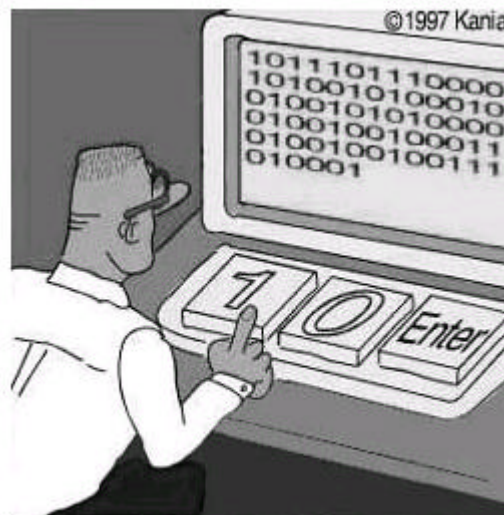


Mikroprozessorenkurs



Real programmers code in binary.

Jann Philipp Oesch, El. Ing. HTL
OESCH.ORG – Schönbühl und Leipzig
Mail: jann@oesch.org

Inhaltsverzeichnis

1	Zahlensysteme	3
1.1	Zahlendarstellungen	3
1.2	Grundrechenarten	4
1.3	Zahleumwandlung Hex - Dez - Bin	5
1.4	Logische Operationen	7
1.5	Zahlenbereich, negative Zahlen und Carry-Bit	8
2	Aufbau eines Computersystems	9
2.1	Central Prozessor Unit (CPU)	9
2.2	Der Speicher (Memory)	10
2.3	Die Peripherie und I/O-Ports	13
2.4	Der Prozessorbus	13
2.5	Chip Select Logik	14
3	Der MC68332 Mikrocontroller von Motorola	16
3.1	Interner Aufbau	16
3.2	Die CPU32	18
3.3	Die Register	19
3.4	Das Status Register (SR) und das Conditions Code Register (CCR)	20
3.5	Der Adressraum	21
4	Inbetriebnahme des Übungsboards	22
4.1	Das Testboard der TEKO Bern	22
4.2	Adressraumeinteilung des Testboards	22
4.3	New Background Debugger NBD32 von OESCH.ORG	23
4.4	Assembler AS32 von Motorola	30
5	Programmierung	34
5.1	Befehlssatz	34
5.2	Adressierungsarten	36
5.3	Standard - Assemblerkonstrukte (Schlaufen, Bedingungen)	40
6	Der Stack	42
6.1	Der Stack als Zwischenspeicher	42
6.2	Subprogramme	43
7	Interrupts und Exceptions	45
7.1	Beispiel interruptgesteuerte Uhr	45
7.2	Programm „Interruptgesteuerte Uhr“	48
8	Anhang	51
8.1	Literaturverzeichnis	51

1 Zahlensysteme

1.1 Zahlendarstellungen

Dezimale Zahlen

Kennung in Schreibweise: keine oder d (234d)

Anzahl Ziffern: 10 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Gewichtung der Stelle: 10^{n-1} , wobei n der Stelle entspricht (von rechts nach links)

Beispiel: $285 = 2 \cdot 10^2 + 8 \cdot 10^1 + 5 \cdot 10^0 = 200 + 80 + 5 = 285$

Binäre Zahlen

Kennung in Schreibweise: % oder b (%1010; 1010b)

Anzahl Ziffern: 2 {0, 1}

Gewichtung der Stelle: 2^{n-1} , wobei n der Stelle entspricht (von rechts nach links)

Beispiel: $1010b = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 0 + 2 + 0 = 10d$

Hexadezimale Zahlen

Kennung in Schreibweise: \$ oder h (\$5A3; 5A3h)

Anzahl Ziffern: 16 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

Gewichtung der Stelle: 16^{n-1} , wobei n der Stelle entspricht (von rechts nach links)

Beispiel: $5A3h = 5 \cdot 16^2 + 10 \cdot 16^1 + 3 \cdot 16^0 = 1280 + 160 + 3 = 1443d$

1.2 Grundrechenarten

Addition

57	1011001011b	44h
+ 4	+ 01100101b	+ 7h
---	-----	----
61	1100110000b	4Bh
===	=====	====

Subtraktion

57	1011001011b	3Eh
- 4	- 01100101b	-1Ah
---	-----	----
53	1001100110b	24h
===	=====	====

Multiplikation

22*13	101b*1010b	12h*3Ah
-----	-----	-----
26	1010	74
26	0	3A
-----	1010	-----
286	-----	414h
=====	110010b	=====
	=====	

Division

168:12=14	11000101b:101b=100111b	414h:3Ah=12h
-12 ==	101 =====	3A ==
---	---	--
48	1001	74
48	101	74
--	---	--
0	1000	0
	101	

	111	
	101	

	10 --> Rest 10b	
	=====	

1.3 Zahlenumwandlung Hex - Dez - Bin

Jede Zahl eines Zahlensystems A kann in das Zahlensystem B konvertiert werden. Im Computer werden alle Zahlen binär dargestellt, was allerdings schlecht lesbare Zahlen ergibt, die oft mehr als 30 Ziffern lang sind. In der Praxis benutzt man für diese Zahlen meist das Hexadezimalsystem, da die Konvertierung Hexadezimal-Binär sehr einfach ist und die Zahl 4x weniger Stellen aufweist. Im Folgenden werden nun die einzelnen Konvertierungen betrachtet.

Hexadezimal oder Binär → Dezimal

Jede Stelle wird mit ihrer Gewichtung multipliziert

Beispiel:

$$5A3h = 5 \cdot 16^2 + 10 \cdot 16^1 + 3 \cdot 16^0 = 1280 + 160 + 3 = 1443d$$

$$101010b = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 32 + 8 + 2 = 42d$$

Dezimal → Hexadezimal oder Binär

Die Zahl wird durch die Basis des neuen Zahlensystems geteilt, der jeweilige Rest entspricht einer Stelle der neuen Zahl. Der Quotient wird immer wieder dividiert bis nur noch Rest bleibt.

Beispiel:

Umwandlung von 254d nach Hexadezimal:

$$254d : 16d = 15d; \text{ Rest } 14d \rightarrow Eh$$

$$15d : 16d = 0d; \text{ Rest } 15d \rightarrow Fh$$

Die Zahl kann jetzt von unten nach oben gebildet werden und lautet dementsprechend FEh.

Für eine binäre Zahl wird berechnet:

$$254d : 2d = 127d; \text{ Rest } 0d \rightarrow 0b$$

$$127d : 2d = 63d; \text{ Rest } 1d \rightarrow 1b$$

$$63d : 2d = 31d; \text{ Rest } 1d \rightarrow 1b$$

$$31d : 2d = 15d; \text{ Rest } 1d \rightarrow 1b$$

$$15d : 2d = 7d; \text{ Rest } 1d \rightarrow 1b$$

$$7d : 2d = 3d; \text{ Rest } 1d \rightarrow 1b$$

$$3d : 2d = 1d; \text{ Rest } 1d \rightarrow 1b$$

$$1d : 2d = 0d; \text{ Rest } 1d \rightarrow 1b$$

Die Zahl kann jetzt von unten nach oben gebildet werden und lautet dementsprechend 11111110b.

Binär → Hexadezimal und Hexadezimal → Binär

Diese Transformation ist mit Hilfe einer Tabelle sehr einfach, nach einigen Umwandlungen kennt man die Tabelle auswendig und kann die Umwandlung innert Sekunden vornehmen. Pro 4 Stellen (Bit) der Binärzahl wird eine Hexadezimalziffer entstehen und umgekehrt.

Hex	Binär	Dezimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Tabelle 1: Hex-Bin Tabelle

Beispiel:

Umrechnen einer Binärzahl nach Hexadezimal:

010011010111010110b Erster Schritt: 4'er Blöcke von rechts nach links bilden
 01'0011'0101'1101'0110b Zweiter Schritt: Suchen der Binärkombinationen in der Tabelle und Hexzahl eintragen
 1 3 5 D 6 h → 135D6h

Umrechnen einer Hexadezimalzahl nach Binär:

E3Ah → gemäss Tabelle Binärkombinationen einsetzen: 1110'0011'1010b

1.4 Logische Operationen

Neben den arithmetischen Operationen gibt es noch 4 wichtige logische Operationen, die aus der Digitaltechnik her bekannt sind. Diese Operationen werden immer auf Binärzahlen angewandt und wirken immer Stellenweise. Eine binäre Stelle nennt man Bit, deshalb spricht man auch von bitweiser Wirkung.

AND Verknüpfung

Bei der AND - Verknüpfung wird ein Bit des Resultates nur dann eins, wenn das zugehörige Bit der beiden betroffenen Zahlen auch eins ist. Als Zeichen dieser Verknüpfung wird „&“ benutzt.

```
Beispiel:    1001110b
             &0111100b
             -----
             0001100b
             =====
```

OR Verknüpfung

Bei der OR - Verknüpfung wird ein Bit des Resultates eins, wenn das zugehörige Bit bei einer oder beiden Zahlen eins ist. Als Zeichen dieser Verknüpfung wird „|“ benutzt.

```
Beispiel:    1001110b
             |0101100b
             -----
             1101110b
             =====
```

EOR / XOR Verknüpfung (Exklusiv - OR)

Bei der EOR - Verknüpfung wird ein Bit des Resultates nur eins, wenn das zugehörige Bit bei einer, aber nicht beiden Zahlen eins ist.

```
Beispiel:    1001110b
             EOR 0101100b
             -----
             1100010b
             =====
```

COM /NOT Operation (Complement)

Bei der COM - Operation werden die einzelnen Bit der Zahl gedreht, aus 1 wird 0 und umgekehrt. Als Operationszeichen wird vielfach „!“ benutzt.

```
Beispiel:    !1001110b
             -----
             0110001b
             =====
```

1.5 Zahlenbereich, negative Zahlen und Carry-Bit

Jede Stelle einer Binärzahl nennt man Bit, eine Zahl mit 4 Stellen ist somit eine 4-Bit-Zahl. In einem Computer können keine beliebig langen Zahlen dargestellt werden. Sie werden durch die technische Eigenschaften des Prozessors bestimmt. Normale Werte sind 4,8,16,32,64 Bit. Grössere Zahlen werden jeweils mittels geeigneter Software aufbereitet und häppchenweise dem Prozessor zugeführt.

Für die üblichen Bitwerte werden folgende Begriffe verwendet:

Nibble:	4-Bit Zahl, Bsp. 1010b; Ah
Byte:	8-Bit Zahl, Bsp. 10100101b; A5h
Word:	16-Bit Zahl, Bsp. A5D4h
Longword:	32-Bit Zahl, Bsp. A643239Eh

Durch die vorgegebene maximale Stellenzahl wird der Zahlenbereich beschränkt, so dass nur noch Zahlen in einem bestimmten Bereich zulässig sind.

Länge	Zahlenbereich
4-Bit	0..15d
8-Bit	0..255d
16-Bit	0..65'535d
32-Bit	0..4'294'967'295d

Negative Zahlen

Wie oben ersichtlich ist, sind nur ganze, positive Zahlen darstellbar. Um den Bedarf an negativen Zahlen abzudecken, wird ein Bit zum Speichern des Vorzeichens verwendet. Aus diesem Grund geht ein Bit verloren, statt 8 Bit stehen deshalb nur noch 7 Bit für die Zahl zur Verfügung, der Wertebereich halbiert sich. Man unterscheidet zwischen Einerkomplement und Zweierkomplement. Wir beschränken uns hier auf die Bildung des Zweierkomplements, die zwar etwas schwieriger, aber zum Rechnen viel einfacher ist. Die Bildung des Zweierkomplements wird an folgendem Beispiel betrachtet.

Wir wollen die Zahl -41d (29h) darstellen (8 Bit):

29h nach Binär gewandelt: 00101001b → 29h

COM - Operation: 11010110b

Die Zahl '1' addieren: 11010111b → D7h

Die Zahl kann nun wie gewohnt mit unseren Rechenoperationen benutzt werden. Die Wandlung zurück zu einer positiven Zahl wird durch nochmaliges Anwenden der obigen Methode erreicht:

D7h nach Binär gewandelt: 11010111b → D7h

COM - Operation: 00101000b

Die Zahl '1' addieren: 00101001b → 29h

Überlauf und Carry-Bit

Im folgenden betrachten wir nur positive Zahlen auf einem Prozessor mit 8 Bit Zahlenverarbeitung.

Wir betrachten an einem Beispiel, welche Fehler durch die Einschränkung des Zahlenbereiches entstehen können. Wir addieren zwei 8 Bit Zahlen:

```

 10110100b    180d
+01110001b    +113d
-----
100100101b    293d
=====

```

Das Resultat dieser Operation wird 9 Bit, unser Rechner wird aber nur 8 Bit darstellen können und uns als Ergebnis 37d präsentieren statt des richtigen Resultats von 293d. Um diesen „Fehler“ zu erkennen, besitzt der Prozessor ein Flag (1 Bit), das Carry-Flag genannt wird. Dieses wird in einem solchen Fall gesetzt (gesetzt = binär „1“). Da uns der Prozessor solche Überläufe signalisiert, können wir den Rechenfehler mit geeigneter Software abfangen und richtigstellen.

Bei vorzeichenbehafteten Zahlen wird die Überlaufbedingung etwas komplizierter und wird deshalb nicht betrachtet.

2 Aufbau eines Computersystems

Ein typisches Computersystem besteht immer aus Mikroprozessor (CPU), RAM, ROM und Peripherie. Sind zwei oder mehr dieser Elemente auf dem gleichen Chip vereint, spricht man von einem Microcontroller (MCU).

2.1 Central Prozessor Unit (CPU)

Die CPU verarbeitet die durch einen Prozess anfallenden Daten nach einer festen, in relativ kleinen Schritten vorgegebenen Anleitung. Diese Anleitung nennt man Programm. Die CPU liest Befehl für Befehl des Programms und führt diese unmittelbar nacheinander aus. Mittels sehr elementarer Befehle, die sinnvoll aneinandergereiht sind, lässt sich eine komplexe Aufgabe lösen. Folgende Aufgaben werden von der CPU bearbeitet:

- Lesen der im Speicher gespeicherten Befehle, Interpretation und Ausführung
- Steuerung des Programmablaufs
- Kontrolle des Bus

Diese Aufgaben werden in der CPU mit Hilfe von folgenden Komponenten gelöst:

- Die Kontrolleinheit (Control Unit)
Die Control Unit steuert den internen Ablauf der CPU gemäss den Informationen des Instruction Decoders
- Der Befehldecoder (Instruction Decoder)
Im Instruction Decoder werden die einzelnen Befehle eines Programms decodiert und die Informationen an alle anderen Teile der CPU verteilt.
- Der Programmzähler (Program Counter, PC)
Der Program Counter wird nach dem Anlegen der Speisespannung auf einen definierten Wert ge-

bracht. Der Inhalt des Program Counter bildet die Speicheradresse, an der der nächste auszuführende Befehl steht. Nachdem der Befehl verarbeitet wurde, wird der PC erhöht und steht so auf dem nächsten Befehl des Programms. Es gibt die Möglichkeit, im Programm den Wert des PC zu verändern. Das Programm wird dann an der neuen Adresse weiterfahren. Dies nennt man einen Sprung.

- Die ALU (Arithmetic Logic Unit)
Die ALU ist das Rechenwerk des Computers. Die ALU kann alle logischen Verknüpfungen (AND, OR, EOR, NOT), addieren, subtrahieren und schieben. Je nach Ergebnis der Operation wird das Flagregister verändert.
- Das Flagregister (Status Register, SR)
Im Flagregister können die internen Zustände der ALU und der CPU gelesen werden. Diese können vom Programm zur Steuerung des Programmablaufs benutzt werden. Die wichtigsten Flags sind:
 - Zeroflag: Wird '1', wenn die letzte Operation in der ALU als Ergebnis 0 war.
 - Signflag: Wird '1', wenn die letzte Operation in der ALU eine negative Zahl gab.
 - Carryflag: Wird '1', wenn die letzte Operation in der ALU einen Übertrag verursachte.
- Die Register
Ein Register ist ein CPU-interner Speicher der vom Programm ansprechbar ist. Bei Operationen mit der ALU werden immer ein oder zwei Register (Quellregister) zu einem Zielregister zusammengeführt. Meist ist das Zielregister mit einem der Quellregister identisch. Früher unterschied man zwischen Akkumulator und Hilfsregister. Man konnte nur mit dem Akkumulator bestimmte Rechenoperationen durchführen. Moderne Prozessoren kennen diese doch massive Einschränkung nicht mehr, die Register werden (quasi) gleichwertig betrachtet. Allerdings gibt es auch Spezialregister, die für bestimmte Aufgaben reserviert sind.
- Die Bus Kontrolleinheit (Bus Control Unit)
Die Bus Control Unit steuert den Ablauf am Bus, also der Schnittstelle zu Speicher und Peripherie. Es werden zum richtigen Zeitpunkt die Daten und Adressen durchgeschaltet und die Kontrollsignale gesetzt (Bustiming).

2.2 Der Speicher (Memory)

Programme und Daten werden im Memory abgelegt (gespeichert). Als Speicher für Daten und Programme werden vorwiegend Halbleiter-Memorychips zum Einsatz. Es wird zwischen RAM (random access memory = Speicher mit wahlfreiem Zugriff = Schreib-Lese-Speicher) und ROM (read only memory = Nur-Lese-Speicher) unterschieden. Bei Mikrocomputeranwendungen wird das Programm und die Konstanten, die bei der Programmierung bereits feststehen, in ROM's gespeichert, die Variablen in RAM's.

Read Only Memory (ROM)

Es gibt verschiedene Arten von ROM, die sich bezüglich Programmierung und Preis unterscheiden.

- Maskenprogrammierte ROM
Maskenprogrammierte ROM werden bereits bei der Chipherstellung programmiert. Diese Variante ist sehr preiswert, eignet sich aber nur bei sehr grossen Serien und wenn das Programm nie mehr verändert wird (Bsp. PC BIOS, Waschmaschine, Drucker).
- PROM (Programmable Read Only Memory)
Anwendung ähnlich der maskenprogrammierten ROM, nur dass der Anwender den Chip selbst programmieren kann. Meist werden fensterlose EPROM eingesetzt (billigeres Gehäuse).
- EPROM (Erasable Programmable Read Only Memory)
Dies ist der häufigste ROM-Typ. Das ROM kann mit UV-Licht gelöscht werden und eignet sich quasi

für alle Anwendungen, besonders für kleine und mittlere Serien und Prototypen. EPROM wird langsam durch FLASH-ROM abgelöst.

- EEPROM (Electrical Erasable Programable Read Only Memory)
Ähnlich EPROM, nur dass der Inhalt elektrisch gelöscht werden kann. EEPROM sind teuer und für grössere Kapazitäten nicht erhältlich. EEPROM eignen sich vorzüglich zur Speicherung von Setups (Bsp. Fernseher, Natel).
- FLASH
Flash sind ähnlich wie EEPROM, mit dem Unterschied dass nicht einzelne Zellen gelöscht werden können, sondern nur das ganze ROM. Flash sind preiswert und eignen sich besonders für Geräte, die in ihrer Lebenszeit zwischendurch neue Software erhalten. Flash können direkt „on board“ programmiert werden und eignen sich speziell für automatische Downloads (Bsp. ISDN-Telefone, Modems, Bi-lettautomaten).

Random Access Memory (RAM)

Es wird zwischen statischen und dynamischen RAM unterschieden. Alle anderen RAM-Bezeichnungen können diesen beiden Grundtypen zugeteilt werden (EDO-RAM, SD-RAM, PS-RAM, Cache-RAM).

- Dynamisches RAM
Das Speicherelement für 1 Bit besteht aus einem Transistor und einer Kapazität. Die Kapazität (der eigentliche Speicher) verliert aber seine Ladung schnell und muss deshalb wieder aufgefrischt werden (Refresh). Dies bedarf einer externen Refresh-Schaltung, die meist mit einem speziellen Baustein realisiert wird. D-RAM haben eine hohe Packungsdichte und weisen deshalb eine sehr hohe Speicherkapazität auf. Der Stromverbrauch von D-RAM ist relativ hoch. Deshalb kommen diese für kleine batteriebetriebene Geräte nicht in Frage.
- Statische RAM
Statische RAM bestehen aus einem Flip-Flop und brauchen deshalb keinen Refresh. Sie benötigen kaum Strom und sind für batteriegepufferte Geräte ideal. S-RAMs mit Lithiumzelle behalten den Inhalt über 20 Jahre. Durch die Flip-Flop-Schaltung ist der Chipflächenbedarf pro Bit wesentlich grösser als bei D-RAM, deshalb ist die Speicherkapazität pro Chip 4-16x kleiner und die Kosten pro Bit höher. S-RAM sind aber wesentlich schneller als D-RAM und in Cache-Anwendungen unersetzlich.

Aufbau eines Speicherchips

Als Beispiel eines Speicherchips soll hier ein 512Kx8 S-RAM betrachtet werden. Es existieren zwei Arten zum Bezeichnen der Kapazität eines Chip. Eine Möglichkeit ist mit der Angabe der verfügbaren Speicherzellen, z.B. 4M, womit aber nicht 4 MByte gemeint sind, sondern 4 MBit. Besser ist die Angabe 512Kx8, mit der die Busbreite gekennzeichnet wird. Üblich ist x1, x4, x8 und x16, meistens wird x8 benutzt. Um eine Busbreite von 16 Bit abzudecken werden 2 8-Bit Speicherbausteine benötigt.

	Adresse, die mit A0-A18 bestimmt wird, geschrieben.
/OE	Output Enable, wenn dieses Signal low ist, wird der Inhalt der Speicherzelle, die mit den Adressleitungen A0-A18 selektiert ist, an D0-D7 ausgegeben.

2.3 Die Peripherie und I/O-Ports

Als Peripherie bezeichnet man alles um die CPU, was nicht Speicher ist. Übliche Peripheriebausteine sind Timer, A/D und D/A - Wandler, Hard-Disk Controller, Video-Controller, Serielle Schnittstelle, Parallele Schnittstelle, etc..

Peripheriebausteine werden meist direkt an den Prozessorbus angeschlossen und mit einer Chip-Select-Logik angesteuert. 90% aller Peripheriebausteine haben einen 8-Bit Datenbus.

I/O-Ports sind die einfachsten Peripheriebausteine. Damit können externe Signale von Sensoren, Taster, etc. an den CPU-Bus angepasst und im Programm gelesen werden. Eventuelle Ausgaben an Kontrolllampen, Motoren, etc. werden vom Bus getrennt und können durch geeignete Schaltungen verstärkt und abgegriffen werden. Als einfache I/O-Port werden häufig die Chips der 74'er Logikserie eingesetzt, namentlich der 74HC245 (Input, Treiber) und 74HC574 (Output mit Buffer). Ein Datenblatt dieser Bausteine befindet sich in Anhang.

Detaillierte Informationen bezüglich Funktionalität und Anschlusschema sind den jeweiligen Datenblättern zu entnehmen und vielfach sehr unterschiedlich. Unbedingt nach Anschlussbeispielen im Datenbuch suchen!

2.4 Der Prozessorbus

Als Prozessorbus werden alle Signale zusammengefasst, die zur Ansteuerung von externen Bausteinen notwendig sind. Es wird unterschieden zwischen Daten-, Adress- und Steuerbus, wobei letzterer je nach CPU unterschiedlich sein kann. Es wird nur ein einfacher Bus ohne DMA (Direct Memory Access) und Sondersignalen betrachtet.

- Adressleitungen A0 - An
Mit dem Adressbus wird festgelegt, worauf zugegriffen werden soll. Dies können RAM, ROM oder Peripheriezugriffe sein. Die höchste erreichbare Adresse entspricht dem Adressraum der CPU, z.B. 4 GByte. Dieser Adressraum wird nun vom Anwender mit Hilfe einer Chip-Select-Logik aufgeteilt in RAM-Bereich, ROM-Bereich und Peripheriebereich. (Das Signal /CS wird aus den Adressleitungen gewonnen.)
- Datenleitungen
Wie der Name sagt, werden mit den Datenleitungen die Daten ausgetauscht. Die Anzahl der Datenleitungen entspricht der Busbreite der CPU, je nach Prozessor gibt es 4, 8, 16 oder 32 Bit.
- Steuersignale
Mit den Steuersignalen wird die Synchronisation mit der Peripherie erzeugt. In unserer Betrachtung ist die CPU immer der Chef, sie sagt der Peripherie was zu tun ist. (CPU ist Busmaster.)

/WE	Write. Wenn dieses Signal low ist, wird ein Datum an die Datenleitungen angelegt, das von
	einem anderen Chip am Bus gelesen wird.
/RD	Read. Wenn dieses Signal low ist, wird ein Datum an den Datenleitungen gelesen, das vom selektierten Chip auf den Bus geschoben wurde.
/IRQ	Interrupt. Dieses Signal wird von einem Peripheriebaustein gesetzt, um die CPU

von einem

speziellen Ereignis zu unterrichten. Was die CPU damit macht, wird im Programmcode festgelegt.

Es existieren noch viel mehr Steuersignale, besonders für spezielle Timings. Der Übersichtlichkeit halber wurden nur die wichtigsten aufgeführt.

2.5 Chip Select Logik

Ein Mikrocomputer besteht immer aus RAM, ROM und Peripherie. Nachfolgend wird die sogenannte Chip-Select-Schaltung erläutert, die nötig ist, um aus dem Adressbus die notwendigen Chip-Selects zu erhalten. Eine Chip-Select-Schaltung wird immer nach der gleichen Reihenfolge entworfen.

1. Bestimmen der Bausteine, die an einen Bus geschaltet werden sollen.
2. Bestimmen des Adressraumes der einzelnen Bausteine nach Datenblatt.
3. Adressraumeinteilung nach Vorgabe oder freiem Ermessen (Achtung, das ROM hat immer einen festen Platz!!). Achtung, wenn sich zwei Bereiche überlappen, muss einem Priorität gewährt werden! Es dürfen nie zwei Bausteine gleichzeitig selektiert werden.
4. Wahrheitstabelle der Chip-Select.
5. Aufbau der Schaltung.

Wie eine solche Logik nach diesem Kochrezept aufgebaut wird, wird an einem Beispiel erläutert:

Gegeben:

- CPU mit 64 KByte Adressraum und 8-Bit Datenbus
- 8 KByte ROM soll an Adresse 0000h sein
- 32 KByte RAM
- 2 I/O-Ports 1 Byte breit

Gesucht:

- a) Chip-Select-Schaltung vollständig ausdecodiert (erweiterungsfähiges System)
- b) Chip-Select-Schaltung mit geringstem Aufwand

Schritt 1 und 2 wurde uns bereits durch die Aufgabenstellung abgenommen, also starten wir bei Schritt 3.

Schritt 3

Die CPU besitzt einen Adressraum von 64 KByte, die Adressen gehen also von 0000h - FFFFh.

Das ROM hat einen festen Platz von 0000h an 8 KByte. 8 KByte entspricht der Hexadezimalzahl 2000h. Somit wird das ROM von 0000h - 1FFFh im Adressraum der CPU Platz finden.

Das RAM belegt 32 KByte, wir könnten es anschliessend an das ROM bringen, der Aufwand ist aber geringer, wenn wir es direkt zuoberst im Adressraum plazieren, also von 8000h - FFFFh.

Die beiden I/O-Ports setzten wir an die Adresse 4000h und 4001h.

Schritt 4

Bilden der Wahrheitstabelle mit den Werten von Schritt 3.

ROM: 0000h -> 0000'0000'0000'0000b
 1FFFh -> 0001'1111'1111'1111b

I/O 1: 4000h -> 0100'0000'0000'0000b

I/O 2: 4001h -> 0100'0000'0000'0001b

RAM: 8000h -> 1000'0000'0000'0000b
 FFFFh -> 1111'1111'1111'1111b

Schritt 5

Nun müssen die Adressleitungen bestimmt werden, nach denen die Chips eindeutig erkannt werden. Dies sind normalerweise die Bits, die sich nicht ändern, demzufolge beim ROM A15-A13, beim RAM A15 und bei den I/O alle.

Wenn A15, A14 und A13 „0“ sind, ist ROM selektiert.

Ist A15 „1“, ist garantiert RAM selektiert.

Die I/O - Bereiche müssen für Aufgabe A vollständig decodiert werden. Für Aufgabe B kann eine Vereinfachung durchgeführt werden. In boolescher Algebra wird geschrieben:

$$/CS\ ROM = /(/A15 \& /A14 \& /A13) \text{ oder } A15 + A14 + A13$$

$$/CS\ RAM = /A15$$

$$/CS\ IO\ 1 = /(/A15 \& A14 \& /A13 \& /A12 \& /A11 \& /A10 \& /A9 \& /A8 \& /A7 \& /A6 \& /A5 \& /A4 \& /A3 \& /A2 \& /A1 \& /A0)$$

$$/CS\ IO\ 2 = /(/A15 \& A14 \& /A13 \& /A12 \& /A11 \& /A10 \& /A9 \& /A8 \& /A7 \& /A6 \& /A5 \& /A4 \& /A3 \& /A2 \& /A1 \& A0)$$

Für die Vereinfachung darf unbenutzter Adressraum beliebig genutzt werden, auf volle Dekodierung wird verzichtet. Es ist aber sehr wichtig, dass nie zwei CS gleichzeitig low werden! Vereinfacht wird:

$$/CS\ ROM = /(/A15 \& /A14) \text{ oder } A15 + A14$$

$$/CS\ RAM = /A15$$

$$/CS\ IO\ 1 = /(/A15 \& A14 \& /A0)$$

$$/CS\ IO\ 2 = /(/A15 \& A14 \& A0)$$

3 Der MC68332 Mikrocontroller von Motorola

Die bisherigen Erläuterungen gelten für alle bekannten Mikrocomputer. Von nun an wird speziell der Microcontroller MC68332 betrachtet und auf seine Eigenschaften eingegangen. Viele der behandelten Themen können aber leicht auf andere Controllerfamilien (Hitachi H8, Siemens x51, Microchip PIC, SGS-Thomson COP, etc) übertragen werden.

Der MC68332 ist ein Mitglied der MC683xx Controllerfamilie, die aus der Kombination des Mikroprozessors MC68020 und leistungsfähiger Peripheriebausteinen hervorgegangen ist. Je nach Anwendung wurde jeder der Controller auf ein anderes Gebiet optimiert:

- Der MC68302 als Kommunikationsprozessor für ISDN-Anwendungen
- Der MC68307 als 8051 Ersatz (besitzt einen 8051-kompatiblen Bus)
- Der MC68322 als Druckerprozessor
- Der MC68332 als Timingprozessor
- Der MC68340/41 für CD-Player oder CD-ROM
- Der MC68360 als Hochleistungs-Kommunikationsprozessor für Ethernet, Teilnehmervermittlungsanlagen (TVA) und anderen TDM-Anwendungen (TDM = Time Division Multiplex = Zeitschlitzmultiplex)

Alle der genannten Controller besitzen den gleichen Prozessorkern, deshalb sind die meisten in diesem Kurs behandelten Aussagen auf alle Mitglieder dieser Controllerfamilie übertragbar.

3.1 Interner Aufbau

Folgende Punkte sprechen für den Einsatz eines MC68332:

- Externer CPU-Bus mit 24 Adressleitungen und einer Datenbusbreite von 16 Bit
- Programmierbare Chip-Select Ausgänge
- Systemüberwachung (Erkennen von Zugriffen in nicht definierte Bereiche)
- Watchdog Timer (Erkennen ob der normale Programmablauf nicht gestört ist)
- 32,768 kHz externer Quarz (spart Kosten und Energie)
- Eingebauter Debugging Port zum Entwickeln von Programmen
- Statische Operation (Taktfrequenz von 0 – 16 MHz)
- Interne Architektur 32 Bit
- Coprozessor für Zeitfunktionen
- Eingebaute serielle Schnittstelle
- 2 KByte RAM eingebaut (reicht für einfache Programme oder TPU-Programme)

Der MC68332 besteht aus folgenden Komponenten:

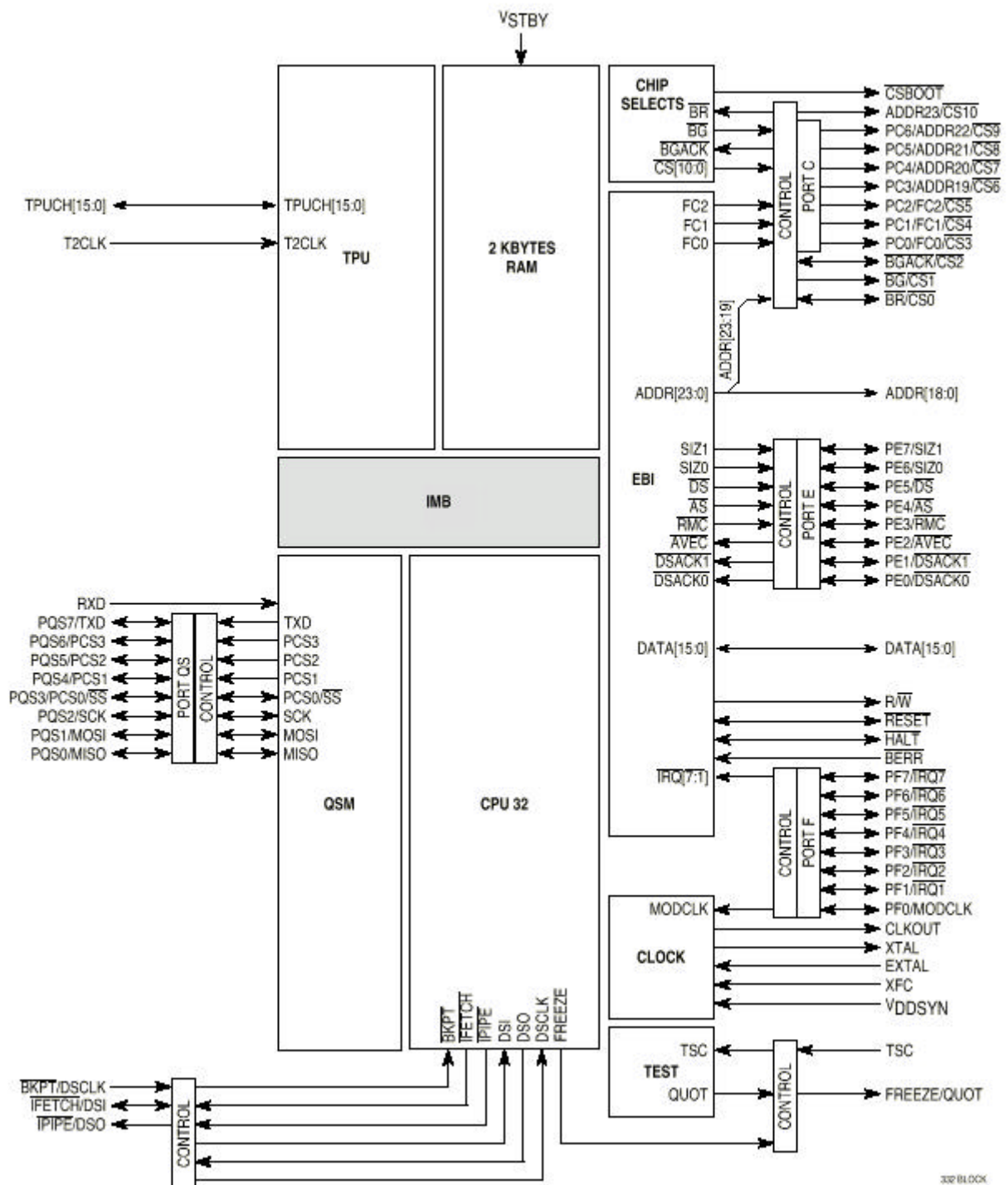


Abbildung 2: Blockschema MC68332

- Der Prozessorkern CPU32
Die CPU32 ist der Prozessor des MC68332. Die CPU32 steuert den Prozessorbus, führt das aktuelle Programm aus und bedient die internen und externen Peripheriebausteine.
- Die Time Processor Unit (TPU) (Coprozessor für Timingaufgaben)
Die TPU ist ein intelligenter Peripheriebaustein für Timingaufgaben. Die TPU ist ein unabhängiger zweiter Mikrocontroller auf dem Chip und wird als Coprozessor für Timingaufgaben verwendet. Sie

kann ihre 16 Ein-/Ausgabekanäle selbständig steuern und somit sehr komplexe Signalgenerierungen eigenständig ausführen. (Beispielsweise die Steuerung eines Schrittmotors, das Ausmessen von Pulsbreiten, das Erzeugen von Rechteckfrequenzen oder PWM-Signalen (Pulsweitenmodulation) und viele andere Timeraufgaben.) Dabei wird die CPU nicht unterbrochen und kann ihre eigentlichen Aufgaben erledigen. Durch diese Parallelarbeit der beiden Mikrocontroller wird eine erhebliche Leistungssteigerung des Gesamtsystems erreicht.

- **Das System Integration Modul (SIM)**
Für den Aufbau einer Chip-Select-Logik waren bisher immer externe Bausteine nötig, die den Aufwand für die Hardware und somit die Fehlerrate vergrößerte. Ausserdem mussten Watchdog - Schaltungen (Schaltung zum Erkennen von Prozessorabstürzen) immer extern aufgebaut werden. All diese Funktionen wurden im Modul SIM eingebaut. Weiterhin wurde eine PLL-Schaltung zur Takterzeugung integriert, aus einem billigen Uhrenquarz von 32,768 kHz wird der Systemtakt von 16 MHz gewonnen. Auch die Steuerung des externen Bus wird im SIM kontrolliert. Zum Einstellen der Chipselect, des Watchdogs und der Taktfrequenz stehen auf dem SIM Register zur Verfügung.
- **Das Queued Serial Modul (QSM)**
Eine asynchrone und eine synchrone serielle Schnittstelle sind im Modul QSM zusammengefasst. Die asynchrone Schnittstelle besitzt eine Sende- und Empfangsleitung und eignet sich zum Anschluss an ein Terminal oder einem PC. Man kann auch ein Netzwerk damit aufbauen. Die synchrone Schnittstelle eignet sich zum Anschluss von seriellen EEPROMs, seriellen A/D und D/A-Wandlern, seriellen LED- und LCD-Anzeigen. Da ein sehr hohen Datendurchsatz bei synchronen Verbindungen gewährleistet ist, kann auch eine Prozessor-Prozessor Schnittstelle gebaut werden.
- **Das Standby RAM**
Der MC68332 hat 2 KByte RAM integriert, welches für Daten oder Programme benutzt werden kann. Dieser Speicher kann von einer externen Batterie weiterversorgt werden, wenn der MC68332 bereits abgeschaltet wurde. Falls das interne RAM für eine Anwendung nicht ausreicht und ein externes RAM verwendet werden muss, ist es empfehlenswert, das interne RAM wegen der schnellen Zugriffszeit als Stackbereich zu verwenden.
- **Der Inter-Module-Bus (IMB)**
Der IMB dient als Verbindungsbus im Innern der MCU. Der Aufbau ist quasi identisch mit dem externen Bus.

3.2 Die CPU32

Die CPU32 ist eine Weiterentwicklung des MC68020. Es wurden aber nicht alle Befehle implementiert, einige selten benutzte Befehle wurden nicht miteingebunden. Der Prozessor hat keine Schnittstelle zu einem Mathematik-Coprozessor. Folgende Punkte sprechen für diesen Prozessor:

- 32 Bit Architektur für alle Datenpfade und die Arithmetik
- 32 Bit x 32 Bit Multiplikation / 64 Bit:32 Bit Division
- Grosse Auswahl an Instruktionen und Adressierungsarten (CISC-Prozessor)
- 8 Datenregister und 8 Adressregister, davon eines als Stackpointer benutzt
- User und Supervisor Zustand
- Aufwärtskompatibel zu Programmen, die für MC68000/10 geschrieben wurden

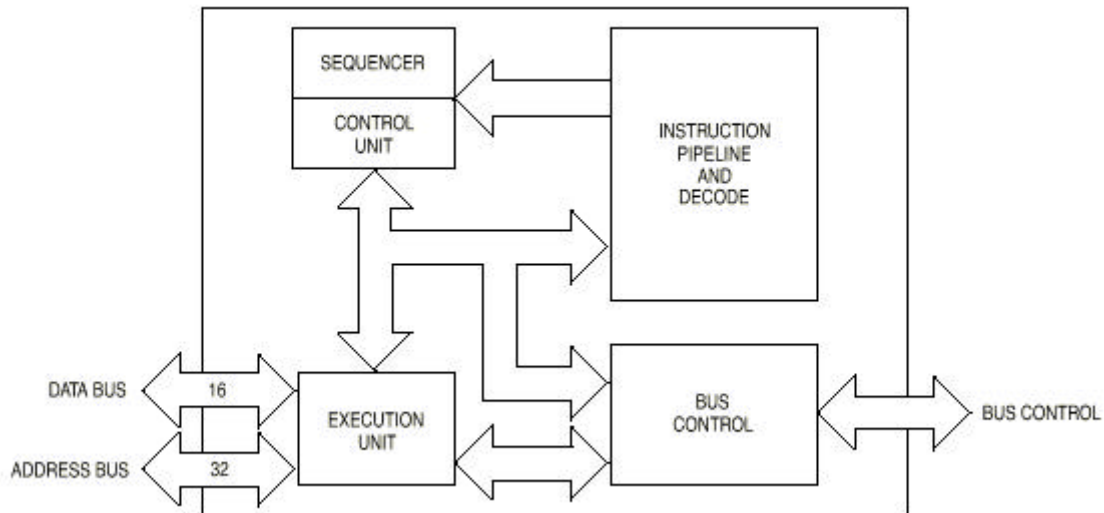


Abbildung 3: CPU32 Blockschaltbild

Der Aufbau der CPU entspricht der Definition in den vorgehenden Kapiteln.

3.3 Die Register

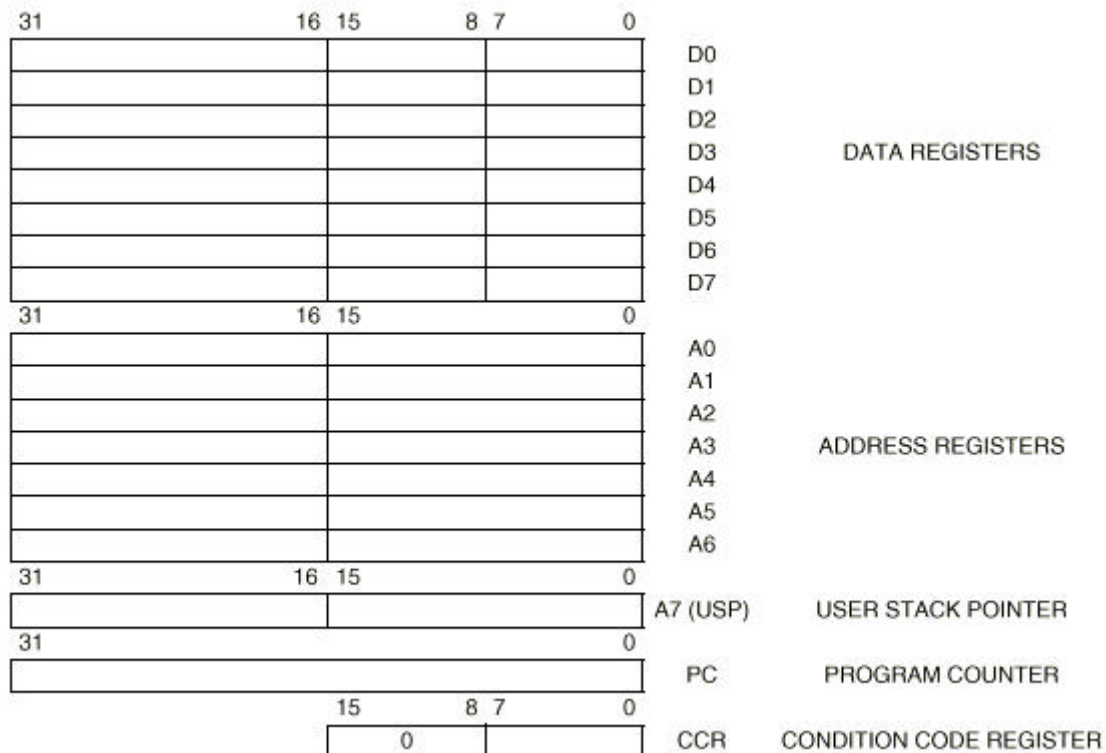


Abbildung 4: Registermodell CPU32 im User-Mode

Die CPU32 besitzt 8 Datenregister und 8 Adressregister. Datenregister werden hauptsächlich zum Speichern von Daten benutzt. Nur mit den Datenregistern kann man arithmetische und logische Operationen durchführen. Dagegen können Adressregister zum Adressieren verwendet werden. Zusätzlich kann man

mit Adressregistern addieren und subtrahieren. Das Adressregister A7 wird als Stackpointer verwendet und sollte vom Programmierer nur in Ausnahmefällen direkt beschrieben werden. Der Program Counter (PC) zeigt immer auf die nächste auszuführende Instruktion. Im Condition Code Register (CCR) stehen die Flags der ALU zur Verfügung.

Zusätzlich zu den genannten Registern stehen im sogenannten Supervisor-Mode weitere Register zur Verfügung. Die Trennung zwischen User- und Supervisor-Mode bewirkt, dass auch bei Programmfehlern (die im User-Mode auftreten) das Betriebssystem (im Supervisor-Mode laufend) stabil bleibt und das fehlerhafte Programm beenden kann. Vielfach ist auch der Speicher hardwaremässig komplett getrennt. Die zusätzlichen Register sind:

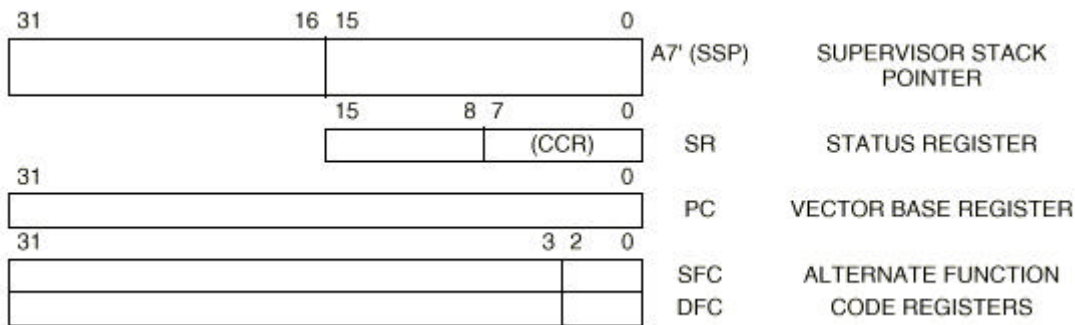


Abbildung 5: Zusätzliche Register der CPU32 im Supervisor-Mode

Der SSP ist ein spezieller Stack, der nur im Supervisor-Mode benutzt wird. Das CCR wird um ein Byte erweitert und wird nun als Status Register (SR) bezeichnet. Im erweiterten Byte befinden sich Flags für die Interrupt-Kontrolle und den Trace (Einzelschrittmodus). Im Vector Base Register (VBR) kann die Tabelle der Interruptvektoren angepasst werden. Die beiden Alternative Funktion Code Register (SFC und DFC) werden für die Hardwaremässige Trennung von Supervisor-Speicher und User-Speicher benutzt.

3.4 Das Status Register (SR) und das Conditions Code Register (CCR)

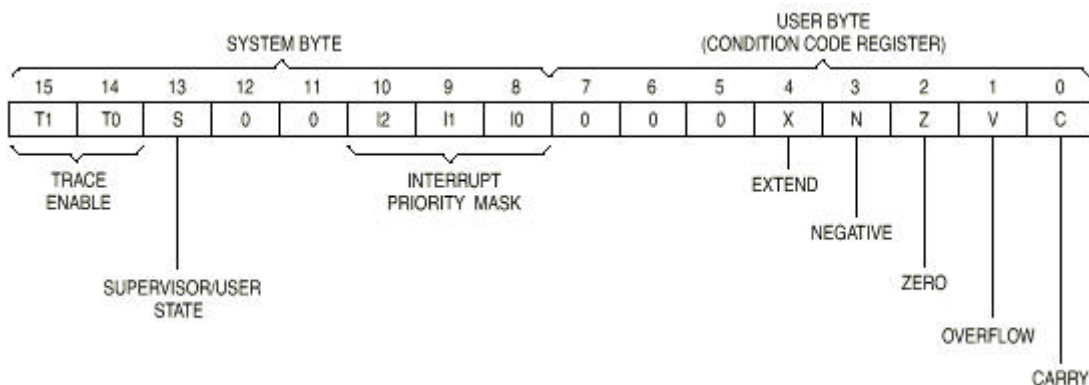


Abbildung 6: SR/CCR - Register

Im befinden sich die bereits erwähnten Flags: Carry, Zero und Negative. Zusätzlich kommen das Overflow (Überlauf) und das Extended -Flag dazu. Die Bedeutung dieser Flags wird im Softwareteil erläutert.

Im SR sind die Interrupt-Flags, die Trace-Flags und das State-Flag zu finden, welche hier nicht betrachtet werden.

3.5 Der Adressraum

Durch die 24 Adressleitungen erhalten wir einen maximalen Adressraum von 2^{24} Bytes. Dies entspricht 16'777'216 Bytes bzw. 16 MBytes. Innerhalb dieses Bereiches befindet sich die Vektortabelle. (normalerweise ab Adresse 0, je nach Programmierung des VBR kann sie an einer anderen Stelle stehen.)

Für die interne Peripherie muss ebenfalls ein Bereich reserviert werden. Es gibt zwei Positionen, wo sich die Peripherieregister befinden können. Dies ist im folgenden Diagramm dargestellt.

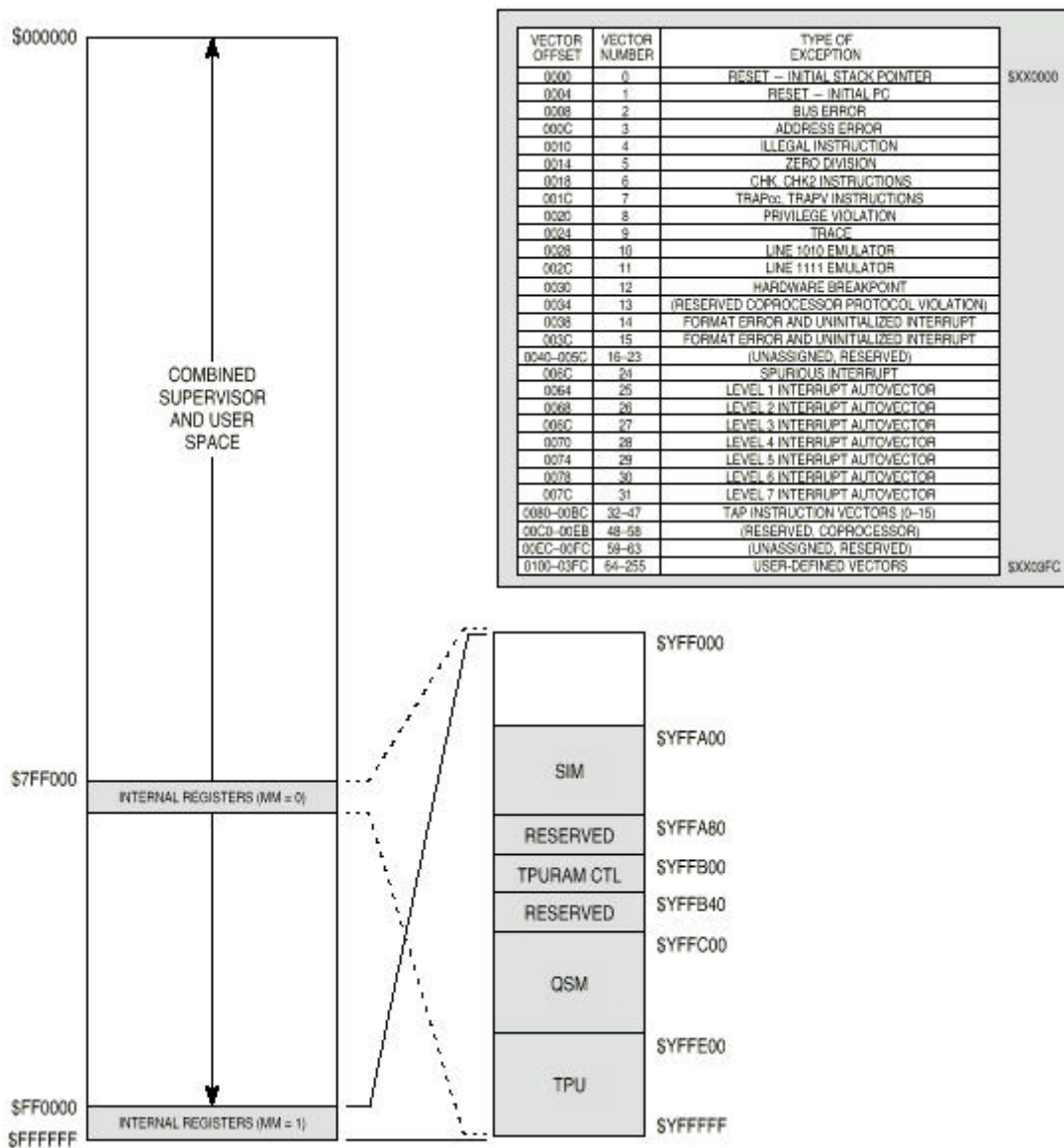


Abbildung 7: Memory Map des MC68332

4 Inbetriebnahme des Übungsboards

Nach diesem Theorieblock wollen wir zur Praxis übergehen. In diesem Kapitel wird das Übungsboard der TEKO Bern vorgestellt. Weiterhin wird der Umgang mit dem Debugging - Werkzeug BD32 erläutert.

4.1 Das Testboard der TEKO Bern

Das Testboard besteht aus zwei Printen, die übereinander angeordnet sind. Auf der oberen Platte befinden sich Controller, eine zweite serielle Schnittstelle, zusätzlich einige I/O Ports und Stecksockel für RAM und ROM. Auf der unteren Platte sind Taster, LED und 4 Anzeigeeinheiten angeordnet. Hier befinden sich ebenfalls die seriellen Schnittstellen und die Spannungsregelung.

Inbetriebnahme

Das Testboard wird mit einem handelsüblichen Speisegerät mit einer Spannung zwischen 12 und 24 Volt betrieben. Die Spannung wird mit dem beigelegtem Kabel an die orangefarbene Buchse angeschlossen (Achtung, Polarität beachten!!!).

Am Stecker oben links (2x5pol) wird das externe BDM-Interface angeschlossen (Auf dem Print sollte sich eigentlich eine BDM-Logik befinden, doch bisher funktionierte sie nicht). Das BDM-Interface wird anschliessend am Parallelport des PC eingesteckt.

Jetzt kann das Speisegerät eingeschaltet werden. Bei korrekter Inbetriebnahme sollte auf der Anzeige die Zahl „0000“ sichtbar sein.

4.2 Adressraumeinteilung des Testboards

Die Aufteilung des Adressraumes ist teilweise hardwaremässig vorgegeben, kann aber noch softwaremässig verschoben werden. Dieser Kurs stützt sich auf folgende Konfiguration:

RAM: 000000h - 0FFFFFFh

ROM: 100000h - 1FFFFFFh

I/O: 200000h - 2007FFh

IRAM: 200800h - 200FFFh

UART: 201000h - 2010FFh

SMM: FFF000h - FFFFFFFh

Der Bereich des I/O kann feiner aufgespalten werden:

200000h untere Tastenreihe bitweise

200100h obere Tastenreihe bitweise

200200h Anzeigeelement 3. + 4. Stelle

200300h Anzeigeelement 1. + 2. Stelle

200400h LED-Reihe oberhalb der Tasten bitweise

200100h LED-Reihe ganz oben bitweise (Gemäss Dokumentation des Boards, geht aber nicht)

4.3 New Background Debugger NBD32 von OESCH.ORG

Zum Testen der Hardware und der Software gibt es von Motorola das Tool BD32, das kostenlos erhältlich ist und die Anschaffung eines teuren Emulators überflüssig macht. Leider wurde dieses Tool nicht mehr modernisiert und ist als DOS-Anwendung mit Parallelport-Zugriff ab Windows NT nicht mehr funktionsfähig. Durch Mailverkehr mit Scott Howard, dem Programmierer des Tools, wurde klar, dass das Tool nicht mehr unterstützt wird. Alternative ist eine UNIX-Lösung, die aber die Grenzen dieses Kurses sprengen würde. Deshalb habe ich mich entschlossen, ein Windows-basierenden Debugger zu schreiben. Für die, die es interessiert: Geschrieben in C mit Dev-C++ (<http://www.bloodshed.net/devcpp.html>)

Es entstand mehr als nur ein Debugger. Da nur 5 Übungsboards zur Verfügung stehen, habe ich einen Simulationsmodus eingebaut, mit dem die meisten Befehle simuliert werden können. Der einzige Unterschied ist die Geschwindigkeit.

Installation

Das Programm befindet sich auf dem Internet an der Adresse <http://www.oesch.org/> unter Mikroprozessortechnik.

Für die Arbeit in diesem Kurs benötigen Sie die folgenden Files:

- Assembler „AS32“
- New Background Debugger „NBD32“
- Software-Emulation des Teko-Übungsboard „TekoSim“

Der Parallelport-Treiber wird nur für die Antsteuerung der Hardware über den Parallelport benötigt und ist bereits auf den Schulrechnern installiert.

Die Programme sind so programmiert, dass sie ohne spezielle Rechte installiert werden können.

Start der Simulation

Als erstes ist immer die Simulation TekoSim zu starten! Folgendes Bild ergibt sich dann:

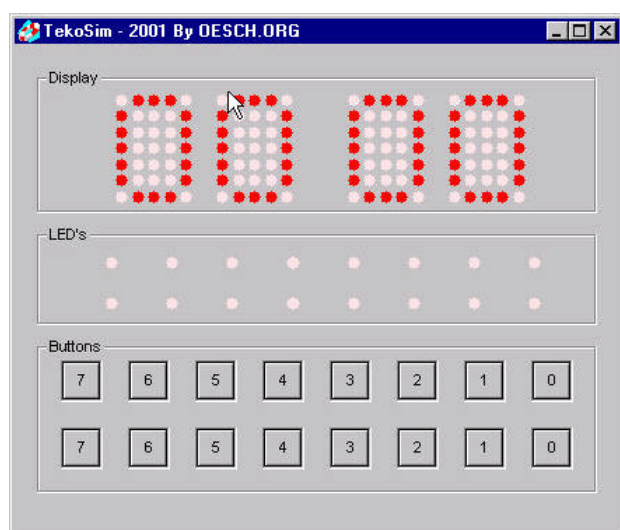


Abbildung 8: TekoSim Oberfläche

Dies entspricht einem Abbild der realen Hardware. Oben die Display, in der Mitte die LED-Reihen und unten die Taster. Der einzige Unterschied sind die unteren Tasten. Auf dem echten Board sind es Taster (nicht rastend), in der Simulation sind beide Tastenreihen einrastend.

Danach kann die Debugging-Umgebung gestartet werden mit Doppelklick auf NBD32.exe. Eventuell erscheint folgende Fehlermeldung:



Abbildung 9: DLL not found Error

Diese Meldung zeigt an, dass die Parallelport-dll fehlt. So lange nur Simuliert werden will, kann diese Meldung ignoriert werden.

Nach dem Start ist folgendes Bild sichtbar:

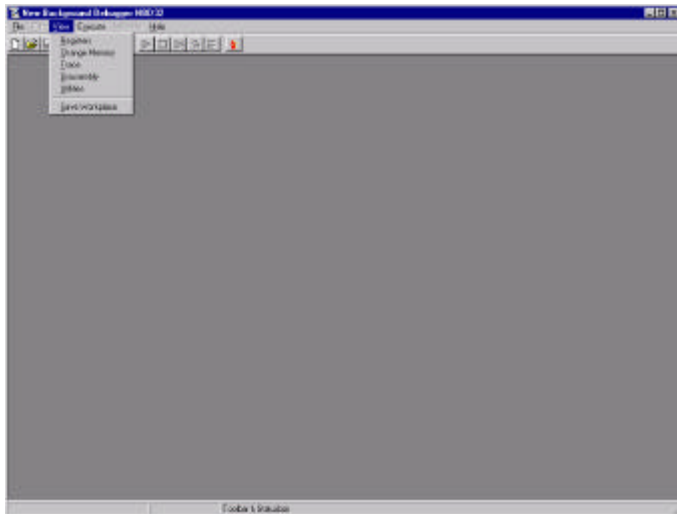


Abbildung 10: NBD32 Startscreen

Als erstes sollte jetzt im Menu Execute/Options angewählt werden:

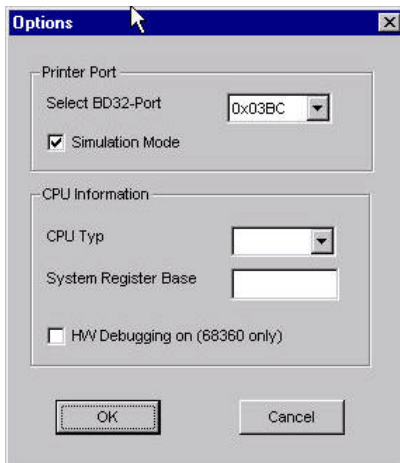


Abbildung 11: Execute/Options

Für die Simulation muss ein „Haken“ im Kästchen „Simulation Mode“ sein. Wird mit der echten Hardware gearbeitet, muss das Feld frei sein und die korrekte Adresse der parallelen Schnittstelle eingetragen werden: (Teko: 0x0378). Mit OK quittieren.

Um das Tool betriebsbereit zu machen, empfiehlt sich gleich im Menu View folgende Einstellung:

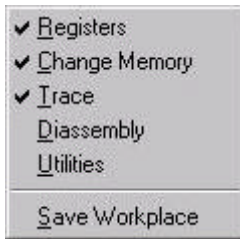


Abbildung 12: Menu View

Jetzt können sie Ihre Arbeitsumgebung nach beliebigen Einstellungen, folgende Ansicht hat sich bewährt:

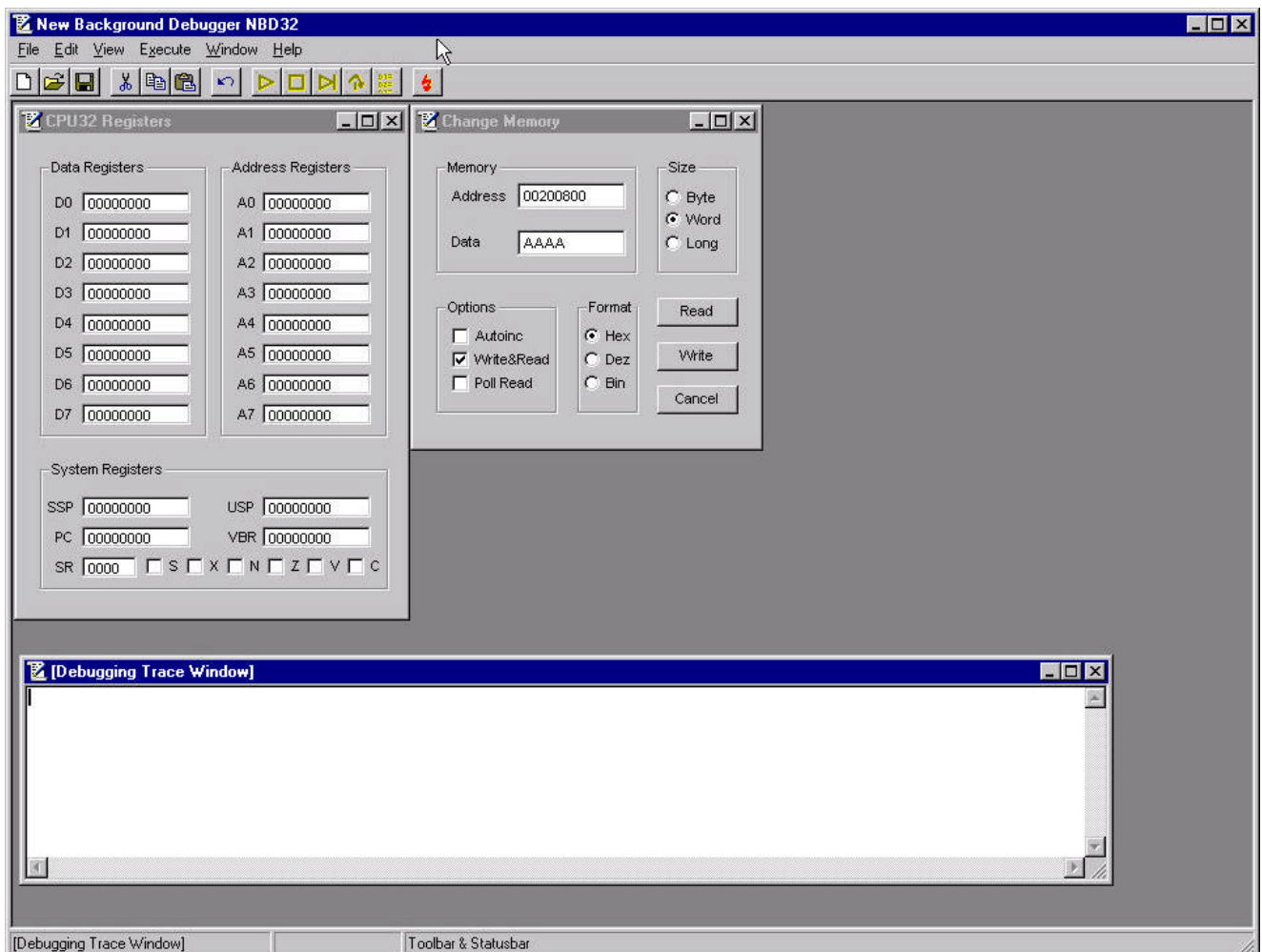


Abbildung 13: Ansicht NBD32, vollständig konfiguriert

Wenn sie zufrieden sind, kann unter „View/Save Workplace“ die gemachte Einstellung gesichert werden.

Die Möglichkeiten und Bedeutungen werden wir anhand von Beispielen lernen.

Das Registerfenster

Im Registerfenster können die Register der CPU32 geändert werden. Ein Ändern wird durch ein direktes überschreiben des Wertes erreicht.

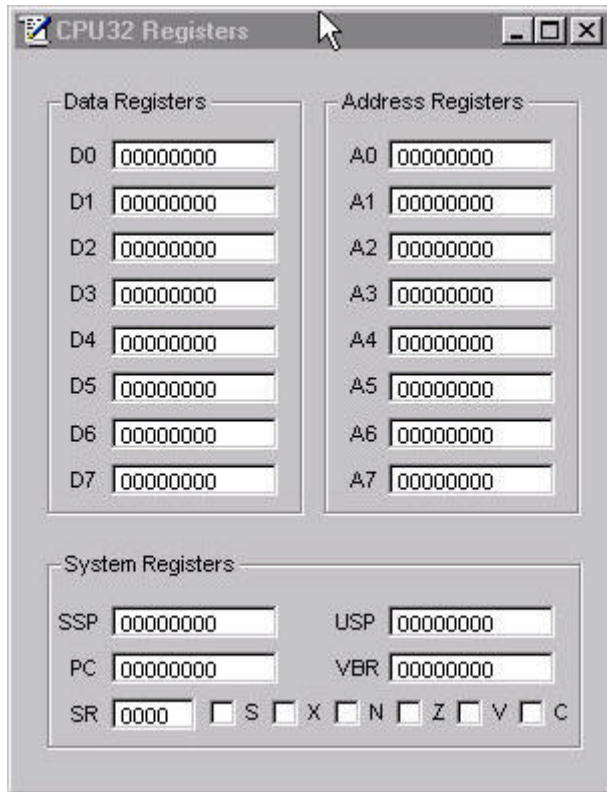


Abbildung 14: CPU32 Register Fenster

Das Change Memory Fenster

Im ChangeMemory Feld kann jede beliebige Speicherzelle geändert werden. Dabei ist es egal, ob es sich um internes Memory der CPU, Externes Memory oder um I/O-Bereiche handelt.

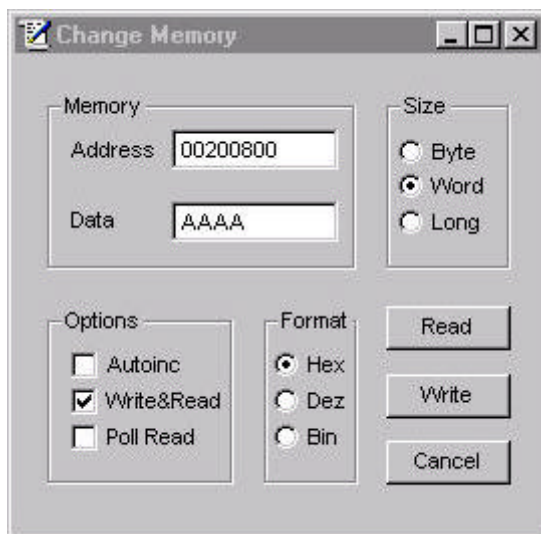


Abbildung 15: Change Memory Fenster

Die Optionen sind nur für erfahrene Benutzer interessant. Als Empfehlung schlage ich vor, alle Felder leer zu lassen.

Schreiben aufs Display

Zum Schreiben von Daten aufs Display ist folgendes Vorgehen nötig:

- Adresse 200200 in das „Address“ – Feld eintragen.
- Daten, z.B. A3 in das „Data“-Feld eintragen.
- Datengrösse Byte in „Size“ wählen.
- Knopf „Write“ drücken.

Lesen von den Tasten

Zum Lesen von Tastendaten

- Adresse 200000 in das „Address“ – Feld eintragen.
- Datengrösse Byte in „Size“ wählen.
- Knopf „Read“ drücken.
- → in „Data“ erscheint der Wert der unteren Tastenreihe.

Realer Zugriff auf die Hardware

Um das „echte“ Board anzuschliessen, muss das Bord erst angeschlossen werden gemäss Angaben im Script Kapitel 4.1. Anschliessend muss unter Execute/Options das Kästchen „Simulation only“ leer sein. Die Parallelport-Adresse sollte korrekt eingestellt sein (Für Teko: 0x0378). Da der „echte“ Prozessor sein I/O Bereich nicht kennt, muss erst die Chip Select Logik für das Display und die Tastatur korrekt eingestellt werden durch die folgende Prozedur:

- Reset des Boards durch die grosse weisse Taste.
- Adresse FFFA54 in „Address“-Feld von Change Memory eingeben
- Datum 2000 ins „Data“-Feld und „Size“ auf Word stellen.
- Knopf „Write“ drücken
- Adresse FFFA56 in „Address“-Feld von Change Memory eingeben
- Datum 5CF0 ins „Data“-Feld und „Size“ auf Word stellen.
- Knopf „Write“ drücken

Ab jetzt kann auf die I/O wie bei der Simulation zugegriffen werden.

Die Scriptsprache

Zum einfacheren Konfigurieren der Hardware steht eine Scriptsprache zur Verfügung. Mit einem normalen Editor kann ein solches Script für Konfigurationseinstellungen geschrieben werden. Folgende Befehle stehen zur Verfügung:

- MW.B [Address] [Data] Schreibt das Byte [Data] an Adresse [Address]
- MW.W [Address] [Data] Schreibt das Word [Data] an Adresse [Address]

- MW.L [Address] [Data] Schreibt das Longword [Data] an Adresse [Address]
- MR.B [Address] Liest ein Byte von Adresse [Address]
- MR.W [Address] Liest ein Word von Adresse [Address]
- MR.L [Address] Liest ein Longword von Adresse [Address]
- RW [Register] [Data] Schreibt das Datum [Data] in Register [Register]
- RR [Register] Gibt das Register [Register] aus
- * Kennzeichnung für einen Kommentar (bis Zeilenende)

Das Script muss dann abgespeichert werden und muss die Endung .scf (ScriptFile) tragen. Unter dem Menu Execute/Script kann dann ein solches Script ausgeführt werden. Im Debuggingfenster kann die Ausführung des Scripts beobachtet werden.

Beispiel eines Scripts:

Das folgende Scripts initialisiert die Chip-Select Logik für die I/O, schreibt AFFE ins Display und setzt Register D2 auf 12345678.

```
MW.W 00FFFA54 2000 *Periferie CS
MW.W 00FFFA56 5CF0
MW.B 00200300 AF *AFFE in Display
MW.B 00200200 FE
RW D2 12345678 *Register ändern
```

Das folgende Script „Initscript.scf“ werden wir in Zukunft für die totale Initialisierung des Boards benutzen.

```
MW.W 00FFFA04 0400 *Systemclock
MW.B 00FFFA21 06 *SW Watchdog off, Busmonitor ein
MW.W 00FFFA44 2ABF *Chipselect on
MW.W 00FFFA46 02A9
MW.W 00FFFA48 1007
MW.W 00FFFA4A 9CB0
MW.W 00FFFA54 2000 *Periferie CS
MW.W 00FFFA56 5CF0
MW.W 00FFFB00 0000 *Internes RAM aktivieren
MW.W 00FFFB02 2008
MW.W 00FFFB04 2008
RW PC 00200800 *PC an Start RAM
RW SSP 00200FF0 *SSP am Ende RAM
RW USP 00200F80
RW SR 00000000 *Switch to USER Mode
```

Der Taskbar

Die rechte Hälfte des Taskbars entspricht dem Menu Execute. Die Schalter werden aber erst aktiv, wenn das Debugging Window aktiv ist.



Abbildung 16: Der Taskbar

Das Play-Symbol entspricht dem Menüpunkt Run. Danach folgen Stop, Trace, Step over, Register to Log und Reset.

Das Menu Execute

R <u>u</u> n	F5
S <u>t</u> op	F6
T <u>r</u> ace	F2
Step <u>O</u> ver	F3
Regs to <u>L</u> og	F4
S <u>e</u> t Breakpoint	
K <u>i</u> ll Breakpoint	
R <u>e</u> set <u>C</u> hip	
E <u>x</u> ecute Script	
<u>D</u> ownload File	
<u>O</u> ptions	

Abbildung 17: Menu Execute

Im Menu Execute sind alle Befehle zur Programmausführung untergebracht. Bitte beachten sie, dass nur die beschriebenen Befehle eine Funktion besitzen.

- Run Startet ein Programm ab Adresse PC. Achten sie darauf, dass das Programm auch wirklich an dieser Adresse beginnt und ein Ende hat.
- Stop Stoppt die Ausführung eines Programms. Das Programm kann mit Run und Trace weitergeführt werden.
- Trace Das Programm wird im Einzelschrittmodus durchgeführt. Im Debugging Fenster und dem Registerfenster kann der Ablauf verfolgt werden.
- Regs to.. Fügt die Register in den Debugging-Log ein.
- Reset.. Reset des Boards oder der Simulation. Achtung, Board muss dann neu konfiguriert werden!
- Execute Execute Script wurde bereits früher erklärt. Führt einen Script aus.
- Download Lädt ein Programm. Das File muss ein S19-File vom Assembler AS32 sein.

Damit sind die wichtigsten Teile erklärt. Ich wünsche Ihnen viel Spass an diesem Tool und hoffe dass Sie mir die eventuellen Programmfehler verzeihen. Für Anregungen und Korrekturen bin ich jederzeit dankbar.

4.4 Assembler AS32 von Motorola

Eine CPU versteht nur Maschinencode, dies sind bestimmte Bitfolgen, die eine entsprechende Operation auslösen. Diese Bitfolgen werden Opcodegenannt. Bei der CPU32 sind diese 2, 4, 6, ..., 12 Bytes lang. Da eine Programmierung, die nur auf Zahlen beruht, sehr mühsam und undurchschaubar ist, gibt es Werkzeuge, die einen Befehl in Maschinencode übersetzen. Diese Tools nennt man Assembler, den Vorgang des Übersetzens assemblieren.

Ein Assembler kennt 2 verschiedene Befehlsarten. Zum einen die sogenannten **Mnemonics**, die Befehle für die CPU, die dann in Maschinencode gewandelt werden. Zum anderen die **Directiven**, die Befehle für den Assembler. Im Zusammenhang mit Assemblern spricht man von **Labels**, **Konstanten**, **Operationen**, **Kommentaren**, etc..

Häufig genutzte Directiven:

Kommentare

Kommentare werden immer durch das Zeichen „*“ eingeleitet und definieren den Start eines Kommentars. Dieser Kommentar ist dann immer bis zum Zeilenende gültig.

Beispiel:

*Das ist ein Kommentar

Labels

Labels definieren eine Adresse, an der der nächste Befehl steht. Als Labels können beliebige Zeichenfolgen stehen. Bedingung ist, dass sie mit einem Buchstaben beginnen. Normalerweise wird nach dem Label ein „:“ gesetzt. Dies ist nicht Bedingung, aber sehr hilfreich.

Beispiel:

Label: *Label enthält nun die Adresse, an der diese Programmzeile im Memory sein wird.

Konstantendeklaration

Es ist hilfreich, wenn Adressen oder Daten, beispielsweise ein I/O-Port als Konstante definiert wird, damit bei einer Hardwareänderung nur die Konstante verändern werden muss und nicht der gesamte Code. Zudem wird der Quelltext viel einfacher lesbar. Eine Konstantenzuweisung wird mit EQU gemacht. Zur Deklaration, um welches Zahlensystem es sich handelt, werden die Zeichen „\$“ für hexadezimal und „%“ für binär benutzt. Steht kein Zeichen, so wird dezimal verwendet.

Beispiel:

I_O_Port EQU \$4001

Muster EQU %10101010

Es können auch mathematische Ausdrücke verwendet werden, zulässig sind:

+, -, *, /, & (AND), | (OR), ^ (Exclusive OR) und ~ (Not). Zu beachten ist, dass diese Operationen beim Assemblieren durchgeführt werden und keinesfalls im Programm!!!

Definition der Startadresse

Zum Definieren der Startadresse des Programms muss dem Assembler vor Beginn der ersten Instruktion mit der ORG Directive gesagt werden, wo das Programm später im Speicher steht.

Beispiel:

```
ORG $200800      *Start des Programms im On-Chip RAM
```

Speicherstellen mit bestimmtem Inhalt füllen

Vielfach ist es erwünscht, Speicherzellen mit einem gewissen Wert zu füllen, zum Beispiel für einen String oder eine Tabelle. Dazu wird die Directive DC.s benutzt. Das s steht wahlweise für B (Byte), W (Word) oder L (Longword).

Beispiel:

```
String:      DC.B 'das ist ein String',0      *Definiert den String und fügt am Ende eine 0 an.
Zahl:       DC.W $4543, %1000110, 453      *Definiert 3 16-Bit Zahlen
```

Adresse gerade machen

Die CPU32 besitzt eine Vorliebe für gerade Adressen. Dies ist darauf zurückzuführen, dass alle Opcodes Vielfache von Words sind und deswegen nie ungerade Adressen entstehen können. Durch die Definition eines Bytes mit DC.B kann aber eine ungerade Adresse entstehen. Dies kann durch EVEN korrigiert werden. EVEN fügt, falls notwendig, ein Byte ein und macht so die nächste Adresse wieder gerade.

Beispiel:

```
String:      DC.B 'das ist ein String',0      *Definiert den String und fügt am Ende eine 0 an.
              EVEN                          *Nächste Adresse liegt an einer geraden Zahl
```

Drucken des Source Listings (in ein File)

Durch die Directive OPT I kann das assemblierte File mit den zugehörigen Opcodes betrachtet werden.

Beispiel:

```
OPT I      *Einschalten des Source Listing Ausdrucks.
```

Damit sind die wichtigsten Direktiven erläutert. Als Beispiel für den Vorgang des Assemblierens dient ein Programm, das den Zustand der hinteren Tastenreihe auf den letzten beiden Stellen des Displays anzeigt. Dazu müssen folgende Zeilen mit einem Texteditor eingegeben werden (Empfehlung: Notepad oder DOS Edit, keinesfalls Word oder ähnliche Programme benutzen!).

```

                OPT I
*Programm zum Testen des Assemblers
                ORG  $200800      *Start im RAM
Display        EQU  $200200
Tasten        EQU  $200100
*
Start:        MOVE.W Tasten,D0    *Wert der Tasten nach D0
              MOVE.W D0,Display  *Nun D0 nach Display
              BRA   Start       *Zurück zum Start, Endlosschleufe
*Ende des Programms
```

Das Programm kann unter dem Namen test.asm abgespeichert werden. Jetzt ist es möglich, den Assembler aufzurufen. Dieser befindet sich im Verzeichnis AS32 der Diskette, bzw. auf der Harddisk des Rechners.

```
A:\as32>as32 test.asm
```

```

                                OPT 1
                                *Programm zum Testen des Assemblers
00200800                        ORG $200800          *Start im RAM
00200200      Display            EQU    $200200
00200100      Tasten             EQU    $200100
                                *
00200800 3039 0020 0100 Start:    MOVE.W Tasten,D0          *Wert der Tasten nach D0
00200806 33c0 0020 0200          MOVE.W D0,Display        *Nun D0 nach Display
0020080c 60f2                    BRA     Start *Zurück zum Start, Endlosschleife
                                *Ende des Programms

====      0 Error(s)
====      0 Warning(s)

```

```
C:\MC683XX\AS32>
```

Falls das obige Bild nicht erscheinen sollte, liegt vermutlich ein Fehler im Text, der im Editor eingegeben wurde, vor. Bei der Überprüfung des Texts ist insbesondere auf die Abstände vom Rand zu achten.

Im Verzeichnis sollte sich nun ein File mit Namen test.s19 befinden. Dies ist der Maschinencode im sogenannten Motorola-S-Format. Der Debugger kann jetzt gestartet und vorbereitet werden. Es ist notwendig, ein paar Grundeinstellungen in den MCU-Registern vorzunehmen. Da mehrere Register betroffen sind, wurden diese in einem Makro zusammengefasst und können so durch den Befehl

BD32>DO TEKO

aufgerufen werden. Der Inhalt des Makros ist (ASCII-File mit Namen TEKO):

```

* Script zum setzen der MCU Register
* May 1999 By Jann P. Oesch, TEKO Bern
*
*Systemclock einstellen
MM $FFFA04
$0400
.
*Software Watchdog abhalten, Bus Monitor Ein
MM $FFFA21;b
6
.
*Chip-Select Freischalten und Boot-CS einstellen
MM $FFFA44
$2ABF
$2A9
$1007
$6CB0
.
*Setzen der Periferie - Chip Selects
MM $FFFA54
$2000
$5CF0
.
*Internes RAM aktivieren (Adresse $200800)
MM $FFFB00
$0000
$2008
$2008
.
*PC am Start
RM PC $200800
*SSP am Ende
RM SSP $200FF0
*USP Bereich $200F00 - $200F80
RM USP $200F80
*Switch to User Mode
RM SR $0

```

Durch die Verwendung dieses Makros können in Zukunft alle Eingriffe in die Systemregister unterlassen werden. Zum Laden des Programms muss folgendes eingegeben werden:

```
BD32>LO test.s19
```

Wichtig ist, dass sich test.s19 im aktuellen Pfad befindet, andernfalls muss dieser mit eingegeben werden.

```
BD32->lo ..\as32\test.s19
...
Download completed OK - 3 records read
BD32->
```

Ist diese Aktion erfolgreich abgeschlossen, kann mittels des DASM - Befehls des Debuggers das Programm diassembliert und kontrolliert werden.

```
BD32->dasm $200800
00200800 3039 0020 0100          MOVE.W  $200100.L,D0
00200806 33C0 0020 0200          MOVE.W  D0,$200200.L
0020080C 60F2                      BRA     $200800.B
BD32->
```

Mit dem Befehl rd sollte kontrolliert werden, ob der Programm Counter (PC) ebenfalls auf das Programm zeigt. Wenn nötig muss mit rm pc \$200800 korrigiert werden.

```
BD32->rd
D0-7 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
A0-7 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00200F80
PC 00200800      USP 00200F80      SFC 00000005      SR 10S--210---XNZVC
VBR 00000000      SSP 00200FF0      DFC 00000005      0000000000000000
00200800 3039 0020 0100          MOVE.W  $200100.L,D0
BD32->
```

Anschliessend kann mit dem Befehl go das Programm gestartet werden.

```
BD32->go
BD32->
```

Die Funktion des Programms kann getestet werden, indem die obrige Tastenreihe bedient und dabei die Anzeige beobachtet wird.

Mit dem Befehl STOP wird angehalten, der PC wieder an den Start gesetzt und das Programm läuft mit dem Befehl T im Einzelschrittmodus (Trace) durch. Mit jedem Druck auf die Return-Taste wird ein weiterer Befehl verarbeitet. Das Register D0 wird oben angezeigt.

```
BD32->t
00200806 33C0 0020 0200          MOVE.W  D0,$200200.L
BD32->
0020080C 60F2                      BRA     $200800.B
BD32->
00200800 3039 0020 0100          MOVE.W  $200100.L,D0
BD32->
00200806 33C0 0020 0200          MOVE.W  D0,$200200.L
BD32->
0020080C 60F2                      BRA     $200800.B
BD32->
```

Mit diesen Ausführungen sollte es nun möglich sein, ein Programm im Editor zu schreiben, es danach mit Hilfe des BD32 ablaufen zu lassen und im Einzelschrittmodus zu verfolgen.

5 Programmierung

In diesem Kapitel wird auf die Programmierung der CPU32 eingegangen. Für die genauen Befehle und Befehlsbeschreibungen wird auf das CPU32 Reference Manual verwiesen, in dem alle Befehle detailliert erläutert sind.

5.1 Befehlssatz

Die CPU32 besitzt einen umfangreichen Befehlssatz, der hier kurz aufgelistet wird. Die genauen Auswirkungen, insbesondere auf die Flags, können im Reference Manual der CPU32 nachgeschlagen werden.

ABCD	Add Decimal with Extend	BCD – Zahlenaddition (BCD = Binary Coded Decimal)
ADD	Add	Addition
ADDA	Add Address	Addition mit Adressregister
ADDI	Add Immediate	Addition mit einer Konstanten
ADDQ	Add Quick	Schnelle Addition mit einer Konstanten im Bereich von 1,2..8
ADDX	Add with Extend	Addition mit Extended-Bit ($D_n = D_n + \text{Operand} + X$)
AND	Logical AND	Logische AND-Verknüpfung
ANDI	Logical AND Immediate	Logische AND-Verknüpfung mit einer Konstanten
ASL	Arithmetic Shift Left	Arithmetisches Linksschieben (identisch mit LSL)
ASR	Arithmetic Shift Right	Arithmetisches Rechtsschieben, Vorzeichen wird beibehalten
Bcc	Branch Conditionally	Bedingter (relativer) Sprung
BCHG	Test Bit and Change	Testen eines bestimmten Bits und von 0 auf 1, bzw. 1 auf 0 wechseln
BCLR	Test Bit and Clear	Testen eines Bits und löschen
BGND	Background	Anhalten des Programmablaufs zum Debuggen (BD32)
BKPT	Breakpoint	Spezieller Befehl für Breakpoints (benötigt spezifische Hardware)
BRA	Branch	Unbedingter relativer Sprung
BSET	Test Bit and Set	Testen eines Bits und setzen
BSR	Branch to Subroutine	Relativer Sprung in ein Subprogramm
BTST	Test Bit	Testen eines Bits
CHK, CHK2	Check Reg. Against Bounds	Vergleich eines Registers gegen Grenzen (spezieller Befehl)
CLR	Clear	Löschen eines Datenregisters (nicht bei Adressregistern)
CMP	Compare	Vergleichen
CMPA	Compare Address	Vergleichen mit Adressregister
CMPI	Compare Immediate	Vergleichen mit Konstante
CMPM	Compare Memory to Memory	Vergleichen von zwei Speicherinhalten
CMP2	Compare Reg. Against Bounds	Vergleichen gegen zwei Grenzen
DBcc	Decrement, Test and Branch	Dekrementieren (eins subtrahieren), Test und bedingter Sprung
DIVS, DIVSL	Signed Divide	Dividieren vorzeichenbehaftet
DIVU, DIVUL	Unsigned Divide	Dividieren ohne Vorzeichen
EOR	Logical Exclusive OR	Exklusiv-OR
EORI	Logical Exclusive OR Immed.	Exklusiv-OR mit Konstante
EXG	Exchange Registers	Austauschen von Registerinhalten
EXT, EXTB	Sign Extend	Vorzeichenerweiterung (Byte->Word->Longword)
LEA	Load Effective Address	Effektive Adresse bilden
LINK	Link and Allocate	Raum auf dem Stack schaffen (spezieller Befehl für Compiler)
LPSTOP	Low Power Stop	Stoppen des Prozessors und Warten auf ein externes Ereignis)
LSL	Logical Shift Left	Linksschieben, die freie Stelle wird mit '0' aufgefüllt
LSR	Logical Shift Right	Rechtsschieben, die freie Stelle wird mit '0' aufgefüllt
ILLEGAL	Take Illegal Instruction Trap	Illegale Instruktion, der Prozessor 'stürzt' kontrolliert ab
JMP	Jump	Unbedingter absoluter Sprung
JSR	Jump to Subroutine	Absoluter Sprung in ein Subprogramm
MOVE	Move	Verschieben von Daten
MOVE CCR	Move Condition Code Register	Verschieben von Daten in das CCR

MOVE SR	Move Status Register	Verschieben von Daten in das SR (nur im Supervisor-Mode)
MOVE USP	Move User Stack Pointer	Verschieben von Daten in den User-SP (nur im Supervisor-Mode)
MOVEA	Move Address	Verschieben von Daten in ein Adressregister
MOVEC	Move Control Register	Verschieben von Daten in das Control Reg. (nur im Supervisor-Mode)
MOVEM	Move Multiple Registers	„Retten“ von mehreren Registern
MOVEP	Move Peripheral	Verschieben von Daten byteweise
MOVEQ	Move Quick	Schnelles Laden von Konstanten in Datenregister (Bereich -128...127)
MOVES	Move Alternate Address Space	Verschieben von Daten in das SR (nur im Supervisor-Mode)
MULS	Signed Multiply	Vorzeichenbehaftet Multiplizieren
MULU	Unsigned Multiply	Vorzeichenlos Multiplizieren
NBCD	Negate Decimal with Extend	Vorzeichenwechsel von BCD-Zahlen
NEG	Negate	Vorzeichenwechsel (2er Komplement)
NEGX	Negate with Extend	Vorzeichenwechsel unter Berücksichtigung von X (2er Komplement)
NOT	Logical NOT	Logische NOT-Verknüpfung (Invertieren)
NOP	No Operation	Keine Operation (Debuggingzwecke)
OR	Logical Inclusive OR	Logische OR-Verknüpfung
ORI	Logical Inclusive OR Immed.	Logische OR-Verknüpfung mit Konstante
PEA	Push Effective Address	Effektive Adresse auf den Stack legen
RESET	Reset External Devices	Reset der Peripheriebausteine
ROL	Rotate Left	Rotieren links
ROR	Rotate Right	Rotieren rechts
ROXL	Rotate with Extend Left	Rotieren links durch das X-Bit
ROXR	Rotate with Extend Right	Rotieren rechts durch das X-Bit
RTD	Return and Deallocate	Spezieller Befehl für Compiler
RTE	Return from Exception	Beenden von Exceptions (privilegierter Befehl)
RTR	Return and Restore Codes	Beenden von Exception und Beibehalten des Supervisor-Mode
RTS	Return from Subroutine	Beenden einer Subroutine (Gegenstück zu BSR und JSR)
SBCD	Subtract Decimal with Extend	Subtrahieren von BCD-Zähler
Scc	Set Conditionally	Bedingtes Setzen eines Bytes gemäss Flags
STOP	Stop	Warten auf Interrupts
SUB	Subtract	Subtrahieren
SUBA	Subtract Address	Subtrahieren von Adressregistern
SUBI	Subtract Immediate	Subtrahieren von Konstanten
SUBQ	Subtract Quick	Schnelle Subtraktion mit einer Konstanten im Bereich von 1,2..8
SUBX	Subtract with Extend	Subtraktion mit Extended-Bit ($D_n = D_n + \text{Operand} + X$)
SWAP	Swap Register Words	Tauschen von Low-Word und High-Word
TAS	Test Operand and Set	Testen und Setzen
TBLS, TBLSN	Table Lookup and Interpolate S	Interpolationsbefehl mit Hilfe einer Tabelle (vorzeichenrichtig)
TBLU, TBLUN	Table Lookup and Interpol. US	Interpolationsbefehl mit Hilfe einer Tabelle (ohne Vorzeichen)
TRAP	Trap	Software-Interrupt (Umschalten auf Supervisormode)
TRAPcc	Trap Conditionally	Bedingter Software-Interrupt
TRAPV	Trap on Overflow	Softwareinterrupt bei Überlauf
TST	Test Operand	Testen eines Operanden
UNLK	Unlink	Gegenstück zu Link (für Compiler)

Die meisten Befehle können zur Byte, Word und Longword-Verarbeitung benutzt werden. Dazu muss dem Assembler mittels .B, .W oder .L mitgeteilt werden, auf was sich die Operation bezieht. Aus ADD wird somit:

- ADD.B Addition für ein Byte, es werden nur 8 Bit des Registers benutzt
- ADD.W Addition für ein Word, es werden 16 Bit des Registers benutzt
- ADD.L Addition für ein Longword, es werden alle 32 Bit des Registers benutzt

5.2 Adressierungsarten

Jeder der oben benannten Befehle kann mit einer oder mehreren Adressierungsarten benutzt werden. Welche Adressierungsart im einzelnen zulässig ist, kann dem Reference-Manual entnommen werden. Im folgenden werden die verfügbaren Adressierungsarten vorgestellt. Dazu werden Symbole benutzt:

- Rn oder Xn steht für ein Daten- oder Adressregister
- Dn steht für ein Datenregister
- An steht für ein Adressregister
- PC steht für den Programm - Counter
- d8 steht für eine 8-Bit Konstante (vorzeichenbehaftet)
- d16 steht für eine 16-Bit Konstante (vorzeichenbehaftet)
- d32 steht für eine 32-Bit Konstante (vorzeichenbehaftet)
- EA steht für Effective Adress, das ist die Adresse die schlussendlich adressiert wird.
- s steht für die Breite des Registers, bei .W wird Word benutzt, bei .L ein Longword

Bei den Beispielen wird immer der MOVE Befehl benutzt und ein Datum in das Register D4 geladen. Die Adressierungsart kann so gut demonstriert werden.

Inherent

Ein inherenter Befehl besitzt keine weiteren Parameter. Durch den Befehl selbst ist die Operation klar. Typische Beispiele sind die Befehle NOP, RTS und RESET.

Register Direkt

Assemblersyntax: = Rn

Der Operand wird direkt durch das Register bestimmt. Der Operand entspricht dem Inhalt des Registers. Dies ist die meistbenutzte Adressierungsart. Einige Befehle unterstützen nur diese Adressierungsart, zum Beispiel SWAP Dn oder EXT Dn.

Beispiel:

Register vor Ausführung des Befehls:	D0: \$12345678	D4: \$AA55AA55
Assemblerbefehl:	MOVE.L D0,D4	
Reg. nach Ausführung des Befehls:	D0: \$12345678	D4: \$12345678

Adressregister indirekt*Assemblersyntax: (An)*

Der Operand wird indirekt durch das Adressregister bestimmt. Im Speicher wird die Adresse geändert/geholt, die an Adresse An steht.

Beispiel:

```

Register vor Ausführung des Befehls: A0: $00200900    D4: $AA55AA55
Speicher:                               $002008FE: $0123
                                         $00200900: $4567
                                         $00200902: $98AB
Assemblerbefehl:                        MOVE.L  (A0),D4
Reg. nach Ausführung des Befehls:  A0: $00200900    D4: $456798AB

```

Adressregister indirekt mit Postinkrement*Assemblersyntax: (An)+*

Der Operand wird indirekt durch das Adressregister bestimmt. Im Speicher wird die Adresse geändert/geholt, die an Adresse An steht. Zusätzlich wird das Adressregister **nach** Beendigung der Operation um 1 bei Byte, 2 bei Word und 4 bei Longwordoperationen inkrementiert.

Beispiel:

```

Register vor Ausführung des Befehls: A0: $00200900    D4: $AA55AA55
Speicher:                               $002008FE: $0123
                                         $00200900: $4567
                                         $00200902: $98AB
Assemblerbefehl:                        MOVE.L  (A0)+,D4
Reg. nach Ausführung des Befehls:  A0: $00200904    D4: $456798AB

```

Adressregister indirekt mit Predektrement*Assemblersyntax: -(An)*

Der Operand wird indirekt durch das Adressregister bestimmt. Im Speicher die Adresse geändert/geholt, die an Adresse An steht. Zusätzlich wird das Adressregister **vor** der Operation um 1 bei Byte, 2 bei Word und 4 bei Longwordoperationen dekrementiert.

Beispiel:

```

Register vor Ausführung des Befehls: A0: $00200904    D4: $AA55AA55
Speicher:                               $002008FE: $0123
                                         $00200900: $4567
                                         $00200902: $98AB
Assemblerbefehl:                        MOVE.L  -(A0),D4
Reg. nach Ausführung des Befehls:  A0: $00200900    D4: $456798AB

```

Adressregister indirekt mit Offset (16 Bit vorzeichenbehaftet)*Assemblersyntax: d16(An)*

Der Operand wird indirekt durch das Adressregister und das Displacement bestimmt. Im Speicher wird die Adresse geändert/geholt, die an Adresse (An+d16) steht. Der Wert des Adressregisters wird nicht verändert.

Beispiel:

Register vor Ausführung des Befehls: A0: \$00200878 D4: \$AA55AA55
 Speicher: \$00200878: \$CCAA
 \$0020087A: \$CCAA
 \$00200900: \$4567
 \$00200902: \$89AB
 Assemblerbefehl: MOVE.L \$88(A0),D4
 Reg. nach Ausführung des Befehls: A0: \$00200878 D4: \$456789AB

Adressregister indirekt mit Index und Offset (8 Bit vorzeichenbehaftet)*Assemblersyntax: d8(An,Xn.s)*

Der Operand wird indirekt durch das Adressregister, das Register Xn und das Displacement bestimmt. Im Speicher die Adresse geändert/geholt, die an Adresse (An+Xn.s+d16) steht. Der Wert des Adressregisters wird nicht verändert. Wenn Xn.W angegeben wird, werden lediglich die unteren 16 Bit des Registers beachtet.

Beispiel:

Register vor Ausführung des Befehls: A0: \$00200878 D0: \$AA550030 D4: \$AA55AA55
 Speicher: \$00200878: \$CCAA
 \$0020087A: \$CCAA
 \$00200900: \$4567
 \$00200902: \$89AB
 Assemblerbefehl: MOVE.L \$58(A0,D0.W),D4
 Reg. nach Ausführung des Befehls: A0: \$00200878 D0: \$AA550030 D4: \$456789AB

Adressregister indirekt mit Index und Skalierung*Assemblersyntax: d8(An,Xn.s * k)*

Der Operand wird indirekt durch das Adressregister, das Register Xn, den Skalierungsfaktor k und das Displacement bestimmt. Im Speicher wird die Adresse geändert/geholt, die an Adresse (An+Xn.s*k+d16) steht. Der Wert des Adressregisters wird nicht verändert. Wenn Xn.W angegeben wird, werden nur die unteren 16 Bit des Registers beachtet. Der Skalierungsfaktor k kann 1, 2, 4 oder 8 sein. Diese Adressierungsart wird nur selten benutzt da sie dem Befehlssatz der MC68k Familie später zugefügt worden ist.

Beispiel:

Register vor Ausführung des Befehls: A0: \$00200878 D0: \$AA55000C D4: \$AA55AA55
 Speicher: \$00200878: \$CCAA
 \$0020087A: \$CCAA
 \$00200900: \$4567
 \$00200902: \$89AB
 Assemblerbefehl: MOVE.L \$58(A0,D0.W*4),D4
 Reg. nach Ausführung des Befehls: A0: \$00200878 D0: \$AA55000C D4: \$456789AB

Absolute Adressierung

Assemblersyntax: \$XXXX

Der Operand wird indirekt durch die Speicherstelle XXXX gegeben. Im Speicher wird die Adresse geändert/geholt, die an Adresse XXXX steht.

Beispiel:

```
Register vor Ausführung des Befehls: D4: $AA55AA55
Speicher:                               $002007FE: $CCAA
                                           $00200800: $4567
                                           $00200802: $98AB
Assemblerbefehl:                        MOVE.L  $00200800,D4
Reg. nach Ausführung des Befehls:      D4: $456798AB
```

Immediate

Assemblersyntax: #XXXX

Der Operand wird direkt angegeben, das heisst, dass es sich um eine Konstante handelt.

Beispiel:

```
Register vor Ausführung des Befehls: D4: $AA55AA55
Speicher:                               $00200800: $CCAA
                                           $00200802: $98AB
Assemblerbefehl:                        MOVE.L  #$00200800,D4
Reg. nach Ausführung des Befehls:      D4: $00200800
```

In diesen Ausführungen wurde die PC-Relative Adressierung bewusst vernachlässigt, da diese nur selten benutzt wird und durch die vorgestellten Adressierungsarten vollständig ersetzt werden kann.

5.3 Standard - Assemblerkonstrukte (Schlaufen, Bedingungen)

Für eine Programmieraufgabe gibt es meist unendlich viele Lösungen. Hier soll an einigen Beispielen gezeigt werden, wie die häufigsten Probleme gelöst werden können. Diese Beispiele können in die eigenen Programme implementiert werden und das Rad muss nicht zweimal erfunden werden.

Warteschlaufe

Eine Warteschlaufe besteht aus einer normalen Schlaufe, die jedoch keinen Inhalt besitzt. Die Wartezeit kann mit der Taktfrequenz der CPU berechnet werden. Wir werden diese jedoch experimentell ermitteln.

```

...                               *letzte Befehlszeile vor dem Warten
MOVE.L  #$100000,D0              *Hexadezimal 100'000 wird
Loop: SUBQ.L  #1,D0                *nun dekrementiert
      BNE.....Loop              *bis auf 0, wenn nicht 0 dann zurück nach Loop
...                               *Erste Programmzeile nach dem Warten

```

Sonstige Schleifen

Schleifen nach dem FOR/NEXT oder DO/WHILE Prinzip werden alle nach dem gleichen Muster aufgebaut. Es gibt aber einen Unterschied die Anzahl der Durchläufe betreffend. Bis 32'000 Durchläufe kann folgende Sequenz benutzt werden:

```

...                               *letzte Befehlszeile vor der Schlaufe
MOVE.W  #99,D0                   *Dez. 99 die Schlaufe wird 100 mal durchgeführt
Loop: ...                         *hier steht der Schlaufeninhalt
      DBF.....D0,Loop            *D0 dekrementieren, wenn D0 positiv dann nach Loop
...                               *Erste Programmzeile nach der Schlaufe

```

Für mehr als 32'000 Durchläufe muss folgendes Konstrukt benutzt werden:

```

...                               *letzte Befehlszeile vor der Schlaufe
MOVE.L  #100000,D0              *Dezimal 100'000 - 100'000 Durchläufe
Loop: ...                         *hier steht der Schlaufeninhalt
      SUBQ.L  #1,D0                *D0 nun dekrementieren
      BNE.....Loop              *bis auf 0, wenn nicht 0 dann zurück nach Loop
...                               *Erste Programmzeile nach der Schlaufe

```

Bedingungen

Vielfach wird während des Ablaufs eines Programms ein Vergleich gemacht. Dem Resultat entsprechend wird ein anderer Programmablauf gewünscht. Dafür stehen die Bcc - Befehle zur Verfügung. Im gesamten Sortiment von gibt es 6 - 8 wichtige:

- BEQ Verzweigt, wenn das Z-Flag gesetzt ist, also wenn die letzte Operation 0 war
Bei CMP: Springen, wenn die verglichene Zahl genau identisch ist
- BNE Gegenstück zu BEQ, springt, wenn letzte Operation nicht 0 gab
Bei CMP: Springen, wenn die verglichene Zahl nicht identisch ist
- BMI Verzweigt, wenn das N-Flag gesetzt ist, also die letzte Operation eine negative Zahl war
- BPL Gegenstück zu BMI, springt bei einer positiven Zahl
- BLO/BCS Nach CMP: Springen, wenn die verglichene Zahl kleiner war
- BLS Nach CMP: Springen, wenn die verglichene Zahl kleiner oder gleich war
- BHSS/BCC Nach CMP: Springen, wenn die verglichene Zahl grösser oder gleich war
- BHI Nach CMP: Springen, wenn die verglichene Zahl grösser war

Eine typische Anwendung ist ein Vergleich:

```

...                               *letzte Befehlszeile vor Abfrage
CMP.W #10,D0                     *Vergleichen mit 10
BEQ   Then                       *Springen wenn genau 10
Else: ...                         *Programm das Abläuft wenn D0 nicht 10
BRA   EndIf                      *Ende des Else-Zweiges
Then: ...                         *hier steht das Programm wenn D0=10
EndIf ...                         *Erste gemeinsame Befehlszeile nach dem Vergleich

```

Eine Abfrage nach kleiner/gleich/grösser sieht so aus:

```

...                               *letzte Befehlszeile vor Abfrage
CMP.W #10,D0                     *Vergleichen mit 10
BEQ   Equal                      *Springen wenn genau 10
BHI   Gross                      *Springen, wenn grösser als 10
Klein:...                        *Programm das Abläuft, wenn D0 < 10
BRA   EndIf                      *Ende des Else-Zweiges
Equal:...                        *hier steht das Programm, wenn D0=10
BRA   EndIf                      *Ende des Else-Zweiges
Gross:...                        *hier steht das Programm, wenn D0>10
EndIf ...                        *Erste gemeinsame Befehlszeile nach dem Vergleich

```

6 Der Stack

Der Stack ist ein spezieller Bereich im Speicher, der von der CPU selbständig verwaltet wird. Der Anwender kann mit Hilfe des sogenannten Stackpointers (SP) auf diesen Bereich zugreifen. Der Stack wird einerseits von der CPU benutzt, um Rücksprungadressen von Subprogrammen und bei Interrupts zwischenspeichern, und andererseits vom Programmierer für die temporäre Zwischenspeicherung von Registern benutzt.

Der Speicherbereich des Stacks wird vom Programmierer in der Systeminitialisierung definiert, bei uns wird dies durch das Makro TEKO erledigt und unser Stack belegt den Bereich von \$200F00 - \$200F80. Der Stack funktioniert nach dem LIFO (Last In, First Out) Prinzip. Das bedeutet, dass die Daten im Stack von oben nach unten abgelegt werden. Für unser Stackbereich heisst dies, dass das erste abgelegte Longword an Adresse \$200F7C liegt. Die höchste Adresse, \$200F80 wird nie beschrieben.

6.1 Der Stack als Zwischenspeicher

Die CPU32 besitzt 8 Daten und 7 Adressregister, die für die Programmierung frei benutzt werden können. (Das 8. Adressregister entspricht dem Stackpointer und sollte nicht anders benutzt werden.) Bei kleinen Programmen reicht dies im Normalfall auch aus, aber schon bei Programmen mit 50-100 Zeilen Code sollte der Inhalt eines Registers temporär zwischengespeichert und das Register für eine andere Operation genutzt werden. In diesem Fall kommt der Stack zum Zug. Der Inhalt beliebiger Register kann auf den Stack zwischengespeichert, die Register für andere Aufgaben benutzt und am Schluss die Register wieder mit den alten Werten geladen werden. Dafür steht der MOVEM – Befehl, mit dem ein oder mehrere Register gesichert oder zurückgeholt werden können.

Beispiel:

```
MOVEM.L D0-D2/D5/A0/A3-A5,-(A7)
```

Die Register D0, D1, D2, D5, A0, A3, A4 und A5 werden auf den Stack = A7 gelegt, anschliessend wird der Stackzeiger um $4 \times 8 = 32$ Byte tiefer sein. Werden die Register wieder geholt, wird folgender Befehl benutzt:

```
MOVEM.L (A7)+,D0-D2/D5/A0/A3-A5
```

Wird nur ein Register auf den Stack gesichert, kann statt MOVEM.L D1,-(A7) auch MOVE.L D1,-(A7) benutzt werden.

!Wichtig!: Die Register werden bei Listen nicht in der aufgeführten Reihenfolge zum Stack geschrieben, sondern als erstes das Register A6, dann A5, A5...A0,D7....D1 und am Schluss D0. Beim Zurücklesen ist die Reihenfolge gerade umgekehrt, also als erstes D0 und als letztes A6.

Als nächstes wird der Stack – Speicherbereich untersucht. Um das Speichern und Holen von Daten zu erläutern, soll die Stackfunktion anhand eines Beispiels veranschaulicht werden:

Beispiel:

Register: D0: \$12345678, D1: \$9ABCDEF0, A0: \$00200400, A7: \$00200780 (Stack-Pointer, mit >< markiert)

	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F	80	
MOVE.L #11223344,D2	\$200770:	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	>XX<	
MOVE.L D2,-(A7)	\$200770:	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	>11<	22	33	44	XX	
MOVEM.L D0-D1/A0,-(A7)	\$200770:	>12<	34	56	78	9A	BC	DE	F0	00	20	04	00	11	22	33	44	XX
MOVEM.L (A7)+,D0-D1/A0	\$200770:	12	34	56	78	9A	BC	DE	F0	00	20	04	00	>11<	22	33	44	XX
MOVE.L (A7)+,D3	\$200770:	12	34	56	78	9A	BC	DE	F0	00	20	04	00	11	22	33	44	>XX<

Im letzten Befehl wird deutlich, dass beim Zurückholen der Registerinhalte nicht das gleiche Register benutzt werden muss. Allerdings sollten bei der Verwendung von Listen (D0-D3/A1-A3,A5) immer die gleichen Listenausdrücke verwendet werden, da sonst nicht gewährleistet ist, dass auch wirklich die richtigen Inhalte in die richtigen Register geraten.

Achtung!: Es muss in jedem Programm und Subprogramm die Bilanz speichern/holen aufgehen. Besondere Vorsicht ist bei Schleifen geboten, dass nicht ein MOVE.L Rn,-(A7) ohne zugehöriges MOVE.L (A7)+,Rn in der Schleife steht, da sonst der Stackbereich überläuft und das Programm abstürzt.

6.2 Subprogramme

Vielfach besteht der Wunsch, einzelne Programmteile von verschiedenen Stellen innerhalb des Programms anzusprechen und dann mit dem weiteren Programmablauf fortzufahren. Als einfaches Beispiel wird die Warteschleife aufgeführt. Der Programmteil, der dabei mehrfach verwendet wird, wird Subprogramm (Subroutine, Unterprogramm) genannt. Ein solches Subprogramm wird mit dem Befehl JSR oder BSR (Jump to Subroutine {absolut}, Branch to Subroutine {relativ}) gestartet. Welcher der beiden Befehle verwendet wird spielt keine Rolle, da der Ablauf identisch ist. Das Subprogramm wird mit dem Befehl RTS (Return From Subroutine) beendet. Nachfolgend wird an einem Beispiel (Blinklicht) die Verwendung von Subprogrammen erläutert.

Programm ohne Subs

```

        ORG $200800
LED     EQU $200400
Loop:   MOVE.B #$FF,LED
        MOVE.L #$100000,D0
Wt1:    SUBQ.L #1,D0
        BNE    Wt1
        MOVE.B #$00,LED
        MOVE.L #$100000,D0
Wt2:    SUBQ.L #1,D0
        BNE    Wt2
        BRA   Loop

```

Programm mit Subs

```

        ORG $200800
LED     EQU $200400
Loop:   MOVE.B #$FF,LED
        JSR   Wait
        MOVE.B #$00,LED
        JSR   Wait
        BRA   Loop

Wait:   MOVE.L #$100000,D0
Wt:     SUBQ.L #1,D0
        BNE   Wt
        RTS

```

Deutlich erkennbar ist, dass das Hauptprogramm kompakter und übersichtlicher geworden ist. Die Verwendung von Subprogrammen ermöglicht es, identische Teile mehrfach zu benutzen. Dadurch werden Programme kürzer, übersichtlicher und weniger stör anfällig.

Zur Erläuterung der Stackveränderung wird das oben abgebildete Programm assembliert und das generierte Listing angezeigt.

```

00200800                ORG      $200800
00200400                LED      EQU      $200400
00200800 13fc 00ff 0020 Loop:  MOVE.B  #$FF,LED
                   0400
00200808 4eb9 0020 081e      JSR      Wait
0020080e 13fc 0000 0020      MOVE.B  #$00,LED
                   0400
00200816 4eb9 0020 081e      JSR      Wait
0020081c 60e2                BRA      Loop
0020081e 203c 0010 0000 Wait:  MOVE.L  #$100000,D0
00200824 5380                Wt:     SUBQ.L  #1,D0
00200826 66fc                BNE     Wt
00200828 4e75                RTS

```

Beim ersten Sprung auf die Subroutine (\$00200808) wird die nachfolgende Adresse (\$0020080E) auf den Stack gelegt und der Programmzähler auf Adresse \$0020081E gesetzt. Der Stack wird also:

```
$200778: XX XX XX XX>00<20 08 0E  XX
```

Nach Ende der Subroutine (\$00200828) wird beim RTS der Programmzähler auf (A7)+ gesetzt (das letztgespeicherte Longword wird vom Stack geholt und in den PC geschrieben). Beim Beispiel wird \$0020080E in den PC geladen und das Programm wird an Adresse \$0020080E fortgesetzt. Der Stack wird deshalb:

```
$200778: XX XX XX XX 00 20 08 0E >XX<
```

Beim zweiten Sprung auf die Subroutine (\$00200816) wird die nachfolgende Adresse (\$0020081C) auf den Stack gelegt und der Programmzähler auf Adresse \$0020081E gesetzt. Der Stack wird also:

```
$200778: XX XX XX XX>00<20 08 1C  XX
```

Nach Ende der Subroutine (\$00200828) wird bei diesem Beispiel \$0020081C in den PC geladen und das Programm geht somit \$0020081C weiter. Der Stack wird deshalb:

```
$200778: XX XX XX XX 00 20 08 1C >XX<
```

Die Funktion bei Interrupts und Exception werden in den zugehörigen Kapiteln behandelt. Prinzipiell ist aber das Vorgehen gleich.

7 Interrupts und Exceptions

Alle Ereignisse, die während des Programmablaufs stattfinden können, werden unter dem Begriff Exception zusammengefasst. Als Spezialfall einer Exception werden in diesem Kapitel die Interrupts betrachtet. Andere Exceptions sind zum Beispiel: Division durch Null, Adressierungsfehler, illegaler Befehl, Zugriffsfehler beim Speicher, etc..

Der Ablauf einer Exception ist immer gleich. Zu Beginn werden das Status-Register und der aktuelle PC auf den Stack gelegt. Danach wird je nach Exception in der Vektortabelle der zugehörige Vektor in den PC geladen und das Exception-Programm dort bis zum RTE (Return from Exception) ausgeführt. Dieser Befehl bewirkt das Zurückholen des Statusworts und des PC, das unterbrochene Programm wird fortgesetzt.

Interrupts (Unterbrechungen) werden benutzt, um auf externe Ereignisse reagieren zu können. Als wichtigstes Beispiel dient die Uhr, die alle Sekunden um eine Sekunde aufwärtsgezählt werden muss. Eine solche Uhr wird als Beispiel vorgestellt. Andere Interrupts können von Tasten kommen, vom Endschalter einer Maschine, einer Lichtschranke, anderen Peripheriebausteinen - das Anwendungsgebiet ist unbeschränkt.

7.1 Beispiel interruptgesteuerte Uhr

Um eine Uhr zu programmieren, muss eine genaue Zeitbasis vorhanden sein. Bei einem Computersystem ist dies meist sehr einfach, da der Quarz des Microcomputers gleich mitbenutzt werden kann. Im System der TEKO Bern ist dies ein 16 MHz Quarz.

Das Uhrenprojekt wird in verschiedene Teilprojekte untergliedert:

- Interruptinitialisierung
Freigeben des Interrupts, Programmierung der Interruptquelle, Eintragen des Interruptvektors
- Interrupt-Programm Tick-Counter
Programmteil, der im Interruptfall ausgeführt werden soll. Ein Tick soll 10 ms entsprechen
- Anzeigeprogramm HH:MM
Normale Uhranzeige auf dem Display im Format HH:MM
Möglichkeit, die Uhr mit den Tasten „1a“ „2a“ „3a“ „4a“ „5a“ „6a“ zu richten
- Anzeigeprogramm MM:SS
Option Anzeige MM:SS bei Druck Taste „1b“
- Programm Stopuhr
- Programm Alarmuhr

Im Folgenden wird auf die ersten 4 Punkte eingegangen, die Punkte 5 und 6 werden im Rahmen einer Übung gelöst.

Interrupt Initialisierung

Für das Uhrenprojekt wird der interne Taktgeber des MC68332 benutzt. Dieser besitzt zwar nicht die Vielfalt eines Timerbausteines, ist dafür in der Handhabung sehr einfach. Mit wenigen Schritten ist der Tick-Interrupt verfügbar.

Als Tick wird die kleinste verfügbare Einheit eines Zeitgebers bezeichnet. In vielen Computersystemen liegt diese zwischen 1 und 20 ms. Für das Beispielprojekt wird ein Tick von 1,024ms verwendet. Für eine Uhrenprogrammierung wäre zwar ein 1ms Tick geeigneter. Dies ist jedoch mit dem internen PIT (Periodic Interrupt Timer) nicht realisierbar.

Auf den Seiten 4-8, 4-9 und D-12 des MC68332 User Manual wird der PIT beschrieben. In das Register **PITR** wird der Wert \$0108 geschrieben, um die Periodendauer von 1,024 ms festzulegen.

Im Register **PICR** wird definiert, welcher Interrupt ausgelöst werden soll. Es gibt zwei Parameter für Interrupts. Der erste ist die Interrupt-Nummer (\$40 - \$FF), der zweite die Priorität (0-7). Stehen zwei verschiedene Interrupts an der CPU an, wird zuerst der Interrupt mit der höchsten Priorität vergeben. Im Beispiel wird die Priorität 7 und der Vektor \$40 benutzt. Somit wird in das Register PICR den Wert \$0740 geschrieben.

Mit diesen beiden Registern wurde ein periodischen Interrupt erzeugt. Der CPU fehlt jedoch die Information, was zu tun ist, wenn der Interrupt aufgetreten ist. Dazu muss nun der Vektor \$40 in der Vektortabelle auf unser Interruptprogramm zeigen. Die vollständige Initialisierung sieht folgendermassen aus:

```

                ORG $200800
VBR_MEM        EQU $200C00 *Start der Vektortabelle
VBR_IRQ        EQU $200D00 *Erster Interrupt = Vektor 40
SIM_PICR       EQU $FFFA22 *PIT Control Register
SIM_PITR       EQU $FFFA24 *PIT Register

                MOVE.L #VBR_MEM,A0 *Vektortabelle startet ab $200C00
                MOVEC.L A0,VBR      *VBR (Vektor Base Register) setzen
*Nun wird die Interruptquelle initialisiert
                MOVE.L #TICK_IRQ,VBR_IRQ *Bei Interrupt Routine TICK_IRQ
                MOVE.W #$0740,SIM_PICR *Interrupt #40, Level 7 (NMI)
                MOVE.W #$0108,SIM_PITR *8x4x512/16'000'000= 1,024ms
                ... (Weitere Initialisierungen)

TICK_IRQ:      *Hier kommt das Programm das im Interruptfall ausgeführt wird
                ...
                RTE *Ende des Interruptprogramm

```

Achtung: Der PIT kann nur im **Supervisor-Mode** programmiert werden! Nach der Ausführung des Makros TEKO muss unbedingt der Befehl

RM SR \$2000

ausgeführt werden, bevor das Programm gestartet wird.

Das Interrupt-Programm Tick-Counter

Der periodische Interrupt, der alle 1,024ms ein Programm aufruft, ist nun aktiviert. Im ersten Schritt beschränken wir uns auf das Hochzählen einer Speicherstelle. Demzufolge wird alle 1,024ms diese Speicherstelle um 1 inkrementiert. Zusätzlich wird geprüft, ob schon ein Tag vorbei ist. In diesem Fall wird das Register auf 0 gesetzt. Damit werden Überläufe vermieden, das Anzeigeprogramm wird einfacher.

Die dazugehörige Interrupt - Routine lautet:

```
TICK_CNT EQU $200BF0 *Long Tick - Counter
TICK_IRQ: MOVE.L D1,-(A7)
          ADDQ.L #1,TICK_CNT *1 Addieren beim Tick-Cnt
          MOVE.L TICK_CNT,D1
          CMP.L #84375000,D1 *Wert = 24h ?
          BCS TI_NE
          CLR.L TICK_CNT
TI_NE:    MOVE.L (A7)+,D1
          RTE *Interrupt beenden
```

Das Anzeigeprogramm für Sekunden auf der rechten Anzeigehälfte

Nun kann im Hauptprogramm die „Zeit in Sekunden Anzeige“ gestartet werden. Dazu muss nur der Tick-Counter gelesen werden und danach eine Konvertierung in Sekunden erfolgen.

Die Konvertierung Ticks in Sekunden erfolgt nach der Formel: $Sec = \frac{Ticks \cdot 1024}{1000000}$

Ein Programm, das diese Aufgabe erfüllt, könnte so aussehen:

```
DISP_2 EQU $200200
Loop:   MOVE.L TICK_CNT,D0
        MULU.L #1024,D1:D0 *In us
        DIVU.L #1000,D1:D0 *Anzeige in ms
        CLR.L D1
        DIVU.L #1000,D1:D0
        MOVE.L D1,D3 *ms in D3
        MOVE.L D0,D2
        DIVU #10,D2 *Hinterste Stelle Sekunden
        SWAP D2 *Wird aus dem Rest gebildet
        MOVE.W D2,D1
        CLR.W D2
        SWAP D2
        DIVU #6,D2 *Vordere Stelle geht von 0-5
        SWAP D2
        LSL.W #4,D2
        ADD.W D2,D1
        CLR.W D2
        SWAP D2
        MOVE.B D1,DISP_2 *Ausgabe der Sekunden
        BRA LOOP
```

Das Programm kann nun beliebig erweitert werden.

7.2 Programm „Interruptgesteuerte Uhr“

Es folgt ein Beispiel einer Uhr mit Stunden, Minuten und Sekundenanzeige, die mit Hilfe der Tasten auf dem Testboard eingestellt werden kann. Das kommentierte Programm zeigt die Anwendung eines Interrupts für zwei Anwendungen, nämlich einer Uhr und einer kleinen Pulsweitenmodulation.

```

* Programm Uhr.asm
* Programm mit Hilfe von Interrupts
* 1999 By Jann P. Oesch, TEKO Bern

*Achtung, Testboard muss im Supervisor-Mode sein!

                ORG $200800 *Startadresse des Programms
TAST1          EQU $200000 *Tastenreihe unten
TAST2          EQU $200100 *Tastenreihe oben
LED            EQU $200400 *LED-Reihe unten
DISP_1        EQU $200300 *Display links
DISP_2        EQU $200200 *Display rechts

TICK_CNT      EQU $200BF0 *Long Tick - Counter
LED_Hell      EQU $200BF4 *BYTE Zwischenspeicher für LED, die hell leuchten
LED_Dim       EQU $200BF5 *BYTE Zwischenspeicher für LED, die gedimmt leuchten
KEY_Flag      EQU $200BF6 *BYTE Flag "Taste gedrückt"

HOUR          EQU $200BF8 *Zwischenspeicher für Stunden
MIN           EQU $200BF9 *Zwischenspeicher für Minuten
SEC           EQU $200BFA *Zwischenspeicher für Sekunden

VBR_MEM       EQU $200C00 *Start der Systemvektoren
VBR_IRQ       EQU $200D00 *Erster Interrupt = Vektor 40

SIM_PICR      EQU $FFFA22 *PIT Interrupt Register
SIM_PITR      EQU $FFFA24 *PIT Control Register

*Start des Programms, prüfen auf Supervisor-Mode
*Wenn nicht im Supermode, erscheint E001 im Display

                MOVE.B  #$E0,DISP_1
                MOVE.B  #$01,DISP_2 *Anzeige E001 für Error: Not in Supervisor
                MOVE.L  #VBR_MEM,A0
                MOVEC.L A0,VBR      *VBR (Vektor Base Register) setzen
                MOVE.B  #$00,DISP_2 *Anzeige E000 für kein Error
                CLR.L   TICK_CNT    *Tick Counter löschen
                CLR.B   KEY_Flag    *Flag "Taste gedrückt"
                CLR.B   LED_Hell    *Keine LED hell
                CLR.B   LED_Dim     *Keine LED gedimmt

*Nun wird die Interruptquelle initialisiert

                MOVE.L  #TICK_IRQ,VBR_IRQ *Bei Interrupt Routine TICK_IRQ
                MOVE.W  #$0740,SIM_PICR  *Interrupt #40, Level 7 (NMI)
                MOVE.W  #$0108,SIM_PITR  *8x4x512/16'000'000= 1,024ms

*Als nächstes folgt das Hauptprogramm

LOOP:          MOVE.B  TAST1,D0        *Taste gedrückt?
              BEQ     NO_KEY          *Nein, weiter
              TST.B   KEY_Flag        *
              BNE     NO_React        *Keine Reaktion auf Dauerdruck
              MOVE.B  D0,KEY_Flag     *
KEY_1:         CMP.B  #%00000001,D0   *Taste 1 Sek?
              BNE     KEY_2          *
              ADD.L   #976,TICK_CNT   *Eine Sekunde addieren
KEY_2:         CMP.B  #%00000010,D0   *Taste 10 Sek?
              BNE     KEY_3          *
              ADD.L   #9765,TICK_CNT  *10 Sekunden addieren
KEY_3:         CMP.B  #%00000100,D0   *Taste 1 Minute?
              BNE     KEY_4          *

```

KEY_4:	ADD.L	#58593,TICK_CNT	*60 Sekunden addieren
	CMP.B	##00001000,D0	*Taste 10 Minuten?
	BNE	KEY_5	*
KEY_5:	ADD.L	#585937,TICK_CNT	*10 Minuten addieren
	CMP.B	##00010000,D0	*Taste 1 Stunde
	BNE	KEY_6	*
KEY_6:	ADD.L	#3515625,TICK_CNT	*1 Stunde addieren
	CMP.B	##00100000,D0	*Taste 10 Stunden
	BNE	NO_React	*
	ADD.L	#35156250,TICK_CNT	*10 Stunden addieren
	BRA	NO_React	*Ende
NO_KEY:	CLR.B	KEY_Flag	*Keine Taste gedrückt? -> Keyflag löschen
NO_React:	MOVE.L	TICK_CNT,D0	*Tickcounter in D0
	MULU.L	#1024,D1:D0	*In us
	DIVU.L	#1000,D1:D0	*Anzeige in ms
	CLR.L	D1	*In D0 sind jetzt die ms
	DIVU.L	#1000,D1:D0	*Dividiert durch 1000 -> D1=ms; D0=Sec
	MOVE.L	D1,D3	*ms in D3
	MOVE.L	D0,D2	*Sekunden nach D2
	DIVU	#10,D2	*Dividiert durch 10
	SWAP	D2	*Rest sich einer der Sekunden (0..9)
	MOVE.W	D2,D1	*Diese Sekundeneiner nach D1
	CLR.W	D2	*Die Sekundenzehner
	SWAP	D2	*Sind wieder in D2
	DIVU	#6,D2	*Dividiert durch 6 ergibt
	SWAP	D2	*Im Rest die echten Sekundenzehner (0..5)
	LSL.W	#4,D2	*Zusammensetzen der Sekunden
	ADD.W	D2,D1	*Form Zehner:Einer in 8 Bit (je 4 Bit)
	CLR.W	D2	*D2 wird für die Minuten bereitgemacht
	SWAP	D2	*Die durch die letzte Division entstanden sind
	MOVE.B	D1,SEC	*Die Sekunden werden nun zwischengespeichert
	DIVU	#10,D2	*In D2 sind jetzt die Minuten, die dividiert werden
	SWAP	D2	*Als Rest sind 1 Minuten (0..9)
	MOVE.W	D2,D1	*Die Minuteneiner nach D1
	CLR.W	D2	*Rest löschen
	SWAP	D2	*Jetzt sind alle Minutenzehner in D2
	DIVU	#6,D2	*Im Rest sind nun die echten Sekundenzehner (0..5)
	SWAP	D2	*Diese holen
	LSL.W	#4,D2	*Zusammensetzten, Zehner vorn
	ADD.W	D2,D1	*Und Minuteneiner hinten
	CLR.W	D2	*D2 wird nun für Stunden bereitgemacht
	SWAP	D2	*Und die erhaltenen
	MOVE.B	D1,MIN	*Minuten werden zwischengespeichert
	DIVU	#24,D2	*24h - Uhr
	CLR.W	D2	*Nur Rest beachten (Überläufe vermeiden)
	SWAP	D2	*Hier sind nun die Stunden (0..23)
	DIVU	#10,D2	*Stundeneiner und Stundenzehner trennen
	SWAP	D2	*Um sie dann wieder
	MOVE.W	D2,D1	*Im richtigen Format
	SWAP	D2	*Zusammensetzten wie schon
	LSL.W	#4,D2	*Bei den Sekunden und Minuten
	ADD.W	D2,D1	*Danach kann das Ergebnis
	MOVE.B	D1,HOURL	*Als Stunden gespeichert werden
	MOVE.B	TAST2,D0	*Testen Prüfen
	AND.B	##\$7F,D0	*Stunden/Minuten ?
	CMP.B	#0,D0	*
	BNE	DISP_OTH	*Nein, Minuten/Sekunden
	MOVE.B	HOURL,DISP_1	*Stunden ausgeben
	MOVE.B	MIN,DISP_2	*Minuten ausgeben
	CLR.B	LED_Hell	*Helle LED löschen
	CLR.B	LED_Dim	*Gedimmte LED löschen
	BRA	LOOP	*Schlaufe weiter
DISP_OTH	CMP.B	#1,D0	*Display Test?
	BNE	DISP_OTH1	*Ja, springen
	MOVE.B	MIN,DISP_1	*Minuten ausgeben
	MOVE.B	SEC,DISP_2	*Sekunden ausgeben

```

        JSR     Disp_Bar           *Anzeige der 1/9-Sekunden mit Leds
        BRA     LOOP              *Zurück nach Loop
DISP_OTH1 MOVE.B  #$FF,DISP_1     *Display Test $FFFF
        MOVE.B  #$FF,DISP_2     *
        CLR.B   LED_Hell         *LED löschen
        CLR.B   LED_Dim          *
        BRA     LOOP              *Nach Loop

```

*Anzeige eines 1/9-Sekunden Ledbalken

```

Disp_Bar: CLR.B   D1              *Disp_Bar
          DIVU   #112,D3          *Aus dem Mikrosekunden 1/9-Sekunden gewinnen
          LEA   TAB,A0           *Und gemäss Tabelle
          MOVE.B (A0,D3.W),D1     *Die LED Leuchten lassen
          TST.B TAST2            *Wenn Taste DIM gedrückt
          BMI   BAR_DIM          *Die Leds gedimmt
          MOVE.B D1,LED_Hell      *Sonst Hell
          CLR.B LED_Dim           *Und nicht gedimmt
          RTS                    *Ende
BAR_DIM:  MOVE.B D1,LED_Dim       *LEDS gedimmt
          CLR.B LED_Hell         *Und nicht hell
          RTS                    *Ende

TAB:      DC.B   $00,$01,$02,$04,$08,$10,$20,$40,$80,$FF
          EVEN

```

*Hier ist die Routine, die im Interrupt-Fall angesprungen wird

```

TICK_IRQ: MOVEM.L D0-D1,-(A7)    *Register retten
          ADDQ.L #1,TICK_CNT     *1 Addieren beim Tick-Counter
          MOVE.L TICK_CNT,D1     *Tick-Counter
          CMP.L  #84375000,D1    *Wert = 24h ?
          BCS   TI_NE            *Nein, weiter
          CLR.L TICK_CNT         *Tick-Counter löschen
          CLR.L D1               *
TI_NE:   MOVE.B LED_Hell,D0      *Puls-Weiten-Modulation für gedimmte LED
          AND.B #$0F,D1          *PWM Period = 16x1,024ms= 16,36ms (61Hz)
          CMP.B #$0E,D1         *
          BCS   Led_Out          *Helle LED ausgeben
          OR.B  LED_Dim,D0       *Gedimmte LED ausgeben
Led_Out: MOVE.B D0,LED           *Und nach HW
          MOVEM.L (A7)+,D0-D1    *Register restaurieren
          RTE                    *Interrupt beenden

```

8 Anhang

8.1 Literaturverzeichnis

- [1] Werner Hilf: MC68000 Grundlagen. Franzis Verlag, München
- [2] Werner Hilf: MC68000 Anwendungen. Franzis Verlag, München
- [3] Josef Fuchs: MC68300 Mikrocontroller. Franzis Verlag, München
- [4] Motorola: CPU32RM/AD 683xx CPU32 Reference Manual
- [5] Motorola: MC68332UM/D 68332 Users Manual
- [6] Motorola: SIMRM/AD System Integration Module Reference Manual
- [7] Motorola Microcontroller Web: <http://mot-sps.com>
- [8] Jann Oesch: Auszug aus dem Befehlssatz der CPU32, OESCH.ORG, Schönbühl und Leipzig