4. Programmierung

4.1. Allgemeine Grundlagen

Es gilt immer folgende Tatsache: **Jeder Computer (und damit auch jeder Mikrocontroller) kann nur das tun, was ihm vorgeschrieben wird!**

"Vorgeschrieben" wird ihm seine Arbeit durch das **Programm**. Es handelt sich dabei um eine Sammlung von Befehlen (= „instructions") in Form **von Hexadezimal-Zahlen**, die der Reihe nach im **Programmspeicher** (= code memory = üblicherweise unser EPROM) angeordnet werden.
Jede „Instruction" hat nach dem Abschluss der Programmierung ihren festen Platz im Programmspeicher und belegt dort 1...3 Speicherplätze. Aus diesem Programmspeicher holt die CPU einen Befehl nach dem anderen heraus und bearbeitet ihn.

Es gilt für alle Computer und Controller:
Nach dem **Einschalten bzw. einem Reset** wird immer bei der **Programmspeicheradresse Null begonnen!** Da dort beim Mikrocontroller nur drei freie Speicherplätze vorhanden sind, findet man üblicherweise ein Sprungbefehl zum eigentlichen Programmstart-Adresse (= „**reset vector**).

Bei der Programmerstellung kann man nun z. B. **direkt die Hex-Codes der benötigten Befehle untereinander schreiben** (= viel zu mühsam, macht bei Mikrocontrollern kein vernünftiger Mensch mehr!) oder man bedient sich einer **Programmiersprache.** Hier müssen wir zwischen der „maschinennahen" Programmierung in „Assembler" und der Programmierung in einer Hochsprache (z. B. „C") unterscheiden.
Bei der **Assemblerprogrammierung** werden die Hexadezimalzahlen der Maschinenbefehle durch einprägsame englische Abkürzungen = **MNEMONICS** (z. B. ADD oder MOV) ersetzt, die man sich leichter merken kann.
Ein Computerprogramm („CROSSASSEMBLER" mit „LINKER" und „HEXCONVERTER") macht daraus die erforderlichen Hexcodes. Man kann damit äußerst schnelle Programme schreiben, muss aber alles bis zum letzten Detail selbst aushecken.
Bei der **Hochsprache „C"** werden vom Hersteller schon viele fertige Unterprogramme in Form ganzer Bibliotheken mitgeliefert -- das spart sehr viel Arbeit. Außerdem schrumpfen die geschriebenen Programme unglaublich zusammen, sind aber jetzt nur noch für Fortgeschrittene lesbar und prüfbar. Der Programmumfang ist allerdings nun größer als bei Assembler, außerdem liegt die Verarbeitungsgeschwindigkeit immer unter der des Assemblerprogramms.
Wir werden uns im Unterricht in beide Methoden einarbeiten.

4.2. Einführung in die professionelle Assemblerprogrammierung

4.2.1. Modulare Programmierung

Ein professionelles Assemblerprogramm soll nicht nur „funktionieren", sondern auch folgende Eigenschaften aufweisen:

Möglichst problemlose **Fehlersuche**.

Leicht zu pflegen (= nachträglich zu erweitern oder zu ändern).

Lösungsweg ohne Mithilfe auch für neue Mitarbeiter **nachvollziehbar.**

Ohne Probleme auf **andere Adressen** im Programmspeicher zu verschieben.

Bei großem Umfang der Aufgabe ohne Schwierigkeiten auf verschiedene Programmierer **aufteilbar.**

Bewährte Programmteile oder Routinen sollten bei neuen Aufgaben „**wiederverwendbar**" sein

Aus diesem Grund werden Assemblerprogramme stets aus **einzelnen „Baugruppen" (= Modulen)** zusammengesetzt. Hierbei sollte man in **jedem einzelnen Modul** stets ein möglichst übersichtliches „**Haupt-Steuerprogramm**" entwerfen, das dann **geeignete Unterprogramme** aufruft. Diese Unterprogramme können entweder im gleichen Modul stehen oder sie werden als „öffentlich zugängliche" Teile aus einem anderen Modul gefischt:

„**EXTERN**" erlaubt hierbei die Einbindung eines solchen „öffentlichen" Unterprogamms aus einem anderen Modul. Dieses Unterprogramm muss aber dort mit der Anweisung „**PUBLIC**" als öffentlich zugänglich erklärt werden.

Alle Einzelmodule werden am Schluss vom „**LINKER**" miteinander zum Gesamtprogramm verknüpft.

Jeder Modul erhält hierbei seinen eigenen Namen und besteht selbst wieder aus einzelnen Teilstücken, den sogenannten „**Segmenten**". Hierbei handelt es sich entweder um

a) Programmteile mit Instruktionen (Name: „**CODE SEGMENT**") oder

b) Speicherplatzreservierungen für Daten, Konstanten usw. im **direk**t adressierbaren **RAM1** auf dem Controllerchip (Name: „**DATA SEGMENT**") oder

c) Speicherplatzreservierungen für Daten, Konstanten usw. im **indirekt** adressierbaren **RAM**-Bereich auf dem Controllerchip, also z. B. im RAM“ (Name: „**IDATA SEGMENT**“) oder

d) Speicherplatzreservierungen für Daten, Konstanten usw. im **externen RAM** (Name: „**XDATA - SEGMENT**“).

Hinweis:
Wir können uns auch nur **255 Speicherplätze** (= 1 Seite) im externen RAM reservieren lassen. Sie werden dann über das Register R0 oder R1 statt über den Datapointer adressiert – das spart Zugriffszeit! (Name: „**PDATA-SEGMENT“**).

Außerdem gibt es die Möglichkeit, verschiedene **Bits** für unsere Anwendung zu definieren (= als Flags oder „Merker“). Sie finden sich im **bitadressierbaren Bereich des RAM1 unter den Adressen 20h bis 2Fh** (Name: „**BIT SEGMENT“**).

Die „**Werkzeuge**“ zur Erstellung eines Moduls lassen sich nun in folgende Gruppen einteilen:

1) **Assembler instructions** („Programmschritte“) = Maschinenbefehle = einzelne Anweisungen im Binärcode, die dem Controller als Hex-Zahl mitgeteilt werden. Sie bilden die einzelnen Elemente eines ausführbaren Programmteiles. Der Programmierer schreibt diese Anweisungen jedoch in Form von „Mnemonics“, also in Form einfacher englischer Abkürzungen, die dann vom PC in Hex-Zahlen übersetzt werden.

2) **Assembler directives**. Sie organisieren die verschiedenen Segmente und Speicherbereiche, legen Symbolnamen für Variable oder Speicherplätze fest, definieren das Modulende usw.
Achtung: Sie sind also „Verwalter“ und produzieren selbst keine Programmcodes.

3) **Assembler controls**. Sie steuern die Assemblier- und Linkvorgänge und sind am „Dollarzeichen“ ($) zu erkennen, mit dem sie eingeleitet werden.

4.2.2. Einteilung der 80C535-Instructions in Funktionsklassen

a) Arithmetische Befehle
Hier steht die **Addition, Subtraktion, Multiplikation, Division und die Dezimalkorrektur** (für BCD-Zahlen). Außerdem das **Inkrementieren** (= Erhöhen eines Registerinhaltes um 1) und das **Dekrementieren** (Erniedrigen eines Registerinhaltes um 1).

b) Sprungbefehle
Man muss zunächst zwischen **SJMP** (= short jump = Sprung um max. 127 nach oben oder unten
AJMP (= absolute jump = Sprung um max. 2 Kilobyte nach oben oder unten) und
LJMP (= long jump = Sprung innerhalb des kompletten 64 Kilobyte-Adressraumes)
unterscheiden.
Die Unterschiede liegen sowohl in der Ausführungszeit wie auch im Speicherplatzbedarf. Außerdem finden sich hier viele **bedingte Sprungbefehle** (z. B. Sprung, wenn Akku leer usw.)
Zu dieser Gruppe gehören auch die **Aufruf-Befehle für Unterprogramme (ACALL, LCALL**) samt **dem Rückkehr-kommando** (RET) und der **Faulenzerbefehl NOP** (no operation = tue gar nix).

Hinweis: Unserem Assembler reicht bereits die Angabe „jmp“ oder „call“ im Programm. Er setzt dann selbst den gerade geeigneten Jump- oder Call-Befehl ein.

c) Logische Befehle
UND (ANL), **ODER** (ORL) , **EXKLUSIV-ODER** (XRL), **INVERTIERUNG** (CPL) stecken in dieser Gruppe. Auch so wichtige Sachen wie **“CLR A”** (= lösche den Akku), Rotations- und Vertauschungsbefehle.

d) Datentransferbefehle
Diese Gruppe enthält alle MOV - Varianten (sowie die Ableger MOVX und MOVC).
Es gilt: 1) Mit **MOV**....werden Registerinhalte irgendwo hinbewegt.
2) Mit **MOVX** wird aus dem externen Datenspeicher (RAM) gelesen oder in ihn etwas hineingeschrieben
3) Mit **MOVC** wird ein Code-Byte aus dem Programmspeicher in den Akku geholt und dort wie eine normale Hex-Zahl behandelt oder verarbeitet (Anwendung: Einsatz von gespeicherten Texten oder Tabellen).
Mit den Befehlen **PUSH....** und **POP....** lassen sich Registerinhalte auf ein sogenannten “Stack” retten bzw. wieder von dort holen.
Mit **XCH**.....können **Registerinhalte vertauscht** werden

e) Bitmanipulationsbefehle
Damit können einzelne Bits in den SFR bzw. im **internen, bitmanipulierbaren Bereich von RAM 1 auf dem Chip** (Adressen 20h bis 2Fh) gesetzt, gelöscht, invertiert oder logisch verknüpft werden.

4.2.3. Befehlsliste für den Mikrocontroller 80C535

a) Erläuterung der Operanden

Operand	Erklärung
A	Akkumulator
addr11	Adresse innerhalb eines 2 Kilobyte-Programmspeicherblocks
addr16	Adresse irgendwo im 64 Kilobyte-Programmspeicherraum
	Register B
bit	Bitadresse im internen RAM
/bit	Inhalt der Bitadresse wird invertiert (= komplementiert)
C	Carry-Bit
#data	8-Bit-Konstante
#data16	16-Bit-Konstante
direct	Adresse eines Platzes im internen RAM
DPTR	Datenpointer
PC	Programmzähler (program counter)
Ri	Indirekte Adressierung eines RAM-Platzes durch R1 oder R2
Rn	Register R0 bis R7 der gerade aktuellen Registerbank
	Zeichen für indirekte Adressierung
rel	Signiertes 8-Bit-Offset für Sprungbefehle (= Relativer Sprung um max. 128 Byte nach oben oder unten)

b) Arithmetische Befehle

Hex-Code	Mnemonik	Beschreibung	Byte	Masch.-Zyklen
28.....2F	ADD A, Rn	Addiere den Inhalt des ausgewählten Registers (R0.....R7) zum Akkumulator	1	1
5	ADD A, direct	Addiere den Inhalt des direkt adressierten RAM-Platzes zum Akku	2	1
26	ADD A,@R0	Addiere den Inhalt des indirekt durch R0 adressierten RAM-Platzes zum Akku	1	1
27	ADD A,@R1	Addiere den Inhalt des indirekt durch R1 adressierten RAM-Platzes zum Akku	1	1
24	ADD A,#data	Addiere die angegebene Konstante zum Akku	2	1
38.....3F	ADDC A,Rn	Addiere den Inhalt des ausgewählten Registers (R0.....R7) zum Akku -- einschließlich Carry	1	1
35	ADDC A,direct	Addiere den Inhalt des direkt adressierten RAM-Platzes zum Akku -- einschließlich Carry	2	1
36	ADDC A,@R0	Addiere den Inhalt des indirekt durch R0 adressierten RAM-Platzes zum Akku -- einschließlich Carry!	1	1
37	ADDC A,@R1	Addiere den Inhalt des indirekt durch R1 adressierten RAM-Platzes zum Akku -- einschließlich Carry!	1	1
34	ADDC A,#data	Addiere die angegebene Konstante zum Akku -- einschließlich Carry	2	1
98.....9F	SUBB A,Rn	Subtrahiere den Inhalt des ausgewählten Registers (R0...R7) vom Akku	1	1
95	SUBB A,direct	Subtrahiere den Inhalt des direkt adressierten RAM-Platzes vom Akku	2	1
96	SUBB A,@R0	Subtrahiere den Inhalt des indirekt durch R0 adressierten RAM-Platzes vom Akku	1	1
97	SUBB A,@R1	Subtrahiere den Inhalt des indirekt durch R1 adressierten RAM-Platzes vom Akku	1	1
94	SUBB A,#data	Subtrahiere die angegebene Konstante vom Akku	2	1
04	INC A	Erhöhe (= inkrementiere) den Akkuinhalt um 1	1	1
08.....0F	INC Rn	Inkrementiere das angegebene Register (R0.....R7)	1	1
05	INC direct	Inkrementiere den direkt adressierten RAM-Platz	2	1
06	INC @R0	Inkrementiere den durch R0 indirekt adressierten RAM-Platz	1	1
07	INC @R1	Inkrementiere den durch R1 indirekt adressierten RAM-Platz	1	1
14	DEC A	Vermindere (= dekrementiere) den Akku um 1	1	1
18.....1F	DEC Rn	Dekrementiere das angegebene Register (R0...R7)	1	1
15	DEC direct	Dekrementiere den direkt adressierten RAM-Platz	1	1
16	DEC @R0	Dekrementiere den durch R0 indirekt adressierten RAM-Platz	1	1
17	DEC @R1	Dekrementiere den durch R1 indirekt adressierten RAM-Platz	1	1
A3	INC DPTR	Inkrementiere den Datenpointer	1	2
A4	MUL AB	Multipliziere Akku und B-Register	1	4
84	DIV AB	Dividiere den Akku durch das B-Register	1	4
D4	DA A	Dezimalkorrektur des Akkus bei BCD-Operationen	1	1

c) Sprungbefehle:

Hex-Code	Mnemonik	Beschreibung	Byte	Masch.-Zyklen
11 oder F1	ACALL addr11	Subroutinenaufruf innerhalb eines 2 Kilobyte Programmspeicherblocks	2	2
12	LCALL addr15	Subroutinenaufruf innerhalb des 64 Kilobyte-Programmspeicher-Raumes	3	2
22	RET	Zurück aus der Subroutine	1	2
32	RETI	Zurück aus der Interrupt-Routine	1	2
01 oder E1	AJMP addr11	Sprung innerhalb eines 2 Kilobyte-Blockes	2	2
02	LJMP addr16	Sprung innerhalb des 64 Kilobyte - Adressraumes	3	2
80	SJMP rel	relativer Sprung um max. 128 Byte nach oben oder nach unten	2	2
73	JMP @A + DPTR	Indirekter Sprung relativ zum Datenpointer	1	2

60	JZ rel	Relativer Sprung um max. 128 Byte, wenn der Akku leer ist	2	2
70	JNZ rel	Relativer Sprung um max. 128 Byte, wen der Akku nicht leer ist	2	2
B5	CJNE A,direct,rel	Springe, wenn Akku und direkt adressierter RAM-Platz ungleiche Inhalte haben, um max. 128 Byte	3	2
B4	CJNE A,#data,rel	Springe, wenn der Akku-Inhalt und die vorgegebene Konstante ungleich sind, um max. 128 Byte	3	2
B8 bis BF	CJNE Rn,#data,rel	Springe, wenn der Inhalt des gewählten Registers (R0 bis R7) nicht mit der vorgegebenen Konstante übereinstimmt, um max. 128 Byte	3	2
B6	CJNE @R0,#data,rel	Springe, wenn der Inhalt der indirekt durch R0 adressierten Speicherzelle nicht mit der vorgegebenen Konstante übereinstimmt, um max. 128 Byte	3	2
B7	CJNE @R1,#data,rel	Springe, wenn der Inhalt der indirekt durch R1 adressierten Speicherzelle nicht mit der vorgegebenen Konstante übereinstimmt, um max. 128 Byte	3	2
D8 bis DF	DJNZ Rn,rel	Dekrementiere das gewählte Register (R0 bis R7) und springe um max. 128 Byte, wenn anschließend noch keine Null im Register steht	2	2
D5	DJNZ direct,rel	Dekrementiere den angegebenen RAM-Speicherplatz und springe um max. 128 Byte, wenn anschließend der Inhalt noch größer als Null ist	3	2
40	JC rel	Springe um max. 128 Byte, wenn das Carry-Bit auf **HIGH** steht (also gesetzt ist)	2	2
50	JNC rel	Springe um max. 128 Byte, wenn das Carry-Bit auf **LOW** steht (also gelöscht ist)	2	2
20	JB bit,rel	Springe um max. 128 Byte, wenn das angegebene Bit auf **HIGH** steht	3	2
30	JNB bit,rel	Springe um max. 128 Byte, wenn das angegebene Bit auf **LOW** steht	3	2
10	JBC bit,rel	Springe um max. 128 Byte, wenn das angegebene Bit auf **HIGH** steht und lösche anschließend dieses Bit	3	2
00	NOP	Keine Wirkung (ergibt nur Zeitverzögerung um 1 Maschinenzyklus, also 1 Mikrosekunde)	1	1

d) Logische Befehle

Hex-Code	Mnemonik	Beschreibung	Byte	Masch.-Zyklen
58 bis 5F	ANL A,Rn	Akku UND gewähltes Register (R0 bis R7)	1	1
55	ANL A,direct	Akku UND direkt angegebener RAM-Speicherplatz (Ergebnis im Akku)	2	1
56	ANL A,@R0	Akku UND indirekt durch R0 adressierter Speicherplatz	1	1
57	ANL A,@R1	Akku **UND** indirekt durch R1 adressierter Speicherplatz	1	1
54	ANL A,#data	Akku **UND** vorgegebene Konstante	2	1
52	ANL direct,A	Direkt angewählter RAM-Platz **UND** Akku (Ergebnis im RAM-Speicherplatz)	2	1
53	ANL direct,#data	Direkt angewählter RAM-Platz **UND** Konstante (Ergebnis im RAM-Speicherplatz)	3	2
48 bis 4F	ORL A,Rn	Akku **ODER** gewähltes Register	1	1
45	ORL A,direct	Akku **ODER** direkt angegebener RAM-Speicherplatz (Ergebnis im Akku)	2	1
46	ORL A,@R0	Akku **ODER** indirekt durch R0 adressierter Speicher-Platz	1	1
47	ORL A,@R1	Akku **ODER** indirekt durch R1 adressierter Speicher-Platz	1	1
44	ORL A,#data	Akku **ODER** vorgegebene Konstante	2	1
42	ORL direct,A	Direkt angewählter RAM-Platz **ODER** Akku (Ergebnis im RAM-Speicherplatz)	2	1
43	ORL direct,#data	Direkt angewählter RAM-Platz **ODER** Konstante (Ergebnis im RAM-Speicherplatz)	3	2
68 bis 6F	XRL A,Rn	Akku **EXKLUSIV ODER** gewähltes Register (R0 bis R7)	1	1
65	XRL A,direct	Akku **EXKLUSIV ODER** direkt angegebener RAM-Speicherplatz (Ergebnis im Akku)	2	1

66	XRL	A,@R0	Akku **EXKLUSIV ODER** indirekt durch R0 adressierter Speicherplatz	1	1
67	XRL	A,@R1	Akku **EXKLUSIV ODER** indirekt durch R1 adressierter Speicherplatz	1	1
64	XRL	A,#data	Akku **EXKLUSIV ODER** vorgegebene Konstante	2	1
62	XRL	direct,A	Direkt angewählter RAM-Speicherplatz **EXKLUSIV ODER** Akku (Ergebnis im RAM-Speicherplatz)	2	1
63	XRL	direct,#data	Direkt angewählter RAM-Speicherplatz EXKLUSIV ODER vorgebene Konstante (Ergebnis im RAM-Speicherplatz)	3	2
E4	CLR	A	Lösche den Akkumulatorinhalt	1	1
F4	CPL	A	Komplementiere den Akkuinhalt	1	1
23	RL	A	Rotiere den Akkuinhalt (um 1 Schritt) nach links	1	1
33	RLC	A	Rotiere den Akkuinhalt (um 1 Schritt) über das Carry-Bit nach links	1	1
03	RR	A	Rotiere den Akkuinhalt (um 1 Schritt) nach rechts	1	1
13	RRC	A	Rotiere den Akkuinhalt (um 1 Schritt) über das Carry-Bit nach rechts	1	1
C4	SWAP	A	Vertausche die beiden Nibbles des Akkumulators	1	1

e) Datentransfer-Befehle

Hex-Code	Mnemonik		Beschreibung	Byte	Masch.-Zyklen
E8 bis EF	MOV	A,Rn	Hole den Registerinhalt (R0.....R7) in den Akku	1	1
E5	MOV	A,direct	Hole den Inhalt des direkt adressierten RAM-Platzes in den Akku	2	1
E6	MOV	A,@R0	Hole den Inhalt der indirekt durch R0 adressierten Speicherzelle in den Akku	1	1
E7	MOV	A,@R1	Hole den Inhalt der indirekt durch R1 adressierten Speicherzelle in den Akku	1	1
74	MOV	A,#data	Lade den Akku mit der gewünschten Konstante	2	1
F8 bis FF	MOV	Rn,A	Speichere den Akkuinhalt im ausgewählten Register (R0.....R7)	1	2
A8 bis AF	MOV	Rn,direct	Lade das ausgewählte Register (R0....R7) mit dem Inhalt des direkt adressierten RAM-Speicherplatzes	2	2

Hex-Code	Mnemonik		Beschreibung	Byte	Masch.-Zyklen
78 bis 7F	MOV	Rn,#data	Lade das gewählte Register (R0...R7) mit der gewünschten Konstante	2	1
F5	MOV	direct,A	Speichere den Akkuinhalt im direkt adressierten RAM-Speicherplatz	2	1
88 bis 8F	MOV	direct,Rn	Speichere den ausgewählten Registerinhalt (R0... R7) im direkt adressierten RAM-Speicherplatz	2	2
85	MOV	direct,direct	Speichere den Inhalt eines direkt adressierten RAM-Platzes in einem anderen direkt adressierbaren RAM-Platz	3	2
86	MOV	direct,@R0	Speichere den Inhalt der indirekt durch R0 adressierten Speicherzelle im direkt adressierten RAM-Platz	2	2
87	MOV	direct,@R1	Speichere den Inhalt der indirekt durch R1 adressierten Speicherzelle im direkt adressierten RAM-Platz	2	2
75	MOV	direct,#data	Lade den direkt adressierten RAM-Platz mit der vorgegebenen Konstanten	3	2
F6	MOV	@R0,A	Speichere den Akkuinhalt in der indirekt durch R0 adressierten Speicherzelle	1	1
F7	MOV	@R1,A	Speichere den Akkuinhalt in der indirekt durch R1 adressierten Speicherzelle	1	1
A6	MOV	@R0,direct	Speichere den Inhalt des direkt adressierten RAM-Platzes in die indirekt durch R0 adressierte Speicher-Zelle	2	2
A7	MOV	@R1,direct	Speichere den Inhalt des direkt adressierten RAM-Platzes in die indirekt durch R1 adressierte Speicher-Zelle	2	2
76	MOV	@R0,#data	Lade die indirekt durch R0 adressierte Speicher-Zelle mit der vorgegebenen Konstanten	2	1
77	MOV	@R1,#data	Lade die indirekt durch R1 adressierte Speicher-Zelle mit der vorgegebenen Konstanten	2	1
90	MOV	DPTR,#data16	Lade den Datenpointer mit einer 16 Bit-Konstante	3	2

93	MOVC	A,@A + DPTR	Lade den Akku mit einem Programmcode (z. B. abgespeicherter Tabellenwert). Die Code-Adresse erhält man als Summe aus Akkuinhalt und DPTR-Inhalt	1 2
83	MOVC	A,@A + PC	Lade den Akku mit einem Programmcode (z. B. abgespeicherter Tabellenwert). Die Code-Adresse erhält man als Summe aus Akkuinhalt und PC-Inhalt	1 2
E2	MOVX	A,@R0	Hole ein Byte aus dem externen Datenspeicher (RAM).Die Adresse steht in Register R0	1 2
E3	MOVX	A,@R1	Hole ein Byte aus dem externen Datenspeicher (RAM).Die Adresse steht in Register R1	1 2
E0	MOVX	A,@DPTR	Hole ein Byte aus dem externen Datenspeicher (RAM).Die Adresse steht im Datenpointer	1 2
F2	MOVX	@R0,A	Speichere den Akkuinhalt im externen Daten-Speicher (RAM). Die Adresse steht im Register R0	1 2
F3	MOVX	@R1,A	Speichere den Akkuinhalt im externen Daten-Speicher (RAM). Die Adresse steht im Register R1	1 2
F0	MOVX	@DPTR,A	Speichere den Akkuinhalt im externen Daten-Speicher (RAM). Die Adresse steht im DPTR	1 2
C0	PUSH	direct	Lege den Inhalt des direkt adressierten RAM-Platzes auf dem Stack ab	2 2
D0	POP	direct	Schreibe vom Stack zurück in direkt adressierten RAM-Platz	2 2
C8 bis CF	XCH	A,Rn	Vertausche die Inhalte von Akku und gewähltem Register (R0....R7)	1 1
C5	XCH	A,direct	Vertausche die Inhalte von Akku und direkt adressiertem RAM-Platz	2 1
C6	XCH	A,@R0	Vertausche die Inhalte von Akku und indirekt durch R0 adressierter RAM-Speicherzelle	1 1
C7	XCH	A,@R1	Vertausche die Inhalte von Akku und indirekt durch R1 adressierter RAM-Speicherzelle	1 1
D6	XCHD	A,@R0	Vertausche die Low-Nibbles von Akku und der indirekt durch R0 adressierten RAM-Speicherzelle	1 1
D7	XCHD	A,@R1	Vertausche die Low-Nibbles von Akku und der indirekt durch R1 adressierten RAM-Speicherzelle	1 1

g) Bitmanipulations-Befehle

Hex-Code	Mnemonik		Beschreibung	Byte	Masch.Zyklen
C3	CLR	C	Lösche das Carry-Bit	1	1
C2	CLR	bit	Lösche das direkt angesprochene Bit	2	1
D3	SETB	C	Setze das Carry-Bit	1	1
D2	SETB	bit	Setze das direkt angesprochene Bit	2	1
B3	CPL	C	Komplementiere das Carry-Bit	1	1
B2	CPL	bit	Komplementiere das direkt angesprochene Bit	2	1
82	ANL	C,bit	Bilde: Carry UND direkt angesprochenes Bit (Ergebnis: als neues Carry-Bit)	2	2
B0	ANL	C,/bit	Bilde: Carry UND invertiertes, direkt ange-sprochenes, Bit. (Ergebnis: als neues Carry-Bit)	2	2
72	ORL	C,bit	Bilde: Carry ODER direkt angesprochenes Bit (Ergebnis: als neues Carry-Bit)	2	2
A0	ORL	C,/bit	Bilde: Carry ODER invertiertes, direkt ange-sprochenes Bit. (Ergebnis: als neues Carry-Bit)	2	2
A2	MOV	C,bit	Benütze das direkt angesprochene Bit als neues Carry-Bit	2	1
92	MOV	bit,C	Schreibe das Carry-Bit in den Platz des direkt angesprochenen Bits	2	2

4.2.4. Programmier-Systematik

Folgende **Reihenfolge** sollte beim Programmentwurf **immer** eingehalten werden:

1. Schritt = **Erfassung und Analyse des Problems** bzw. der zu lösenden Aufgabe.

2. Schritt = Formulierung eines geeigneten **Lösungsweges (= Algorithmus).**

3. Schritt = **Umsetzung** des Algorithmus in ein **Struktogramm oder Flussdiagramm ("Programm-Ablauf-Plan").**

4. Schritt = **Festlegung der dazu erforderlichen Module**, ihrer Aufgaben und ihrer Namen.

5. Schritt = Dann folgt für jeden **Einzelmodul** der Entwurf seines Steuerprogramms, die Festlegung der erforderlichen Unterprogramme, die Vergabe von Namen für die Segmente und Symbole, die Reservierung von Datenbereichen usw.

6.Schritt = **Programmierung der Einzelsegmente jedes Moduls** mit Hilfe eines geeigneten Schreibprogramms (= "Texteditor") auf dem PC (in Assemblersprache mit den entsprechenden Mnemonics).

Wichtig: immer nur ASCII-Text ohne Formatierung und Sonderzeichen erzeugen! (Sonst: Fehlermeldung..)

7. Schritt = **Übersetzung** des mit dem Editor erstellten Programms in Maschinenbefehle durch den **"Assembler**". Er erzeugt einen "noch verschiebbaren Objekt-Code" (= eine **Objekt - Datei**). Dann Verbindung unseres geschriebenen Programmteils mit weiteren Makros, Bibliotheksroutinen etc. und Zuteilung der echten Programmspeicheradressen durch den **"Linker**". Zum Schluss noch die Übersetzung des kompletten, fertigen Programms in eine *.**hex - Datei** -- meist im INTEL - HEX - Format.

8. Schritt = **Übertragung** dieses HEX-Files in den **Programmspeicher** des Controllerboards.

9. Schritt = **Testen** des Programms.

10. Schritt = meist: **Fehlersuche (= Debugging**). Sie beginnt immer mit der Wiederholung ab Schritt 5....

Hinweise zum Programmtest (= Schritte 8 und 9):

Im "normalen" Betriebsfall ist das Programm im EPROM eingebrannt und der Controller beginnt nach dem Reset oder dem Einschalten **immer** bei der Code-Adresse

0000H

mit der Bearbeitung. Will man aber Programme erstellen, prüfen, erweitern oder korrigieren, dann kann nicht jedes Mal sofort ein neues EPROM gebrannt werden. Für diesen Fall gibt es drei Möglichkeiten:

a) Der **komplette Controller wird auf einem PC simuliert** und dort das Programm getestet. (Beispiel: Programmteil „dscope" bei der KEIL-Software)

b) Man **täuscht dem Controllerboard ein EPROM vor**. In Wirklichkeit ist das eine "EPROM-Emulator-Schaltung" (also eine weitere Karte), in die das Programm vom PC aus geladen wird.
Sie ist über ein Flachbandkabel mit der **EPROM-Steckfassung** verbunden und erhält das Programm selbst wieder vom PC über das Druckerkabel (= über den Parallelport LPT1).

c) Man benützt die **"von Neumann - Architektur**" und lädt das geschriebene Programm **vom PC aus über die Serielle Schnittstelle (= Mausanschluss**) in das **RAM** des Mikrocontrollerboards. Vom PC aus wird dann das Programm durch Eingabe der Anfangsadresse gestartet.

Wir werden im Unterricht die Methoden a) und c) benützen und lassen unsere geschriebenen Programme i**mmer** bei der RAM-Adresse

4000HEX = 0x4000

beginnen.

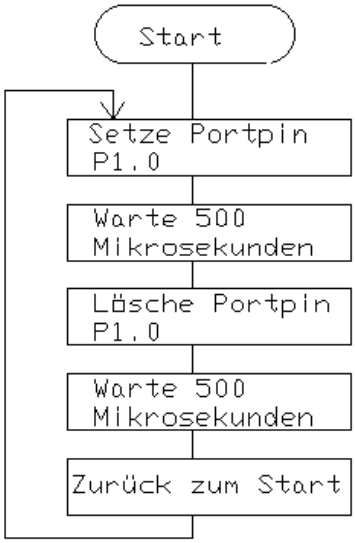
4.3. Erstes Anwendungsbeispiel: Umgang mit einem Controllerport

Die Controllerports sind bekanntlich die “Verbindungsstellen” zur Außenwelt. Über sie können Bits und Bytes ausgegeben oder eingelesen werden - ohne sie wäre das ganze Controllersystem sinnlos.
Wir wollen uns - zur Einarbeitung - deshalb folgende Aufgabe vorgeben:

“Erstellen Sie ein Programm, das am Portpin P1.0 ein Rechtecksignal (mit der Frequenz 1 kHz) erzeugt”.

4.3.1. Der Programm-Ablaufplan als Entwurfshilfe

Zunächst wird die grundsätzliche Arbeitsweise des Programms (der sogenannte **Algorithmus**) festgelegt und z. B. durch einen “PAP” (Programm - Ablauf -Plan) ausgedrückt:



4.3.2. Erstellung des Assemblerprogramms „1kHz.a51“

Dieser Teil ist auch in der Gebrauchsanleitung für die „KEIL-µvision“-Software enthalten!

Wir wollen an einem ersten, einfachen Programm die grundsätzliche Struktur von Assemblerprogrammen kennen lernen und anschließend zeigen, wie man sie zum Leben erweckt.

Wichtig: In der Industriepraxis sind die **Programmstrukturen meist sehr aufwendig**. Deshalb versuchen wir gleich, uns an die dafür erforderlichen, korrekten Entwurfs- und Organisationsmethoden zu gewöhnen.

Dabei gelten folgende Spielregeln:

a) **Assembler Controls und Directives** wollen wir zur besseren Übersicht **groß schreiben**.

b) Hinter einem **Sprungziel (“Label”) muss** ein **Doppelpunkt** stehen.

c) **Niemals einen Bindestrich** in einem Namen oder Label verwenden! Es ist nur der **Unterstrich** erlaubt!

d) Bitte Labels **nie mit Zahlen beginnen.**

e) Es genügt, “**call**” bzw. “**jmp**” zu schreiben. Der Assembler wählt selbst den genau passenden Sprung- oder Call-Befehl aus.

f) ***Bitte jeden Befehl mit einem Kommentar versehen!***

g) Ein **Strichpunkt** bedeutet immer, dass der darauffolgende Text als „Kommentar“ ignoriert wird.

h) **“end”** muss wirklich ganz am Ende stehen.

Es folgen nun **zwei Musterblätter** zur eigenen Assemblerprogrammierung, in denen die entsprechenden Controls, Directives, Symboldeklarationen und Segmentdefinitionen für unser Programm „1khz.a51“ bereits eingetragen sind.

Blatt 1 stellt die „Minimalanforderung“ dar, ist deshalb einfach gehalten und soll den Programmieranfänger nicht gleich komplett verwirren. **Blatt 2** dient für ***Fortgeschrittene*** und reagiert auf die meisten praktischen Probleme.

Wir tippen also Blatt 1 in unseren Keil-Editor ein und speichern es anschließend zweimal ab, nämlich
a) für die **Erzeugung des 1kHz-Tones** als Datei „**1khz.a51**“ im zugehörigen Projektordner „**1kHz**“ und
b) als „template“ (= Mustervorlage) mit der Bezeichnung „**muster.a51**“ an einer Stelle in unserem Verzeichnis, wo wir es später garantiert wiederfinden.

==========

Noch einige Tipps, die wir dauernd brauchen werden:

1) Bei unserem „Compuboard 80C535“ können wir **erst ab der Adresse 4000h** die Programmcodes vom PC aus zum Controllerboard über die RS232 überspielen (denn ab dieser Adresse ist der RAM-Speicher als „von Neumann-Architektur“ organisiert). Wir berücksichtigen das durch einen **„Offset“ mit der Größe 4000h bei der Organisation unseres Programmstarts**.

2) Alles, was hinter einem **Strichpunkt** steht, ist ein **Kommentar** und wird von der KEIL-Software **ignoriert**.

3) Prägen Sie sich bitte genau ein, wie verschiedene Programmteile mit „**SEGMENT CODE ....“** definiert und anschließend mit „**RSEG ....“** aufgerufen werden.

4) Achten Sie genau darauf, dass nie die **STACK- Reservierung** in Ihrem Programm vergessen wird und dass der Label „**START**“ Ihres Assemblerprogramms stets die **STACK - Adresszuweisung** vornimmt.

==========

(Als Vorlage für die Muster diente das auf der KEIL-CD-ROM mitgelieferte Assembler-Template
C51EVAL\ASM\TEMPLATE.A51
Es ist sehr ausführlich gehalten und liefert nahezu alle Informationen -- auch für nicht hier aufgeführte Möglichkeiten. Bitte ggf. selbst ausdrucken und sich informieren!)

Aber jetzt folgt das erste, vereinfachte Programmblatt.
(Die fetten Markierungen sollen Sie darauf hinweisen, dass diese Teile unseren 1kHz-Ton erzeugen werden...)

```
;1khz.a51 / 01. Januar 2005 / G. Kraus

$NOMOD51                        ;Schalte den Standard – 8051 – Modus aus
$INCLUDE (REG515.INC)           ;Verwende stattdessen den Registersatz des 80515 / 80535

OFFSET  EQU 4000h               ;Programmcodes können erst ab Adresse 4000h gespeichert werden.

ausgang bit P1.0                ; Portpin P1.0 heißt nun "ausgang"

?STACK  SEGMENT IDATA
        RSEG ?STACK
        DS 5

CSEG    AT 0+OFFSET             ; Programm-Codes beginnen bei 4000h
        ljmp   START            ; reset location (spring zu START)

PROG1   SEGMENT    CODE         ; Erstes Codesegment heißt "Prog1"
        RSEG PROG1

START:  mov    SP,#?STACK-1     ; Zuweisung der Stack - Adresse
loop:   call   zeit             ; Zeitschleife aufrufen
        cpl    ausgang          ; Portpin P1.0 invertieren
        jmp    loop             ; Springe zu LOOP und wiederhole alles

PROG2   SEGMENT CODE            ;Zweites Codesegment heißt "Prog2"
        RSEG PROG2

zeit:   mov R7,#250             ;Lade Register R7 mit der Zahl 250
        djnz   R7,$             ;Dekrementiere R7 bis auf Null
        ret                     ;Zurück aus dem Unterprogramm

END
```

So sieht ein Profiblatt aus:

```
;1khz.A51
;Erstellt am 01. Januar 2005 durch G. Kraus
;-------------------------------------------------------------------------------------------
$NOMOD51                        ; 8051-Standard-Modus abschalten
$INCLUDE (REG515.INC)           ; dafür 80535-Registersatz bereitstellen
;===========================================================
;Assembler-Direktiven und Deklarationen
;===========================================================
NAME    PIEPS                   ; Dieser Modul soll "PIEPS" heißen
;-------------------------------------------------------------------------------------------

OFFSET  EQU 4000h               ; Offset von 4000h beim Compuboard nötig

ausgang bit P1.0                ; Portpin P1.0 heißt nun "ausgang"
;-------------------------------------------------------------------------------------------
?STACK  SEGMENT IDATA           ; Stack soll im IDATA-Bereich liegen.
        RSEG ?STACK             ; Schalte auf ?Stack-Segment um.
        DS 5                    ; Reserviere 5 Bytes für den Stack.
;-------------------------------------------------------------------------------------------
MY_BYTES  SEGMENT DATA    ; Reservierung von  Bytes im DATA-Bereich
          RSEG MY_BYTES   ; Schalte auf das Segment „MY_BYTES“ um

LISA:     DS 1            ; Reserviere für mich 1 Platz mit Namen „LISA“
SUSI:     DS 5            ; Dazu noch „SUSI“ mit 5 Plätzen
;-------------------------------------------------------------------------------------------
MY_WORDS SEGMENT DATA     ; Reservierung von Wörtern im DATA-Bereich
         RSEG MY_WORDS    ; Schalte auf das Segment „MY_WORDS“ um

ANNELIESE: DW 1           ; „ANNELIESE“ soll 1 Wort = 2 Bytes belegen
FRAUEN:    DW 5           ; Für meine Frauen sind 5 Worte nötig
;-------------------------------------------------------------------------------------------
MY_BITS   SEGMENT BIT     ; Reservierung im BIT-Bereich
          RSEG MY_BITS    ; Schalte auf das Segment „MY_BITS“

KUNO:     DBIT 1          ; Reserviere 1 Bit mit dem Namen „KUNO“
HUGO:     DBIT 5          ; Die Bitvariable „HUGO“ habe 5 Bits
;-------------------------------------------------------------------------------------------
DATEN   SEGMENT XDATA     ; Speicherplätze im externen RAM. reservieren
        RSEG DATEN        ; Schalte auf "Daten" um und halte zur Gaudi
array:  DS 500            ; ein Feld „array“ mit 500 Plätzen frei
;-------------------------------------------------------------------------------------------
CSEG    AT 0+OFFSET       ; Programm-Codes beginnen bei 4000h
        jmp   START       ; reset location (springe zu START)
;===========================================================
;Erst hier beginnt unser Programm (= assembler instructions)
;===========================================================
PROG1   SEGMENT   CODE    ; Erstes Codesegment heißt "Prog1"
        RSEG PROG1        ; Schalte auf dieses Codesegment PROG1 um
        USING    0        ; Registerbank 0 soll verwendet werden

START:  mov   SP,#?STACK-1  ; Zuweisung der Stack-Adresse
loop:   call  zeit          ; Zeitschleife aufrufen
        cpl   ausgang       ; Portpin P1.0 invertieren
        jmp   loop          ; Springe zu LOOP und wiederhole alles


PROG2   SEGMENT CODE        ;Zweites Codesegment heißt "Prog2"
        RSEG PROG2          ;Schalte auf dieses Codesegment um

zeit:   mov R7,#250         ;Lade Register R7 mit der Zahl 250
        djnz  R7,$          ;Dekrementiere R7 bis auf Null
        ret                 ;Zurück aus dem Unterprogramm
;-------------------------------------------------------------------------------------------
END                         ;“END" muss immer ganz am Ende stehen!
```

Erläuterungen der Elemente:

a) Controls

b) Directives:

NAME kann auch weggelassen werden

Adreßoffset (4000h) wegen „von Neumann“-Architektur

Bit-Deklaration für „ausgang“

Der **STACK** ist eine wichtige „Zwischenablage“ und sollte **nie weggelassen werden**

So können für irgendwelche Variablen die nötigen Speicherplätze in RAM 1 reserviert werden

Auch 16-Bit-Variable (= „Wörter“) können dort gezielt gespeichert werden

In „MY_BITS“ reservieren wir für uns zum Spaß noch einige „Merker“ oder „Flags“, z. B. einen „KUNO“ und 5 „HUGO“s.

So legt man sich ein Datenfeld im externen RAM an

Reset-Vector ist nun 4000h

c) Codes:

Erstes CODE-SEGMENT

Zweites CODE-SEGMENT

END nie vergessen!!!!!!!!!!

Und nochmals einige wichtige Hinweise für den Anwender:

a) Bitte „**DS**“ (= define store = definiere einen Speicherplatz mit einem bestimmten Namen im angegebenen Memorybereich) **nicht mit „DB“ (= define Byte) verwechseln**!
DB darf nur innerhalb eines Programmteiles (= CODE SEGMENT) verwendet werden und speichert einen oder mehrere vorgegebene **Werte im EPROM** (z. B. eine Tabelle oder einen Text).

Beispiele:
TABELLE: DB 0,1,2,3,4,5,6,7,8,9,10,20 ;Diese **Zahlenwerte** werden als „TABELLE“ im EPROM gespeichert

Aber Vorsicht!
TABELLE: DB ‘01234567890’ ;Jetzt werden die **ASCII-Zeichen** der Zahlen 0...9 gespeichert!!

Auch das Bereitstellen von Text (z. B. zur Display-Anzeige) funktioniert:
TEXT: DB ‘Alle Lehrer sind doof’ ;Das ergibt einen diskriminierenden Text für eine Rache-Anzeige

Zahlen und Text dürfen sogar gemischt werden:
EXAMPLE: DB 10,20,50,100, ‘$’, ‘Das geht ja prima!’

b) Anstelle der verwendeten Direktive „**EQU**“ (= equate = „bedeutet“ oder „entspricht“) zur Zuweisung eines Wertes oder Namens bei einem Byte kann auch **„SET“** verwendet werden. Mit „EQU“ gilt die Zuweisung immer und ewig, solange dieses Programm verwendet wird. Mit „SET“ kann man dagegen mal schnell den Namen ändern....

Zur Simulation und zum Hardwaretest des fertigen 1khz-Assemblerprogrammes:
Siehe hierzu die getrennte „KEIL-µvision2“-Anleitung!

4.3.3. Erstellung des C-Programms „1kHz.c“

Die KEIL-Software enthält auch einen für seine Qualität weltweit bekannten C-Compiler. Außerdem werden immer mehr Programmieraufgaben heute in C gelöst. Wir wollen deshalb zweigleisig fahren und jedes Beispiel sowohl in C wie auch in Assembler bearbeiten.

Wichtig: wir arbeiten hier mit „Standard - ANSI -C“ (also nicht in C++).

Noch einige Tipps für die C-Programmierung:

a) Einzeilige Kommentare werden mit **//** eingeleitet

b) Mehrzeilige Kommentare werden zwischen **/*……*/** gesetzt

c) Werden „Unterprogramme“ verwendet -- sie heißen „**Funktionen**“ in C -- so müssen sie **vor** dem Aufruf des Hauptprogramms als „**Prototypen**“ mit einem Strichpunkt am Ende deklariert werden.

d) Am Anfang unseres C-Programms stellen wir die Liste der Spezial-Bits und Spezial-Funktionsregister unseres Controllertyps sowie die „Standard - Ein- und Ausgaberoutinen“ über die **„Header“**

#include <reg515.h> und **#include <stdio.h>**

bereit.

e) De häufigste Fehlermeldung wird durch den **vergessenen (und oft erforderlichen) Strichpunkt** am Ende einer Anweisung ausgelöst....

f) Da wir meist irgendwelche Variablen verwenden, sollte man bei ihrer Deklaration stets folgende Liste über die Wertebereiche bzw. über die Anzahl der Bits und Bytes im Kopf haben:

char (= character) bedeutet immer eine „8 Bit - = 1 Byte-Zahl“ mit Werten zwischen **-128 und +127**

unsigned char ergibt die **positiven Zahlen zwischen 0 und 255**

int (= integer) liefert 16 Bit- = 2 Byte - Zahlen mit Werten zwischen **-32768 und +32767**

unsigned int gehört dann zu Zahlen zwischen **0 und 65535**

So sieht unser 1kHz-Oszillator in C aus:

```
#include<reg515.h>
#include<stdio.h>

sbit ausgang=0x90;                  // Portpin P1.0 hat die Bitadresse 90h und heißt nun „ausgang“

void zeit(void);                    // Prototypenanmeldung

void main(void)                     // Hauptprogramm
{       while(1)                    // Endlosschleife
        {   ausgang=~ausgang;       // Zustand des Bits „ausgang“ umkehren
            zeit();                 // Zeitverzögerung aufrufen
        }
}

void zeit(void)                     // Funktion „zeit“ soll eine Zeitverzögerung von 500 Mikrosekunden liefern
{   unsigned int x;                 //Definiere deshalb dafür eine vorzeichenlose Integer-Variable x
    for(x=0;x<=200;x++);            //und zähle sie einfach von Null aus hoch
}
```

Wir tippen dieses Programm mit dem Editor und speichern es als „1kHz.c“ in unserem Projektordner „1kHz“ ab.
Zum weiteren Vorgehen: Siehe hierzu die getrennte Keil-µvision2-Anleitung.
==========

Damit sollten Sie jetzt in der Lage sein, diese **Aufgabe** erfolgreich zu lösen:

„Schließen Sie die 8 Drucktasten der entsprechenden Zusatzplatine an Port P1 des Mikrocontrollers an. Ebenso sollten Sie den Port P5 über ein weiteres Kabel mit dem Eingang des LED-Treibers verbinden, denn dadurch leuchten 8 LED entsprechend den logischen Zuständen an Port P5.
Schreiben Sie nun ein C-Programm, das die Signale an P1 einliest und dann wieder an P5 ausgibt. So sehen Sie genau, welche Taste gerade gedrückt ist und können beide Ports sowie die Hardware auf korrekte Funktion testen!“

4.4. Anwendung von Sprungbefehlen: Programmierung einer Tastenabfrage

Aufgabe: es ist eine Alarmanlage aufzubauen, die folgende Eigenschaften aufweist:

a) Am Portpin **P1.0** ist die Taste T1 angeschlossen. Wird sie betätigt (= verbindet sie den Portpin mit Masse), dann erzeugt der Controller am Portpin **P5.0** folgendes Alarmsignal:
Der Controller piepst 200 Millisekunden lang mit 500Hz, dann folgt eine Pause mit 200 Millisekunden. Anschließend wird wieder 200 Millisekunden lang gepiepst usw.

b) Das Alarmsignal kann nur durch Schließen einer weiteren Taste T2 am Portpin **P1.1** wieder abgeschaltet werden. Anschließend muss die Alarmanlage sofort wieder "scharf" sein.

Skizzieren Sie die **Schaltung** samt dem **Signalverlauf**.
Zeichen Sie einen **PAP**.
Benennen Sie die verwendeten **Portpins** mit den symbolischen Namen **„alarm", „back" und „signal**".
Entwerfen Sie die erforderlichen **Segmente.**
Legen Sie einen passenden **Projektordner** „Alarm" an.
Schreiben Sie das vollständige Programm in das **Musterformular**, speichern Sie den fertigen Modul unter dem Namen „**alarm.a51**" darin ab, wählen Sie alle erforderlichen **Optionen von Assembler und Linker**. Testen Sie das fertige Programm sowohl mit **dem „dscope"-Simulator wie auch mit der echten Hardware**. Halten Sie sich dabei an Ihre KEIL-µvision2-Anleitung.

Zusatzaufgabe:
Schreiben Sie anschließend dieses **Programm in „C"** und speichern Sie es unter dem Namen „alarm.c" im gleichen Programmordner „Alarm". Binden Sie es samt der erforderlichen StartupA51-Datei in Ihr Projekt ein und testen Sie Ihr Werk.

Lösungsbeispiel in Assembler:

```
;alarm.A51
;Erstellt am 10. Januar 2005 durch G. Kraus
;-------------------------------------------------------------------------------
$NOMOD51
$INCLUDE (REG515.INC)
;===============================================================
;Assembler-Direktiven und -
;===============================================================
OFFSET   EQU 4000h

alarm    bit P1.0
back     bit P1.1
ausgang  bit P5.0
;-------------------------------------------------------------------------------
?STACK   SEGMENT IDATA          ; Stack soll im IDATA-Bereich liegen.
         RSEG ?STACK            ; Schalte auf ?Stack-Segment um.
         DS 5                   ; Reserviere 5 Bytes für den Stack.
;-------------------------------------------------------------------------------
CSEG     AT 0+OFFSET            ; Programm-Codes beginnen bei 4000h
         jmp   START            ; reset location (jump to START)
;===============================================================
;Programmteil (Codes)
;===============================================================
STEUERUNG   SEGMENT   CODE          ; Erstes Codesegment heißt "Steuerung"
            RSEG STEUERUNG          ; Schalte auf dieses Codesegment um

START:      mov   SP,#?STACK-1      ; Zuweisung der Stack-Adresse
loop:       jnb alarm,jaulen        ; Zeitschleife aufrufen
            jmp   loop              ; Springe zu LOOP und wiederhole alles

;-------------------------------------------------------------------------------
SIGNAL      SEGMENT CODE            ;Zweites Codesegment heißt "Signal"
            RSEG SIGNAL             ;Schalte auf dieses Codesegment um

jaulen:     call pieps              ;0,2 Sekunden Piepsen
            call pause              ;0,2 Sekunden Pause
            jnb back,loop           ;Resetknopf gedrückt?
            jmp jaulen              ;wenn nicht: weiterpiepsen

pieps:      mov r7,#200             ;200 mal 1 Millisekunde vorsehen
loop1:      call zeit_1ms           ;
            cpl ausgang
            djnz r7,loop1
            ret
;
pause:      clr ausgang             ;Ausgang auf LOW
            mov r7,#200             ;200 mal 1 Millisekunde als Pause
loop2:      call zeit_1ms
            djnz r7,loop2
            ret
;
zeit_1ms:   mov r6,#2               ;2 Durchgänge zu je 500 Mikrosekunden vorsehen
nochmal:    mov r5,#250             ;Zahl 250 ergibt 2 mal 250 = 500 Mikrosekunden
loop3:      djnz r5,loop3
            djnz r6,nochmal
            ret

end
```

Musterlösung für die Alarmanlage in C:

```
#include<reg515.h>
#include<stdio.h>

sbit ausgang=0xf8;              //Ausgang an P5.0
sbit alarm=0x90;                //Alarm durch P1.0
sbit reset=0x91;                //Reset durch P1.1


void main(void)
{   unsigned int x,y;

    while(1)
    {   do {} while(alarm==1);          //Tue nix, wenn "alarm" nicht betätigt wird

        while(1)                        //Endlosschleife für Alarm (mit Reset-Abfrage)
        {   for(y=0;y<=200;y++)
            {
                for(x=0;x<=62;x++);
                ausgang=~ausgang;
            }

            if(reset==1)                //reset gedrückt?
            {   ausgang=0;              //Wenn Nein: weiterpiepsen
                for(x=0;x<=12000;x++);
            }
            else {break;}               //Wenn Ja: abbrechen und zur Tastenabfrage zurückkehren
        }

    }
}
```

4.5. Alphabetischer Stichwortkatalog zur Assemblerprogrammierung

Sie finden hier die wichtigsten Begriffe, Direktiven und Controls (aber keine Befehle!) aufgelistet, die in professionellen Programmen verwendet werden. Nach Möglichkeit ist auch ein passendes Beispiel aufgeführt. Bei selteneren Ausdrücken muss das Original-Handbuch der Firma KEIL zu Rate gezogen werden

Ausdruck	Bedeutung
BDATA	Bitadressierbarer Bereich in Ram1. Registeradressen von 20h bis 2Fh. Hier können Flags oder „Merker“ zum eigenen Gebrauch definiert werden. Festlegung des benützten Segmentes: z. B. mit MY_BITS SEGMENT BIT Nach der darauffolgenden Direktive „RSEG MY_BITS“ können darin die gewünschten, einzelnen Bits mit „.....DBIT.....“ definiert werden.
BIT	Damit wird einem Bit ein symbolischer Name zugewiesen. Beispiel: ausgang BIT P1.0 weist dem Portpin P1.0 den Namen „ausgang“ zu.
CODE	Andere Bezeichnung für „Programm“ bzw. die Hex-Codes, die ein Programm ergeben
CSEG	CODE SEGMENT = Programmteil oder Unterprogramm. Mit der Direktive „CSEG AT...“ wird ein solcher CODE-Block durch den Linker an einer bestimmten Stelle im Programm-speicher angeordnet.
DATA	Bezeichnung für Daten, Konstanten, Messergebnisse. Mit der Direktive „SEGMENT DATA“ wird für diese Größen der Speicherplatz im direkt adressierbaren RAM 1 reserviert.
DB	= DEFINE BYTE. Damit kann innerhalb eines CODE-Segmentes eine Tabelle mit irgendwelchen Konstanten usw. im Programmspeicher angeordnet werden. **Hinweis: werden die definierten Konstanten oder Zahlen in Hochkomma eingeschlossen, dann erfolgt die Ablage der Werte als ASCII-Zeichen! Verwendet man Anführungszeichen, erzeugt man eine Zeichenkette (String array), bei dem ebenfalls ASCII-Zeichen entstehen, aber am Ende „\0“ entsprechend einer Hex-Zahl 0x00 angehängt wird.....**
DBIT	Bedeutet „DEFINE BIT“. Damit können in einem mit „SEGMENT BIT“ angelegten Bitsegment sogenannte „Bitvariable“ definiert werden. Anschließend muss die Anzahl der reservierten Bits angegeben werden. Beispiel: HUGO: DBIT 5 reserviert 5 Bits im Feld „HUGO“.
DS	= DEFINE STORE. Damit können für selbstdefinierte Variable die nötigen Speicherplätze in RAM 1 reserviert werden.
EQU	= EQUATE. Damit wird z. B. einer Zahl ein bestimmter symbolischer Name zugewiesen. (Beispiel: OFFSET EQU 4000h).
END	END muss wirklich und absolut ganz am Ende eines Moduls stehen und darf nicht vergessen werden.
EXTERN	Damit werden Unterprogramme in unser Programm eingebunden, die selbst wieder zu anderen Modulen gehören.
IDATA	= indirekt adressierbarer DATA - Bereich, z. B. in RAM 2
interrupt vector	= Programmspeicherplatz-Adresse, die beim Auslösen des betreffenden Interrupts angesprungen wird. Dort findet der Controller normalerweise einen Sprungbefehl zur eigentlichen Interruptbearbeitung (= Interrupt Service Routine).
NAME	Assemblerdirektive, mit der einem Modul ein bestimmter Name zugewiesen wird.
ORG	Erzwingt, dass ein Programm ab einer bestimmten Adresse im Programmspeicher beginnt (z. B. wird mit „ORG 0000h“ unser Programm ab Adresse 0000 ins EPROM gebrannt.
PDATA	= DATA SEGMENT im externen RAM. Durch indirekte Adressierung über die Register R0 und R1 erhält man eine Möglichkeit zum schnelleren Abspeichern einer „Seite“ mit 255 Bytes.
PUBLIC	Damit wird ein Unterprogramm als „öffentlich zugänglich“ erklärt und kann von einem anderen Modul über die Direktive „EXTERN“ mitbenutzt werden.

reset location	Hier befindet sich der Label „START“ unseres Programms. Dort soll begonnen werden.
reset vector	Physikalische EPROM - Adresse, die beim RESET oder Neustart angesprungen wird.
RSEG	Nachdem ein CODE- oder DATA - Segment mit der zugehörigen Direktive (...SEGMENT CODE bzw. .....SEGMENT DATA) definiert wurde, schaltet man mit dieser Anweisung auf das betreffende Segment um. Erst dann darf mit dem Einrippen der Programmcodes usw. begonnen werden.
SEGMENT	Speicherplatzreservierungen für einen Programmteil (= CODE SEGMENT) oder eine Anordnung von Datenworten (= DATA SEGMENT bzw. XDATA SEGMENT oder IDATA SEGMENT) oder von erforderlichen Bits (SEGMENT BIT)
stack	Zwischenablage für wichtige Daten und Adressen (Beispiel: Interrupt - Bearbeitung). Wichtig: dieser Speicher ist nach dem Prinzip „LAST IN -- FIRST OUT“ aufgebaut. Abgelegt wird mit „PUSH...“, zurückgeladen mit „POP.....“
stack pointer	Adresszähler des Stacks
USING	Mit dieser Direktive wird die Auswahl einer Bank aus den vier vorhandenen Registerbänken in RAM1 bekannt gegeben. Achtung: im folgenden Programmteil muss unbedingt noch der entsprechende Umschaltbefehl (über 2 Bits im PSW - Register) enthalten sein.
XDATA	Daten-Speicherbereich im externen RAM, definiert mit „SEGMENT XDATA“.
XSEG	Festlegung der Startadresse des Daten-Speicherbereiches im externen RAM Beispiel: XSEG AT 7000h.

4.6. Projekt: Programmierung einer Mäuse-Orgel

4.6.1. Aufgabenstellung

Sie sollen mit Hilfe der beiden Zusatzplatinen „Drucktasten-Eingabe“ an Port P1 und „NF-Verstärker mit Lautsprecher“ an Portpin P5^0 eine Mini-Orgel programmieren, bei der nach dem Drücken einer Taste (....8 Stück stehen zur Verfügung.....) der zugehörige Ton aus der C-Dur-Tonleiter erklingt. Das Spielen von Akkorden oder mehrstimmiger Stücke ist nicht möglich. Schreiben Sie dieses Programm in „C“.

4.6.2. Etwas Musik-Grundlagen

Bei einer Tonleiter geht man immer von einem ***Grundton*** mit einer bestimmten Frequenz aus. In 8 Schritten (= „Intervallen“) erhöht man nun die Tonfrequenzen, bis man bei der ***doppelten Frequenz*** des Grundtones angelangt ist. Auf diese Weise hat man eine „***Oktave***“ durchlaufen. Dann wiederholt man erneut die einzelnen Schritte, bis sich wieder die Frequenz verdoppelt hat usw. Die verschiedenen Oktaven werden durch „Strichbezeichnungen“ oder Nummern (z. B. „C1“)unterschieden.

Innerhalb der C-Dur-Tonleiter haben wir es im deutschsprachigen Raum mit folgenden Einzeltönen zu tun:

C D E F G A H C

Nun ist es wichtig zu wissen, welchen Frequenzabstand die einzelnen Töne voneinander haben. Und hier geht es schon los: eigentlich ist eine Oktave sogar in 12 Einzeltöne (= „**Halbtöne**“) aufgeteilt. Leider klingt das nicht sehr angenehm, wenn man damit Musik macht -- aber es wird gemacht...(Name: „Zwölfton-Musik“)
Am angenehmsten waren und sind für uns Menschen immer noch folgende Abstände in einer Tonleiter:

C	**D**	**E**	**F**	**G**	**A**	**H**	**C**
2 Halbtöne	2 Halbtöne	1 Halbton	2 Halbtöne	2 Halbtöne	2 Halbtöne	1 Halbton	

Also, ganz perfekt stimmt das auch nicht, aber es geht da nur noch um einzelne Abweichungen in der 1... 2 Hz – Größenordnung (die man natürlich beim Stimmen eines Orchesters merkt). Johann Sebastian Bach war der Erste, der eine Angleichung vorschlug und in seiner „**wohltemperierten Stimmung**“ festlegte, dass alle Halbtöne mit demselben Faktor in ihre Brüder zur rechten oder linken Seite umgerechnet werden können.

Dieser Faktor ist $\mathbf{k} = \sqrt[12]{2}$ (Das ergibt die Zahl 1,059463094)

Und die letzte nötige Information ist die exakte Frequenz eines „Referenztons“, von dem aus alle übrigen Ganz- und Halbtonfrequenzen mit diesem Faktor bestimmt werden können. Das ist der

Kammerton „A“ mit f = 440 Hz

Beispiel: der Ton „G“ liegt um zwei Halbtöne tiefer als „A“. Also beträgt seine Frequenz

$$G = \frac{440Hz}{1,059463094 \bullet 1,059463094} = 392Hz$$

Der Ton „H“ liegt dagegen um zwei Halbtöne höher als „A“. Deshalb gehört dazu die Frequenz

$$H = 440Hz \bullet 1,059463094 \bullet 1,059463094 = 493,88Hz$$

Aufgabe: Bestimmen Sie alle Einzelfrequenzen für die C-Dur-Tonleiter und tragen Sie die Ergebnisse in eine kleine Tabelle ein:

4.6.3. Programmierung des Kammertones „A“

Wir wollen ein C-Programm schreiben, bei dem alle 8 Tasten nacheinander abgefragt werden und bei einer gedrückten Taste der gewünschte Ton erzeugt wird.

Bitte beachten:

a) Die Tastenabfrage erfolgt mit bedingten „while“-Schleifen, die in eine Endlosschleife eingebettet sind.

b) Um auch zu sehr tiefen Tönen zu kommen, muss für die Zeitverzögerung eine „unsigned integer“ – Zahl als Vorgabe benützt werden, denn damit sind Werte zwischen Null und 64535 zulässig.

Außerdem sollte man jetzt schon die einzelnen Tasten (die über Port P1 abgefragt werden!) mit den passenden Tönen bezeichnen.

Das an Portpin P5^0 ausgegebene Tonsignal erhält die Bezeichnung „ausgang“.

Damit erhalten wir folgendes Listing für unser Programm:

```
#include <reg515.h>
#include <stdio.h>

sbit ausgang=P5^0;                  // Bit-Deklaration für den Tonausgang

sbit C1=P1^0;                       // Klaviertasten für Tonleiter von Ton „C1“ bis „C2“
sbit D=P1^1;
sbit E=P1^2;
sbit F=P1^3;
sbit G=P1^4;
sbit A=P1^5;
sbit H=P1^6;
sbit C2=P1^7;

unsigned int x;                     // Global deklarierte Zählvariable vom Integer-Typ

void main(void)
{       while(1)                    // Endlosschleife
        {
                while(A==0)         // Taste für Ton A” gedrückt?
                {
                        for(x=0;x<=140;x++);        // Zeitverzögerung für eine halbe Periode von 440 Hz
                        ausgang=~ausgang;           // Ausgangsbit umkehren
                }
        }
}
```

Nehmen Sie dieses Programm in Betrieb und gleichen Sie es auf die richtige Frequenz des Kammertons “A” ab.

4.6.4. Vollständige Orgel

Ergänzen Sie nun die fehlenden 7 Töne, bringen Sie die Frequenzen auf die richtigen Werte aus Ihrer Tabelle und spielen Sie anschließend ein kleines Lied.

Auf der nächsten Seite folgt das Programm-Listing. Damit stimmen die erzeugten Frequenzen mit einer maximalen Abweichung von 1,5 Hz.....

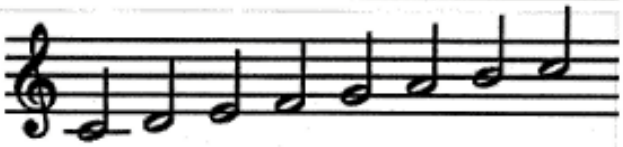
Anhang: Listing des Musterprogramms mit korrekt abgeglichenen Frequenzen

```
#include <reg515.h>
#include <stdio.h>

sbit ausgang=P5^0;

sbit C1=P1^0;
sbit D=P1^1;
sbit E=P1^2;
sbit F=P1^3;
sbit G=P1^4;
sbit A=P1^5;
sbit H=P1^6;
sbit C2=P1^7;
unsigned int x;


void main(void)
{       while(1)
        {
                while(C1==0)
                {   for(x=0;x<=236;x++);
                    ausgang=~ausgang;
                }

                while(D==0)
                {   for(x=0;x<=210;x++);
                    ausgang=~ausgang;
                }

                while(E==0)
                {   for(x=0;x<=187;x++);
                    ausgang=~ausgang;
                }

                while(F==0)
                {   for(x=0;x<=177;x++);
                    ausgang=~ausgang;
                }

                while(G==0)
                {   for(x=0;x<=157;x++);
                    ausgang=~ausgang;
                }

                while(A==0)
                {   for(x=0;x<=140;x++);
                    ausgang=~ausgang;
                }

                while(H==0)
                {   for(x=0;x<=125;x++);
                    ausgang=~ausgang;
                }

                while(C2==0)
                {   for(x=0;x<=118;x++);
                    ausgang=~ausgang;
                }
        }
}
```

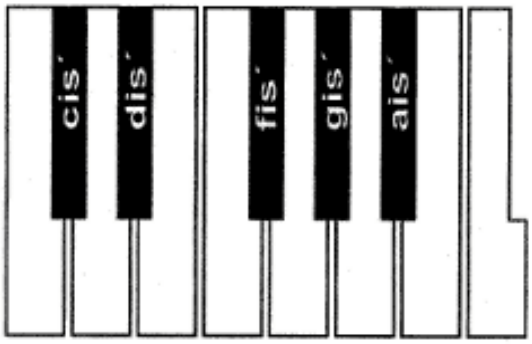
4.6.5. Anhang: Download aus dem Internet zur Kontrolle der berechneten Tonfrequenzen

Tonhöhe und Frequenz:

Die C-Dur-Tonleiter



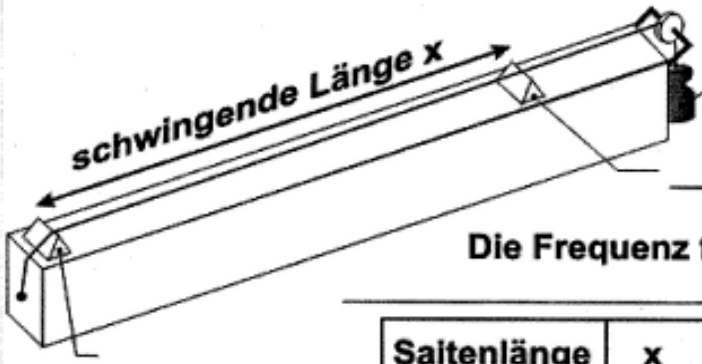
umfasst die__ Töne (_____)
von einem C-Ton zum nächsten C-Ton (hier von__ zu__).
Auf der Klaviatur benötigt man dafür _____
Töne, die eine Oktave höher liegen haben immer die_____ Frequenz.
Beispiel: c´ _____
c´´ _____



261,6Hz 329,6Hz 392,0Hz 493,9Hz
293,7Hz 349,2Hz **440,0Hz** 523,2Hz

Der **Kammerton a´ mit 440Hz** ist der Ton, _____

Monocord: (Instrument mit nur einer Saite)



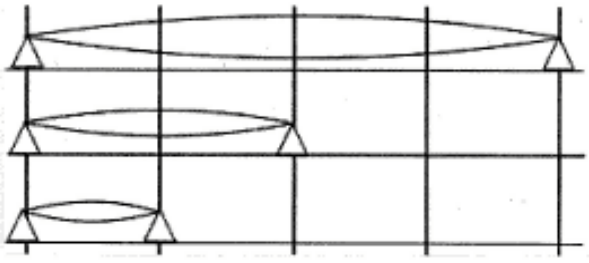
_____
_____
_____

Die Frequenz f hängt von _____ _____ ab.

_____

Saitenlänge	x	$\frac{x}{2}$	$\frac{x}{4}$
Frequenz	f		

Saitenschwingung:



Schwingungsbild im Oszilloskop: