Sphinx C-- für MenuetOS

Programmieren in Sphinx C--

Dirk Kamp
Barry Kauler

Sphinx C– Dokumentation für MenuetOS

Letzte Original Änderung 14. September 2002 von Barry Kauler.
Letzte übersetzte Änderung 27. November 2002 von Dirk Kamp.

Original Dokumentation von Peter Cellik, Mai 1995
Erweiterte Dokumentation von Barry Kauler
Deutsche Übersetzung/Erweiterung von Dirk Kamp

Inhaltsverzeichnis

1. Einführung

Normalerweise sind Computer nichts weiter als ein gewöhnlicher Schalter. Sie kennen nur die zwei Zustände “ ein” oder “ aus”. Deshalb wurden Computer in der Anfangszeit noch mit Kippschaltern bedient. Die Ausgabe der berechneten Werte erfolgte entweder über Leuchtdioden oder kleinen Lämpchen. Die damaligen Computer wurden noch richtig binär programmiert, nicht so man es heute kennt. Diese binäre Programmierweise kennt auch nur zwei Zustände, nämlich “0” und “1”. Genauso wie ein normaler Lampenschalter.

Im Laufe der Zeit entwickelten sich die CPU’s (das Herz eines jeden Computers) immer weiter, die Anzahl der Transistoren nahm zu und somit auch die Komplexität. Die ersten richtigen Betriebssysteme entstanden zu jener Zeit. Bei manchen Computern war es das einfache Basic, welches fest im ROM verankert war und somit das programmieren des Computers schon um einiges erleichterte. Bei anderen, wie beispielsweise der PDP 11 - den man damals schon als Großrechner angesehen hatte - konnte man mit dem integrierten Basic im ROM nicht viel anfangen, da es keines hatte. Dieser Rechner lief auf etwas das nachher zu Unix (es gibt wohl kaum jemanden der nicht etwas von Linux gehört hat) werden sollte. Dieses Ur-unix wurde damals noch komplett in Assembler geschrieben. Doch da die Technik weiter voranschritt, mußte eine Möglichkeit gefunden werden, dieses Betriebssystem so einfach wie möglich auf andere Rechner zu portieren. Also entwickelte sich aus einer noch älteren Sprache nach und nach die Sprache C. Das alte Unix ähnliche Betriebssystem wurde nun, bis auf ein ganz kleinen Assembler Teil , für die Sprache C umgeschrieben.

Die Programmiersprache C wurde so flexibel ausgelegt, daß sie zum einen die sehr Hardwarenahe Programmierung ermöglichte, aber auch selber leicht auf andere Rechner portiert werden konnte. So wurde diese Programmiersprachen von einem Rechner zum nächsten portiert. Als sich langsam herausstellte das diese Programmiersprache sehr großes Interesse auch bei anderen Computer Freunden weckte, wurde es langsam Zeit mal ein Buch über diese zu schreiben. So entstand von den Entwicklern dieser Programmiersprache ein Standardwerk, welches noch viele Jahre später auch als die Bibel der Programmiersprache C bezeichnet wurde. Die beiden Autoren Kernighan & Ritchie hatten diese Sprache entwickelt, um das damalige Unix ähnliche Betriebssystem von einem Computer auf einen anderen zu portieren. Sie hatten dieses Buch geschrieben um die Anwender zum einen in die Programmiersprache C einzuführen und zum anderen um ein gewissen Standard zu setzen. Doch leider ließ das Buch zu viele Fragen offen, so daß sich kein kompletter einheitlicher Standard durchsetzte. Viele der damals in Umlauf befindlichen “C” Varianten, hatten ihre besonderen Spezialitäten, so daß die Quelltexte, trotz der eigentlich identischen Sprache, nicht mehr untereinander austauschbar waren. Dieses führte wieder zu Problemen und Verwirrung.

Etliche Jahre später machte sich das ANSI Konsortium dran die Programmiersprache C zu standardisieren. Dieser Standard regelt bei der Programmiersprache C hauptsächlich drei Dinge.

Als allererstes den Grundwortschatz. In diesem Grundwortschatz sind nur die Fundamentalsten Befehle enthalten. Alles weitere, was zur Programmierung eines Computers benötigt wird, ist in sogenannten Bibliotheken ausgelagert. Egal ob es sich dabei um Ein oder Ausgabe Funktionen, Mathefunktionen oder das bearbeiten von Zeichenketten handelt. Alles was nicht irgendwie von Nöten war, wurde in die Bibliotheken ausgelagert. Das wäre Punkt 2 der Änderungen des ANSI Konsortium. Und der 3 Punkt betraf den Präprozessor.

So entstand die erste Standardisierung der Sprache “C”, auch bekannt als ANSI “C”. Doch nichts ist perfekt. Dieser ANSI C Standard wurde etliche Jahre später von dem ISO “C” Standard abgelöst, der wiederum einige Ungereimtheiten des damaligen ANSI Standards beseitigte.

Die Programmiersprache C lief inzwischen auf jeden nur erdenklichen Computer und Betriebssystem. Die Computer wurden immer leistungsfähiger, die Betriebssysteme immer größer und vor allem immer mehr grafisch basiert. Langsam aber sicher verlangten die Programmierer nach neuen Ansätzen um das programmieren für die neuen Betriebssysteme zu erleichtern. Inzwischen hatten die Programmierer ein neues Schlagwort, daß hieß “objektorientierten Programmierung” oder kurz “OOP” genannt.

Die objektorientierte Programmierung ermöglicht es dem Programmierer nicht jedesmal das Rad neu zu erfinden. Diese Art der Programmierung bringt dem Programmierer natürlich sehr viele Vorteile, gerade was das wiederverwenden von Programmmodulen angelangt. Aber, wo Licht ist, ist auch Schatten. Viele dieser Vorteile erkaufen sich die Programmierer leider dadurch, daß die Programme künstlich aufgebläht werden. Man schaue sich mal so manch ein Windows Programm an. In den ersten Tagen der Betriebssysteme, wie beispielsweise MS-DOS, paßte eine Textverarbeitung ohne Probleme auf eine einzige Diskette (natürlich ohne Rechtschreibkorrektur und etlichen grafischen Spielereien). Doch das was der Programmierer will, soll er auch bekommen.

Inzwischen ist fast jedes Programm, zumindest die, die für Windows geschrieben wurden, meistens in einer Programmiersprache geschrieben, die “OOP” tauglich ist. Für Anwendungen ist diese Art der Programmierung natürlich sehr gut geeignet, da es sehr vieles vereinfacht. Doch will man Hardwarenah programmieren oder ein neues Betriebssystem entwickeln, so ist diese Sprache absolut nicht geeignet. Ich will nicht sagen, das es unmöglich ist, aber eben doch - vorallem wenn auf die Codegröße achten muss - nicht so das wahre.

Wie heißt es immer so schön bei den Amerikanern:” Back to the roots”. Also wieder zurück zum Ursprung. Für eine wirklich schnelle und effiziente Programmierung der CPU gibt es nur die Assembler Sprache. Doch auf einen gewissen Komfort möchte der Programmierer natürlich auch nicht verzichten. Weshalb man dann doch lieber wieder zur Programmiersprache “C” greift.

C– bietet dem Programmierer nun etwas von beiden Welten. C– ist eigentlich weder ein C Compiler noch ein Assembler einzelnen, sondern ein C Compiler mit integrierten Assembler, oder ein Assembler mit integrierten C Compiler. Je nachdem wie es der Programmierer gerne hätte. (Obwohl der Grundstock doch bei C liegt)

C– bietet dem Programmierer nun die Möglichkeit, nicht nur einfach zwischen der Programmiersprache C und Assembler hin und her zuschalten, sondern beide zusammen und parallel zu benutzen. Dadurch ist der Programmierer in der Lager die Vorteile beider Programmiersprachen zu verwenden. Er kann den kompletten Befehlssatz, sowie die Register, der jeweiligen CPU (angefangen vom 8086 bis Pentium irgendwas) voll ausnutzen und verwenden. Zusätzlich kann er Teile seines Programmes schön strukturiert in der Programmiersprache C verfassen..

Ich werde nun in dieser Übersetzung versuchen, neben der eigentlichen Übersetzung, auch ein wenig mehr auf Anfänger der Programmiersprache C einzugehen. Nicht nur deshalb weil es vielleicht einige davon gibt, sondern weil ich selbst dazu gehöre. Bis vor etwa vier Wochen war mir die Programmiersprache C zwar bekannt, ich hatte auch schon einiges darüber gelesen und ein wenig damit rumgespielt, aber praktische Erfahrungen waren auch bei mir bisher nicht vorhanden.

Durch einen kleinen Zufall bin ich auf der Suche im Internet auf ein Betriebssystem gestoßen, welche sich noch in der Entwicklung befindet. Es ist sehr klein, bietet eine grafische Oberfläche, ist zu 100 Prozent in Assembler geschrieben und paßte inklusive etlicher kleiner Programme und Tools auf eine einzelne Diskette. Das Betriebssystem nennt sich “ MenuetOS” und wird von einigen Leuten im Internet programmiert. Es haben sich schon einige weitere Programmierer, die Tools, Anwendungen und Programmiersprachen für dieses Betriebssystem entwickeln, zusammengefunden. Da ich mich bisher vor der Programmierung in Assembler gescheut habe, suchte ich für dieses Betriebssystem einen C- Compiler und wurde fündig. Daß er durch seinen integrierten Assembler geradezu prädestiniert ist um Programme für MenuetOS zu schreiben, war für mich erst recht ein Grund mich mit diesem genauer auseinander zu setzen.

Lange Rede kurzer Sinn. Ich werde versuchen meine eigenen Erfahrungen, was das erlernen der Programmiersprache C angelangt, in Worte zu fassen und diese hier in der Dokumentation zu diesem Compiler nieder zu schreiben. Ich hoffe diese sind auch für Anfänger gut verständlich.

Jetzt kann es losgehen!!

2. Bezeichner und Symbole
2.1 Namensgebung von Bezeichnern

Alles in einer Programmiersprache wie Funktionen, interne Befehle und Variablen müssen einen eindeutigen Namen haben, damit der Compiler auch weiß was er damit machen soll. Wie in jeder anderen Programmiersprache, so gibt es auch bei C entsprechende Richtlinien die erfüllt werden müssen, damit man von dem Compiler keine Fehlermeldung erhält. C— Bezeichner (darunter fallen interne Funktionen, externe Funktionen und Variablen) müssen entweder mit einem Unterstrich (_) oder einem Klein- oder Großbuchstaben beginnen..

Danach können Kombinationen von Unterstrichen, Groß- oder Kleinenbuchstaben, so wie Zahlen folgen (0-9). Die maximale Länge eines Bezeichners darf 64 Zeichen nicht überschreiten. Einige Beispiele für korrekte Bezeichner:

```
_DOG
CoW
loony12
HowdYBoys_AND_Girls
WOW___
x
```

Einige Beispiele von falschen Bezeichnern:

```
12bogus
y_es sir
the-end
```

Alle diese drei Beispiele werden vom Compiler mit einer Fehlermeldung quittiert. Das erste Beispiel fängt mit einer Zahl an, was aber in C nicht zulässig ist. Das zweite Beispiel fängt zwar korrekt an beinhaltet aber ein Leerzeichen (“ “) in dem Bezeichner. Dadurch kann der Compiler nun aber nicht unterscheiden, ob es sich dabei um einen oder zwei Bezeichnern handelt. Deshalb bekommt man auch eine Fehlermeldung ausgespuckt. In dem letzten Beispiel wird ein Bindestrich (“-”) in dem Namen verwendeten, was in C auch nicht zulässiges.

In C ist der Bindestrich “-“ ein Operator. Das Bedeutet, das dieses Zeichen etwas mit etwas anderem macht und daraus dann wieder ein Ergebnis herauskommt. Natürlich kennt man dieses Zeichen auch aus der Mathematik. Der Compiler könnte somit annehmen, das er den zweiten Bezeichner vom Ersten abziehen sollte. Doch wohin soll der Compiler das Ergebnis ablegen ? Da er dieses nicht weiß, ist diese Schreibweise bei Bezeichnern auch nicht erlaubt und führt zu einer Fehlermeldung.

2.2 Symbole

Symbole sind Einzelzeichen oder Zeichenkombinationen die einfache Funktionen ausführen und die lesbarkeit erleichtern. Mal ein kleines Beispiel.

C— kennt eine sehr geniale Funktion die das vertauschen von Inhalten erleichtert. Anstatt jetzt großartig mit sogenannten Hilfsvariablen herum zu hantieren, und alles kreuz und quer zuzuweisen, verwendet man in C— einfach das Symbol “><“, wobei davor, wie auch danach, jeweils einer der Variablen steht. Und schon sind die Inhalte ausgetauscht. In einem späteren Kapitel wird das “Swap“ vertausch Symbol aber noch genauer beschrieben. Nun ein Liste der vorhandenen Symbole:

Der Aufbau zeigt das Symbol, danach eine Kurzbeschreibung gefolgt von einem kleinen Beispiel :

```
/*   =    Anfang eines Kommentarbereiches
          Bsp.: /* Kommentar */

*/   =    Ende eines Kommentarbereiches
          Bsp.: /* Kommentar */

//   =    Einzelne Kommentarzeile
          Bsp.: // Kommentar

=    =    Zuweisung
          Bsp.: AX = 12;

+    =    Addition
          Bsp.: AX = BX + 12;

-    =    Subtraktion
          Bsp.: house = dog - church;

*    =    Multiplikation
          Bsp.: x = y * z;

/    =    Division
          Bsp.: x1 = dog / legs;

&    =    bitweises UND
          Bsp.: polution = stupid & pointless;

|    =    bitweises inklusive ODER
          Bsp.: yes = i | mabe;

^    =    bitweises exklusive ODER
          Bsp.: snap = got ^ power;

<<   =    Bits nach links schieben
          Bsp.: x = y << z;
```

```
>>   =    Bits nach rechts schieben
          Bsp.: x = y >> z;

+=   =    Addition + Zuweisung
          Bsp.: fox += 12; entspricht fox = fox +12;

-=   =    Subtraktion + Zuweisung
          Bsp.: cow -= BX; entspricht cow = cow - BX;

*=   =    Multiplikation + Zuweisung
          Bsp.: cow *= dog; entspricht cow = cow * dog;

/=   =    Division + Zuweisung
          Bsp.: cow /= dog; entspricht cow = cow / dog;

&=   =    bitweises UND + Zuweisung
          Bsp.: p &= q; entspricht p = p & q;

|=   =    bitweises inklusive ODER + Zuweisung
          Bsp.: p |= z; entspricht p = p | z;

^=   =    bitweises exklusive ODER + Zuweisung
          Bsp.: u ^= s; entspricht u = u ^ s;

<<=  =    Bits nach links schieben + Zuweisung
          Bsp.: x <<= z; entspricht x = x << z

>>=  =    Bits nach rechts schieben + Zuweisung
          Bsp.: x >>= z; entspricht x = x >> z

><   =    vertausche
          Bsp.: x >< y;  vertauscht Wert von x und y

==   =    gleich wie
          Bsp.: IF(AX == 12)

>    =    größer als
          Bsp.: IF(junk > BOGUS)

<    =    kleiner als
          Bsp.: if( x < y )

>=   =    größer oder gleich wie
          Bsp.: if(AX >= 12)

<=   =    kleiner oder gleich wie
          Bsp.: IF(BL <= CH)

!=   =    nicht gleich wie
          Bsp.: IF(Frau != Mann)

<>   =    unterschiedlich wie
          Bsp.: IF(Katze <> Hund) identisch mit !=
```

```
@    =    Befehl einfügen
          Bsp.: @ COLDBOOT(); insert COLDBOOT code

:    =    dynamische Prozedur
          Bsp.: : funktionname () Funktionsnamen definieren

$    =    Assemblerbefehl
          Bsp.: $ PUSH AX   Legt AX auf den Stapel

#    =    offset Adresse von
          Bsp.: loc = #cow;  loc = addresse von cow

#    =    Compiler Präprozessor Befehl
          Bsp.: #define cow dog;

!    =    NICHT Operation
          Bsp.: !x_var; if(!xflag)

...  =    jegliche Anzahl an Parametern
          Bsp.: void proc(...);

::   =    erlaubt Sichtbarkeit
          Bsp.: ::var=0;

~    =    Symbol momentan nicht benutzt.
```

2.3 Reservierte Bezeichner

Die folgende Liste zeigt die reservierten Bezeichner von C-- , welche nicht als Bezeichner für Variablen oder Funktionsnamen in ihrem Programm verwendet werden dürfen (stand C- - v0.238). Diese Bezeichner sind schon intern im Compiler definiert oder reserviert und die Benutzung als neue Bezeichner in Ihrem Programm würde zu einer Fehlermeldung führen. Deshalb sollten Sie darauf achten, diese nicht zu verwenden.

Diese Liste können sie sich jederzeit anzeigen lassen, indem sie den C- - Compiler mit der Kommandozeilenoption /WORDS starten :

```
BREAK           CARRYFLAG       CASE            CONTINUE
ELSE            EXTRACT         FALSE           FOR
FROM            GOTO            IF              LOOPNZ
MINUSFLAG       NOTCARRYFLAG    NOTOVERFLOW     NOTZEROFLAG
OVERFLOW        PLUSFLAG        RETURN          SWITCH
TRUE            WHILE           ZEROFLAG        __CODEPTR__
__COMPILER__    __DATAPTR__     __DATESTR__     __DATE__
__DAY__         __FILE__        __HOUR__        __LINE__
__MINUTE__      __MONTH__       __POSTPTR__     __SECOND__
__TIME__        __VER1__        __VER2__        __WEEKDAY__
__YEAR__        _export         asm             break
byte            case            cdecl           char
continue        default         do              dword
else            enum            extern          far
fastcall        float           for             goto
if              inline          int             interrupt
long            loop            loopnz          pascal
return          short           signed          sizeof
static          stdcall         struct          switch
union           unsigned        void            while
word            ESCHAR          ESBYTE          ESINT
ESWORD          ESLONG          ESDWORD         ESFLOAT
CSCHAR          CSBYTE          CSINT           CSWORD
CSLONG          CSDWORD         CSFLOAT         SSCHAR
SSBYTE          SSINT           SSWORD          SSLONG
SSDWORD         SSFLOAT         DSCHAR          DSBYTE
DSINT           DSWORD          DSLONG          DSDWORD
DSFLOAT         FSCHAR          FSBYTE          FSINT
FSWORD          FSLONG          FSDWORD         FSFLOAT
GSCHAR          GSBYTE          GSINT           GSWORD
GSLONG          GSDWORD         GSFLOAT

AX    CX    DX    BX    SP    BP    SI    DI
EAX   ECX   EDX   EBX   ESP   EBP   ESI   EDI
AL    CL    DL    BL    AH    CH    DH    BH
ES    CS    SS    DS    FS    GS    ST(0) ST(1) ST(2) ST(3) ST(4) ST(5) ST(6)
ST(7) ST
st(0) st(1) st(2) st(3) st(4) st(5) st(6) st(7) st
```

Verwenden sie beim compilieren Ihres Programmes die Kommandozeilenoption “/ia” (oder “#pragma option ia” im Quelltext) verwenden, können Sie Assemblerbefehle ohne die Schlüsselwörter “asm” oder “$” verwenden. Dadurch werden aber auch alle Assembler Befehle zu reservierten Bezeichner, so daß Sie diese für Ihre eigenen Variablen und Funktionen nicht mehr verwenden können. Sie müssen sich somit zwangsweise neue einfallen lassen.

Beachte Sie, das die Prozessor Registernamen groß geschrieben werden, wenn sie nur für C— verwendet werden. Verwenden Sie allerdings Assembler Befehle, so können Sie die Registernamen sowohl klein als auch groß schreiben. Folgende definierte Bezeichner gibt es:

```
ax     cx     dx     bx     sp     bp     si    di
eax    ecx    edx    ebx    esp    ebp    esi    edi
al     cl     dl     bl     ah     ch     dh    bh
es     cs     ss     ds     fs     gs
DR0    DR1    DR2    DR3    DR4    DR5    DR6    DR7
CR0    CR1    CR2    CR3    CR4    CR5    CR6    CR7
TR0    TR1    TR2    TR3    TR4    TR5    TR6    TR7
MM0    MM1    MM2    MM3    MM4    MM5    MM6    MM7
XMM0   XMM1   XMM2   XMM3   XMM4   XMM5   XMM6   XMM7
dr0    dr1    dr2    dr3    dr4    dr5    dr6    dr7
cr0    cr1    cr2    cr3    cr4    cr5    cr6    cr7
tr0    tr1    tr2    tr3    tr4    tr5    tr6    tr7
mm0    mm1    mm2    mm3    mm4    mm5    mm6    mm7
xmm0   xmm1   xmm2   xmm3   xmm4   xmm5   xmm6   xmm7
```

2.4 Automatische Register

Beim erstellen von Bibliotheks Prozeduren wäre es normalerweise notwendig unterschiedliche Prozeduren für den 16 Bit und 32 Bit Modus des Prozessors zu schreiben, da einmal die 16 Bit Register oder die 32 Bit Register verwendet werden. Um den Aufwand so klein wie möglich so halten, bietet C— dem Programmierer die Möglichkeit, Prozeduren für beide Prozessor Modi zu entwickeln. Dabei wird für die entsprechenden Register eine leicht abgewandelte Form der normalen Schreibweise verwendet:

```
(E)AX = 0;
```

Der Compiler wird während des compilierens nun je nach eingestellten Modus mal das AX Register verwenden (16 Bit) oder EAX benutzen (32 Bit). Nun verwendet der Compiler beim Compilieren einmal das AX Register bei 16 Bit Code oder EAX bei 32 Bit Code. Durch Verwendung dieser Schreibweise wird dazu beigetragen die Bibliotheksdateien klein und kompakt zu halten. Das verwenden der automatischen Register erlaubt es die Bibliotheksdateien zu vereinfachen und hält diese kleiner.

2.5 Vordefinierte Bezeichner

Diese Bezeichner geben Auskunft darüber, welche Art von Programm gerade compiliert wird.

__TLS__	Compilieren unter Windows (w32,w32c,dll).
__DLL__	Compilieren einer dll Datei.
__CONSOLE__	Compilieren einer Windows Konsolenanwendung.
__WIN32__	Compilieren einer grafischen Windowsanwendung.
__FLAT__	Compilieren im 32 bit modus.
__MSDOS__	Compilieren im 16 bit modus.
__TINY__	Verwende Speichermodel “tiny” im 16 bit modus.

__SMALL__	Verwende Speichermodel “small” im 16 bit modus.
__DOS32__	Compiliere im 32 bit DOS modus.
__COM__	COM Datei wird compiliert.
__SYS__	SYS Datei wird compiliert.
__ROM__	ROM Datei wird compiliert.
__OBJ__	OBJ Datei wird compiliert.
__TEXE__	EXE Datei im “tiny” Speichermodel wird compiliert.
__EXE__	EXE Datei im “small” Speichermodell wird compiliert.
codesize	Datei wird im Hinblick auf die Größe optimiert.
speed	Datei wird im Hinblick auf die Geschwindigkeit optimiert.
cpu	Definiert den Prozessortyp für den die Datei erstellt wird : 0 - 8086 1 - 80186 2 - 80286 3 - 80386 4 - 80486 5 - Pentium 6 - Pentium MMX 7 - Pentium II

Diese Bezeichner - oder auch Variablen genannt - können mit Hilfe der Präprozessor Befehle “#ifdef” oder “#ifndef” auf ihre Werte oder Wertigkeit abgefragt werden. Der Bezeichner “cpu” kann verwendet werden um den Prozessor abzufragen, für den das gerade compilierende Programm gedacht ist. Folgend ein Beispiel:

```
#ifdef cpu > 3   //ist der Prozessortyp größer als 80386
```

Mit dieser Abfrage können sie nun während der Compilierung feststellen, für welchen Prozessortyp das Programm gerade Compiliert wird. Doch welchen nutzen soll das ganze haben ?

Ganz einfach. Stellen Sie sich mal vor, sie haben eine kleine Textverarbeitung geschrieben, die sowohl unter DOS wie auch als Windowskonsolen Anwendung laufen soll. Die Compilierungsarten sind für beide Betriebsysteme unterschiedlich. Einmal muß ihre kleine Textverarbeitung für DOS im 16 Bit Modus und einmal für Windows im 32 Bit Modus Compiliert werden. Um nun nicht für jedes Betriebssystem einen extra Quellcode anzulegen, können Sie ihr Programm auch so auslegen, das es für beide funktioniert. Mit Hilfe der Präprozessor Befehle können Sie nun einfach bestimmte Programmteile, die nur auf das jeweilige Betriebssystem passen, ausklammern. Das ganze nennt man “Bedingte Compilierung“. Es wird also nur dann etwas Compiliert, wenn eine Bedingung erfüllt ist.

Sie rufen dann den Compiler einfach mit den entsprechenden Kommandozeilenoptionen für das jeweilige Betriebssystem auf und fragen diese Optionen in Ihrem Programm während der Compilierung ab. So Compilieren Sie einmal eine kleine Textverarbeitung für DOS und einmal als Windowskonsolenanwendung und das ganze mit ein und demselben Quellcode.

3. Konstanten
3.1 Numerische Konstanten

Numerische Konstanten sind fest vorgegebene Werte die auch innerhalb des Programmes nicht mehr verändert werden können. Solche konstanten werden sehr oft eingesetzt um das Programm besser lesbar zu machen. Als Beispiel sei mal die deutsche Mehrwertsteuer genannt, die zur Zeit 16% beträgt. Um nicht überall im Programm diese Zahlen schreiben zu müssen kann man auch hergehen und definiert die Konstante als “Mwst”. Diese Definition erfolgt mit Hilfe eines Präprozessorbefehles und würde wie folgt aussehen:

```
#define Mwst 16
```

Nun würde überall in ihrem Programm, wo sie die Konstante “Mwst” verwenden, diese beim compilieren durch den Werte 16 ersetzt.

Die Schreibweise von numerische Konstanten in Dezimal (Basis 10) oder Hexadezimal (Basis 16) ist in C identisch. Um eine Konstante als Binäre (Basis 2) Zahl anzugeben, schreibt man vor den 0´en und 1´sen einfach “0b” ohne Leerzeichen. Um eine numerische Konstante in Oktaler (Basis 8) Schreibweise dazustellen, schreibt man vor den Oktalen Zahlen einfach “0o”, wiederum ohne Leerzeichen.

Einige Beispiele :

```
0b11111111      // Binär. Entspricht 255 Dezimal
0x00F           // Hexadezimal. Entspricht 15 Dezimal.
0o10            // Oktal. Entspricht 8 Dezimal.
```

Diese Schreibweisen werden sowohl bei C— als auch bei der Programmierung mit dem Assembler verwendet. Zusätzlich, wegen der traditionellen Assembler Programmierung unter MASM (der Assembler der Firma Microsoft), können Hexadezimale Zahlen auch mit einem “h” oder “H” am Ende dargestellt werden. Beispiel :

```
000Fh
127AH
```

Es ist möglich Konstanten direkt als entsprechenden Datentyp zu definieren, indem man die Kürzel “L”, “U” und “F” verwendet. Momentan ignoriert der Compiler noch diese Information.

Groß und Kleinschreibung kann verwendet werden. Einige Beispiele :

```
#define DEF1 1023L      //”long” (Vorzeichenbehaftet 32-bit integer) Wert.
#define DEF2 2561Lu     //”unsigned long” (Vorzeichenlos 32-bit integer).
#define DEF3 3.02F      //”float” (32-bit fließkommazahl).
```

3.2 Zeichen Konstante

Einzelzeichen Konstanten sind in der Sprache C in Hochkommas (‘) eingebettet.

Zusätzlich gibt es in C noch besondere Sonderzeichen, die mit einem Backslash (\) beginnen und aus einem oder mehrere zusätzlichen Zeichen bestehen.

Unterstützte Sonderzeichen sind :

```
‘\a’        /* gibt einen piepton aus*/
‘\b’        /* Korrekturtaste <- (Schreibmaschine läßt grüßen;-)*/
‘\f’        /* formular vorlauf */
‘\l’        /* Zeilen Vorschub */
‘\n’        /* Neue Zeile */
‘\r’        /* Zeilen Anfang */
‘\t’        /* nächsten Tabulatorpunkt anspringen */
‘\x??’      /* das ASCII Zeichen wird aus den beiden ?? Erzeugt.
            Es werden zwei Hexadezimal Zahlen benötigt. */
‘\???’      /* ASCII Zeichen welches mit Hilfe von drei dezimal Zahlen darge
               stellt wird. */
```

Jedes andere Zeichen das nach einem Backslash (\) folgt, wird akzeptiert. Damit ist es möglich u. a. auch das “NULL” Zeichen zu verwenden, welches verwendet wird um bei String Zeichenketten das Ende zu markieren. Dabei sollte man beachten das das “NULL” Zeichen zwar wie folgt dargestellt wird “\0”, aber nicht das Zeichen “0” repräsentiert, sondern die Zahl 0 darstellt.

Zeichenketten Konstanten - also solche die aus mehr als nur einem Zeichen bestehen - werden auch in C— unterstützt. Hier einige Beispiele für Zeichenketten Konstanten:

‘ab’
‘the’
‘this is large’

Es gibt keine Begrenzung von der Anzahl der Zeichen in einer Zeichenketten Konstante, aber nur die letzten vier Zeichen sind wichtig. Diese vier Zeichen sind das maximale was in 32 Bit gespeichert werden kann, also in vier Byte. Das bedeutet aber auch, daß die Konstante ‘this is large’ identisch ist mit der Konstante ‘arge’. Da die letzten vier Zeichen ja identisch sind. Deshalb sollten sie darauf achten, daß sich Zeichenketten Konstanten in den letzten vier Buchstaben unterscheiden. C— behandelt alle Zeichenketten Konstanten wie Numerische Zahlen.

Verwendet also nicht die Zeichen direkt, sondern die entsprechenden ASCII Werte. Bei mehrfach Zeichenketten Konstanten ist das erste Zeichen signifikant, deshalb ist der Wert von ‘ab’ gleich ‘a’*256+’b’..

Um den letzten Satz genauer zu verstehen, müssen wir mal kurzzeitig in die Binäre Rechnerei einsteigen. Wie wir wissen (oder besser gesagt, nehmen wir es mal an ;-) ist ein Byte - was ein einzelnes Zeichen entspricht - 8 Bit lang. Mit 8 Bit kann man nun 256 verschiedene Zeichen darstellen (von 0 bis 255). Rechnen wir nun mit 16 Bit, so bedeutet die obrige Zeile nichts weiter, als das wir den Buchstaben “a“ mit 256 multiplizieren und somit um 8 Bit verschieben. Dabei werden die unteren 8 Bit wieder frei und es kann der Buchstabe “b“ dort abgelegt werden.

3.3 String Konstanten

String Konstanten werden in Anführungszeichen (‘’) eingebettet. Auch hier gilt für die speziellen- und Sonderzeichen die mit einem Backslash (\) beginnen, genau das gleiche wie für die Zeichenketten Konstanten. Diese Sonderzeichen sind auch die gleichen wie sie schon bei den Zeichenketten Konstanten beschrieben worden sind. Verwendet man also bei einer String Konstante beispielsweise das Sonderzeichen “\n”, so bewirkt dies, daß der Text Cursor einer Zeile tiefer und zum Text Anfang springt.

Die maximale Länge einer String Konstante ist 2048 Zeichen inklusive dem 0 Zeichen zur Kennzeichnung des String Endes. Somit hat man maximal 2047 Zeichen zur Verfügung.

3.4 Konstante Ausdrücke

Ein Konstanter Ausdruck ist eine einzelne numerische Konstante oder eine Liste von Numerischen Konstanten die mit mathematischen Operationen, während der Compilierung, zu einer einzelnen Numerische Konstante berechnet werden. Wie bei allen Ausdrücken in C— werden diese von links nach rechts berechnet..

Dies ist ein großer Unterschied zu allen anderen Programmiersprachen, wenn man bedenkt das bei dieser Art des Rechnens 2 + 3 *2 = 10 und nicht 8 ist, wie es bei der Beachtung der Notationsregeln eigentlich sein sollte.

Einige Beispiele von berechneten Konstanten Ausdrücken:

```
45 & 1 + 3              // ergibt 4
14 - 1 / 2                   // ergibt 6
1 * 2 * 3 / 2 + 4       // ergibt 7
```

Die oberen Beispiele sind integer (Ganzzahl) Konstanten, weil “14-1/2” als Ergebnisse “6” ergibt. Also keine Nachkommastellen berücksichtigt werden. Natürlich unterstützt C— auch float - Gleitkomma - Datentypen.

Der Compiler erkennt automatisch eine Gleitkomma Konstante, wenn diese einen Nachkomma Anteil hat..

```
float y;            //definiert eine 32-bit Gleitkomma variable
  y=80+0.375;
```

Dieses legt den 32 Bit Wert " 0x42A0C000" in " y". Dieser Wert entspricht der Gleitkomma Zahl "80. 375". Folgende Darstellungs-formate werden vom Compiler für Gleitkommazahlen akzeptiert:

```
   0.98
   -15.75
   3.14e2
   1.234567E-20
```

"E" oder "e" bedeutet "Exponent". Das vorletzte Beispiel hat somit den Wert 3. 14*(10^2). Der Compiler rechnet diese Werte automatisch in einer Gleitkommazahl um.

4. Datentypen
4.1 Typen von Variablen

Es gibt acht Speicher Variablentypen in C—, diese sind byte, word, dword, char, short, int, long, und float. Die folgende Tabelle zeigt die Größe und den Wertebereich jedes Variablentyps

NAME	Größe (Bytes)	Werte Bereich (dezimal)	Werte Bereich (hex)
byte	1	0 to 255	0x00 to 0xFF
word	2	0 to 65535	0x0000 to 0xFFFF
dword	4	0 to 4294967295	0x00000000 to 0xFFFFFFFF
char	1	-128 to 127	0x80 to 0x7F
short	2	-32768 to 32767	0x8000 to 0x7FFF
long	4	-2147483648 to 2147483647	0x80000000 to 0x7FFFFFFF
float	4	+/-1.17E-38 to +/-3.37E38	

Äquivalente Datentypen:

byte	unsigned char
int	short or long <<<CAREFUL HERE!
word	unsigned short
dword	unsigned long

Beachten Sie das die Grundeinstellung des Compilers so ist, das es jede 16 Bit Variable auf einer gerade Adresse legt.

In dem Beispiel unten, bekam die Variable "vchar" noch ein zusätzliches Byte im Speicher verpaßt, so daß die Variable "vshort" an einer geraden Adresse im Speicher beginnt. 32 Bit Variablen beginnen immer an Adressen die ein Vielfaches von 4 sind.

So bekommt im unteren Beispiel die Variable “vshort” noch einmal 16 Bit (2 byte) Speicher angehängt, so daß die Variable “vint” (32 Bit) ihr direkt folgen kann.

Das Ergebnis ist ein sehr kompakter Code.

```
cpuspeed.c— 22: char vchar=0;;
00000036 0000       db       0,0
cpuspeed.c— 23: short vshort=0;;
00000038 00000000   dw       0,0
cpuspeed.c— 24: int vint=0;;
0000003C 00000000        dd       0
cpuspeed.c— 25: long vlong=0;;
00000040 00000000        dd       0
cpuspeed.c— 26: word vword=0;;
00000044 0000                      dw       0
```

!!! Warnung wegen des Datentyps ‘int’

Michael folgte der C Praktik, in der die Größe einer “int” Variablen durch die Größe der verwendeten Register Bits des jeweiligen Prozessors bestimmt wird. Beispielsweise sind bei DOS Programmen die Register nur 16 Bit breit, während sie bei Windowsanwendungen (oder der direkten 32 Bit Programmierung) 32 Bit breit sind.

Um irgendwelchen Problemen schon im Vorfeld aus dem Weg zu gehen, sollten Sie diesen Datentyp deshalb besser nicht verwenden. Sie können statt dessen den “Short” Datentyp für 16 Bit Vorzeichenbehaftete , und den Datentype “long” für 32 Bit Vorzeichenbehaftete Variablen verwenden, ebenso die Datentypen “word” und “dword” für Vorzeichenlose 16 Bit und 32 Bit Variablen.

Es gibt noch einen weiteren Grund warum man den Datentype “int” nicht verwenden sollte. Es ist nämlich auch noch ein Befehl. “Int” ist ein Assembler Befehl um ein Interrupt auszulösen. Benutzen sie die Kommandozeilenoption “/ia”, so werden alle Assembler Befehle zu C— Schlüsselwörter und sind somit reserviert. Was dazu führt, das Sie den Datentyp “int“ somit sowieso nicht verwenden können, ohne auf Probleme zu stoßen.

4.2 Variablen Deklaration

Wie deklariert man Variablen. Vielleicht sollten wir erst einmal klären, was das überhaupt ist. Immerhin haben wir den Ausdruck ja schon mehrfach verwenden. Variablen sind Behälter in denen während der Programmausführung Daten gespeichert werden. Egal ob Sie irgend etwas berechnen, der Anwender eine Eingabe macht oder Sie Daten in irgendeiner Form ausgeben. Jederzeit sind dabei Variablen im Spiel. Ohne sie geht nichts. Nur woher soll der Compiler wissen, was er da eigentlich speichern soll. Wir müssen ihm also sagen, wie der Inhalt dieser Variable aussieht. Deshalb muß jede Variable vorher definiert und einem Datentyp zugewiesen werden.

Dabei ist es dem Compiler nicht nur wichtig zu wissen, welchen Datentyp die entsprechende Variable hat, sondern auch, wieviel Speicher er dafür reservieren muß. Wie wir schon gesehen haben, muß der Compiler für die verschiedenen Datentypen auch unterschiedlich viel Speicher reservieren.

Die Variable selbst ist ja nur ein Bezeichner. Also nur ein Name, damit der Compiler weiß womit er es zu tun hat. Intern, also nach der Compilierung, verwendet das Endprogramm nicht mehr den Namen sondern eine entsprechende Speicheradresse und kann mit dem alten Namen nichts mehr anfangen.

Das deklarieren von Variablen sieht wie folgt aus :

```
Datentyp Bezeichner;
```

Wobei “Datentyp” einer von folgenden sein kann : char, byte, int, word, long, dword und float.

Weitere Variablen desselben Typs lassen sich die folgt deklarieren:

```
Datentyp Bezeichner, Bezeichner,..., Bezeichner;
```

Eindimensionale Felder werden wie folgt deklariert:

```
Datentyp Bezeichner[Anzahl];
```

Wobei “Anzahl” ein konstanter Ausdruck für die Anzahl der Einträge des jeweiligen Datentyps ist.

Was Felder sind - auch Arrays genannt - wird noch in einer der folgenden Kapitel genauer beschrieben, weshalb ich hier nicht weiter darauf eingehen möchte, ausser das es sich dabei um eine hintereinander liegende Kette von Variableneinträge der selben Variable und des selben Datentyps handelt. Man denk an eine einzeilige Tabelle. Beispielsweise das Mathe-Schreibheft aus Schulzeiten, wo in jedem Kästchen eine Zahl paßte. ;-)

Felder können auch ohne eine “Anzahl” deklariert werden. Dabei werden dann die bei der Initialisierung zugewiesenen Werte als “Anzahl” genommen. Dabei bedeutet ‘Initialisierung’ nichts weitere, als das man der Variable einen Startwert zuweist. Beim Start des Programmes hat eine Variable noch keinen definierten Wert. Oder anders ausgedrückt - wer weiß schon was in der Speicherstelle steht wenn das Programm startet!

Deshalb sollte man jeder Variable, wenn möglich, initialisieren und ihr einen Wert zuweisen. Felder ohne direkte Angabe einer Anzahl werden nun wie folgt deklariert :

```
Datentyp Bezeichner [] = {const1, const2};
```

Folgende Schreibweise ist bei String Felder erlaubt:

```
char msg1[]=”dies ist ein String”;
char msg1=”dies ist ein String”;
```

Manch ein Leser wird sich nun Fragen, wieso Strings als eine Feld von “char“ Einzelzeichen definiert wird. Ganz einfach : C– (wie auch C oder C++) kennen keine Strings !!

Wenn Sie ein paar Seiten zurückgehen, um sich die Auflistung der vorhandenen Datentypen anzuschauen, werden Sie feststellen, das dort nirgendwo der Datentyp “string“ zu finden ist. Strings in der Programmiersprache C sind nichts weiter als Felder vom Datentyp “char“in dessen letztem Eintrag, nach dem letzten Buchstaben, das NULL Zeichen steht. C ist einer der wenigen Programmiersprachen die keinen Datentyp für Zeichenketten (also Strings) hat.

Nun einige Beispiele für das deklarieren von globalen Variablen:

```
byte  i,j;              /* deklariert die Variablen i und j vom Typ Byte */
word  ss[10];           /* es wird ein Feld mit 10 Einträgen vom Typ word
                           erstellt */
short h,x[27];          /* deklariert die Variable h vom Typ short und ein Feld
                           mit 27 Einträge vom selben Typ als x */
long zz = 0;            /* deklariert die Variables zz vom Typ long und
                           initialisiert sie direkt mir dem Wert 0 */
```

4.3 Compilierung und Wirkungsbereich von Variablen
4.3.1 Globale Variablen

Der Wirkungsbereich einer globalen variable ist dort wo sie definiert wird. Das bedeutet das eine Variables für den Programmbereich unterhalb ihrer Deklaration sichtbar ist. Das klingt jetzt ein wenig merkwürdig, hat aber schon seine Richtigkeit, wie wir noch sehen werden.

Man kann eine Variable nicht verwenden, wenn diese erst nach dem Programmcode, indem sie verwendet wird, deklariert worden ist. Wie wir ja schon vorher gelernt haben, muß der Compiler für jede Variable Speicher reservieren, damit er diese verwenden kann.

Ein Compiler ist dumm und kann nicht denken. Deshalb geht er den Programmcode von vorne nach hinten durch. Reserviert als erstes mal Speicher, damit das Programm nachher diesen verwenden kann und compiliert dabei die einzelnen Prozeduren und was sonst noch alles im Programm vorhanden ist. Doch was passiert, wenn der Compiler auf einen Bezeichner einer Variable stößt für die er keinen Speicher reserviert hat. Der Compiler wird natürlich meckern, das er diese Variable nicht kennt. Damit der Compiler Speicher reservieren kann, muß die Variable also immer vor dem Bereich deklariert worden sein, indem sie nachher auch benutzt wird.

Hinweis: Auch wenn der Compiler das vorwärts Referenzieren zuläßt.

Von Interesse dürfte nun sein, wo der Compiler den Speicherbereich der einzelnen Variablen im Programmcode der Ausführbaren Datei ablegt. Für die eigentliche Programmausführung ist das uninteressant, aber in manchen Situationen kann es doch schon günstig sein, wenn man weiß, wo der Speicherplatz einer Variablen hingelegt wird.

Initialisierte Variablen, also solche die direkt einen Wert bei der Deklarierung zugewiesen bekommen haben, werden bei der Compilierung auch genau dort plaziert wo sie sich befinden. Also auch mitten im eigentlichen Programmcode. Da diese Variablen ja direkt einen Wert beim Start des Programmes zugewiesen bekommen haben, weiß der Compiler vorher ja schon, was sich in dieser Variablen befindet und wie groß der benötigte Speicher sein muß.

Bei nicht initialisierten Variablen steht der Compiler beim Compilieren des Programmes auf dem Schlauch. Er kennt zwar den Datentyp der Variable, aber er weiß zu dem Zeitpunkt ja noch nicht, was sie damit alles anstellen wollen. Deshalb wird bei solchen Variablen der Speicherplatz für diese ans Ende des Programmes gelegt.

Das hat den Vorteil, das sie mit der Variablen einiges mehr machen können, als der Datentyp es eigentlich zulassen würde (einer der Vorteile der Sprache C, welchen die Systemprogrammierer sehr lieben), aber auch den Nachteil, das wenn sie zu weit gehen, dabei auch die Werte anderer Variablen einfach überschreiben können, obwohl sie garnicht auf diese per Variablennamen zugreifen.

Folgendes Beispiel soll das ganze etwas besser verdeutlichen :

```
char smsg1=”CPU speed:”;      //Der Speicher dafür wird genau hier abgelegt,also
                                mitten im Programm. Eckige Klammern sind nicht
                                erforderlich
dword cpuspeed;               //Der Speicher für diese Variable wird ans Ende
                                des Programmes gelegt, da der Compiler keinen
                                Startwert dafür hat.

void draw_window(void)
{
  sys_window_redraw(1);
  sys_draw_window(100<<16+300,100<<16+200,0x001111CC,0x8099BBFF,0x00FFFFFF);
  sys_write_text(8<<16+8,0xFFFFFF,”C— CPU speed example for MenuetOS”,34);;
  sys_draw_button(281<<16+12,5<<16+12,1,0x5599CC);
  sys_write_number(4<<16,cpuspeed,25<<16+40,0xFFFF80);
  sys_window_redraw(2);
}

//Die Funktion “sys_write_text()” übergibt die Adresse eines Strings. Der String
  wird sofort hinter der Funktion compiliert und abgelegt.
```

```
dword test1=0;                      //Speicher wird hier angelegt.
dword test2;                        //Speicher kommt ans Ende des Programmes.

void main(void)
{
  dword test3=0;                    //Nur temporär, wird auf dem Stapel erzeugt.
  cpuspeed=sys_service(5,0)/1000000;      //CPU Geschwindigkeit holen.
  draw_window();
  while(1)
  {
   switch(sys_wait_event())
   {
    case 1:
     draw_window();
     continue;
    case 2:
     /*sys_get_key();*/
     continue;
    case 3:
     if(sys_get_button_id()==1) sys_exit_process();
     continue;
   }
  }
}
```

Es wäre besser wenn man eine Möglichkeit hätte, das der Speicherplatz für nicht initialisierte Variablen auch dort abgelegt wird, wo sich die Variablen im Programm direkt befinden. Und die Möglichkeit hat man bei C— auch !!

Mit Hilfe des folgenden Präprozessorbefehles kann man dieses Compilerverhalten erzwingen :

```
# initallvar TRUE
```

Oder man verwendet die Kommandozeilenoption “/iv” beim Starten des Compiliervorganges.

4.3.2 Lokale Variablen

Alle Variablen die innerhalb einer Funktion deklariert wurden, gelten als Lokal und sind innerhalb dieser Funktion nur solange gültig, wie diese ausgeführt wird. Außerhalb der Funktion sind diese Variablen unbekannt.

Solche Variablen werden nur temporär (also kurzzeitig) auf dem Stapel (Stack) erzeugt und beim verlassen der Funktion wieder von diesem entfernt. Schauen Sie sich das Beispiel “test3“ einmal an.

C— unterstützt auch den “static“ Befehl. Mit diesem werden aus lokalen Variablen permanente, die dadurch auch entsprechenden Speicherplatz zugeweisen bekommen und nichtmehr auf dem Stapel erzeugt werden.

Die Schreibweise um eine Statische Variable zu erzeugen sieht folgendermaßen aus :

```
static dword zz;
static long xx=0;
```

Der “static“ Befehl kann zusätzlich auch auf globale Objekte (Variablen, Strukturen, Prozeduren) angewendet werden. Solche Objekte sind normalerweise nur in den Dateien sichtbar in denen sie deklariert wurden. Durch den “static“ Befehl lassen sich diese sonst unsichtbaren Bezeichner auch für andere Programme verwendbar machen.

4.3.3 Dynamische Variablen

Dynamische Variablen werden nur dann mit in den Programmcode aufgenommen, wenn diese irgendwo im Programm referenziert wurden, also irgendwo verwendet werden. Ob das auch bei lokalen Variablen funktioniert ist unbekannt.

Hier Beispiele für globale dynamische variablen:

```
 : dword zz;             // “:” macht sie Dynamisch.
 : struct RECT xx;
```

Wofür werden diese nun verwendet?

Barry schreibt dazu :
... Möglicherweise in header Dateien (include Dateien) oder Bibliotheks Dateien.

Ich hätte dazu noch ein anderes kleines Beispiel. Kommen wir nochmals auf unsere kleine Textverarbeitung für DOS und Windows zurück. Vielleicht kommen in der DOS Version ja andere Variablen zum Einsatz, da wir es ja auch mit unterschiedlichen Modi zu tun haben. Wir wissen ja schon, einmal 16 Bit und einmal 32 Bit. Definieren wir nun alle Variablen für die entsprechenden Modes als Dynamisch, dann werden die nicht benötigten Variablen, dadurch das sie nicht verwendet werden, auch nicht mit compiliert. Diese sind ja durch die Präprozessorabfrage in den einzelnen Programmzeilen ausgeschlossen und werden auch nicht referenziert, also verwendet.

Aber zurück zum Thema. Neben den dynamischen Variablen gibt es, wie sollte es auch anders sein, auch dynamische Prozeduren. Doch dazu weiter unten mehr.

Ein weiterer Punkt bei dynamischen Variablen ist, wenn sie in das Programm hineincompiliert werden, das sie am Ende des Programmes abgelegt werden. Hinter den dynamischen Prozeduren.

5. Ausdrücke

Konstante Ausdrücke hatten wir vorher ja schon besprochen. Nun wollen wir uns den Ausdrücken annehmen, die auch Konstanten, Variablen und Register beinhalten.

Es ist eine Interessante Frage was passiert, wenn wir mal normale Ausdrücke mit Registernamen mischen. Ob das funktioniert ?

5.1 Verwendung von Register in C– Ausdrücke

Dieser Beispielcode ist korrekt und funktioniert auch:

```
EAX=y+100<<7;
EBX=EAX<<2;
offset1=EAX+EBX+x;
```

Was uns nun bei diesem kleinen Programm besonders interessiert, ist das Ergebnis der ersten Zeile. Dieses wird berechnet und ohne das erzeugen einer temporären Variable direkt in das Register EAX abgelegt. In der dritten Zeile sind die Register mit einer normalen Variable gemischt und das Ergebnis wird in der Variable “offset1“ abgelegt.

5.1.1 EAX, EDX

Ich experimentiere mit einem C Ausdruck der keine expliziten Register verwendet:

```
long y,x=5;
void main(void) {
  long time=1;
  y=100+90*cos(time*1.5+2.0)/x;
```

Der mathematische Ausdruck wird dabei zu folgendes compiliert :

```
cpuspeed.c– 52: y=100+90*cos(time*1.5+2.0)/x;;
00000131  DB45FC                          fild    [ebp-4]
00000134  D80DB8010000                    fmul    [1B8h]
0000013A  D805BC010000                    fadd    [1BCh]
00000140  D9FF                            fcos
00000142  50                              push    eax
00000143  DB1C24                          fistp   [esp]
00000146  58                              pop     eax
00000147  69C0BE000000                    imul    eax,eax,0BEh
0000014D  99                              cwd
0000014E  F73D04010000                    idiv    dword ptr [104h]
00000154  A3D0010000                      mov     [1D0h],eax
```

Eines ist dabei sehr interessant - es wird nur das EAX Register dabei verwendet. (das ist änderbar)

Was aber passiert, wenn ich das EAX Register direkt selbst in diesem Ausdruck verwende. Schauen wir einfach mal nach:

```
cpuspeed.c— 52: y=100+90*EAX*cos(time*1.5+2.0)/x;;
00000131  69C0BE000000                imul    eax,eax,0BEh
00000137  50                          push    eax
00000138  DB45FC                      fild    [ebp-4]
0000013B  D80DBC010000                fmul    [1BCh]
00000141  D805C0010000                fadd    [1C0h]
00000147  D9FF                        fcos
00000149  50                          push    eax
0000014A  DB1C24                      fistp   [esp]
0000014D  58                          pop     eax
0000014E  5A                          pop     edx
0000014F  F7EA                        imul    edx
00000151  99                          cwd
00000152  F73D04010000                idiv    dword ptr [104h]
00000158  A3D4010000                  mov     [1D4h],eax
```

Nicht schlecht, der Compiler ist nicht dumm. Zumindest schlau genug um zu bemerken, das es eine Kollision gibt und das obwohl ich das EAX Register extra an einer Stelle gesetzt habe, die den Compiler eigentlich hätte durcheinander bringen müssen.

Aber nein, direkt in der ersten Zeile wird die Anweisung “190 *EAX” ausgeführt, daß Ergebnisse auf den Stapel (auch Stack genannt) gelegt und nach den zwischenzeitlichen Berechnungen wieder von diesem mit der “pop edx” Anweisung zurück geholt. Nun wollen wir den Compiler noch ein wenig mehr in Bedrängnis bringen, indem wir das “EDX” Register an einer Stelle mit einbeziehen, die bei der Berechnung zu Schwierigkeiten führen könnte.

```
cpuspeed.c— 52: y=100+90*EAX*cos(time*1.5+2.0)*EDX/x;;
00000131  69C0BE000000                imul    eax,eax,0BEh
00000137  50                          push    eax
00000138  DB45FC                      fild    [ebp-4]
0000013B  D80DBC010000                fmul    [1BCh]
00000141  D805C0010000                fadd    [1C0h]
00000147  D9FF                        fcos
00000149  50                          push    eax
0000014A  DB1C24                      fistp   [esp]
0000014D  58                          pop     eax
0000014E  5A                          pop     edx
0000014F  F7EA                        imul    edx
00000151  F7EA                        imul    edx
00000153  99                          cwd
00000154  F73D04010000                idiv    dword ptr [104h]
0000015A  A3D4010000                  mov     [1D4h],eax
```

Na also, ich habe es geschafft!

Die Zeile 151 ist falsch, da es den Werten des EDX Registers verwendet, der in der Zeile 14F schon berechnet wurde.

Zeile 14F führt eine Vorzeichenbehaftete Ganzzahl Multiplikation von EAX*EDX. Dadurch geht das 64 Bit Ergebnisse im Register Paar EDX:EAX verloren.

5.1.2 ECX

Was ist mit dem Register ECX? Es scheint so als wenn C— dieses Register zur Schleifensteuerung verwendet, dem ist aber nicht so::

```
cpuspeed.c— 52: loop(count) y=100+90*cos(time*1.5+2.0)/x;;
00000138  DB45F8                          fild    [ebp-8]
0000013B  D80DC4010000                    fmul    [1C4h]
00000141  D805C8010000                    fadd    [1C8h]
00000147  D9FF                            fcos
00000149  50                              push    eax
0000014A  DB1C24                          fistp   [esp]
0000014D  58                              pop     eax
0000014E  69C0BE000000                    imul    eax,eax,0BEh
00000154  99                              cwd
00000155  F73D04010000                    idiv    dword ptr [104h]
0000015B  A3DC010000                      mov     [1DCh],eax
00000160  FF4DFC                          dec     dword ptr [ebp-4]
00000163  75D3                            jne     138h
```

Jedoch habe ich herausgefunden, wie man das Register ECX problemlos verwenden kann:

```
  loop(ECX) y=100+90*cos(time*1.5+2.0)/x;
```

Durch diese Anweisung verwendet der Compiler den “loop” Assembler Befehl. Somit wird die Schleife mit einem Ergebnis im Register ECX=0 verlassen.

5.1.3 EBX, ESI, EDI

Woran wir noch nicht gedacht haben, sind die Register ESI,EBX und EDI. Diese werden wahrscheinlich benutzt, wenn mit Zeigern und Feldern Berechnungen angestellt werden. Zum Beispiel:

```
cpuspeed.c— 55: EBX=*z+EAX<<2/x;;
00000178  8B1D18020000                    mov     ebx,[218h]
0000017E  8B1B                            mov     ebx,[ebx]
00000180  01C3                            add     ebx,eax
00000182  B102                            mov     cl,2
00000184  D3E3                            shl     ebx,cl
00000186  93                              xchg    ebx,eax
00000187  31D2                            xor     edx,edx
00000189  F73504010000                    div     dword ptr [104h]
0000018F  93                              xchg    ebx,eax
```

Ein anderes Beispiel:

```
cpuspeed.c— 53: *z=100+90*cos(time*1.5+2.0)/x;;
00000138  DB45F8                      fild    [ebp-8]
0000013B  D80DC0010000                fmul    [1C0h]
00000141  D805C4010000                fadd    [1C4h]
00000147  D9FF                        fcos
00000149  50                          push    eax
0000014A  DB1C24                      fistp   [esp]
0000014D  58                          pop     eax
0000014E  69C0BE000000                imul    eax,eax,0BEh
00000154  99                          cwd
00000155  F73D04010000                idiv    dword ptr [104h]
0000015B  8B1DDC010000                mov     ebx,[1DCh]
00000161  8903                        mov     [ebx],eax
```

Ja, es verwendet das EBX Register. Können wir so dem Compiler einen Schraubenschlüssel in seine Rädchen werfen ?

Hinweis, die Zeile mit dem Code “fistp [esp]” sieht ein wenig unheimlich aus, ist aber ein sehr schlauer Zug. Die “fist/fistp” Assembler Anweisung kann einen Wert nicht direkt in ein Register schreiben, sondern nur in einem Speicherplatz. Somit wird mit den Anweisungen “push eax, fistp [esp], pop eax” die Berechnungen der FPU (Fließkommaprozessor) in das Register EBX gespeichert. Es ist einfach den Compiler dazu zu bringen das EBX Register nicht mehr verwenden zu können:

```
cpuspeed.c— 55:     EBX=*z+EBX+EAX<<2/x;     //stoppt   die   Werte
Zuweisung ins EBX Register!
```

Können wir erreichen, daß der Compiler die Register ESI und EDI verwendet? Natürlich können wir dies, wenn wir beispielsweise mehrdimensionale Felder oder mehrfach Zeiger in derselben Anweisung verwenden. Lassen Sie uns dies am Beispiel der zwei Zeiger “z” und “zz” versuchen:

```
cpuspeed.c— 57: EBX=*z+*zz;;
00000138  8B1DB8010000                mov     ebx,[1B8h]
0000013E  8B1B                        mov     ebx,[ebx]
00000140  8B35BC010000                mov     esi,[1BCh]
00000146  031E                        add     ebx,[esi]
```

Versuchen wir dieses nun noch einmal mit drei Zeigern :

```
cpuspeed.c— 58: EBX=*z+*zz+*zzz;;
00000138  8B1DC0010000                mov     ebx,[1C0h]
0000013E  8B1B                        mov     ebx,[ebx]
00000140  8B35C4010000                mov     esi,[1C4h]
00000146  031E                        add     ebx,[esi]
00000148  8B35C8010000                mov     esi,[1C8h]
0000014E  031E                        add     ebx,[esi]
```

Irgendwie schaffe ich es nicht den Compiler dazu zu bringen das EDI Register zu verwenden. Er versucht auf alle Fälle zu vermeiden das ESI Register zu benutzen. Der Compiler benutzt dieses Register in den obigen Beispiel nur, weil ich direkt das Register EBX in der Anweisung benutzt habe. Doch habe ich noch folgende Anweisung:

```
cpuspeed.c— 58: y=*z+*zz+*zzz;;
00000138  8B05C8010000                mov     eax,[1C8h]
0000013E  8B00                        mov     eax,[eax]
00000140  8B1DCC010000                mov     ebx,[1CCh]
00000146  0303                        add     eax,[ebx]
00000148  8B1DD0010000                mov     ebx,[1D0h]
0000014E  0303                        add     eax,[ebx]
00000150  A3C4010000                  mov     [1C4h],eax
```

Man sieht, daß nur die Register EAX und EBX verwendet werden. Natürlich gibt es auch Ausdrücke wo der Compiler die Register ESI und EDI verwendet, ohne daß ich das Register EBX gebrauche. Ein Beispiel:

```
struct test {short a;char b[8];long c;} rr[5];
dword i,k;

cpuspeed.c— 71: rr[i].b[k] >< rr[i+1].b[k+2];;
00000127  8B352C020000                mov     esi,[22Ch]
0000012D  8B3D28020000                mov     edi,[228h]
00000133  6BFF0E                      imul    edi,edi,0Eh
00000136  8A843EE4010000              mov     al,[esi+edi+1E4h]
0000013D  8B3D2C020000                mov     edi,[22Ch]
00000143  83C702                      add     edi,2
00000146  8B1528020000                mov     edx,[228h]
0000014C  42                          inc     edx
0000014D  6BD20E                      imul    edx,edx,0Eh
00000150  868417E4010000              xchg    [edi+edx+1E4h],al
00000157  88843EE4010000              mov     [esi+edi+1E4h],al
```

Der “><” Operator tauscht die Inhalte eines jeden Parameters. Hier haben wir die Register ESI, EDI, EAX und EDX verwendet.

Können wir nun empfehlen bei C— Ausdrücken die Register zu verwenden? NEIN, tun sie es nicht! Der Compiler versucht hauptsächlich bei Ausdrücken das Register EAX zu verwenden, ansonsten zusätzlich noch das EDX Register, falls das nicht klappt noch das EBX Register - und wenn alles schief geht als letztes das ESI Register. Nur der Compiler weiß Probleme zu umgeben, wenn ich das Register EAX verwende..

Sie können zwar ohne Probleme die Register EAX und ECX verwenden, doch können sie damit rechnen, das nach der Ausführung des Ausdrucks, ihre Werte nicht korrekt sind. Für die Mehrheit der Ausdrücke werden die Register ESI und besonders EDI nicht verwendet, was bedeutet, daß sie diese in ihren Ausdrücken verwenden können und wenn dies nicht ausreichen sollte bleibt ihnen als allerletztes noch das

Register ESI übrig. Ich kann ihnen nichts garantieren, doch lassen Sie es mich wissen, wenn sie Ausnamen dieser Regel finden.

Natürlich, sollten Funktionen oder Makros in diesen Ausdrücken verwendet oder aufgerufen werden, hängt es davon ab, was die Funktion oder das Makro mit den Registern macht.

Doch kommen wir nun zu unserem vorherigen Beispiel zurück:

```
EAX=y+100<<7;            //do NOT insert another statement between...
EBX=EAX<<2;              //do NOT always expect EAX to be preserved!
offset1=EAX+EBX+x;       //AVOID using EBX in an expression!
```

Bei diesem Beispiel gibt es keine Probleme, da die Werte den Registern zugewiesen worden sind und beim nächsten Ausdruck wieder ausgelesen werden. Sollte aber zwischen den Anweisungen der zweiten und letzten Zeile eine zusätzliche Zeile hinzugefügt werden, könnte dies schon wieder Probleme verursachen. Jedoch ist dieses Code Fragment, daß ich aus einem vorhandenen Programm herausgenommen habe, nicht gerade unproblematisch. Beachten Sie, daß dem Register EAX in der ersten Zeile ein Werte zugewiesen wird, in der letzten Zeile aber auf diesen Wert zugegriffen wird, was zu Problemen führen könnte. Das ist deshalb gefährlich da in der letzten Zeile das Register EBX auf der rechten Seite des Ausdrucks steht !!

Wenn sie diesen Abschnitt genau studiert haben werden sie verstehen, unter welchen Bedingungen Sie ohne Probleme die entsprechenden Register in ihren Ausdrücken verwenden können. Zum Beispiel wissen sie, das Sie das Register ECX nicht als Zähler für eine Schleife verwenden dürfen. Ansonsten können sie das Register ECX frei verwenden.

Sollten sie sich jedoch nicht sicher sein, daß sie in ihren Ausdrücken die entsprechenden Register verwenden können, besteht die Möglichkeit den Compiler aufzufordern ihnen dies nötigenfalls anzuzeigen. Der Compiler ist in der Lage entsprechende Warnungen über das verwenden der Register auszugeben. Damit der Compiler diese Warnungen auch ausgibt müssen sie diesen mit der Kommandozeilenoption “/ws” aufrufen oder innerhalb ihres Quellcodes den Präprozessorbefehl wie folgt setzten:

```
#warning TRUE
```

Um die Ausgabe der Warnungen vom Bildschirm in eine Datei umzuleiten, ist es notwendig eine weitere Kommandozeilenoption zu verwenden:

```
wf=file_name
```

5.2 Bedingungsausdrücke

Bedingungsausdrücke sind Ausdrücke, die zum erstellten von “ja” oder “nein” Abfragen verwendet werden, wie Beispielsweise für die “if” Anweisungen und für die “do {} while” Schleife. Es gibt zwei Arten von Bedingungsausdrücken, die einfachen und die komplexen (wie sollte es auch anders sein ;-).

5.2.1 Einfachen Bedingungsausdrücke

Einfache Bedingungsausdrücke sind einzelne Ausdrücke, dessen Ergebnisse nach der Berechnung ein “Ja” ergibt wenn der berechnete Wert ungleich Null ist oder andernfalls ein “nein” falls der berechnete Wert null ergibt.

5.2.2 Komplexe Bedingungsausdrücke

Komplexe Bedingungsausdrücke haben die folgende Form:

```
( linke Seite -  Vergleichs Operation - rechte Seite  )
```

Wobei:

“linke Seite”

Ist entweder AL/AX/EAX oder ein konstanter Ausdruck. Der Ausdrucks Typ wird vom ersten Ausdruck (Register oder Variable) bestimmt: dabei ist der Standardwert bei “word” 16 Bit, “dword” für 32 Bit Code. Soll ein anderer Ausdruckstyp verwendet werden so kann dies mit den Schlüsselwörtern “byte”, “char”, “int”, “word”, “long”, “dword” oder “float” explizit angegeben werden.

“Vergleichsoperation”

Ist eine der folgenden “==”, “!=”, “<>”, “<”, “>”, “<=”, oder “>=”.

“rechte Seite”

Ist ein Einzelregister, Variable oder Konstanter Ausdruck.

Einige Beispiele von korrekten komplexen Bedingungsausdrücken:

```
( x+y > z )
(int CX*DX <= 12*3 )
(byte first*second+hold == cnumber )
```

einige Beispiele von falschen komplexen Bedingungsausdrücken:

```
( x+y >= x-y )      //Die rechte Seite ist kein einzelner Ausdruck
                      oder Konstante Ausdruck.
( z = y )           //Es muß heißen “==” und nicht “=”, da diese
                      Schreibweise nur bei Zuweisungen verwendet
                      wird.
```

5.3 Datentypausweitung von Registern

In Standard-C gibt es das Konzept des “casting”. Dabei wird der Compiler aufgefordert eine Variable als anderen Datentyp zu definieren, oder den Datentyp in einen anderen zu ändern, in dem sie eigentlich standardmäßig definiert war und bisher so auch behandelt wurde.

Dabei geht es vorrangig nicht darum sich Gedanken zu machen, das der Datentyp sich ändert, sondern wie der Compiler damit umgeht. Die Daten (also der Inhalt einer Variable) änderte sich nicht, aber der Datentyp (indem die Variable bisher definiert war). Die Inhalte ändern sich nicht, aber wie sie zu interpretieren sind. Ich nenne das Datentypausweitung.

Wie sie wissen kann bei einem Bedingungsausdrucks auf der linken Seite einer der folgenden Datentypen stehen -”byte”, “char”, “int”, “word”, “long”, “dword” oder “float”.
Dieses wird ihnen im folgenden auch die grundsätzlichen Probleme bei der Benutzung von Registern aufzeigen.

In C– können Register genauso verwendet werden, als wären sie normale Variablen. Sowohl bei einfachen wie auch bei komplexen Ausdrücken. Und genau da hätten wir dann auch schon das Problem. Woher soll der Compiler wissen, für welchen Datentyp sie nun das entsprechende Register verwenden oder einsetzten. Standardmäßig ist der Compiler so eingestellt, das er die Register als Vorzeichenbehaftete Ganzzahlen, je nach Registergröße, behandelt. Zum Beispiel ist das Register AX eine Vorzeichenlose Ganzzahl. Nun gibt es die Möglichkeit, diese Voreinstellungen des Compilers zu umgehen, wie das Beispiel zeigt :

```
float f = 1.0;

void PROC ()
{
  IF (f < signed ECX)        // im Register ECX ist eine
Vorzeichenbehaftete
                         // Zahl.
  IF (unsigned EBX > f)      // im Register EBX ist eine
Vorzeichenlose Zahl.
  IF (f == float EAX)        // Im Register EAX ist eine
Fließkommazahl. //Achtung: der Inhalt von EAX wir zerstört!
}
```

Die erlaubten Modifizierer sind ‘signed’, ‘unsigned’ und ‘float’.

5.4 Datentypkonvertierung

Beachten Sie das in Standard C das “casting” folgende Schreibweise hat “if(x==(float)zz)”, der Modifizierer also in Klammern sitzt. Diese Schreibweise wird von C— nicht unterstützt.

Der C— Compiler konvertiert Werte in den benötigten Ziel Datentyp. Folgende Variablen sollen dies verdeutlichen. Die variable “a” ist vom Typ “float”, “j” ist vom Typ “long” und “i” ist vom Typ “short”. Vergleichen sie folgendes:

```
cpuspeed.c— 59: i=a+j;;
0000011F    50              push    eax         //konvetiert “a” zu integer.
00000120    D905C0010000    fld     [1C0h]       //    /
00000126    DB1C24          fistp   [esp]         //    /
00000129    58              pop     eax         //    /
0000012A    660305BC010000  add     ax,[1BCh]  //Ergebniss als short.
00000131    66A3B8010000    mov     [1B8h],ax
```

Was sie hier sehen ist, daß die Werte auf der rechten Seite in den Datentyp “short” umgewandelt werden bevor die Berechnung beginnt. Danach wird das Ergebnis der Variable “i” zugewiesen.

Ein anderes Beispiel ist dieser Ausdruck, wobei “z” vom Vorzeichenlosen Datentyp “short” ist:

```
z = a+j;
```

Warnung:

Diese Zeile erzeugt denselben Code wie oben, jedoch gibt es keine Typenumwandlung. Auf diese Weise, wenn das Ergebnis des Ausdrucks auf der rechten Seite ausgewertet worden und Negativ ist, wird das Ergebnis des Ausdrucks unveändert in der Variable “z” übernommen. Doch ist das Ergebnis nicht korrekt.

Dieser Fehler entsteht deshalb, weil es für Vorzeichenbehaftete Zahlen zwei Schreibweisen gibt. Dabei wird die positive Vorzeichenbehaftete Zahl genauso geschrieben wie die Vorzeichenlose Zahl. Nur wenn die Zahl negativ ist, wird diese anders dargestellt.

Die Konvertierung des Variableninhaltes in den Zieltyp sollte man vor der Berechnung vermeiden, da es sich bei diesem um ein Ganzzahl handelt, die Werte auf der Linken Seite aber Fließkommazahlen sind. Würde man nun die Werte vorher Konvertieren, so würden zum einem die Nachkommastellen verloren gehen und zum anderen kann es dadurch zu Rundungsfehlern bei der Berechnung kommen.

C— erlaubt Ihnen den Datentyp, indem der zu berechnende Ausdruck nachher umgewandelt werden soll, anzugeben. Folgendes Beispiel zeigt dieses. Dabei ist “a” vom Datentyp “float”, “j” ist “long” und “i” ist “short”.

```
cpuspeed.c— 59: i= float a+j;
0000011F  D905BC010000                fld     [1BCh]
00000125  DA05B8010000                fiadd   [1B8h]
0000012B  DF1DB4010000                fistpw  [1B4h]
00000131  9B                          fwait
```

In diesem Beispiel werden die Parameter vor der Berechnung in das Fließkomma Format umgewandelt. Beachten Sie das der generierte Assembler Code die Variable “a” und “j” direkt in die FPU lädt und alle Berechnungen dort direkt im Fließkommaformat gemacht werden.

5.5 Zuweisung zu einem Register

Direkt benannte Register werden in C— Ausdrücken vom Compiler als Vorzeichenlose Ganzzahlen behandelt. Im Kapitel “Bedingungsausdrücke“ wurde beschrieben wie man dieses umgehen kann (siehe dazu auch “Datentypkonvertierung“). Die oben beschriebene Datentypkonvertierung gilt natürlich auch für das Ziel-Register.

```
EAX = a*j;  //a ist float, j ist long.
```

Die Variablen “a” und “j” werden vor der Berechnung in den “long” Datentyp umgewandelt. Doch da sie als Vorzeichenlose Zahlen behandelt werden, wird vom Compiler mehrfach die Vorzeichenlose “mul” Instruktion erzeugt.
Andererseits:

```
EAX = long a*j;  //a ist float, j ist long.
```

Dies bewirkt das der Compiler die Ausdrücke auf der rechten Seite als Vorzeichenbehaftet ansieht und somit “imul” Instruktionen erzeugt. Jedoch als Warnung: wird keine tatsächliche Datentypumwandlung vorgenommen, so ist das Ergebnis der rechten Seite, das gleiche wie auf der Linken..

5.6 Mehrfach Zuweisungen

Sollte es mal nötig sein, mehreren Variablen einen gleichen Wert zuzuweisen, so können sie dieses wie folgt schreiben :

```
  var1 = 0;
  var2 = 0;
  var3 = 0;
```

Was natürlich auch einfacher geht, wie dieses Beispiel zeigt :

```
  var1 = var2 = var3 = 0;
```

Die Schreibweise hat nicht nur den Vorteil, das sie weniger tippen müßen, sondern erzeugt auch einen kompakteren Programmcode, da der Compiler direkt sehen kann, das jeder der Variablen ein und denselben Wert erhält.

6. Definiton von Funktionen und Makros

In jeder heutigen Programmiersprache kennt man Funktionen und Prozeduren die das Programmieren erleichtern und dafür sorgen, das gewisse Programmteile mehrfach verwendet werden können. In der Sprache C gibt es genaugenommen eigentlich nur Funktionen und gar keine Prozeduren. Ich erkläre mal zum besseren Verständnis den Unterschied. Der Unterschied ist relativ simpel. Beide werden mit entsprechenden Parametern aufgerufen, allerdings liefert nur die Funktion einen direkten Wert, also ein Ergebnis auch wieder zurück. In C ist jede Prozedur eigentlich eine Funktion, nur das diese eben “nichts“ oder etwas unbestimmtes zurückliefert. Damit Funktionen und Prozeduren überhaupt etwas machen können, müssen sie also mit Werten gefüttert werden.

C– kennt nun zwei verschieden Arten, wie diese Wert oder Parameter übergeben werden. Einmal hätten wir da die stapelbasierte Übergabe und die andere Möglichkeit ist die Parameter mit Hilfe der Register zu übergeben..

Bei der stapelbasierten Übergabe werden die Parameter einem nach dem anderen auf den Stapel gelegt und die aufzurufende Prozedur kann diese vom Stapel lesen und verarbeiten. Dies hat den Vorteil, das die zu übergebenen Parameter fast Datentyp unabhängig ist. Hat leider aber auch den Nachteil, das alles erst im Speicher abgelegt und dann von dort wieder zurück geholt werden muß. Und das kostet natürlich Geschwindigkeit. Folgende Datentypen können so direkt an eine Prozedur über den Stapel übergeben werden : ‘byte’, ‘char’, ‘word’, ‘int’, ‘dword’ oder ‘long’.

Dabei werden die Parameter der Pascal Aufrufskonvertion ähnlich auf den Stapel gelegt. Das bedeutet, das der Erste Parameter als erstes auf den Stapel gelegt wird, dann der zweite usw. Diese Aufrufskonvention hat leider den Nachteil, das man nicht mit variabler Parameteranzahl bei Prozeduren arbeiten kann. Deshalb muß man sich immer sicher sein, das die Anzahl der auf den Stapel gelegten Parametern auch der Anzahl der Prozedure entsprechen. Die folgende Beispiel Prozeduren (Stapelbasiert) gibt die Summe aller übergebenen Parameter als “word” Datentypen zurück, auch wenn die Datentypen der Parametern unterschiedlich sind :

```
word add_them_all (short a,b; long c,d)
word x;
dword y;
{
  return( a+b+c+d );
}
```

Folgendes liegt auf dem Stapel während des Aufrufs der Funktionen “add_them_all()”:

```
    EBP - 6    Lokale variable “y”.
    EBP - 2    Lokale variable “x”.
    EBP + 0    gespeichertes EBP Register.
    EBP + 4    Rücksprungadresse.
    EBP + 8    Der letzte rechte Parameter, “d”.
    EBP + 12   Parameter “c”.
    EBP + 16   Parameter “b”.
    EBP + 20   Parameter “a”.
```

Wird 32 Bit Code verwendet, übergibt der Compiler alle Parameter einer Funktion, ohne Rücksicht auf den Datentyp, in 32 Bit . Der Grund warum das EBP Register hier verwendet wird ist, das der Compiler diesen Assemblercode direkt an den Anfang der Prozedur legt:

```
push ebp        //speichere EBP.
mov ebp,esp     //benutze EBP um auf den Stapel zugreifen zu können
sub esp,value   //Wert = Anzahl der Bytes die für die Lokalen
                  Variablen benötigt wird.
```

Am Ende der Prozedure plaziert der Compiler nun foglendes :

```
mov ebp,esp         // EBP wiederherstellen.
pop esp             // ESP wiederherstellen.
ret 12              //entferne -EBP, lokal “x”, lokal “y”.
```

Für die Pascal Aufrufkonvention werden die lokalen Variablen mit der “ret” Instruktion entfernt, in diesen Beispiel also 12 byte vom Stapel vor dem Rücksprung entfernt.

Der Datentyp der übergebenen Parameter läßt sich effizienter, schneller und einfacher deklarieren,als wir es bisher gemacht haben. Als Beispiel:

```
void afunc(word a, b, c; dword d, e; byte f);        //Kurzschreibweise.
void afunc(word a, word b, word c, dword d, dword e, byte f);
//Komplett ausgeschrieben.
```

Die einzelne Datentyp Angabe “word“ reicht aus um alle folgenden Parameter durch Komma getrennt, bis zum Semikolon, zu deklarieren. Danach folgen zwei Parameter die als “dword“ und “f“ der als “byte“ deklariert wurden.

Die Parameter für eine Register Prozedur werden mit Hilfe der Register übergeben. Register Prozeduren haben maximal sechs Parameter. Die Parameter als Register sind vom Datentyp ‘int’ oder ‘word’, in der Reihenfolge AX, BX, CX, DX, DI, und SI. Die ersten vier Parameter können auch vom Datentyp ‘char’ oder ‘byte’ sein und werden in diesem Fall als AL, BL, CL und DL Register verwendet. Einige der sechs Parameter können auch vom Datentyp ‘long’ oder ‘dword’ sein, wobei dann die Register EAX, EBX, ECX, EDX, EDI, oder ESI verwendet werden.

Eine Macro ist eine ‘Inline’- Register Prozedur. Das Schlüsselwort “inline” wird weiter unten beschrieben. Die Wertrückgabe von Funktion erfolgt mit Hilfe der Register. Folgende Tabelle zeigt welches Register welchen Datentyp darstellt:

Rückgabetyp	Rückgabe im Register
byte	AL
word	AX
dword	EAX
char	AL
short	AX
long	EAX

Der einfachste Weg einen Wert von einer Funktion zurückzugeben ist der Befehle “return()”. Aber auch entsprechende Register können Rückgabe Werte zugewiesen werden. Folgende Beispiele dienen zur Erläuterung:

```
dword proc_one ()
{
return( 42 );
}

dword proc_two ()
{
EAX = 42;
}
```

Wenn man es ganz genau nimmt, wie schon oben beschrieben, gibt eine Prozedur eigentlich keinen Wert zurück, denn nur eine Funktion tut dieses (in C— wird der Rückgabe Wert im EAX Register zurückgegeben). Jedoch kümmere ich mich nicht um diese Beschränkung da die Wörter Funktion und Prozedur austauschbar sind.

Es ist auch möglich ein Prozessor Flag zu setzen. Das folgende Beispiel setzt das Carry Flag:

```
long CARRYFLAG fopen();             //Prozedur Definition.

    if(fopen()) { code here }       //Fragt das zurückgegebene Carry flag auf
                                      TRUE/FALSE ab.
    if(handle=fopen()) { code here //immernoch Carry flag gesetzt ?.
```

Dieser Bedingungsausdruck wird als korrekt angesehen wenn die Funktion mit dem gesetzten Carry Flag zurückkehrt.

Wir sind es ja gewohnt das Rückgabewerte im Register EAX vorliegen, welche wir vergleichen können. Doch woher wissen wir ob das Register EAX oder das Carry Flag gemeint ist?

Jede Verwendung eines Bedingungsausdruckes bewirkte bisher, daß der Compiler bei einem Vergleich grundsätzlich das EAX Register verwendete. Als Beispiel:

```
if(fopen() == 5) { code here }  //Kehrt zurück zum Rückgabewert.
```

Man auch auch ‘OVERFLOW’ oder ‘ZEROFLAG’ Flags verwenden.

6.1 Aufrufkonventionen

Michael beschreibt einen Mechanismus zum Spezifizieren der Aufrufkonventionen. Das grundsätzliche Format sieht wie folgt aus:

```
rettype modif procname ();
```
where “modif” can be “pascal”, “cdecl”, “stdcall”, or “fastcall”.

Wobei “modif” folgendes sein kann - “pascal”, “cdecl”, “stdcall”, oder “fastcall”.

6.1.1 Erläuterung der Aufrufkonventionen

cdecl - dieser Typ des Prozeduraufrufes ist der Standard bei der Sprache C. Er zeigt, wie die Parameter über den Stapel an die Prozedur übergeben werden und zwar von rechts nach links. Das entfernen der Parameter vom Stapel erfolgt nach der Rückkehr aus der Prozedur. Dieser Art des Prozeduraufrufes ist für Prozeduren mit einer unbekannten Anzahl an Parametern sehr gut geeignet, aber nicht sehr effizient.

pascal - bei dieser Art des Aufrufes erfolgt die Übergabe der Parameter in der Reihenfolge von links nach rechts, genauso wie sie auch im Programm stehen. Allerdings muß die Prozedur dabei selbst dafür sorgen das die Parameter wieder vom Stapel entfernt werden, in dem sie die “ret n” Instruktion verwendet.

stdcall - Diese Art des Aufrufes ist eine Mischung aus den ersten Beiden. Die Parameter werden in der Reihenfolge von rechts nach links übergeben (ungekehrt Reihenfolge wie sie in der Deklaration geschrieben stehen). Die Parameter werden von der Prozedur selbst wieder vom Stapel entfernt.

fastcall /register bei diesem Typ des Aufrufs werden Parameter mit Hilfe der Register übergeben. Dabei brauchen keine Parameter vom Stapel entfernt werden. Maximal können sechs Parameter mit den Registern EAX, EBX, ECX, EDX, EDI und ESI verwendet werden. 16 Bit Parameter benutzen die Register AX, BX, CX und DX. 8 Bit Parameter äquivalent dazu AL, BL, CL und DL.

6.1.2 Standardmäßig eingestellte Aufrufskonventionen

“register” — Obwohl standardmäßig die Aufrufkonvention “pascal“ verwendet wird, wird bei Funktionen die komplett großgeschrieben wurde, die Aufrufkonvention “fast call” (oder auch “register” genannt) verwendet.

“pascal” — Jegliche Funktionen mit einem oder mehrerer Kleinbuchstaben.

“stdcall” — Wenn in der Kommando Zeile folgende Parameter mit angegeben werden “/w32”, “/w32c” oder “/DLL” (ein MS Windows Programm oder eine DLL) .

6.2 Variable Anzahl an Parametern

Die Standard C Aufrufkonvention erlaubt dieses. Folgendes Beispiel zeigt wie man eine Funktion mit variabler Anzahl an Parametern deklariert:

```
void cdecl printf (word,...);
```

Dieses Beispiel hat wenigstens einen Parameter vom Datentyp “word”, aber auch noch weitere unbekannte.

6.3 Inline Stapel Funktionen

Dieses definiert eine Inline Prozedur:

```
inline void sys_write_text(dword EBX, ECX, EDX, ESI){
 EAX = 4;
 $int 0x40;
}
```

Mit Hilfe des Schlüsselwort “Inline” erreicht man dieses.
Der erzeugte Programm Code dieser Funktionen sieht wie folgt aus (cpuspeed ist ein Programm, msys ist die System Bibliothek):

```
cpuspeed.c— 26: sys_write_text(8<<16+8,0xFFFFFF,”C— CPU speed exam-
ple for MenuetOS”,34););
0000005F  B808000800                    mov     eax,80008h
00000064  50                            push    eax
00000065  68FFFFFF00                    push    0FFFFFFh
0000006A  B8E0000000                    mov     eax,0E0h
0000006F  50                            push    eax
00000070  6A22                          push    22h
msys.h— 99:         EAX = 4;
00000072  B804000000                    mov     eax,4
msys.h— 100:        $int 0x40;
00000077  CD40                          int     40h
00000079  83C410                        add     esp,10h
```

Wenn das ein bißchen Durcheinander aus sieht, dann ist das ebenso ;-). Obwohl die Register in der Funktionsdefinition angegeben wurden,

werden sie nicht verwendet. Die Registernamen werden hier nur als Dummy´s verwendet (was auch logisch ist und dem Standard C entspricht).

Deshalb dürfte es weniger verwirrend sein wenn man den Dummy´s aussagekräftigere Namen gibt, wie beispielsweise:

```
inline void sys_write_text(dword xystart, color, #message, length){
 EAX = 4;
 $int 0x40;
}
```

Beachten Sie das es erlaubt ist, die “return” Anweisung in einer Inline Funktion zu verwenden. Der Compiler ist intelligent genug dieses zu erkennen und keine “ret” Anweisung zu erzeugen und nötigenfalls die Parameter selbst vom Stapel zu entfernt.

6.4 Inline Register Funktionen

Folgendes Zeile definiert eine Inline Prozedur, wobei die Parameter mit Hilfe der Register übergiben werden:

```
inline fastcall void sys_write_text(dword EBX, ECX, EDX, ESI){
 EAX = 4;
 $int 0x40;
}
```

Das Schlüsselwort “fastcall” bewirkt das die Parameter in die Register EBX, ECX, EDX, und ESI übergeben werden (zumindest in diesem Beispiel). Beachten Sie das sie das Schlüsselwort “fastcall” zwischen dem Rückgabetyp und dem Funktionsnamen setzen. Der erzeugte Code für diesen Funktionsaufruf:

```
cpuspeed.c— 26: sys_write_text(8<<16+8,0xFFFFFF,”C— CPU speed example for MenuetOS”,34););
0000005F  BB08000800                    mov     ebx,80008h
00000064  B9FFFFFF00                    mov     ecx,0FFFFFFh
00000069  BAE0000000                    mov     edx,0E0h
0000006E  BE22000000                    mov     esi,22h
msys.h— 99:         EAX = 4;
00000073  B804000000                    mov     eax,4
msys.h— 100:        $int 0x40;
00000078  CD40                          int     40h
```

Persönlich mag ich den Namen “fastcall” nicht, geeigneter wäre dafür das Schlüsselwort “register”. Doch das kann man mit Hilfe der folgenden simplen #define Anweisung erreichen:

```
#define register fastcall
```

Inline Prozeduren sind Dynamisch. Das bedeutet, das sie nur eingefügt werden, wenn sie im Programm auch verwendet werden.

6.5 “nicht Inline“ Register Funktionen

bei dem folgenden Beispiel:

```
fastcall void sys_write_text(dword EBX, ECX, EDX, ESI){
 EAX = 4;
 $int 0x40;
}
```

müssen die Parameter in Registern übergeben werden, aber die Funktion selbst ist nicht als Inline Funktion compiliert. Der compilierte Code sieht wie folgt aus:

```
cpuspeed.c— 26: sys_write_text(8<<16+8,0xFFFFFF,”C— CPU speed exam-
ple for MenuetOS”,34););
00000067  BB08000800                    mov     ebx,80008h
0000006C  B9FFFFFF00                    mov     ecx,0FFFFFFh
00000071  BAE4000000                    mov     edx,0E4h
00000076  BE22000000                    mov     esi,22h
0000007B  E8A4FFFFFF                    call    24h
```

wobei die Funktion ansich wie folgt compiliert wird :

```
msys.h— 99:         EAX = 4;
00000024  B804000000                    mov     eax,4
msys.h— 100:        $int 0x40;
00000029  CD40                          int     40h
0000002B  C3                            ret
```

Dieses Beispiel ist keine dynamische Funktion, was bedeutet das der compilierte Programm Code sich immer im Programm befindet, auch wenn sie nicht verwendet wird

6.6 “Nicht-Inline“ Stapel Funktionen

Beispiel:

```
void sys_write_text(dword xystart, color, #message, length){
 EAX = 4;
 $int 0x40;
}
```

Dieses ist eine konventionelle nicht-Inline Funktion, in der die Parameter über den Stapel übergeben werden. Beachte das C— die PASCAL Aufrufs Konvention als Standard verwendet, in dem der linke Parameter als erstes auf dem Stapel gelegt wird und die Parameter vor der Rückkehr aus der Funktion wieder von diesem entfernt werden müßen.

6.7 Dynamische “nicht-Inline” Funktionen

Nicht-Inline Funktionen werden normal compiliert und in das Programm mit aufgenommen egal ob sie verwendet werden oder nicht. Einen Doppelpunkt “:” vor der Funktion definiert diese als dynamisch:

```
 : sys_write_text(dword xystart, color, #message, length){
 EAX = 4;
 $int 0x40;
}
```

Das bedeutet das diese Funktion nur Compilierte in das Programm mit aufgenommen wird wenn das Programm diese auch verwendet. Beachten Sie das dieses Zeichen als allererstes stehen muß. Bei einer Nicht-Inline Register Funktion ist dieses Zeichen auch erlaubt.

Interrupt Prozeduren

Interrupt Prozeduren, die zur Verwaltung und Bearbeitung von Interrupts dienen, werden wie folgt definiert:

```
interrupt procedure_name ()
{
// put code here
}
```

Interrupt Prozeduren erhalten nicht automatisch die Register und es werden auch keine Register verändert, solange nicht ein Interrupt ausgelöst wurde und die Prozedur die Kontrolle erhält. Das bedeutet, das die Prozedur dafür sorgen muß, das die Register per “push“ und “pop“ auf dem Stapel gesichert und wieder restauriert werden. Als Beispiel:

```
interrupt safe_handle ()
{
$PUSHAD                  //alle 32-bit Register sichern.
/* hier können sie nun alles machen */
$POPAD
}
```

7. Bedingte Ausdrücke
7.1 ‘if’ und ‘else’

Auswahl Anweisungen, besser bekannt als “if” Ausdruck sind in C fast genauso wie in anderen Programmiersprachen. C— hat zwei Auswahl Anweisungen. “if” und “IF”. “if” stellt einen weiten Sprung (near jump) dar, und “IF” einen kurzen Sprung (short jump) . “IF” wird schneller ausgeführt und spart drei Bytes im Programm Code ein, kann aber nur innerhalb von 127 Bytes im Programm springen..

nach einer Auswahl Anweisung in C kann einer einzelne Anweisung oder ein Anweisungsblock (eingeschlossen neben “{“ und “}”) folgen. C— Auswahl Anweisungen sind auf C— bedingte Ausdrücke beschränkt (s. Beschreibung in Kapitel 1.4). Sollten mehr als 127 Bytes nach einer “IF” Anweisung folgen, gibt der Compiler folgende Fehlermeldung aus:s:

```
IF jump distance too far, use if.
```

diese Meldung kann man leicht beseitigen in dem man die “IF” Anweisung durch “if” ersetzt. Die “else” und “ELSE” Anweisungen werden wie gewohnt verwendet. Beachten Sie das “ELSE” denselben 127 byte Beschränkungen unterlegt wie “IF”. “else” erzeugt 1 byte mehr als “ELSE”. “IF” und “else”, und “if” und “ELSE” Anweisungen lassen sich wie folgt auch mischen:

```
if( x == 2 )
    WRITESTR(“Two”);
ELSE{WRITESTR(“not two.”);
    printmorestuff();
    }
```

Sollten nach einer “ELSE” Anweisung mehr als 127 Bytes folgen gibt der Compiler folgende Fehlermeldung aus:

```
ELSE jump distance too far, use else.
```

Einfach die “ELSE” Anweisung durch “else” ersetzen und der Fehler ist beseitigt.

7.2 ‘do {} while’ Schleife

Die ‘do {} while’ Schleife wiederholt ein Programmblock solange wie ein Bedingungsausdruck als erfüllt angesehen wird. Dieser Programmblock wird wenigstens einmal ausgeführt, da die Abfrage des Bedingungsausdrucks erst am Ende ausgeführt wird.

Bei dem folgenden Beispiel wird der Programmblock in der Schleife fünfmal durchlaufen:

```
count = 0;
do {
    count++;
    WRITEWORD(count);
    WRITELN();
    } while (count < 5);
```

Die Bedingungsanweisung in der ‘do {} while’ muß denselben Regeln folgen wie bei den “IF” und “if” Anweisungen.

7.3 “loop” Schleife

Die “loop” Schleife wiederholt einen Programmblock solange, wie eine Variable oder ein Register nicht den Wert 0 hat. Am Ende des Programmblocks wird die entsprechende Variable oder das Register um 1 erniedrigt und dann getestet ob sie = 0 ist. Ist die Variable nicht = 0, wird der Programmblock von vorne ausgeführt und das ganze wiederholt sich. Das folgende Beispiel der “loop” Schleife verwendet die Variable “count” als Zähler:

```
count = 5;
loop( count )
    {WRITEWORD(count);
    WRITELN();
    }
```

Das verwenden des Registers CX bringt die beste Code Effizienz für die “loop” Schleife. Dabei wird der Maschinensprachbefehl “LOOP” verwendet. Sollte der Schleifenzähler schon 0 sein bevor die Schleife anfängt, wird diese Schleife mit der maximale Anzahl an Durchgängen gestartet. Dieses sind 256 Durchläufe bei 8 Bit (byte oder char), 65536 bei 16 bit (word oder int) und 4294967296 bei 32 bit (dword oder long). Bei dem folgenden Beispiel wird die Schleife 256 mal durchlaufen:

```
BH = 0;
loop( BH )
    {
    }
```

Wird kein Zähler bei diesem Schleifentyp angegeben, so wird der Programmblock endlos durchlaufen. Daß folgende Beispiel schreibt endlos “*” auf dem Bildschirm:

```
loop()
    WRITE(‘*’);
```

Der Programmierer kann, wenn er oder sie es möchte, innerhalb des Schleifenblockes den Zähler verwenden oder auch verändern.

Bei dem folgenden Beispiel wird der Zähler auf drei gesetzt, so daß die Schleife nur dreimal wiederholt wird:

```
CX = 1000;
loop( CX )
    {
    IF( CX > 3 )
     CX = 3;
    }
```

7.4 “for” Schleife

Auch die “for” Schleife vom Standard C wird unterstützt. Natürlich ist auch die Schreibweise “FOR” ist erlaubt, für Sprünge innerhalb von +/- 127 Bytes im Speicher. Folgendes Beispiel demonstriert dieses:

```
FOR(ECX=0;ECX<200;ECX++){
  EAX=ECX<<7;
  EBX=EAX<<2;
  EDI=EAX+EBX;
  FOR(EDX=0;EDX<320;EDX++,EDI++){
    ESI=EDI+offset1;
    c1=flowers[ESI];
    ESI=EDI+offset2;
    c2=flowers[ESI];
    ESI=EDI+offset3;
    c3=flowers[ESI];
    EAX=ECX<<6;
    EBX=EAX<<2;
    ESI=EAX+EBX+EDX;
    screen[ESI]=c1+c2+c3;
  }
}
```

Auch beschränkte Sprünge wie “break”,”continue”,”BREAK” und “CONTINUE” sind erlaubt. Letztere wieder für Sprünge innerhalb von +/- 127 Bytes.

7.5 “switch” Vergleich

Der Standard C “switch” Vergleich arbeitet genauso, wie folgendes Beispiel zeigt:

```
switch(sys_wait_event())
{
 case 1:
  draw_window();
  continue;                      //Mache weiter und setze den switch zurück.
 case 2:
  /*sys_get_key();*/
  break;                         //switch Bereich verlassen.
 case 3:
  if(sys_get_button_id()==1) sys_exit_process();
  continue;
}
```

Auch beschränkte Sprünge wie “break”, “continue”, “BREAK” und “CONTINUE” sind erlaubt. Letzteren wieder für Sprünge innerhalb von +/- 127 Bytes.

8. Felder (Arrays)

8.1 Felder Indizierung

Die Version von C— aus dem Jahre 1996, erstellt von Peter Cellik, erforderte das Felder mit einem byte Offset vom Start des Feldes referenziert wurden, ohne Rücksicht auf den Datentyp. Als Beispiel:

```
dword array[10];
dword i;

  EBX=4;
  array[EBX]=0;      //*byte* offset.
  array[4]=0;        //*byte* offset.
```

Jedoch hat Michael Sheker die Sache verbessert, so daß Felder auch mit einem entsprechenden Index des jeweiligen Datentyps referenziert werden können. Als Beispiel:

```
  i=1;
  array[i]=0;        //index bestimmt durch den Datentyp.
```

Das letzte Beispiel greift auf das zweite Element des Feldes zu. Hier kann man die Compilierte Version sehen:

```
cpuspeed.c— 66:   EBX=4;
0000011F    BB04000000                  mov     ebx,4
cpuspeed.c— 67:   array[EBX]=0;
00000124    C783C401000000000000        mov     dword ptr [ebx+1C4h],0
cpuspeed.c— 68:   array[4]=0;
0000012E    C705C801000000000000        mov     dword ptr [1C8h],0
cpuspeed.c— 69:   i=1;
00000138    C705EC01000001000000        mov     dword ptr [1ECh],1
cpuspeed.c— 70:   array[i]=0;
00000142    8B35EC010000                mov     esi,[1ECh]
00000148    C704B5C401000000000000      mov     dword ptr [1C4h+esi*4],0
```

Das Feld startet bei der Adresse 1C4h und man sieht das die ersten zwei Einträge des Feldes als byte Offset berechnet wurden. Jedoch, verwendet man eine Variable als Index, indiziert diese Variablen das Feld korrekt, wie im normalen C.

Um auf ein Feld mit Hilfe eines byte Offsets zugreifen zu können, kann folgendes Format verwendet werden, wobei “index” ein Numerischer Wert ist:

```
variable[index]
variable[index+EBX+ESI]
variable[index+EBX+EDI]
variable[index+EBP+ESI]
variable[index+EBP+EDI]
variable[index+ESI]
variable[index+EDI]
variable[index+EBP]
variable[index+EBX]
```

beachten Sie das die Verschachtelung von Feldern erlaubt ist. Dabei wird ein korrekter Index durch den jeweiligen Datentyp erzeugt, wie folgendes Beispiel zeigt :

```
cpuspeed.c— 69:     buf[array[i]]=0;
0000013A  8B35C0010000                  mov     esi,[1C0h]
00000140  8B34B514020000                mov     esi,[214h+esi*4]
00000147  C704B5C401000000000000        mov             dword   ptr
[1C4h+esi*4],0
```

Im obrigen Beispiel, sind die beiden Felder “buf[]” und “array[]” als Datentyp “dword” definiert. Man kann das “*4” in beiden Anweisungen sehen.

Die korrekte Offset byte Adressierung von Felder und Indizierung mit Hilfe des Datentyps in C ist nicht einfach. Michael beschloß deshalb folgendes einzuführen:

```
  array[*i] = 0;   //Hier beinhaltet Variable i das absolute Byte,
als die Anzahl der Elemente (index).
```

...hmmm.
...hmmm.

Felder können Ansammlungen von Strukturen sein, und wiederum Strukturen können Felder enthalten. Dieses wird genauer im Abschnitt “Strukturen” beschrieben.

9. Strukturen
9.1 Struktur Schreibweise

Dies ist das Format wie eine Struktur definiert wird:

```
  struct <tag> {<fields>};
```

wobei

<tag> ist der Name einer Struktur Definition (nicht deren Zuweisung),
<fields> Die Felddefinition der Struktur

Es gibt zwei erlaubte Formate um eine Struktur und deren Felder zu definieren:

```
  struct [<tag>] {<fields>} <name>[,<name>...];
  [struct] <tag> <name> [, <name>...];
```

wobei

  [ ] eckige Klammern optional Angaben sind,

  <name> Ist der Name der Zuweisung der Struktur

Das erste Format definiert eine Struktur und dessen Felder. Hier ein Beispiel für das erste Format:

```
struct rect {long x; long y;} myrect;
struct {long x; long y;} myrect;
struct {long x; long y;} rect1, rect2;  //Zwei Strukturen werden initialisiert.
```

Beachten Sie das der Ausdruck <tag> den Namen der Struktur definiert, dieser jedoch optional ist wenn alle Zuweisungen in diesem Format gemacht werden.

Das zweite Format definiert die Struktur separat ohne dessen Felder. Ein Beispiele für das zweite Format:

```
struct rect {long x; long y;};          //definition einer Struktur.
struct rect myrect;                      //Struktur Zuweisung zu myrect.
struct FileInfo{dword read,firstBlock,qnBlockRead,retPtr,Work; byte filedir;};
struct FileInfo myfile;                 //Struktur FileInfo Zuweisung zu myfile.
```

Beachten Sie das die Felder einer Struktur in keinster Weise ausgerichtet werden – nur am Anfang der Struktur Zuweisung werden diese ausgerichtet, wenn beim compilieren die Ausrichtung eingeschaltet wurde. Hier ist ein Beispiel wie innerhalb von Strukturen Felder verwendet werden:

```
struct test {
  int a;
  char b [8];
  long c;
} rr, ff [4];
```

In diesem Beispiel wird die Strukturen “test” den Variablen “rr” und einem Feld vor vier dieser Strukturen mit dem Namen “ff” zugewiesen. Die Struktur “test” kann auch mit Hilfe des zweiten Formates definiert werden, wie folgendes Beispiel zeigt:

```
struct test dd;
```

Weiterhin ist es zulässig das Schlüsselwort “struct” auszulassen:

```
  test dd;
```

9.2 Initialisierung von Strukturen bei ihrer Deklaration

Strukturen können nicht bei ihrer Definition Initialisiert (mit Werten aufgefüllt) werden, nur bei ihrer Zuweisung. Weiterhin können nur globale Strukturen bei ihrer Zuweisung Initialisiert werden . C— unterstützt einige Wege Strukturen bei ihrer Initialisierung mit Werten zu füllen:

9.2.1 Einzelner Wert

```
struct test dd = 2;
```

In diesem Beispiel wird der Speicher Bereich der Struktur “dd” mit dem Werten 2 gefüllt. Standardmäßig ist alles auf 8 Bit eingestellt es sei denn es ist eine Datentypausweitung vorhanden. In dem obrigen Beispiel wird der komplette Speicherbereich der Struktur mit dem Wert 2 ausgefüllt.

9.2.2 Felder von Werten

```
struct test dd = {1,2,,6};
```

In diesem Beispiel bekommt das erste Feld der Struktur “dd” den Wert 1 zugewiesen, daß Zweite den Wert 2, das Vierte den Wert 6. Nicht benutzte oder nicht initialisierte Felder bekommen den Wert 0 zugewiesen.

9.2.3 FROM Befehl

```
struct test dd = FROM “file.dat”;
```

In diesem Beispiel ist an dem Platz, wo die Struktur “dd” normalerweise initialisiert wird, der Name einer Datei <filename> angegeben. Sollte die Dateigröße die Größe der Struktur überschreiten, so werden die überschüssigen Bytes einfach in den Programmen Code hinein geschrieben. Sollte die Dateigröße kleiner als die Strukturgröße sein, so wird der Rest der Struktur mit dem Wert 0 gefüllt.

9.2.4 EXTRACT Befehl

```
struct test dd = EXTRACT “file.dat”,24,10;
```

In diesem Beispiel ist an dem platzt, wo die Struktur “dd” normalerweise initialisiert wird, eine Teilangabe der Datei <filename> mit der Länge von 10 Bytes und mit einem Offset von 24 angegeben. Die nicht gefüllten Bytes der Struktur werden mit 0 aufgefüllt.

9.3 Initialisierung von Strukturen während des Programmstartes.

Werte können den Feldern einer Struktur während der Ausführung zugewiesen werden. Beispiel:

```
struct test {short a;char b[8];long c;};
void proc() {
struct test aa[5], rr;
short i;
   aa[0] = 0x78;              //1. Gesamter Speicher wird gefüllt mit 0x78.
   aa[0] = 0x12345678;        //2.  ebenso
   aa[i] = int 0x12345678;    //3. Gesamter Speicher wird gefüllt mit 0x5678.
   aa = long 0x12345678;      //4. Alle Felder werden gefüllt mit 0x12345678.
   rr = i;                          //5. 16-bit Wert füllt den Speicher.
```

In dem ersten Beispiel wird der Speicher des ersten Strukturfeldes (von fünf Strukturen) mit dem Wert 0x78 (standardmäßig) komplett ausgefüllt. Im zweiten Beispiel, wird der höhere Anteil des Wertes einfach ignoriert. Im dritten Beispiel wird der niedrigere Teil des Wertes ignoriert und die Struktur (i + 1) mit dem Anteil des “word” Wertes 0x5678 gefüllt. Im vierten Beispiel wird der komplette Speicher aller fünf Strukturen mit dem 32 Bit long Wert des vollen 0x12345678 gefüllt. Im fünften Beispiel wird der Speicher der Struktur “rr” mit dem 16 Bit Wert der Variable i aufgefüllt.

Es ist möglich den Inhalt einer Struktur in eine andere zu kopieren. Als Beispiel:

```
  rr = aa [2];
```

Kopiert den Inhalt des dritten Strukturfeldes von “aa” in die Struktur “rr“.

9.4 Zugriff auf Strukturen

Zugriff auf Felder von Strukturen und Feldern in Strukturen

Hier Beispiele dafür:

```
struct test {short a;char b[8];long c;} rr[5];
    rr.a = 1;
    rr.b[i] = 2;
    rr[i].c = 3;
    rr[j].b[i] = 4;
```

Hinweis:

Für Operationen in der ein Strukturfeld selbst ein Feld ist und in dem der Feld Index einer Variable ist, kann der Compiler die Register ESI und EDI verwenden, und in einigen Situationen (Beispiel:rr[i].b[j] >< rr[i+1].b[j+2]) wird das Register EDX zusätzlich verwendet.

9.5 Adressen von Strukturen

Es ist möglich die Speicher Adresse einer Struktur und eines zugehörigen Struktur Feldes zu erhalten. Hier sind Beispiele:

```
struct bb               // Name der Struktur.
{
  word b;                     // Erste Feld.
  dword c;                    // Zweite Feld.
} ss;                         // Zuweisung.
void proc ()
{
  EAX=#ss.b;      // Hole Adresse von Feld “b” der Struktur “ss”.
  EAX=#bb.b;      // Hole den offset vom selben Feld der Strukturdefinition
“bb”.
}
```

Es gibt einen sehr wichtigen Unterschied bei diesen Beispielen. Der Ausdruck “EAX=#ss.b;” gibt die absolute Adresse des Feldes wieder, wohingegen der Ausdruck “EAX=#bb.b;” nur den Offset Anteil vom Start der Struktur wiedergibt.

9.6 Verschachtelte Strukturen

bei der Verwendung von Strukturen innerhalb neuer Strukturen ist es wichtig darauf zu achten das diese schon vorher definiert worden sind. Ein Beispiel von Verschachtelten Strukturen:

```
struct RGB
{
  byte Red;
  byte Green;
  byte Blue;
  byte Reserved;
};
```

```
struct BMPINFO
{
  struct BMPHEADER header;    //Die Beschreibung dieser Struktur fehlt.
  struct RGB color [256];     //Feld der RGB Strukturen.
} info;                       //Zuweisung.
```

Nehmen wir nun einmal an, daß der Inhalt des Feldes “red” der zehnte Eintrag des Feldes “color” ist. Das Ganze können wir dann so schreiben:

```
  AL = info.color[10].Red;
```

allerdings gibt es auch eine Begrenzung bei der Benutzung von verschachtelten Strukturen in C—. Es ist nicht erlaubt mehr als eine Variable zu verwenden. Folgendes Beispiel soll dieses verdeutlichen:

```
struct ABC {
  int a;
  int b;
  int c;
};
struct {
  struct ABC first [4];     // 4 Kopien der Struktur ABC
  int d;
} second [4];

int i, k;
void proc ()
{
  AX=second[i].first[k].a; //Ein Fehler, zwei Variablen werden
benutzt.
  AX=second[2].first[k].a; //Diese Schreibweise ist erlaubt.
  AX=second[i].first[3].a; //ebenso erlaubt.
}
```

9.7 Bit Felder von Strukturen

Bit Felder von Strukturen werden verwendet um Speicher zu sparen. Dies erlaubt es Werte in kompakter Form zu speichern und ermöglicht es somit, Werte von Peripheriegeräten besser zu organisieren und direkten Zugriff auf diese zu erhalten, bei denen die einzelnen Bits unterschiedliche Funktionen haben.

Die Definition von Bit Feldern hat folgende Schreibweise:

```
<typ>[<Bezeichner>]:< eine Konstante >;
```

hier ein Beispiel:

```
struct test {byte a:1;b:2;c:1;d:1;long vv;};
struct test rr=0;
```

Das Bit Feld besteht aus einer gewissen Anzahl an Bits, welche durch den Numerische Ausdruck <constant> gesetzt werden.

Der Wert sollte eine positive Zahl seinen und darf die Anzahl der Bits des definierten Datentyps <type> nicht überschreiten.

in C– können Bit Felder nur Vorzeichenlose Werte annehmen. Es ist nicht möglich Felder (Arrays) von Bitfeldern anzulegen oder mit Hilfe von Zeigern darauf zu zugreifen. Hier Beispiele für den Zugriff::

```
cpuspeed.c– 73:     EAX=rr.a;
0000012D  A104010000                    mov     eax,[104h]
00000132  83E001                        and     eax,1

cpuspeed.c– 74:     EAX=rr.b;
00000135  A104010000                    mov     eax,[104h]
0000013A  83E006                        and     eax,6
0000013D  D1E8                          shr     eax,1

cpuspeed.c– 75:     EAX=rr.c;
0000013F  A104010000                    mov     eax,[104h]
00000144  83E008                        and     eax,8
00000147  C1E803                        shr     eax,3

cpuspeed.c– 75:     rr.c=1;
0000013F  800D0401000008                or      byte ptr [104h],8
```

An dem erzeugten Assembler Code können sie sehen, daß die Bit Felder “a”,”b”,”c” und “d” alle in einem einzigen Byte im Speicher abgelegt werden.

<identifier> ist der Name des Bit Feldes. Seine Anwesenheit ist unwichtig. Das unbestimmte Bit Feld definiert ein angemessen Anzahl an Bits vor dem folgenden Feld der Struktur. Bei einem unbestimmten Bit Feld, für welches eine Größe von 0 angezeigt wird, gibt es eine besondere Zuweisung: diese garantiert, daß der Speicher für das folgende Bit Feld an der Grenze dieses Typs anfängt, welches für dieses unbestimmte Bit Feld bestimmt ist.

In C– sind alle Bit Felder hintereinander gepackt deren Größe durch den Datentyp definiert ist. Ist in der Abfolge der Felder mal kein Bit Feld, so werde die restlichen Bits des Bytes nicht verwendet. Die maximale Größe eines einzelnen Bit Feldes ist 32 Bit für den Datentype dword/long, 16 Bit für den Datentyp word/short und 8 Bit für den Datentyp byte/char. die Bit Felder können untereinander mit dem Operator “union” verknüpft sein. “sizeof” auf ein Bit Feld angewendet gibt die Größe des Feldes in Bits zurück..

9.8 die Deklaration von Prozeduren innerhalb von Strukturen

C– unterstützt das deklarieren von Prozeduren in Strukturen, welches vergleichbar ist mit dem Konzept der “classes” von C++. Das bedeutet das eine Prozeduren eine “methode” der “classe” bekommt. Ein Beispiel::

```
struct Point                  //Deklaration der Class.
{
     int x;                        //Dateneintrag
     int y;                        // der Class vom Typ Point.
     void SetX (int);         //Deklaration der Methode
     void SetY (int);         // vom Typ Point.
};

void Point::SetX (int _x)          //Deklaration der Prozedure der
Point Classe.
{
    IF((_x>=0)&&(_x<=MAX_X)) x=_x;

//Die Variablen x und y sind bestandteile dieser Classe und der kon-
squente Zugriff
//auf ihnen durch die Prozeduren der gleichen Classe wird direkt aus-
geführt.
}

void main ()
Point p;                      //Struktur  p  wird  über  den  Stapel
zugewiesen.
{
  p.y = p.x = 0;
  p.SetX(1);
}
```

Der Aufruf von “p.SetX(1)” legt die Adresse der Struktur (classe) auf den Stapel für die aufzurufende Prozedure (methode). Dieser Extraparameter wird nicht explizit spezifiziert in der Anweisung angegeben. In der Prozedur ist dieser Adresse über die Parametrische Variable “this” verfügt war.

Ist die Prozedur mit dem Schlüsselwort “static” deklariert, wird die Variable “this” nicht an die Prozedur übergeben und steht somit der Prozedur nicht zur Verfügung.

Die Prozedur die in der Struktur deklariert würde danach dynamisch sein. Dafür muß diese bei ihrer Definition am Anfang mit einem “:” versehen sein (genauso wie bei den normalen dynamischen Prozeduren). Doch solche dynamische Prozeduren können nicht als Makros verwendet werden.

9.9 Vererbung

C– bietet Mechanismen zur einfachen und mehrfachen Vererbung an. Die Deklaration von Struktur mit Vererbung hat die folgende Schreibweise::

```
struct Derived: Base1, Base2... Basen
{
  long x0;
};
```

Die Anzahl der Grundstrukturen ist nicht limitiert.

Für die mehrfach Vererbung kann die Struktur zwei oder mehrerer Kopien der Grundstruktur weiter vererben. Bei dieser Art gibt es allerdings Mehrdeutigkeiten. Ein Beispiel:

```
struct A
{
  long x, y;
  ...
};
struct B: A         //Die Struktur ‘B’ erbt ‘A’.
{
  ...
};
struct C: A     //Die Struktur ‘C’ erbt ‘A’.
{
  ...
};
struct D: B, C      //Die Struktur ‘D’ erbt  ‘B’ und ‘C’.
{
  ...
};
void main ()
D d;                //Die Struktur ‘D’ bekommt ‘d’ über den Stapel zugewiesen.
{
  d.x = 0;
```

in diesem Beispiel enthält die Struktur “D” zwei Kopien der Struktur “A” und in diesem sind zwei Felder mit dem Namen “x” enthalten. Wenn einen C++ Compiler auf den Ausdruck “d.x=0” stößt, erzeugte eine Fehlermeldung. C– produziert keine Fehlermeldung, sondern greift standardmäßig bei dem Zugriff auf das Feld “x”, auf die letzte Grundstruktur des Feldes “x” zu. Um Zugriff auf die Struktur von “x” zu erhalten muß folgende Schreibweise angewendet werden:

```
  d.B::x=0;
```

von dieser Schreibweise folgt dann

```
  d.x=0;
```

und

```
d.C::x=0;
```

welche beide Äquivalent sind.

Wie man mit Hilfe von Zeiger auf die Inhalte von Struktur zugreifen kann, wird im Abschnitt “Zeiger” beschrieben.

10. Zeiger (pointers)

Dieser Abschnitt zeigt die allgemeine Verwendung von Zeiger. Das verwenden von Zeigern ist nicht komplett implementiert wie man es von normalen C Compilern gewohnt ist.

Hier einige Beispiele wie man Zeiger in C— verwendet::

```
char *string [4] = {“string1”, “string2”, “string3”, 0}; // Ein Zeigerfeld
char *str = “string4”;

main ()
int i;
char *tstr;
{
    FOR (i = 0; string [i] != 0; i++) {
       WRITESTR (string [i]);
       WRITELN ();
    }
    FOR (tstr = str; byte *tstr != 0; tstr++) {
       WRITE (byte *tstr);
    }
}
```

Zeiger können als Parameter für Prozeduren übergeben werden. Zeiger können auch in Strukturen verwendet werden. Es ist auch erlaubt das ein Zeiger auf einen anderen zeigt.

Ein Zeiger kann auch auf eine Prozedur zeigen. Folgend die benötigte Schreibweise:

```
void (*ptr)(); // Deklariert einen Pointer auf diese Prozedur
```

In diesem Beispiel gibt die Funktion den Wert “void”, also nichts, zurück.

Der Compilers führt keine Typenüberprüfung durch, wenn einem Zeiger etwas zugewiesen wird. Als Beispiel:

```
char *z;
  z = #main;
```

Obwohl die Variable “z” als ein Zeiger vom Datentype “char” definiert wurde, wird in der nächsten Zeile diesem Zeiger die Adresse einer Funktion zu gewiesen. Der Compilers merkt nicht daß dies ein Fehler ist, weshalb sie selbst auf solche Konstellationen der Zuweisung aufpassen müssen. Andererseits gibt ihnen dieses natürlich jegliche Freiheit mit Zeiger zu machen was immer sie wollen ohne durch den Compilers behindert zu werden.

10.1 Zeiger auf Strukturen

Die Nutzung von Zeiger auf Strukturen (also die Referenzierung) ist beim normalen C ein wenig anders als bei C—, da C— den “->” Operand nicht kennt. Noch erlaubt der Compilers mir einen Zeiger wie folgt zu deklarieren:n:

```
struct FileInfo *pmyfile;  //Nicht akzeptiertes Format.
```

Auch wenn ich einen Zeiger deklarieren kann, kann ich nicht den “->” Operand einsetzen. Doch wie verwende ich nun einen Zeiger, der auf eine Struktur verweisen soll? Betrachtet man den Beispielcode so scheint es, daß wir ein zusätzliches Register bemühen müssen:

```
cpuspeed.c— 73:     ESI.FileInfo.read=0;
0000013E  C70600000000                  mov     dword ptr [esi],0
```

Beachten Sie das die Schreibweise im Quellcode wie folgt lautet :”ESI.FileInfo.read=0;”.

Standardmäßig greifen wir auf das Daten Segment zu, obwohl wir im FLAT- Speicher Modell normalerweise nicht mit Segmenten arbeiten. Natürlich können wir die ES und FS Segment Register verwenden um auf eine physikalische Speicher Adresse zu zugreifen, weshalb wir einmal ein alternatives Segment einfügen. Nehmen wir ES für folgendes Beispiel:

```
//Während der Initialisierung weise ich die Elemente zu...
cpuspeed.c— 56: struct FileInfo myfile=={0,0,0,0,0,0};
00000104 000000000000000000000000 dd      0,0,0
00000110 0000000000000000         dd      0,0
00000118 00                         db      0

//Jetzt stellen Sie sich vor das “myfile” im ES Register ist (extra
segment):
cpuspeed.c— 74: ESDWORD[#myfile.read]=0;;
0000015A 26C705040100000000000   mov     dword ptr es:[104h],0

cpuspeed.c— 75: ESDWORD[myfile.read]=0;;
00000165 8B3504010000             mov     esi,[104h]
0000016B 26C70600000000           mov     dword ptr es:[esi],0
```

... ich habe es mit und ohne “#” ausprobiert... interessant.

11. andere Interessante Dinge
11.1 Sprungmarkierungen

Sprungmarkierungen werden verwendet um im Programm Code bestimmte Stellen eindeutig wiederfinden zu können, die mit dem Assembler Sprung Befehle oder mit dem C “goto” Befehle angesprungen werden können. Beachten sie wieder das die Schreibweise “GOTO” nur den Sprung innerhalb von +/- 127 Bytes erlaubt.

Es gibt zwei verschiedene Arten von Sprungmarkierungen, globale und lokale. Globale Sprungmarkierungen, wie der Name schon vermuten läßt, sind Markierungen die folglich überall aus dem Programm angesprungen werden können. Lokale Sprung Markierungen sind nur innerhalb des jeweiligen Prozedurblocks sichtbar und für alle weiteren Programmteile außerhalb dieses Blocks nicht sichtbar. Markierungen bestehend aus normalen Bezeichnern gefolgt von einem “:”. Wird in den Bezeichnern ein oder mehrere Kleinbuchstaben verwendet, so ist dieser global, ansonsten (wird er komplett Groß geschrieben) ist es eine lokale Sprungmarkierung.

Globale Sprung Markierungen brauchen in “Inline” Prozeduren nicht verwendet werden, nur lokale Markierungen. An dieses sollte man sich erinnern, da Inline-Prozeduren ja mehrfach während der compilierung im Programm eingefügt werden.

11.2 Vertausch “Swap” Operator

C– besitzt eine Operator, den man in keiner andere Sprache finden kann, den vertausch “Swap” Operator. Diese Operator vertauscht zwei Werte - besser ausgedrückt die Inhalte zweier Variablen oder Register - miteinander. Die Schreibweise dafür ist “><”. Die jeweiligen Inhalte von Variablem werden gegeneinander ausgetauscht. 8 Bit und 8 Bit, 16 Bit und 16 Bit oder 32 Bit und 32 Bit. Einige Beispiele dafür::

```
AX >< BX;                // Sichere den Wert von BX in AX und den
Wert von AX in BX
CH >< BL;                // Vertausche die Werte von CH und BL
dog >< cat;              // Vertausche die Werte der Variablen dog
und cat
counter >< CX;      // Vertausche die Werte von counter und CX
```

Werden zwei 8 Bit Speichervariablen miteinander vertauscht, so wird der Inhalt des Registers AL zerstört. Werden zwei 16 Bit Speichervariablen miteinander vertauscht, so wird der Inhalt des Registers AX zerstört. Sollten zwei 32 Bit Speichervariablen vertauscht werden, so geht der Inhalt des Registers EAX verloren. In allen anderen Fällen, wenn eine Speichervariable und ein Register vertauscht werden, behalten alle Register ihre Werte.

11.3 Der “Neg” Operator

C– verfügt über ein sehr schnellen Befehl um das Vorzeichen einer Variable umzukehren. Plaziert man ein “-” Zeichen vor der Speichervariable oder dem entsprechenden Register gefolgt von einem “;” nach dem Bezeichner, so wir das Vorzeichen des Registers oder der Speichervariable umgekehrt. “Neg“ kommt, wie man sich denken kann, von “negieren“. Einige Beispiele:

```
-AX;      // Dasselbe wie ‘AX=-AX;’, allerdings schneller.
-tree;    // Dasselbe wie ‘tree = -tree;’ ,allerdings schneller.
-BH;      // wechsle das Vorzeichnen von BH.
```

11.4 NOT Operator

C– verfügt über eine schnelle Schreibweise eines Befehls mit den man ein logisches nicht (NOT), denn NOT Operator, auf variablen anwenden kann. Plaziert man ein “!” vor der Variable oder dem Register gefolgt von einem “;”, so wird der Wert der Variable oder des Registers mit dem logischen “NOT” verknüpft und ändert. Einige Beispiele::

```
!AX;      // Dasselbe wie ‘AX ^= 0xFFFF;’ aber schneller.
!node;    // Wechsele den Inhalt von ‘node’ in sein logischen NICHT
(NOT).
!CL;      // Dasselbe wie ‘CL ^= 0xFF’ aber schneller.
```

11.5 spezielle Bedingungsausdrücke

C– unterstützt sechs spezielle Bedingungsausdrücke::

```
CARRYFLAG
NOTCARRYFLAG
OVERFLOW
NOTOVERFLOW
ZEROFLAG
NOTZEROFLAG
```

dieser können genauso verwendet werden wie die normalen Bedingungsausdrücke. Wollen sie beispielsweise einen bestimmten Programmbereich nur dann ausführen, wenn das “carry” Flag gesetzt ist, verwenden Sie einfach folgenden Programmcode:

```
IF( CARRYFLAG )
    {
    // Machen sie hier, was sie wollen
    }
```

wollen sie eine Programm Blocks so lange ausführen wie das “overflow” Flag gesetzt ist, benutzen sie einfach folgende Programmzeilen:

```
do {
     // Machen sie hier, was sie denken
     } while( NOTOVERFLOW );
```

11.6 ‘sizeof’ Operator

Die Operation “sizeof” definiert die Größe der Speichers, der von dem entsprechenden Objekten belegt wird. Das Format ist:

```
 sizeof (<Name eines Datentyps>)
```

Das zurück gegebene Ergebnis ist die Größe des Speichers der belegt wird in Bytes. Der Operator kann auf Variablen, Register, Variablentypen, Strukturen, Textketten und Dateien angewendet werden.

Beispiele:

```
sizeof (“Test”)     //Ergebnis=5. inklusive des null Endes.
char a = “Test”;
sizeof (a)                    //Ergebnis=1.    weil    “a”    ein
Einzelzeichen Datentyp ist.
sizeof (file “filename.dat”)       //Ergebnis= Größe der Datei.
sizeof (func1)      //Gibt die Größe der Prozedur zurück.
sizeof (ss.bb)      //Gibt  die  Größe  von  “bb”  der  Struktur  “ss”
zurück.
sizeof (FileInfo)   //Gibt die Größe der Struktur “FileInfo” zurück.
```

Schreibweisen:

```
  z = sizeof (FileInfo.read);
```

Wollen sie die Größe einer Prozedur erfahren, so muß diese schon vorher im Programm definiert worden sein. Handelte es sich dabei um eine dynamische Prozedur, so wird eine Größe von 0 zurückgegeben.

11.7 Unions

Der “unions” Operand erlaubt es unterschiedliche Variablen, das sie ein und denselben Speicher Bereich verwenden. Die Größe der Speichers wird dabei durch den größten Datentype bestimmt.
Als Beispiel:

```
union
{
  dword regEAX;
  word regAX;
  byte regAL;
}; // Hat 3 Variablen irgendwo im Speicher daklariert.

void test ()
{
    regEAX = 0x2C;
    BL = regAL;  //Register BL bekommt den Wert 0x2C zugewiesen.
}
```

Es ist möglich Variablen von verschwinden Typen, Felder,

Zeichenketten Variablen und Strukturen zu vereinigen. Die Zuordnung kann global und lokal erfolgen, auch innerhalb von Strukturen. Die globale Verknüpfung kann initialisiert sein oder auch nicht. Um das Ganze zu initialisieren, muß nur die erste Einheit initialisiert werden. Ist erste Einheit nicht initialisiert aber einer der folgenden, gibt der Compiler eine Fehlermeldung aus.

12. interne Makros

Diese Makros sind im Compiler integriert und liegen nicht in einer externen Bibliotheken vor. Es sind zwei Gruppen vorhanden, solche für die FPU (Fließkomma Arithmetik Einheit) mit mathematischen Befehlen und solche für den Zugriff auf die I/O Ports.

Hier nun die FPU Makros:

```
atan(x);       //Berechnet den Arctangens der Zahl x.
atan2(x,y);    //Berechnet den Arctangens der Position x/y.
cos(x);        //Berechnet den Cosinus des Wertes x.
exp(x);        //Berechnet den Exponenten der Zahl x.
fabs(x);       //Berechnet den Absolut Wert der Zahl x.
log(x);        //Berechnet den Logarithmus der Zahl x.
log10(x);      //Berechnet den 10 Logarithmus der Zahl x.
sin(x);        //Berechnet den Sinus Wert der Zahl x.
sqrt(x);       //Zieht die 2. Wurzel aus der Zahl x.
tan(x);        //Berechnet den Tangens Wert der Zahl x.
```

Man kann an diese Makros Fließkomma und Vorzeichenbehaftete Ganzzahlen (“long”) übergeben oder bekommt diese Datentypen auch wieder zurück. Beachte das innerhalb der FPU nur Fließkommazahlen verarbeitet werden, aber die Lade und Speicher Anweisung diese in die entsprechenden Ganzzahl konvertiert. Jeglicher Parameter im Winkel Format muß als Radiant angegeben werden.

Hier die I/O Port Makros:

```
inp(port)                 //Ließt ein Byte aus dem Port
inportb(port)   //Ließt ein Byte aus dem Port
inport(port)         //Ließt ein Word aus dem Port
inportd(port)   //Ließt ein double word aus dem Port
```

port - der “port” Parameter ist nicht wesentlich. Wird kein Wert übergeben, so lautet die erzeugte Befehlszeile “in al,dx”. Ist der Port Wert allerdings kleiner 256 so wird folgende Befehlszeile erstellt “in al,port”.

```
outp(val,port)          //Schreibt den byte Wert “val” in den port
outportb(val,port)      //Schreibt den byte Wert “val” in den port
outport(val,port)       //Schreibt den word Wert “val” in den port
outportd(val,port)      //Schreibt den double word Wert “val” in den port
```

val - zu schreibender Wert
port - die Portadresse auf die geschrieben werden soll. Wird kein zu schreibender Wert angegeben, so wird folgende

Befehlszeile erzeugt “out dx,al”.
Ist die angegeben Port Adresse kleiner als 256, wird der Wert über folgenden Befehle ausgegeben “out port,al”.*/

13. Inline Assembler

Der C— Inline Assembler unterstützt alle 8088/8086 Befehle, zusätzlich noch die vom 80286, 80386, 80486 und Pentium bis zu den Pentium III erweiterten Befehle. Hier ein Beispiel für Inline Assembler:

```
//auf der C Ebenen werden variable deklariert:

dword cpuspeed;

    $mov  eax,cpuspeed       //lädt den Inhalt von “cpuspeed” (eckige Klammern
                               sind nicht erlaubt).
    $mov  edi,#smsg1+16      //das “#” Zeichen bedeutet den direkten Modus
                               (Adresse von  “smsg1”).
    $mov  ecx,5
  $newnum:                   //Sprung Markierungen sind erlaubt.
    $xor  edx,edx
    $mov  ebx,10
    $div  ebx
    $add  dl,48
    $mov  DSBYTE[edi],dl     //das Schlüsselwort “DSBYTE” ist erforderlich ist.
    $sub  edi,1
    $loop newnum
```

Beachte das alle Inline Assembler Befehle mit einem “$” beginnen.

Dabei entsteht ein Problem mit den Assemblern NASM und FASM, da die Instruktionen “mov eax,cpuspeed” den Wert der Variable “cpuspeed” lädt. Dabei ist es nicht erlaubt eckige Klammern “[]” zu verwenden. Um die Adresse der Variable zu laden verwenden Sie folgende Befehlszeile “mov eax,#cpuspeed”.

Grundsätzlich erfolgt die Assembler Schreibweise denen des MASM (Microsoft Assembler).

Es ist nicht erlaubt folgendes zu verwenden “mov [edi],dl”. Sie müssen dabei das Segment und die Größe angeben, deshalb das “DSBYTE” Präfix.

Obwohl wir das “FALT”- Speicher Modell für Menuetos verwenden, in welchem CS=DS=SS ist (das ist der Startpunkt des virtuellen Speicherbereiches für das jeweilige Programm), muß die Assembler Anweisung immer noch das Segment decodieren auf welches sie zugreifen will.

Sie können natürlich auch die Inline Assembler Befehle in einem gesamten Block zusammenfassen. Das sieht folgendermaßen aus:

```
asm {
//Assembler Code hier.
}
```

Diese Schreibweise ist erlaubt und vereinfacht einiges, indem sie einfach das Schlüsselwort “asm” verwenden. Ich habe herausgefunden das “${ }” leider nicht funktioniert.

Wollen sie eine komplette Integration von asm und C erreichen, verwenden Sie einfach folgende Kommandozeilenoption:

#pragma option ia //”$” und “asm” Schlüsselwörter nicht benötigt.

Somit brauchen sie für ihren Assemblercode keinen extra Block mehr. Als Beispiel :

```
dword cpuspeed;
void main(void)
{
  cpuspeed=sys_service(5,0)/1000000;      //ein C Ausdruck.
   push eax                                     //Oha, kein Assembler Block!
   mov eax,cpuspeed
   pop eax; xor eax,eax
  draw_window();                                 //zurück zum normalen C
```

Zur Illustration habe ich irgendwelchen zufälligen Programm Code eingefügt.

Sie schreiben eine Assembleranweisung genauso wie in C und können auch das “;” Zeichen für das Befehls Ende verwenden, falls sie mehrere Befehle in einer Zeile schreiben möchten.

Bei manchen Schlüsselwörter kann es zu Kollisionen kommen, Bspw. das Schlüsselwort “int”, daß sowohl ein Assembler Befehl wie auch einen Datentyp darstellt.

Jedoch ist C– sehr oft in der Lage aus der direkten Verwendung des Schlüsselwortes “int” zu erkennen für welche Programmiersprache es gerade verwendet wird..

Um eine Konflikt mit dem Schlüsselwort “int” in header (include Dateien) und die Definition als Befehlsanweisung zu vermeiden, ist es angebracht die folgende Präprozessorzeile “#pragma option ia” nach den “#include” Anweisungen einzufügen. Folgendes Beispiel zeigt die Ursache eines Konfliktes:

EAX = int 1 + a;

Um zu verdeutlichen das der Inline Assembler grundsätzlich MASM kompatibel ist, hier einige Beispiele:

```
    $jmp  short place1       //”short” Schlüsselwort, im +/-127
                               Bytes Bereich.
    $mov eax,cpuspeed        //lädt den Inhalt einer Variable.
    $mov eax,#cpuspeed       //lädt die Adresse einer variable.
```

14. Präprozessor Schalter und Kommandozeilen Optionen

Kommandozeilen Optionen

Sie können sich die Kommandozeile Optionen anzeigen lassen, indem sie einfach “c—” im DOS prompt, ohne jegliche Parameter, eingeben. Folgendes wird ihnen angezeigt::

```
SPHINX C— Compiler   Version 0.238   Jun 03 20022
USAGE: C— [options] [FILE_NAME.INI] [SOURCE_FILE_NAME]]
```

14.1 C— Compiler Optionen

Optimierung

```
/OC       optimiere Code auf Größe
/DE       Aktivierung der Temporären Erweiterungsvariablen
/OS       optimiere Code auf Geschwindigkeit
/OST      Aktivierung der String Optimierung
/ON       Aktivierung für Optimierungnummer
/AP[=n]   Ausrichten der Start Prozedur
/UST      benutze Starteinstellungen für Variablen
/AC[=n]   ausrichten der Start Wiederholungen
```

Code Erzeugung

```
/2        80286 code Optimierung
/3        80386 code Optimierung
/4        80486 code Optimierung
/5        pentium code Optimierung
/SA=####  Start Adresse
/AL=##    Gesetze wert als byte
/WFA      fast call API Prozeduren
/IV       initialisieren alle Variablen
/A        Aktivierung der Adresse Ausrichtung
/SUV=#### Staat Adresse der variablen
```

Präprozessor

```
/IP=<path>     include Datei Pfad
/IA            Assembler Befehle als gültige Bezeichner
/D=<idname>    definiere Bezeichner
/CRI-          Include Dateien nicht auf Wiederholung überprüfen
/MIF=<file>    Haupteingabe Datei
/IND=<name>    importiere Name aus einer DLL
```

linken

```
/AT             füge ATEXIT Unterstützungsblock ein
/STM            Startcode in der main Prozedur
/ARGC           füge “parse” Kommandozeile ein
/NS             Deaktiviere stub
/P              füge parse Kommandozeile ein
/S=#####        Setze Stapelgröße
/C              füge CTRL<C> ignorierung ein
/WIB=#####      Setze Bild Basisadresse
/R              füge größenveränderbaren Speicherblock ein
/WFU            Füge Fix Up table (für Windows32) hinzu
/ENV            füge Variable mit Umgebung ein
/WMB            Erzeuge windows Einzelblock
/J0             Deaktiviere Initialisierungssprung zu main()
/WS=<name>      Setze stub Datei für win32
/J1             Initialisierungssprung nach main() ist short
/WBSS           setze post data in bss Bereich
/J2             Initialisierungssprung nach main() ist near
/WO             Rufe API Prozeduren als ordinal auf
/STUB= <name>   Setzte stub Dateiname
/CPA            Lösche post Bereich
/DOS4GW         Datei arbeitet mit DOS4GW
```

Ausgabe Datei

```
/TEXE     DOS EXE file (model TINY)
/D32      EXE Datei (32bit code für DOS)
/EXE      DOS EXE file (model SMALL)
/W32      EXE für Windows32 GUI
/OBJ      OBJ output file
/W32C     EXE für Windows32 console
/SOBJ     slave OBJ output file
/DLL      DLL für Windows32
/SYM      COM file symbiosis
/DBG      Erzeuge debug informationen
/SYS      device (SYS) file
/LST      Erzeuge assemblerlisting
```

Verschiedenes

```
/HELP /H /?    hilfe, diese Infos
/WORDS         Liste von C— reservierten Bezeichnern
/W             Schalte Warnungen ein
/LAI           Liste der Assembler Befehle
/WF=<file>     Warnungen in eine Datei
/ME            Zeige mein Name und meine Adresse
/MER=##        Maximale Anzahl an Fehlern
/X             Deaktiviere SPHINXC— header ausgabee
/NW=##         Deaktiviere ausgewählte Warnung
```

Compiler Schalter

Viele der Kommandozeilen Optionen können von Compilerschaltern in Quelle Code überschrieben werden. Compilerschalter fangen mit dem Zeichen “?” oder “#” an. Der Präprozessor Befehl “#pragma option” wird verwendet und entsprechende Kommandozeilen Optionen direkt im Quelle Code zu setzen. Als Beispiel zeigen folgende Schalter Einstellungen die ich am Anfang eines Programmes setze das ich für MenuetOS schreibe:

```
#startaddress 0
#code32 TRUE
#pragma option X          //Deaktiviere SPHINXC— header Ausgabe..
#pragma option LST        //Erzeuge Assembler Listingdatei.
//#pragma option OC       //Optimierung der Codegröße.(BK: macht die
                            Startdatei größer!!!)
//#pragma option 4        //für i80486.
//#pragma option A        //Richtet Daten auf Paritätsadresse aus.
#pragma option J0         //Schaltet den Initialisierungssprung zu
                            main() aus.
#resize 0                 //Deaktiviert Speicher Größenveränderung
                            beim Start der Ausgabedatei.
```

Verweise

#includepath

bewirkt dasselbe wie die Kommandozeilen Option “/IP”. Diese Option sagt dem Compiler wo er die entsprechenden “include” Dateien findet. Also in welchem Verzeichnis er suchen soll. Beispiel :#includepath C:\progra~2\c—-\inc

#include

Diese Anweisung kann zwei Formen annehmen. Erstens:#include “windows.h—” sucht als erstes im aktuellen Verzeichnis, wo sich auch das Hauptprogramm befindet. Befindet sich die angegebene Datei dort nicht, wird in den Verzeichnis gesucht welches mit der Anweisung “#includepath” oder der Kommandozeilen Option “/ip=path” definiert worden ist. Befindet sich diese Datei nichts im aktuellen Verzeichnis oder in dem per Kommandozeilen Option angegebenen, wird die entsprechende Datei in dem Verzeichnis gesucht welches zusätzlich noch in der Datei “C—.INI” mit den Befehle “ip=” angegeben wurde (Hinweis, die “c—.ini” muß sich im selben Verzeichnis befinden wo auch der Compiler liegt). Sollte die Suche trotzdem vergeblich sein, so sucht C— nun auch in den Verzeichnis, von wo aus der Compiler gestartet worden ist.ist.

#inline

ich bin mir nicht sicher ob dieses hundertprozentig korrekt ist, aber hier ist die Beschreibung:

Ab und zu ist es vorteilhaft hinzu geladene Routinen auf ihre Größe hin zu optimieren. Und dies zu erreichen verwendet man den Präprozessor Schalter “#inline TRUE”. Setzt man diesen Schalter auf “#inline FALSE”, so wird der hinzugeladen Programm Code auf Geschwindigkeit hin optimiert. Dieser Schalter dient nur zur Optimierung der Geschwindigkeit oder der Programm größe. Weitere Präprozessor Befehle für die Optimierung sind #codesize, #speed und der #inline Befehl, die auch nur innerhalb von Prozeduren gesetzt werden können und deshalb dort nur lokale Auswirkungen haben. Sollen diese Schalter auch globale (also auf das gesamte Programm auswirkende) Funktion haben, ist es wichtig die Schalter außerhalb jeglicher Prozedur zu setzen.

‘C—.INI’ Dateii

In der “c—.ini” Datei lassen sich Compiler Schalter und Präprozessor Optionen schon vordefinieren, die beim Laden des Compiler direkt gesetzt werden. Dies ist eine alternative zu der Kommandozeilen Eingabe oder dem Präprozessor Befehle “#pragma option” innerhalb des Quellcodes. Die Schreibweise ist identisch mit den Kommandozeilen Optionen, allerdings ohne einen vorangestelltes “/” oder “-” Zeichen. Wird die Datei “c—.ini” in dem Verzeichnis, welches durch die Umgebungsvariable “set c—=<path>” definiert wurde oder - wenn diese Variable nicht gesetzt sein sollte - oder in demselben Verzeichnis wo sich auch der Compiler befindet, werden die dort eingetragenen Parameter für alle zu compilierenden Programme verwendet. Wird die Datei allerdings in dem aktuellen Verzeichnis gefunden wo sich auch ihr Quellcode befindet, so werden die dort eingestellt Parameter nur für ihr aktuelles Projekt verwendet.et.

Beispiel einer “C—.INI” Datei::

```
R-
X
3  ; Kommentare sind erlaubt, wenn diese ein “;” vorangestellt haben.
os
```

Die .ini - Datei kann jeglichen Namen haben (die Namenserweiterung muß allerdings .ini lauten). Der Name dieser Datei wird dem Compiler über die Kommandozeilen Option mitgeteilt. Die Datei c—.ini wird vorher ausgeführt, bevor die vom Anwender angegebenen .ini Datei geladen wird. Die “*.ini” Dateien kann man deshalb fast wie Make- Dateien ansehen, da in ihnen spezifische Einstellungen für die Compilierung vorgenommen werden können.

Assembler Instruktionen

Sie können sich über die Kommandozeilenoption “c— /lai” die unterstützten Assembler Befehle des Compiler anzeigen lassen. Für die Version 0. 238 sind das folgende:

SPHINX C— Compiler Version 0.238 Jun 03 20022

Liste der ünterstützten Assemblerbefehle :

AAA	AAD	AAM	AAS	ADC	ADD
ADDPS	ADDSS	ADRSIZE	AND	ANDNPS	ANDPS
ARPL	BOUND	BSF	BSR	BSWAP	BT
BTC	BTR	BTS	CALL	CALLF	CBW
CDQ	CLC	CLD	CLI	CLTS	CMC
CMOVA	CMOVAE	CMOVB	CMOVBE	CMOVC	CMOVE
CMOVG	CMOVGE	CMOVL	CMOVLE	CMOVNA	CMOVNAE
CMOVNB	CMOVNBE	CMOVNC	CMOVNE	CMOVNG	CMOVNGE
CMOVNL	CMOVNLE	CMOVNO	CMOVNP	CMOVNS	CMOVNZ
CMOVO	CMOVP	CMOVPE	CMOVPO	CMOVS	CMOVZ
CMP	CMPPS	CMPSB	CMPSD	CMPSS	CMPSW
CMPXCHG	CMPXCHG8B	COMISS	CPUID	CVTPI2PS	CVTPS2PI
CVTSI2SS	CVTSS2SI	CVTTPS2PI	CVTTSS2SI	CWD	CWDE
DAA	DAS	DB	DD	DEC	DIV
DIVPS	DIVSS	DW	EMMS	EMMX	ENTER
F2XM1	FABS	FADD	FADDP	FBLD	FBSTP
FCHS	FCLEX	FCMOVB	FCMOVBE	FCMOVE	FCMOVNB
FCMOVNBE	FCMOVNE	FCMOVNU	FCMOVU	FCOM	FCOMI
FCOMIP	FCOMP	FCOMPP	FCOS	FDECSTP	FDISI
FDIV	FDIVP	FDIVR	FDIVRP	FENI	FFREE
FIADD	FICOM	FICOMP	FIDIV	FIDIVR	FILD
FILDQ	FIMUL	FINCSTP	FINIT	FIST	FISTP
FISUB	FISUBR	FLD	FLD1	FLDCW	FLDENV
FLDL2E	FLDL2T	FLDLG2	FLDLN2	FLDPI	FLDZ
FMUL	FMULP	FNCLEX	FNDISI	FNENI	FNINIT
FNOP	FNSAVE	FNSETPM	FNSTCW	FNSTENV	FNSTSW
FPATAN	FPREM	FPREM1	FPTAN	FRNDINT	FRSTOR
FSAVE	FSCALE	FSETPM	FSIN	FSINCOS	FSQRT
FST	FSTCW	FSTENV	FSTP	FSTSW	FSUB
FSUBP	FSUBR	FSUBRP	FTST	FUCOM	FUCOMI
FUCOMIP	FUCOMP	FUCOMPP	FWAIT	FXAM	FXCH
FXRSTOR	FXSAVE	FXTRACT	FYL2X	FYL2XP1	HALT
HLT	IDIV	IMUL	IN	INC	INSB
INSD	INSW	INT	INTO	INVD	INVLPD
INVLPG	IRET	IRETD	JA	JAE	JB
JBE	JC	JCXZ	JE	JECXZ	JG
JGE	JL	JLE	JMP	JMPF	JMPN
JMPS	JNA	JNAE	JNB	JNBE	JNC
JNE	JNG	JNGE	JNL	JNLE	JNO
JNP	JNS	JNZ	JO	JP	JPE
JPO	JS	JZ	LAHF	LAR	LDMXCSR
LDS	LEA	LEAVE	LES	LFS	LGDT
LGS	LIDT	LLDT	LMSW	LOADALL	LOCK

```
LODSB     LODSD     LODSW     LOOP      LOOPD     LOOPE
LOOPNE    LOOPNZ    LOOPW     LOOPZ     LSL       LSS
LTR       MASKMOVQ  MAXPS     MAXSS     MINPS     MINSS
MOV       MOVAPS    MOVD      MOVHLPS   MOVHPS    MOVLHPS
MOVLPS    MOVMSKPS  MOVNTPS   MOVNTQ    MOVQ      MOVSB
MOVSD     MOVSS     MOVSW     MOVSX     MOVUPS    MOVZX
MUL       MULPS     MULSS     NEG       NOP       NOT
OPSIZE    OR        ORPS      OUT       OUTSB     OUTSD
OUTSW     PACKSSDW  PACKSSWB  PACKUS    PACKUSWB  PADDB
PADDD     PADDSB    PADDSW    PADDUSB   PADDUSW   PADDW
PAND      PANDN     PAVGB     PAVGW     PCMPEQB   PCMPEQD
PCMPEQW   PCMPGTB   PCMPGTD   PCMPGTW   PEXTRW    PINSRW
PMADD     PMADDWD   PMAXSW    PMAXUB    PMINSW    PMINUB
PMOVMSKB  PMULH     PMULHUW   PMULHW    PMULL     PMULLW
POP       POPA      POPAD     POPF      POPFD     POR
PREFETCHNTA    PREFETCHT0     PREFETCHT1     PREFETCHT2
PSADBW    PSHUFW    PSLLD     PSLLQ     PSLLW     PSRAD
PSRAW     PSRLD     PSRLQ     PSRLW     PSUBB     PSUBD
PSUBSB    PSUBSW    PSUBUSB   PSUBUSW   PSUBW     PUNPCKHBW
PUNPCKHDQ PUNPCKHWD PUNPCKLBW PUNPCKLDQ PUNPCKLWD PUSH
PUSHA     PUSHAD    PUSHF     PUSHFD    PXOR      RCL
RCPPS     RCPSS     RCR       RDMSR     RDPMC     RDTSC
REP       REPE      REPNE     REPNZ     REPZ      RET
RETF      ROL       ROR       RSM       RSQRTPS   RSQRTSS
SAHF      SAL       SAR       SBB       SBC       SCASB
SCASD     SCASW     SETA      SETAE     SETALC    SETB
SETBE     SETC      SETE      SETG      SETGE     SETL
SETLE     SETNA     SETNAE    SETNB     SETNBE    SETNC
SETNE     SETNG     SETNGE    SETNL     SETNLE    SETNO
SETNP     SETNS     SETNZ     SETO      SETP      SETPE
SETPO     SETS      SETZ      SFENCE    SGDT      SHL
SHLD      SHR       SHRD      SHUFPS    SIDT      SLDT
SMSW      SQRTPS    SQRTSS    STC       STD       STI
STMXCSR   STOSB     STOSD     STOSW     STR       SUB
SUBPS     SUBSS     SYSENTER  SYSEXIT   TEST      UCOMISS
UD2       UNPCKHPS  UNPCKLPS  VERR      VERW      WAIT
WBINVD    WRMSR     XADD      XCHG      XLAT      XLATB
XOR       XORPS
```

15. Anhand
Wie installiere ich C— auf meinem Computer..

Das installieren von C— auf dem Computer ist sehr einfach. Nehmen wir an, daß sie C— auf dem Laufwerk C: installieren möchten.

1. Erzeugen sie auf dem Laufwerk C einen Ordner (als Beispiel mit dem Befehle:”MD C—”)..

2. Dann kopieren sie die Bibliotheksdateien (Dateien mit der Änderung *.H).

3. Kopieren sie in denselben Ordner auch folgende Dateien: C—.EXE; MAINLIB.LDP; STARTUP.H— (die Datei C—.INI ist nicht so wichtig).g).

4. dann fügen sie in der Datei “autoexec.bat” folgende Zeile ein: “SET C—=C::\C—”

5. die Bibliotheksdateien können auch in einem anderen Ordner kopiert werden, aber diesen müssen sie dann in der “c—.ini” Datei mit der Zeile “ip=c::\bib-verzeichnis” angeben.

Befindet sich der Compiler im momentanen Verzeichnis, so ist die Umgebungsvariable für C— nicht so wichtig..

Und nun viel Spaß mit C—

16. Kontakt

Über Kommentare und Anregungen freut man sich natürlich immer. Auch konstruktive Kritik ist erwünscht. Einfach eine mail an :

dirk@am-user.de

Kontakt zu Barry Kauler der auch das Englische Original geschrieben hat bekommen sie über seine Homepage unter :

http://www.goosee.com/explorer/