wenn man von der parallelen Schnittstelle (Ports PA bzw. PB) oder der seriellen Schnittstelle (ACIA)[13] Daten einliest oder abspeichert. Dieser Vorgang wird durch ein Diskettensymbol verdeutlicht, wobei bei Datenabspeicherung PA, PB und/oder ACIA links von dem Pfeil steht bzw. stehen, der auf das Diskettensymbol zeigt. Im Falle des Einlesens einer Datei steht der jeweilige Empfänger rechts von dem Diskettensymbol. Ein Pfeil, der von dem Diskettensymbol auf den Empfänger weist, verdeutlicht schon optisch in eindeutiger Weise die Datenübertragungsrichtung.

2. Allgemeine Informationen zur Benutzung des Simulators

Die Bedienung des Simulators soll mit Hilfe der Erläuterungen zu den einzelnen Menüpunkten nähergebracht werden. Um hierbei eine möglichst gute Übersicht zu gewährleisten, werden die jeweiligen Funktionen der einzelnen Menüpunkte in der Folge ihres Auftretens erläutert. Zuerst erfolgt eine Auflistung der Menüpunkte mit einer kurzen Funktionsbeschreibung. Im Anschluß daran wird eine genauere Beschreibung der einzelnen Punkte, soweit nötig, gegeben.

Datei	Hauptmenü-Eintrag für grundsätzliche Systemfunktionen
Anleitung Programm Emulator Mikrorechner	Unter diesen Menüpunkten finden Sie allg. Informationen zu: - der Bedienung der graphischen Oberfläche - der Bedienung des Simulators - der Bedienung des simulierten Mikrorechners
Speicherbereich laden speichern	Diese Menüpunkte erlauben: - ein auf der Festplatte gespeichertes Programm oder einen Datenbereich (über eine Dateiauswahlbox) einzulagern - ein Programm oder einen Datenbereich (über eine Dateiauswahlbox) auf der Festplatte abzuspeichern
Format	Auswahl zwischen zwei Speicherformaten: *.HEX, *.PTI Endung wirkt als Dateifilter für die Auswahlboxen
Übertragung ? Mikrorechner ? Mikrorechner	Übertragung von Daten zwischen Simulator und Praktikumsrechner: - Datenübertragung vom Praktikumsrechner zum Simulator - Datenübertragung vom Simulator zum Praktikumsrechner
Definitionen Schnittstellen ROM\RAM-Erweiterungen	Hier werden Systembedingungen festgelegt: - Zuweisung der simulierten seriellen und parallelen Schnittstellen zu den Hardware-Schnittstellen des PCs - Erweiterung des simulierten Arbeitsspeichers von 16 auf max. 32 kbyte
Ende	Beenden des Simulators (auch über die Tastenkombination "Strg+Q" möglich)

[13] Diese Schnittstellen werden in Kapitel 4 erklärt.

Sie können eine allgemeine Einführung in die Benutzung des Programms und in die wesentlichen Bedienschritte über den Menüpunkt **Datei** \ (Anleitung) **Programm** erhalten. Die in einigen Menüpunkten und auf der Benutzeroberfläche dargestellten Fragezeichen liefern nach Anklicken der linken und unmittelbar danach der rechten Maustaste Hilfetexte zum angewählten Thema.

Speicher	Hauptmenü-Eintrag zur Speicher- und Registerdarstellung
anzeigen	Eröffnet ein Fenster, in dem der Speicherinhalt in hexa-dezimaler Form dargestellt wird und verändert werden kann
Disassembler	Eröffnet ein Fenster, in dem der aktuelle Speicherinhalt disassembliert dargestellt wird
Register	Die Registerinhalte (nach einem Trace-Trap) werden in einem separaten Fenster dargestellt
Assembler	
Anleitung	Anleitung zur Benutzung des Assemblers (vgl. Abschnitt 2.2)
Aufruf	Aufruf des Assemblers, nach Verlassen des Simulators !

Bausteine	Hauptmenü-Eintrag zur Darstellung und Bedienung der Rechnerkomponenten
Timer 6840	Darstellung und Bedienung der 3 Timer des Zeitgeber-/Zählerbausteins (s. Kapitel 5)
Anleitung	Unter diesem Punkt kann eine Anleitung zur Benutzung der Timer aufgerufen werden
Timer #1 Timer #2 Timer #3	Es wird für jeden Timer ein eigenes Fenster geöffnet, in dem der momentane Zustand des Timers dargestellt wird
PIA 6821	Darstellung und Bedienung des Parallelschnittstellen-Bausteins MC6821 (s. Kapitel 4)
Anleitung	Unter diesem Punkt kann eine Anleitung zur Benutzung der zwei Ports des Bausteins MC6821 aufgerufen werden
Port PA Port PB	Für jeden Port wird ein Fenster geöffnet, in dem der Schaltzustand der Ein-/Ausgangsleitungen zu erkennen ist; deren Zustände können auch durch Betätigen der im Fenster dargestellten Schalter verändert werden
ACIA R6551	Darstellung und Bedienung des seriellen Schnittstellen-Bausteins R6551 (s. Kapitel 4)
Anleitung	Unter diesem Punkt kann eine Anleitung zur Benutzung des R6551 aufgerufen werden
ACIA einschalten	In einem separaten Fenster kann die aktuelle Konfiguration des R6551 und die serielle Datenübertragung beobachtet werden

Dokumentation	Hauptmenü-Eintrag zur Bedienung des Dokumentationssystems
Anleitung	Hier werden weitere Informationen zur Benutzung des Dokumentationssystems gegeben
Menue-Wechsel	Wechsel vom Hauptmenü zum Dokumentationsmenü[14]
Tabellen	Anzeige von Tabellen zur Programmierung des simulierten Prozessors (vgl. Kapitel 1):
CPU-Befehlssatz	Tabellen B-1 und B-2 aus Anhang von Kapitel 1
Befehlssatz alphanumerisch	Tabellen C-1 und C-2 aus Anhang von Kapitel 1
Branch-Befehle	Tabelle B-3 aus Anhang von Kapitel 1
relative Adressierung	Tabelle 1.3-5 aus Kapitel 1
Postbyte	Bild A-1 aus Anhang von Kapitel 1
Hilfsroutinen	Tabelle D-1 aus Anhang von Kapitel 1

Kurse	Hauptmenü-Eintrag zur Bearbeitung der im Simulator implementierten Kurse[15]
Anleitung	Anzeige von allgemeinen Informationen zur Benutzung der Kurse:
CPU-Hardware	- Kurs zur Prozessorhardware
CPU-Software	- Kurs zur Prozessorsoftware
Peripherie	- Kurs zur Bedienung der Peripheriebausteine

Tutor	Hauptmenü-Eintrag zur Bearbeitung des im Simulator implementierten Tutors[16]
Anleitung	Anzeige von allg. Informationen zur Benutzung des Tutors:
Hardware	- Tutor zur Rechnerhardware
Software	- Tutor zur Rechnersoftware

[14] In der vorliegenden Praktikumsversion nicht implementiert.
[15] In der vorliegenden Praktikumsversion nicht implementiert.
[16] In der vorliegenden Praktikumsversion nicht implementiert.

3. Bedienung der simulierten Mikrorechner-Tastatur

In der linken oberen Ecke des Rechner-Bildes ist ein großes Fragezeichen dargestellt. Klickt man dieses mit der linken Maustaste an, so werden in der Umgebung der Tasten kleine Fragezeichen sichtbar und in dem Display werden den jeweiligen Stellen der Anzeige die zugehörigen Funktionen zugewiesen. Wünscht man nun eine Erklärung zu den einzelnen Tastenfunktionen, so erscheint nach Drücken der rechten und der linken Maustaste (zuerst die rechte Taste drücken, festhalten, dann Drücken der linken Maustaste) ein separates Hilfefenster mit Erläuterungen zu der Funktion der jeweiligen Taste.

Die Anwahl der Tasten zur Eingabe der Zahlenwerte sowie der mit festen Funktionen vorbelegten Tasten kann auf zwei Arten erfolgen:

?? Die erste Möglichkeit ist die Anwahl der Tasten mit der Maus, wobei hierfür die linke Maustaste zu benutzen ist.

?? Die zweite Möglichkeit besteht in der Anwahl der Tasten über die PC-Tastatur in folgender Weise:

  ? Die Anwahl der Tasten für die Eingabe von hexadezimalen Zahlen sowie der "+" bzw. "-" Tasten und der Funktionstasten F1 bis F4 erfolgt durch direkte Benutzung der gleichnamigen Tasten der PC-Tastatur.

  ? Die Tasten mit fest vorbelegten Funktionen, also die Tasten der Tastengruppen drei und vier (siehe hierzu den entsprechenden Hilfetext), sind über die Tastenkombination <Strg>+<Taste> bzw. <Ctrl>+<Taste> anzuwählen.

Hinweis:
In einigen Bedienfunktionen des Mikrorechners muß eine Taste über längere Zeit "gehalten" werden. Dies wird im Simulator durch Benutzung der Umschalt- (*Shift*-) Taste der PC-Tastatur ermöglicht. Drücken Sie hierbei die Umschalttaste und selektieren Sie mit der Maus die zu haltende Taste.

2.4 Übungen zur Assemblerprogrammierung

Im diesem Abschnitt sollen die in Abschnitt 2.3.5 beschriebenen Unterprogrammtechniken an praktischen Beispielen vertieft und erweitert werden.

"Taschenrechner"

Das Ihnen bekannte Assemblerprogramm "P24-1.ASM" ist nur für eine Zahlenbreite von zwei Ziffern einsetzbar. Wünschenswert wäre ein Programm, das abhängig von einer Konstanten eine veränderbare Ziffernbreite zuläßt. Hierfür muß der Kern des Programms "P24-1.ASM", die BCD-Addition, aus dem Hauptprogramm herausgenommen werden und in einer eigenen Unterprogrammroutine abgelegt werden. Tabelle 2.5-1 zeigt eine mögliche Implementierung dieses Unterprogramms.

Im Unterschied zum ersten Programm, soll nun bei einem Überlauf ein Symbol (kleines Rechteck) in der Anzeige erscheinen. Wird danach die Taste "S" gedrückt, so soll das Programm beendet werden. Im Falle einer gültigen BCD-Ziffer soll das Programm von vorne beginnen.

Tabelle 2.5-1: Unterprogrammroutine UP_ADD zur BCD-Addition

```
;========================
; Unterprogramm UP_ADD: Addition von zwei BCD-Ziffern
; In:   Y: Speicherstelle der ersten BCD-Ziffer (liegt im unteren Nib)
;       B: zweite BCD-Ziffer (liegt im unteren Nibble)
; Out:  A: gleich Null: kein Uebertrag, sonst Uebertrag fuer naechsth.
;       B: korrigierte Ziffer, vorbereitet zur Anzeige
;
        ORG   $0500  ;Beginn Unterprogramm

UP_ADD: ADDB  0,Y    ;Addition der Ziffer
        TFR   B,A    ;
        ANDA  #$F0   ;Test auf Ueberlauf durch vorherige Addition
        BNE   BCDADD ;wenn ja, Korrektur
        TFR   B,A    ;
        CMPA  #$09   ;Test auf gueltige BCD-Ziffer
        BLS   CARRY2 ;wenn ja, Korrektur ueberspringen
BCDADD: ADDB  #$06   ;Korrektur, als Ergebnis gueltige BCD-Ziffer
CARRY2: TFR   B,A    ;
        ANDA  #$F0   ;Vorbereitung zum Test auf Ueberlauf (Hauptp.)
        ANDB  #$0F   ;eventl. Uebertrag der korrig. Ziffer loeschen
        STB   0,Y    ;korrigierte Ziffer abspeichern
        RTS          ;Ruecksprung in das Hauptprogramm
```

Praktische Übung P2.5-1:

Analysieren Sie das in Tabelle 2.5-1 gezeigte Unterprogramm UP_ADD und entwerfen Sie ein Flußdiagramm. Wie reagiert die Routine auf einen eventuellen Übertrag auf die nächsthöhere Ziffer?

Praktische Übung P2.5-2:

Entwerfen Sie ein Flußdiagramm für das zugehörige Hauptprogramm zu dem Unterprogramm UP_ADD von Tabelle 2.5-1. Im Hauptprogramm soll die verarbeitete Ziffernbreite mittels einer Konstantendefinition "einstellbar" sein. Im Falle eines Überlaufes soll ein Symbol (kleines Rechteck) in der Anzeige erscheinen. Wird danach die Taste "S" gedrückt, so soll das Programm beendet werden. Im Falle der Eingabe einer gültigen BCD-Ziffer soll das Programm von vorne beginnen.

Implementieren Sie Ihr Flußdiagramm und testen Sie die Programme auf Ihrem Praktikumsrechner aus.

Werden für das Programm weitere arithmetische Operationen implementiert, so erhält man einen kleinen Taschenrechner für Dezimalzahlen, der mit variabler Ziffernbreite arbeiten kann.

"Vierstufen-Zähler"

Das folgende Beispiel verdeutlicht die verschiedenen Möglichkeiten zur Zwischenspeicherung von Registerinhalten in Unterprogrammen.

Es ist ein Programm zu implementieren, das einen vierstufigen Zähler repräsentiert. Die einzelnen Zählerstufen sollen über Unterprogramme realisiert werden. Der Einfachheit halber soll jede Zählstufe von $00 bis $FF zählen. Die Zähler sollen hierarchisch angeordnet sein, so daß jeder Zähler seinen darunterliegenden nächsten Zähler in jedem Zählschritt einmal aufruft. Dem höchstwertigen Zähler sind die Anzeigestellen S7,S6 zugeordnet, dem niederwertigsten Zähler die Anzeigestellen S1,S0.

Die folgende Tabelle 2.5-2 zeigt das Hauptprogramm des Zählers. Die Register X und B werden während der Abarbeitung des Unterprogramms über Speicherbereiche zwischengespeichert.

Tabelle 2.5-2: Hauptprogramm des Zählers

```
; P25-3.ASM
;
; Zaehlprogramm ueber drei Unterprogramme zur Darstellung der ver-
; schiedenen Moeglichkeiten zur Zwischenspeicherung sowie der ver-
; schachtelten Progr. von Unterprogrammen.
;    Zaehler des Hauptprogramms erscheint in den Anzeigestellen S7,S6
;    Register B und X werden ueber Speicherbereiche gerettet
;
```

```
          ORG    $0400      ;Beginn des Programmbereiches

          JSR    CLRDISP    ;Anzeige loeschen
          LDX    #$0006     ;Vorbereitung der Anzeige (S7,S6)
          LDB    #$FF       ;Zaehler initialisieren
NEXT0:    INCB              ;Zaehler inkrementieren
          JSR    SHOWB7SG   ;Zaehlerstand anzeigen
          STB    MEM1       ;Zaehler und Anzeigeposition ueber
          STX    MEM2       ;Speicherbereich retten
          JSR    UP1        ;Aufruf des ersten UP
          LDB    MEM1       ;Zaehler und Anzeigeposition einlesen
          LDX    MEM2       ;
          CMPB   #$FF       ;Abbruchbedingung erfuellt?
          BNE    NEXT0      ;wenn nein, zurueck zu NEXT0
          SWI               ;Programmende

MEM1      EQU   $0500  ;Speicherbereiche zum Zwischenspeichern
MEM2      EQU   $0502  ;von Daten
CLRDISP   EQU   $F110  ;Anzeige des Praktikumsrechners loeschen
SHOWB7SG  EQU   $F120  ;Anzeige der Hexzahl, die in B steht; Pos. in X
```

Praktische Übung P2.5-3:

Implementieren Sie für obiges Hauptprogramm die drei geforderten Unterprogramme. Im ersten Unterprogramm sollen die Register X und B über den Stack gerettet werden. In dem zweiten Unterprogramm sollen sie über Prozessorregister zwischengespeichert werden.

Testen Sie das Programm auf Ihrem Praktikumsrechner aus.

"Interrupt-Routinen"

Interrupt-Routinen sind eine spezielle Art von Unterprogrammen, die ausschließlich durch Hardware- und Software-Interrupts aktiviert werden. An dieser Stelle soll die grundsätzliche Struktur von Interruptroutinen besprochen werden.

Interruptbehandlung durch den Monitor:

Wird ein Interrupt vom 6809-Prozessor erkannt, so rettet er abhängig vom Interrupttyp einige (mindestens PC und CC)[17] Register oder alle Prozessorregister auf den Systemstack und ruft die Interruptbehandlungsroutine des Monitors auf. Diese ermittelt die Interruptquelle und setzt den Programmzähler auf die Einsprungadresse der jeweiligen Behandlungsroutine der Interruptquelle. Die gültige Einsprungadresse lädt die Monitorroutine aus der sogenannten Interruptvektortabelle, in der für alle Interruptquellen eines Systems die entsprechenden Einsprungadressen hinterlegt sind. Beispielsweise ist im Falle des Praktikumsrechners die Interrupt-Vektoradresse für den SWI3 (Software-Interrupt3) die 16bit-Adresse $004C, $004D.

[17] Vgl. Kapitel 1

Struktur einer Interrupt-Routine

Eine Interruptroutine ist ein in sich abgeschlossenes Programm, daß mindestens einen RTI-Befehl (*return from interrupt*) besitzen muß. Das Hauptprogramm integriert die Interruptroutine in das Gesamtprogramm, in dem es die Einsprungadresse der Interruptroutine an der entsprechenden Adresse in der Interruptvektortabelle hinterlegt. Tritt im nachfolgenden Programmfluß ein Interrupt auf, so wird durch die Monitorroutinen automatisch zu der zugehörigen Interruptbehandlungsroutine verzweigt. Am Ende des Kontrollflußes der Interruptroutine wird durch den RTI-Befehl die Programmflußkontrolle wieder an das verdrängte Programm zurückgegeben und Stackinhalte zurückgeschrieben.

Wie Sie bereits aus Kapitel 1 wissen, unterstützt der 6809-Prozessor drei Hardware-Interruptleitungen (NMI, FIRQ, IRQ) sowie drei Software-Interrupts (SWI1, SWI2, SWI3). Da in den Kapiteln 4 und 5 die Interruptmechanismen für die verschieden Peripheriekomponenten des Praktikumsrechners beschrieben werden, beschränken wir uns im folgenden Beispiel auf den Software-Interrupt SWI3.

Zu beachten ist, daß der 6809-Prozessor bei allen drei Software-Interrupts alle Register auf den Stack rettet. Werden an die Interruptroutine Übergabeparameter mittels Register übertragen, so liegen diese beim Einsprung in die Interruptroutine auf dem Stack. Eine oft genutzte Möglichkeit diese Werte in den Prozessor zu laden ist die Benutzung der Stack-relativen Adressierung. Hierbei werden die zu lesenden Werte relativ zum aktuellen Stackpointer adressiert und geladen. Rückgabeparameter an das aufrufende Programm können in der gleichen Weise (Stack-relativ !) auf den Stack geschrieben werden. Bei Beendigung der Interruptroutine werden diese durch den RTI-Befehl automatisch in die zugehörigen Prozessor-Register übertragen. Die Reihenfolge in der die einzelnen Registerwerte auf den Stack gelegt werden, ist in Kapitel 1 bei der Beschreibung des RTI-Befehls angegeben.

Praktische Übung P2.5-4:

Schreiben Sie ein Programm, daß die Betriebssystemschnittstelle eines PCs auf dem Praktikumsrechner beispielhaft nachbildet. Hierzu werden alle Monitorroutinen nicht mehr direkt angesprochen, sondern durch ein sogenanntes Gate[18] (Interruptroutine mit SWI3) aktiviert. Dem Gate wird als Übergabeparameter die Nummer der zu aktivierenden Monitorroutine nach folgender Tabelle in Register X übergeben:

[18] Durch die Implementierung von (Call-) Gates ist es möglich, unberechtigte Betriebssystemzugriffe durch Anwendungsprogramme zu verhindern. Moderne Betriebssysteme arbeiten hierzu in völlig gekapselten Adreßbereichen und Ebenen (z.B. bei Intel-Prozessoren im Protected Mode Level 0).

Monitorroutine	codierte Nummer
SHOWT7SG, Anzeigestelle S0	0
SHOWT7SG, Anzeigestelle S1	1
SHOWT7SG, Anzeigestelle S2	2
SHOWT7SG, Anzeigestelle S3	3
SHOWT7SG, Anzeigestelle S4	4
SHOWT7SG, Anzeigestelle S5	5
SHOWT7SG, Anzeigestelle S6	6
SHOWT7SG, Anzeigestelle S7	7
CLRDISP	8
HALTKEY	9

Im Register B soll abhängig von der gewünschten Monitorroutine der zu verarbeitende Wert übergeben werden, oder der zu erwartende Wert zurückgegeben werden (im Falle von HALTKEY).

Das Hauptprogramm soll einen Ziffernzähler für die Ziffern "0" und "1" realisieren. Dazu sollen die eingelesenen Ziffern in der Anzeigestelle S0 dargestellt werden und die Anzahl der bereits eingegebenen Ziffern in den Anzeigestellen S7 (Ziffer 0) und S6 (Ziffer 1) ausgegeben werden. Bei Betätigen der "+"-Taste soll das Programm die beiden Zähler zurücksetzen und erneut die Auftrittshäufigkeit der beiden Ziffern "0" und "1" zählen und darstellen. Andere Tasten sollen keine Auswirkungen haben.

Hinweis:

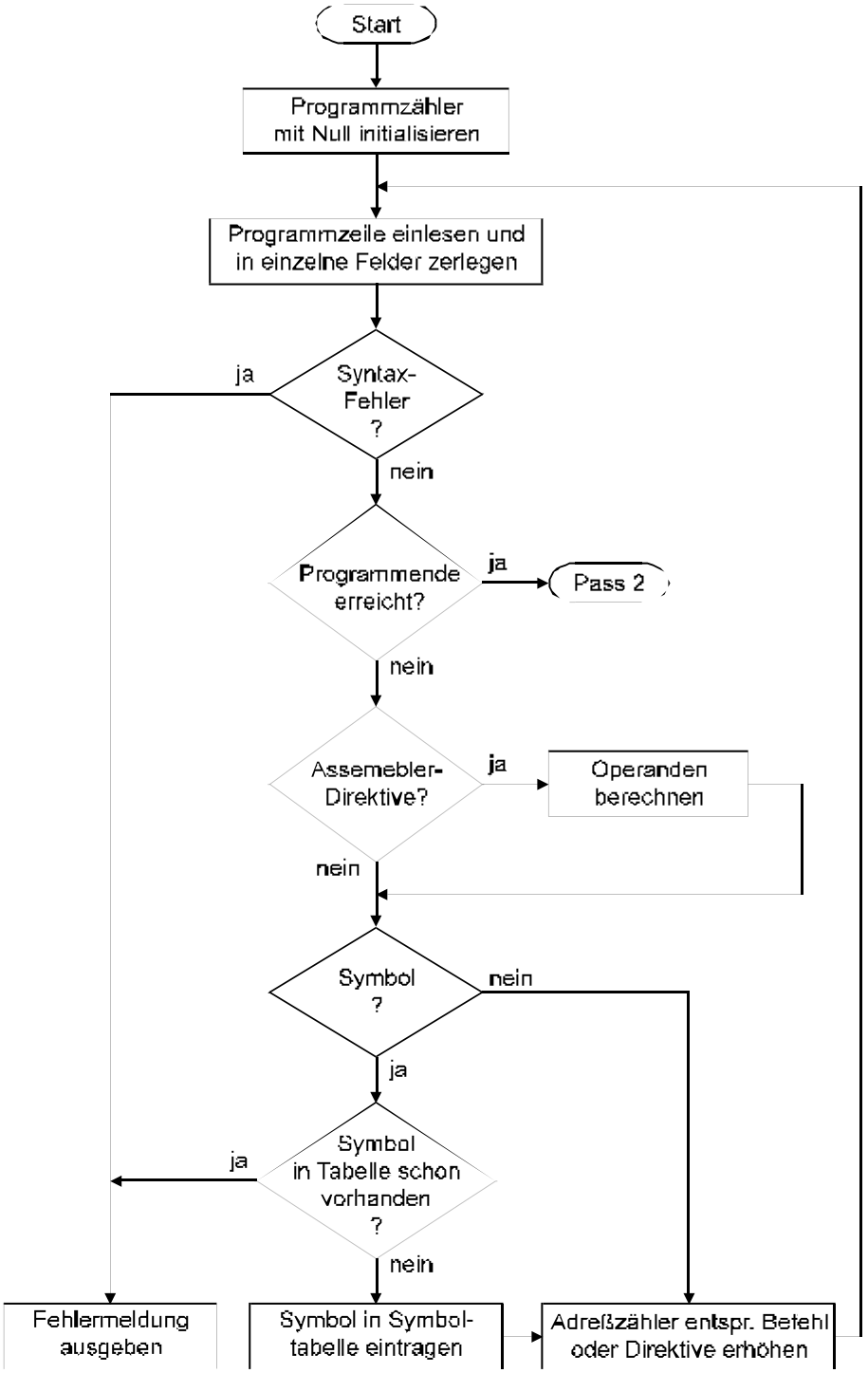
> Die Aktivierung von Monitorroutinen durch das Hauptprogramm soll nur durch das obengenannte Gate mit dem Software-Interrupt 3 stattfinden.

> Die Interrupt-Vektoradresse für den SWI3 (Software-Interrupt3) ist die 16bit-Adresse $004C, $004D.

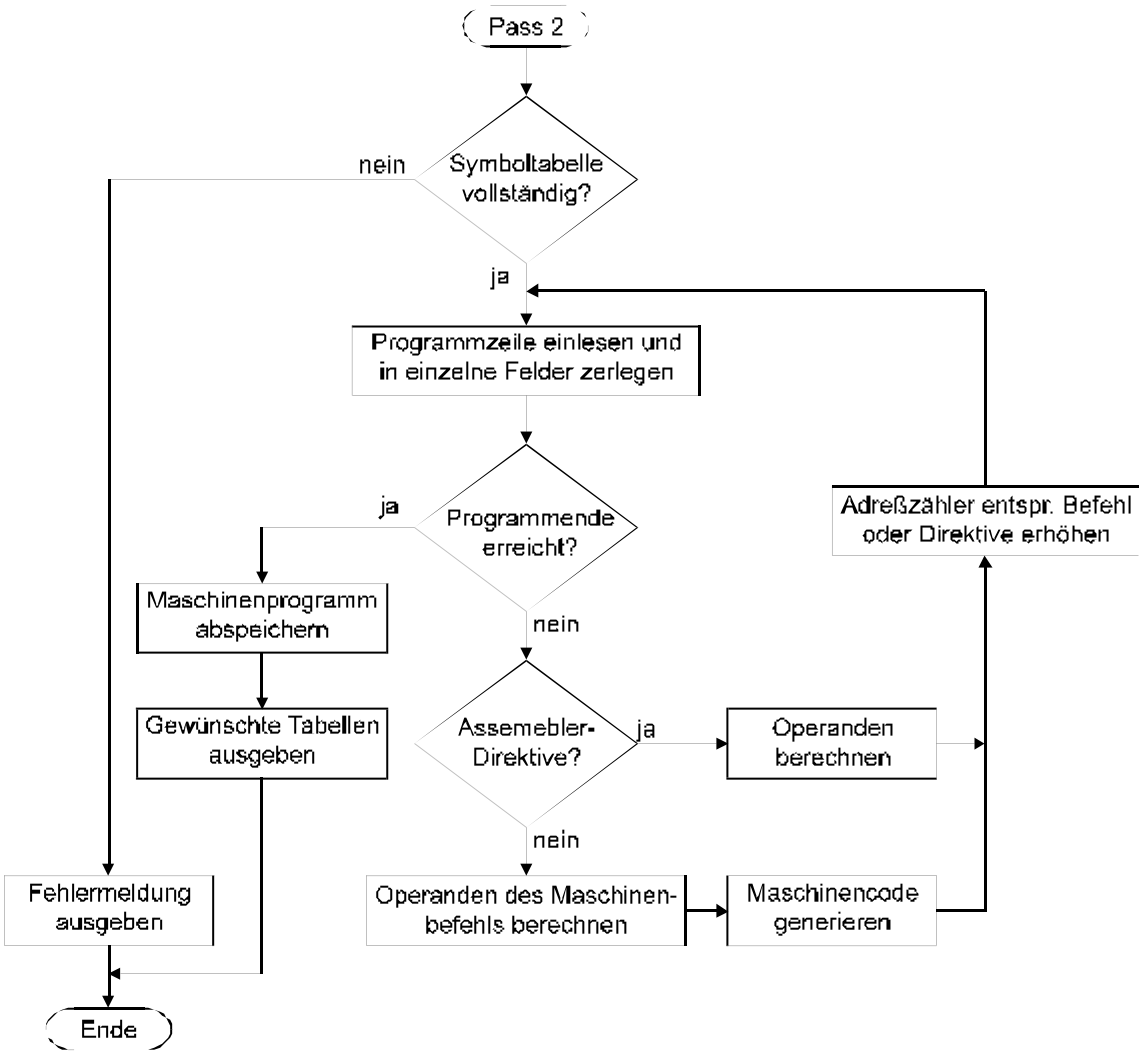
Lösungsvorschläge zu den Praktischen Übungen

Zu P2.3-1:

Für *pass 1* des Assemblers ergibt sich als Flußdiagramm:



Für *pass 2* des Assemblers ergibt sich als Flußdiagramm:



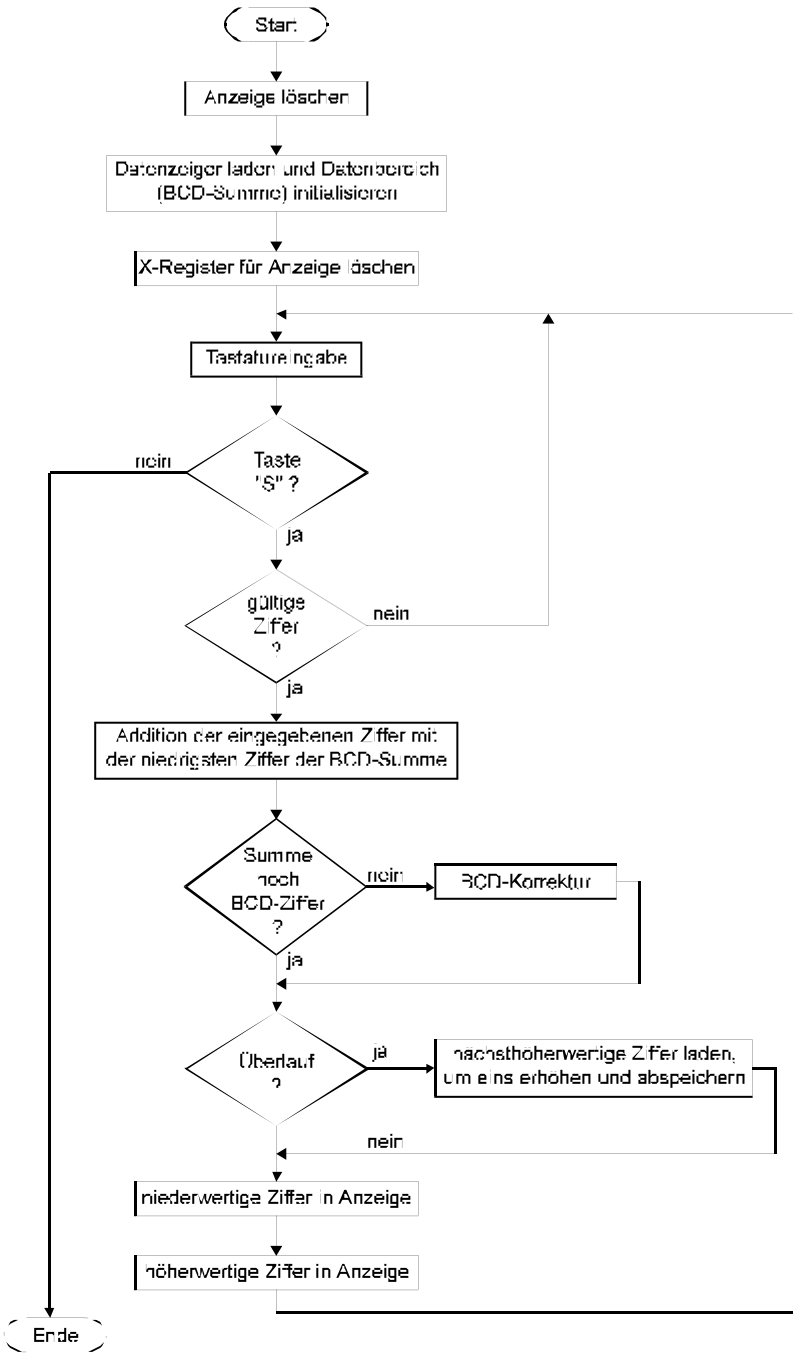
Zu P2.4-1:

1. Wird das Programm mit der Option "-L" assembliert, ergibt sich das folgende Listing. Ganz links steht die Zeilennummer des Listings. Rechts daneben steht die Speicheradresse mit dem OP-Code. Weiter rechts folgt das Quellcodeprogramm.

```
0001                        ************************************************
0002                         * P24-1.ASM
0003                         *
0004                         * Programm zur Addition von BCD-Zahlen ohne Verwen
0005                         * des DAA-Befehls
0006                         *
0007 0400                    ORG   $0400   ;Beginn des Programmbereichs
0008
0009 0400 bd f1 10           JSR   CLRDISP ;Anzeige loeschen
0010 0403 10 8e 06 00        LDY   #$0600  ;Datenzeiger laden
0011 0407 6f a4              CLR   0,Y     ;Datenbereich mit Null initialisieren
0012 0409 6f 21              CLR   1,Y     ;
0013 040b 8e 00 00  START:   LDX   #$0000  ;X loeschen
0014 040e bd f1 43           JSR   HALTKEY ;Zeichen von Tastatur lesen
0015 0411 c1 86              CMPB  #$86    ;Vergleich, auf Ende der Eingabe "S"
0016 0413 27 30              BEQ   ENDE    ;
0017 0415 c1 09              CMPB  #$09    ;Test der Eingabe auf gueltige Ziffer
0018 0417 22 f2              BHI   START   ;bei ungueltiger Ziffer zurueck
0019 0419 eb a4              ADDB  0,Y     ;Addition der Eingabe mit LSB der Sum
0020 041b 1f 98              TFR   B,A     ;
0021 041d 84 f0              ANDA  #$F0    ;Test auf Ueberlauf durch vorherige A
0022 041f 26 06              BNE   BCDADD  ;wenn ja, Korrektur
0023 0421 1f 98              TFR   B,A     ;
0024 0423 81 09              CMPA  #$09    ;Test auf gueltige BCD-Ziffer
0025 0425 23 02              BLS   CARRY2  ;wenn nein, Korrektur
0026 0427 cb 06     BCDADD:  ADDB  #$06    ;Korrektur, als Ergebnis gueltige BCD
0027 0429 1f 98     CARRY2:  TFR   B,A     ;
0028 042b 84 f0              ANDA  #$F0    ;
0029 042d 27 06              BEQ   SHOW    ;Test auf Ueberlauf durch vorherige A
0030 042f 86 01              LDA   #$01    ;
0031 0431 ab 21              ADDA  1,Y     ;naechsthoeherwertige Ziffer um Eins
0032 0433 a7 21              STA   1,Y     ;naechsthoeherwertige Ziffer abspeich
0033 0435 c4 0f     SHOW:    ANDB  #$0F    ;eventl. Uebertrag der korrigierten Z
0034 0437 e7 a4              STB   0,Y     ;korrigierte Ziffer abspeichern
0035 0439 bd f1 1c           JSR   SHOWT7SG;korrigierte Ziffer in Anzeige
0036 043c e6 21              LDB   1,Y     ;naechsthoeherwertige Ziffer laden
0037 043e 30 01              LEAX  1,X     ;Anzeigestelle nach links verschieben
0038 0440 bd f1 1c           JSR   SHOWT7SG;naechsthoeherwertige Ziffer in Anz
0039 0443 20 c6              BRA   START   ;zurueck zur naechsten Eingabe
0040 0445 3f        ENDE:    SWI1
0041
0042 f110           CLRDISP  EQU  $F110   ;Loeschen der Anzeige, In:-,
0043 f11c           SHOWT7SG EQU  $F11C   ;unteres Nibble von B in Anzeige
0044 f143           HALTKEY  EQU  $F143   ;Lesen der Tastatur mit Warten,
```

```
Number of errors 0
```

2. Das Programm addiert die eingegebene BCD-Ziffer mit der niederstwertigen Ziffer der Summe. Die BCD-Addition wird ohne den DAA-Befehl durchgeführt. Überschreitet die Summe den Wert "99", so wird hexadezimal in der zweiten Stelle weitergezählt. D.h. eine Überprüfung auf eine gültige BCD-Ziffer in der zweiten Stelle findet nicht statt. Beispielsweise ergibt 99 + 1 = A0.
   Das folgende Flußdiagramm zeigt die Problematik:



Zu P2.4-2:

1. In dem Programm sind drei syntaktische Fehler vorhanden:
   Für Zeile 29 gibt der Assembler die Fehlermeldung "29: *Branch out of Range*". Hier fehlt das Sprungziel "SHOW" im Verzweigungsbefehl.
   Für Zeile 34 meldet er den Fehler "34: Unrecognized Mnemonic". Es fehlt beim *Store*-Befehl das Quellregister "B".
   Für Zeile 37 meldet der Assembler: "37: Illegal Register for Indexed". In diesem Fall ist bei den Operanden des "LEAX"-Befehls das erste Komma zuviel.
2. Der logische Fehler des Programms besteht aus einer falschen Wahl eines Verzweigungsbefehls. Statt einen vorzeichenlosen Verzweigungsbefehl zu benutzen, wurde in Zeile 18 der "BGT"-Befehl benutzt. Hierdurch werden die Tasten des Funktionsblockes akzeptiert, so daß die Addition zu falschen Ergebnissen führt.

Zu P2.4-3:

Der "DS9" erzeugt aus der Datei "P24-1.HEX" das folgende Listing:

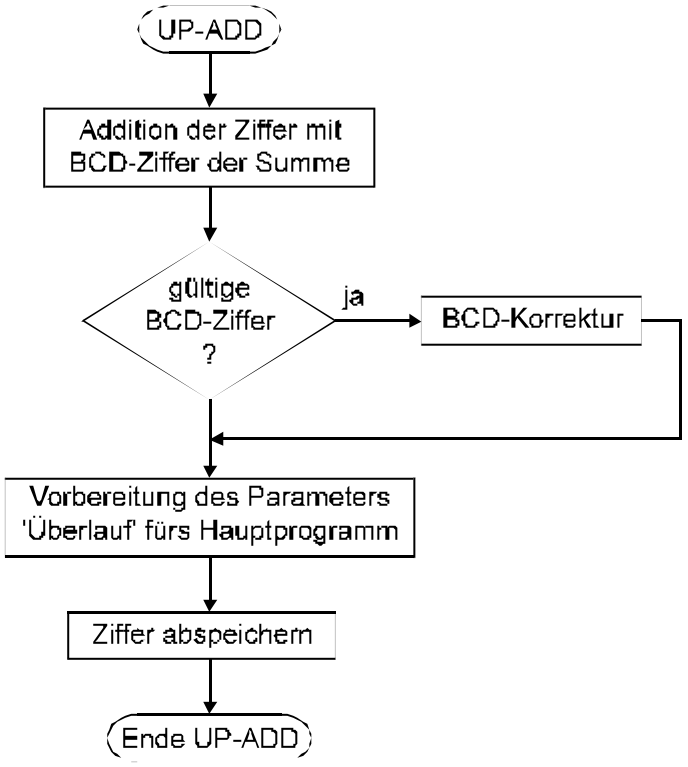
```
;                    ***   6809 - Disassembler Ver. 1.3   ***


0400: BD F1 10         JSR    $F110
0403: 10 8E 06 00      LDY    #$0600
0407: 6F A4            CLR    ,Y
0409: 6F 21            CLR    $01,Y
040B: 8E 00 00         LDX    #$0000
040E: BD F1 43         JSR    $F143
0411: C1 86            CMPB   #$86
0413: 27 30            BEQ    $30   ;$0445
0415: C1 09            CMPB   #$09
0417: 22 F2            BHI    $F2   ;$040B
0419: EB A4            ADDB   ,Y
041B: 1F 98            TFR    B,A
041D: 84 F0            ANDA   #$F0
041F: 26 06            BNE    $06   ;$0427
0421: 1F 98            TFR    B,A
0423: 81 09            CMPA   #$09
0425: 23 02            BLS    $02   ;$0429
0427: CB 06            ADDB   #$06
0429: 1F 98            TFR    B,A
042B: 84 F0            ANDA   #$F0
042D: 27 06            BEQ    $06   ;$0435
042F: 86 01            LDA    #$01
0431: AB 21            ADDA   $01,Y
0433: A7 21            STA    $01,Y
0435: C4 0F            ANDB   #$0F
0437: E7 A4            STB    ,Y
```

```
0439: BD F1 1C          JSR    $F11C
043C: E6 21             LDB    $01,Y
043E: 30 01             LEAX   $01,X
0440: BD F1 1C          JSR    $F11C
0443: 20 C6             BRA    $C6   ;$040B
0445: 3F                SWI(1)
```

Symbolische Adressen für Sprungziele und Unterprogramme können bei einer Disassemblierung nicht berücksichtigt werden. Ein "Nach-Disassemblieren" von Hand ist notwendig. Der generierte Quellcode stimmt ansonsten mit dem "Original"-Code überein.
Beachten Sie bitte, daß der "DS9" bei Verzweigungen nicht nur den Offset zum Sprungziel angibt, sondern auch die absolute Adresse des Sprungzieles.

Zu P2.5-1

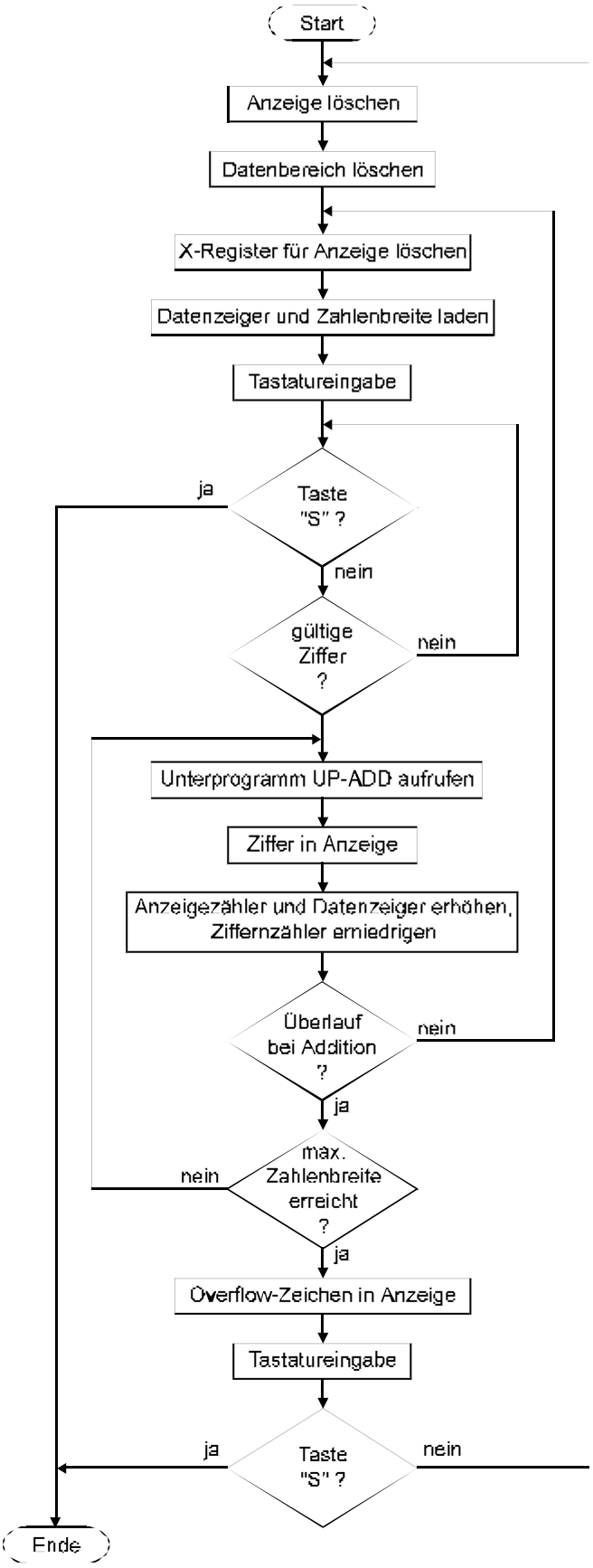
Für das Unterprogramm von Tabelle 2.4-6 ergibt sich das folgende Flußdiagramm:



Im Falle eines Übertrages auf die nächsthöhere Ziffer wird das Register A auf einen Wert ungleich Null gesetzt und dem Hauptprogramm übergeben. Die berechnete Ziffer wird abgespeichert. Die Übergabe der zugehörigen Speicheradresse erfolgt durch das Register Y.

Zu P2.5-2:

Das Flußdiagramm für das Hauptprogramm ist auf der folgenden Seite abgebildet. Erzeugt die BCD-Addition einen Übertrag, so wird das Unterprogramm nochmals aufgerufen, um den Übertrag zur nächsthöheren Ziffer der Summe zu addieren. Im *worst case* muß über die gesamte Zahlenbreite eine Übertragsaddition erfolgen. Wird ein Überlauf erkannt, so wird das Overflow-Symbol zur Anzeige gebracht. Wird danach die "S"-Taste gedrückt, so wird das Programm beendet, ansonsten bei Eingabe einer gültigen Ziffer mit einer neuen Addition begonnen.



Eine mögliche Implementierung für das Hauptprogramm folgt auf der nächsten Seite.

```
;*****************************************************************************
; P25-1.ASM
;
; Programm zur Addition von BCD-Zahlen mit maximal 7 Ziffern ohne Verwendung
; des 6809-Befehls DAA. Der Befehl wird durch das Unterprogramm UP_ADD er-
; setzt.
; Erweiterung der Wortbreite ohne Probleme moeglich
; (Praktikumsrechner hat allerdings max. 8 Stellen).
; Bei Ueberlauf Aktivierung der Anzeige.
; Abbruch der Eingabe mit der Taste "S" des Praktikumsrechners.
;
        ORG    $0400       ;Beginn des Programmbereiches

NEW:    JSR    CLRDISP     ;Anzeige loeschen
        CLRA               ;A loeschen
        CLRB               ;B loeschen
        STD    >$0600      ;Datenbereich $0600 - $0607
        STD    >$0602      ;mit Null initialisieren
        STD    >$0604      ;
        STD    >$0606      ;
START:  LDX    #$0000      ;X loeschen
        LDY    #$0600      ;Datenzeiger laden
        LDU    #ZWIDTH     ;Zahlenbreite laden
        JSR    HALTKEY     ;Zeichen von Tastatur lesen
        CMPB   #$86        ;Vergleich, auf Ende der Eingabe (Taste "S")
        BEQ    END         ;
        CMPB   #$09        ;Test der Eingabe auf gueltige Ziffer (0..9)
        BHI    START       ;bei ungueltiger Ziffer zurueck zur Eingabe
CARRY:  JSR    UP_ADD      ;Aufruf UP zur Addition von zwei BCD-Ziffern
        JSR    SHOWT7SG    ;korrigierte Ziffer in Anzeige
        LEAX   1,X         ;X:=X+1
        LEAY   1,Y         ;Y:=Y+1
        LEAU   -1,U        ;U:=U-1
        LDB    #$01        ;Reg. B fuer Addition des Uebertragsbits auf 1
        CMPA   #$00        ;Vergleich auf Ueberlauf
        BEQ    START       ;wenn nein, zurueck zur naechsten Eingabe
        CMPU   #$0000      ;Vergleich auf max. Zahlenbreite
        BEQ    OVERFL      ;Overflow, Uebertrag ignorieren; zur OV-Behandlung
        BRA    CARRY       ;Uebertrag addieren und anzeigen
OVERFL: LDA    #$5C        ;
        JSR    SHOWA       ;Overflow-Symbol in Anzeigestelle S7
        JSR    HALTKEY     ;Zeichen einlesen
        CMPB   #$86        ;Programmende?
        BNE    NEW         ;nein, neuer Lauf
END:    SWI                ;Programmende

ZWIDTH  EQU   $0007 ;Maximale Ziffernanzahl einer zu speichernden BCD-Zahl

;-----------------------------------------------------------------------------
; Hier beginnen die Definitionen der Monitorroutinen und der Adressen der
; Hardware-Register.
;                        Monitorroutinen
```

```
;
CLRDISP    EQU  $F110      ;Loeschen der Anzeige, In:-, Out:-
SHOWT7SG   EQU  $F11C      ;unteres Nibble von B in Siebensegmentcode wandeln
                           ;und anzeigen, Position in X
SHOWA      EQU  $F113      ;Direkte Ansteuerung der Segmente der Anzeige-
                           ;stelle durch das Byte in A, Position in X
HALTKEY    EQU  $F143      ;Lesen der Tastatur mit Warten, In:-, Out:B
```

Zu P2.5-3:

Die folgenden drei Unterprogramme repräsentieren die geforderten Zähler. In "UP1" werden die Register über den Stack gerettet, während in "UP2" nicht benötigte Register verwendet werden.

```
;=====================================================================
;Unterprogramm  UP1: Zaehler, für die Anzeigestellen S5,S4
;                Register B und X werden ueber den Stack gerettet.
;
UP1:    LDX    #$0004     ;Anzeigeposition laden (S5,S4)
        LDB    #$FF       ;Zaehler initialisieren
NEXT1:  INCB              ;Zaehler inkrementieren
        JSR    SHOWB7SG   ;Zaehlerstand anzeigen
        PSHS   B          ;Zaehler und Anzeigeposition ueber
        PSHS   X          ;den Stack retten
        JSR    UP2        ;Aufruf des zweiten UP
        PULS   X          ;Zaehler und Anzeigeposition einlesen
        PULS   B          ;
        CMPB   #$FF       ;Abbruchbedingung erfuellt?
        BNE    NEXT1      ;wenn nein, zurueck zu NEXT1
        RTS               ;Ende Unterprogramm

;=====================================================================
;Unterprogramm  UP2: Zaehler, für die Anzeigestellen S3,S2
;                Register B und X werden ueber Register gerettet.
;
UP2:    LDX    #$0002     ;Anzeigeposition laden (S3,S2)
        LDB    #$FF       ;Zaehler initialisieren
NEXT2:  INCB              ;Zaehler inkrementieren
        JSR    SHOWB7SG   ;Zaehlerstand anzeigen
        TFR    B,A        ;Zaehler und Anzeigeposition ueber
        TFR    X,Y        ;andere Prozessorregister retten
        JSR    UP3        ;Aufruf des dritten (letzten) UP
        TFR    A,B        ;Zaehler und Anzeigeposition wieder
        TFR    Y,X        ;zurueckschreiben
        CMPB   #$FF       ;Abbruchbedingung erfuellt?
        BNE    NEXT2      ;wenn nein, zurueck zu NEXT2
        RTS               ;Ende Unterprogramm

;=====================================================================
```

```
;Unterprogramm  UP3: Zaehler, für die Anzeigestellen S1,S0
;
UP3:    LDX    #$0000     ;Anzeigeposition laden (S1,S0)
        LDB    #$FF       ;Zaehler initialisieren
NEXT3:  INCB              ;Zaehler inkrementieren
        JSR    SHOWB7SG   ;Zaehlerstand anzeigen
        CMPB   #$FF       ;Abbruchbedingung erfuellt?
        BNE    NEXT3      ;wenn nein, zurueck zu NEXT3
        RTS               ;Ende Unterprogramm
```

Im Unterprogramm "UP2" werden je zwei Stackoperationen benutzt, um die Registerinhalte auf den Stack zu schreiben bzw. einzulesen. Es ist auch möglich, die Stackoperationen mit jeweils einem Befehl auszuführen. Hierzu ist entweder der numerische Wert "immediate" anzugeben; oder die einzelnen Register durch Kommata getrennt aufzuzählen.

Zu P2.5-4:

Eine mögliche Implementierung des Hauptprogramms und der Interruptroutine zeigt das folgende Listing:

```
;*****************************************************************
; P25-4.ASM
;
; Ziffern-Zaehlprogramm mittels einer Software-Interruptroutine.
; Software-Interrupt wird als Monitor- (Betriebssystem-) Gate benutzt,
; um Funktionen des Monitors zu benutzen. Eingegebener Wert wird in S0
; dargestellt, Anzahl der eingebenen Werte fuer "0" in S7, fuer "1" in
; S6. Uebergabeparameter: X-Reg.: call-by-value, B-Reg.: call-by-refr.
; Parameter werden in der Interruptroutine Stack-relativ geladen.
;
        ORG  $0400   ;Beginn des Programmbereiches

        LDX  #MRGATE ;Interruptroutine in der Vektortabelle
        STX  $004C   ;an Adresse $004C, $004D eintragen
NEW:    LDX  #$0008  ;Uebergabeparameter nach A (Anzeige loeschen)
        SWI3         ;Software-Interrupt ausfuehren
        CLRB         ;Zaehler in den Speicherstellen
        LDX  #$0006  ;$06 und $07 in der Zero-Page
        STB  ,X      ;mit Null initialisieren
        SWI3         ;und jeweiliger Wert (0)
        LEAX 1,X     ;in den Anzeigestellen
        STB  ,X      ;S6 und S7 anzeigen
        SWI3         ;
INPUT:  LDX  #$0009  ;Uebergabeparameter fuer Tastatur nach X und
        SWI3           ;zur Interruptroutine verzweigen
        CMPB #$01    ;Wurde Taste "1" betaetigt?
        BNE  INPUT0  ;Wenn nein, vielleicht "0"
```

```
        LDX  #$0000  ;Wert in Anzeigestelle
        SWI3           ;S0 anzeigen
        LEAX 6,X     ;Zaehler fuer den
        LDB  ,X        ;Wert "1" laden,
        INCB           ;inkrementieren und wieder
        STB  ,X        ;abspeichern
        SWI3           ;In S6 anzeigen
        BRA  INPUT   ;zurueck zur naechsten Eingabe
INPUT0: CMPB #$00    ;Vergleich auf Taste "0"
        BNE  INPUT+  ;wenn nein, vielleicht "+"-Taste
        LDX  #$0000  ;Wert in Anzeigestelle
        SWI3           ;S0 anzeigen
        LEAX 7,X     ;Zaehler fuer "0"
        LDB  ,X        ;laden,
        INCB           ;inkrementieren und
        STB  ,X        ;abspeichern
        SWI3           ;In S7 anzeigen
        BRA  INPUT   ;zurueck zur naechsten Eingabe
INPUT+: CMPB #$80    ;Vergleich auf "+"-Taste
        BEQ  NEW     ;Wenn ja, Zaehler neu initialisieren
        BRA  INPUT   ;ungueltige Taste, zurueck zur Eingabe



; Interruptroutine
MRGATE: LDX  4,S     ;X-Register vom Stack holen (Stack-relativ)
        CMPX #$0008  ;Routine fuer Anzeige gefordert ?
        BHS  DISP    ;Wenn nein, vielleicht Anzeige loeschen
        JSR  SHOWT7SG;Anzeige des unteren Nibbles aus B
        BRA  SUBEND  ;zum RTI
DISP:   CMPX #$0008  ;Routine zum loeschen der Anzeige gefordert?
        BNE  TAST    ;Wenn nein, vielleicht Routine zur Tastatureing.
        JSR  CLRDISP ;Anzeige loeschen
        BRA  SUBEND  ;zum RTI
TAST:   CMPX #$0009  ;Routine zur Tastatureingabe gefordert?
        BNE  SUBEND  ;Wenn nein, falscher Uebergabeparameter; zurueck
        JSR  HALTKEY ;auf Eingabe von Tastatur warten
        STB  2,S     ;B-Register auf den Stack legen (Stack relativ)
SUBEND: RTI          ;Rueecksprung zum Hauptprogramm


CLRDISP   EQU  $F110  ;Loeschen der Anzeige, In:-, Out:-
SHOWT7SG  EQU  $F11C  ;unteres Nibble von B in Anzeige, Position in X
HALTKEY   EQU  $F143  ;Lesen der Tastatur mit Warten, In:-, Out:B
```

Es hat sich als gute Programmierpraxis erwiesen, in Unterprogrammroutinen, und insbesondere in Interruptroutinen, nur eine Aussprungstelle zu programmieren. In der obigen Interruptroutine wäre es möglich nach den einzelnen Abfragen nicht mit einem

BRA-Befehl zum Ende der Routine zu verzweigen und erst dann einen RTI-Befehl durchzuführen, sondern statt eines BRA-Befehls sofort einen RTI-Befehl zu programmieren. In Bezug auf ein gutes Software-Engineering und einfache Wartungs- und Pflegeoptionen hat sich herausgestellt, daß sich eine Beschränkung auf eine einzige Aussprungstelle positiv auf die Programmierfehlerrate auswirkt.