Inhaltsverzeichnis

1 Der Sprachumfang

1.1 Datentypen

1.1.1 Zahlentypen

WORD sind positiv im Bereich von $0 \ldots 2^{16}-1$. Hexadezimalzahlen (vierstellig) können direkt mit einem vorgesElse If wert.r < 8 Then Begin Writeln(Taus, chr(9), 'movb', chr(9), HexWord(adr.a), ', RL', wert.r) End Else Mark('not a byteregister!');tellten Dollarzeichen ($) geschrieben werden, wobei führende Nullen weggelassen werden können.

Binärzahlen (sechzehnstellig) können direkt mit einem vorgestellten Prozentzeichen (%) geschrieben werden. Auch hier können führende Nullen einfach weggelassen werden.

Dies ist der generische Zahlentyp des Compilers, für den viele Optimierungen greifen. Er sollte daher bevorzugt verwendet werden.

INTEGER sind ganzzahlig im Bereich -32768 ... 32767 und belegen ein Word. Für diesen Typ können viele Optimierungen des Compilers nicht genutzt werden und daher sollte er nur verwendet werden wenn Werte negativ werden können.

INTEGER und **WORD** sind zueinander nicht kompatibel. Zur Umwandlung vom einen in den anderen Typ müssen die Inlineprozeduren **TRUNC** bzw. **ABS** verwendet werden.

Einen Fehler bekomme ich mit Integer nicht abgestellt: Wird der Modulo einer negativen Konstanten bezüglich einer zweiten Konstanten berechnet, ergibt dies ein negatives, also falsches, Ergebnis. In der Praxis wird dieser Fall kaum auftreten, er sähe z.B. so aus:

`j := -1000 MOD 16;` ... ergibt ein negatives Ergebnis

Um den Fehler zu vermeiden, muss die negative Konstante eingeklammert werden:

`j := (-1000) MOD 16;` ... ergibt das korrekte Ergebnis

Damit wertet der Parser zuerst den Klammerausdruck aus. so dass ein Integerwert resultiert. Im Generator wird dann die Berechnung des Gesamtausdrucks nicht vom

Compiler übernommen sondern eine `div`–Anweisung eingefügt, die zum korrekten Ergebnis führt.

Liegt einer der Werte in einer Variablen vor, was in der Praxis der Normalfall sein sollte, wird ebenso vorgegangen und ein korrektes Ergebnis geliefert.

BOOLEAN belegen ein Word, **TRUE** = 1, **FALSE** = 0.

1.1.2 Bitweise Wordoperationen

Da die logischen Operatoren natürlich schon vergeben sind, gibt es hierfür drei Extraoperatoren: **ANDB, ORB, XORB**. Beispiel:

```
i := j ANDB k;
```

Achtung: Die logischen Operatoren (OR, &) dürfen nicht auf Wordwerte angewandt werden! Der Compiler erkennt den Fehler nicht und produziert dann unsinnigen Code.

1.1.3 Operatoren

Zuweisung: :=

Arithmetisch: +, -, *, DIV, MOD.

Logisch: &, OR, ~ (not).

Bitweise: ANDB, ORB, XORB.

Bei den arithmetischen Operationen wird nicht auf Über– oder Unterlauf geprüft, die Flags hierzu werden schlichtweg nicht ausgewertet, sondern direkt das Resultat zurückgeliefert. Im Zweifelsfall ist also vor der jeweiligen Operation eine entsprechende Prüfung der Zahlen notwendig. Ob die Zahlen vor der Operation verglichen werden oder die Flags nach der Operation ausgewertet werden, macht keinen großen Unterschied.

Multiplikation und Division mit bzw. durch eine Konstante, die eine Zweierpotenz ist, wird für WORDS durch Schiebeoperationen, bzw. durch Und-Maskierung bei `MOD`, ersetzt. Dieses wird vom Parser jedoch nur erkannt, wenn der zweite Operand diese Konstante ist, was bei `DIV` und `MOD` selbstverständlich ist, nicht jedoch bei der Multiplikation. D.h.:

`i := 2 * i;` wird in eine `mulu`–Sequenz umgesetzt,

`i := i * 2;` in ein einfaches `shl`.

1.1.4 Relationen

= , < , <= , > , >=, #.

1.1.5 Relation IN

Auch ohne den Typ SET ist diese Relation sehr praktisch, zumal sie vom Compiler sehr effektiv umgesetzt werden kann und damit viel weniger Code erzeugt wird als mit verknüpften Bedingungen. Die Auswahlmenge in geschweiften Klammern kann durch Aufzählungen, separiert durch Kommata, und Bereiche gebildet werden. Ein Bereich wird als *Anfangswert .. Endwert* gebildet. Für die C16X als 16–Bit–Controller ist die Menge auf die Werte von 0 bis 15 begrenzt.

Beispiel:

```
IF i IN {0 .. 2, 3, 5, 11} THEN
```

Der Wertebereich wird nur mit Debugglevel > 1 kontrolliert. Ist im obigen Beispiel $i > 15$ bildet der C16X bei der verwendeten SHL–Anweisung automatisch den Modulo 16. Dies kann zu einem unerwünschten Ergebnis führen. Die Konstanten, die die Auswahlmenge bilden, werden vom Compiler kontrolliert und ggf. wird eine Fehlermeldung (out of range) erzeugt.

1.1.6 RECORDs

Sind in üblicher Weise implementiert. Zuweisungen zwischen diesen sind komponentenweise, für identische Typen (vom selben Namen) auch komplett möglich. Bis zu 5 Words kopiert der Compiler hierbei direkt, für größere Strukturen fügt er eine Kopierschleife ein. Zum Kopieren benötigt der Compiler drei Register. Sollten diese knapp werden, bemerkt er dieses daran, dass ihm R1 nicht zugewiesen wird. Dann speichert er R1 vorübergehend auf dem Stack und benutzt dieses.

Die tatsächliche Adresse der jeweiligen Komponente wird beim Zugriff vom Compiler vorab berechnet und entsprechend eingesetzt, so dass kein zusätzlicher Overhead im Programm entsteht.

Ab Version 0.2 können Records als variable Parameter übergeben werden. Hierbei wird nur deren Basisadresse - 2 übergeben, die einzelnen Komponentenadressen werden im laufenden Programm mittels `field` berechnet.

Die Objekteigenschaft *val* einer Recordvariablen enthält den Offset des letzten Elementes zu *Ramstart*. Die Typeigenschaft *size* liefert die Gesamtlänge des Records in Bytes.

1.1.7 ARRAYs

Sind global und für lange Prozeduren implementiert.

In kurzen Prozeduren werden diese nicht implementiert, da es wenig Sinn macht die wenigen Register mit strukturierten Variablen zu füllen. Natürlich kann aber auch aus allen Prozeduren auf globale Arrays zugegriffen werden, so dass diese als Buffer genutzt werden können.

Auch für Arrays sind Zuweisungen elementweise oder komplett möglich (s 1.1.6), .

Ist der Index einer Komponente konstant, wird er vom Compiler berechnet. Wird mit einer Variablen indiziert, muss die effektive Adresse des jeweiligen Elementes zur Laufzeit berechnet werden. Dies erzeugt natürlich einen etwas größeren Overhead.

Grundsätzlich sind auch Ausdrücke als Index möglich. Dabei ist aber etwas Vorsicht angebracht, insbesondere mit der Laufvariablen in FOR–Schleifen, da diese eventuell bei der Auswertung des Ausdruckes verändert werden. Das folgende Beispiel verursacht diesen Fehler, der vom Compiler nicht erkannt wird:

```
FOR i := 0 TO n DO
    a[i + 1] := 0;
END;
```

Hierbei wird verbotenerweise die Laufvariable in der Schleife geändert, was letztendlich zum falschen Ergebnis führt. Ursache ist die Sonderbehandlung der Laufvariablen in FOR–Schleifen, die aus Effizienzgründen immer in einem Register gehalten wird. Die Abhilfe ist verblüffend einfach:

```
FOR i := 0 TO n DO
    a[1 + i] := 0;
END;
```

Da hierbei der erste Faktor des Ausdruckes eine Konstante ist, wird diese in ein eigenes Register geladen und der Wert von `i` hierzu addiert. Die Laufvariable selbst wird dabei nicht verändert. Dieser Fehler wird auch vermieden, wenn der Wert der Laufvariablen in eine andere Variable geladen und der Ausdruck mit dieser geformt wird:

```
FOR i := 0 TO n DO
    k := i; k := k + 1;
    a[k] := 0;
END;
```

Damit ist man in jedem Fall auf der sicheren Seite und die Länge des erzeugten Assemblercodes ist exakt dieselbe wie für das letzte Beispiel. Besser wäre es natürlich, wenn dieser Fehler vom Compiler vermieden würde, dies ist aber nicht so einfach, da es sich syntaktisch um eine krasse Ausnahme handelt. Genau diese wird aber doch recht häufig verwendet, so dass ich mittelfristig wohl eine Lösung finden sollte.

Arrayelemente können jetzt auch als Werteparameter und als variable Parameter an Prozeduren übergeben werden. Verwendet man als variablen Parameter daa erste Element, kann damit in Assembler auf das ganze Array zugegriffen werden. Dies ist jedoch noch sehr wenig getestet.

Ab Version 0.2 können Arrays als variable Parameter an Prozeduren übergeben werden, wobei nur deren Basisadresse - 2 übergeben wird. Hiermit belegen sie nur ein Word. Die Adresse der Komponenten wird im laufenden Programm mittels `index` berechnet.

Entgegen der dringenden Empfehlung N. Wirths wird bei einem Zugriff auf Arrays der Index zur Laufzeit nicht auf Gültigkeit überprüft. Einerseits würde diese Überprüfung recht viel Code an sehr vielen Stellen im Programm ergeben, andererseits greift sie auch nur dann, wenn eine allgemeingültige Fehlerbehandlung zu Verfügung steht. Dieses ist aber bei einem Mikrocontrollersystem normalerweise nicht der Fall. Trotzdem ist eine solche Prüfung sinnvoll, weil man nicht vorhersehen kann, was bei einer Überschreitung der Array–Grenzen passiert. Mit der Compileroption “dbn” mit n $>$ 1 kann u.a. solch eine Überprüfung aktiviert werden (s. Abschnitt 3 ff.).

Doe Objekteigenschaft *val* enthält den Adressoffset des letzten Elementes (das mit dem höchstem Index) zu *Ramstart* und wird von *MakeItem* in die Itemeigenschaft *a* kopiert. Wird von *Selector* für ein indiziertes Element *Index* aufgerufen, dann berechnet diese die Adresse des Elementes und setzt *a* auf 1. Die Typeigenschaft *size* enthält die Gesamtlänge des Arrays in Bytes. Damit kann einfach die Anfangsadresse des Arrays im RAM berechnet werden:

```
sa := ramstart + object↑.val - object↑.o_type↑.size
```

Doe Obejekteigenschaft *len* enthält den höchsten Index und wird zur Überwachung der Indizes genutzt.

1.1.8 Strings

Strings sind in die doppelten Anführungsstriche (Shift + 2) eingeschlossene Zeichenketten und werden vom Typ `STRING` deklariert. Sie werden aus dem Programmtext zunächst in einer Pointerliste gesammelt und hinter den Code der aktuellen Prozedur eingefügt. Sie werden mit “String” + Modulname + Zahl gelabelt, die Zahl wird von 0 beginnend fortlaufend inkrementiert. Momentan beträgt die maximale Länge eines Strings 64 Zeichen inklusive des abschließenden Null–Zeichens (IdLen im Scanner). Diese Begrenzung kann bei Bedarf leicht erweitert werden. Das den String abschließende Null–Zeichen wird

direkt vom Scanner eingefügt, er sorgt auch dafür, dass ein String immer eine gerade Anzahl an Bytes enthält. Falls nicht, wird einfach ein weiteres Nullzeichen angefügt.

An Prozeduren können Strings als variable Parameter übergeben werden, um sie zum Beispiel auf einem LCD auszugeben. Als Parameter wird die (gerade) Adresse des ersten Zeichens (Byte!) übergeben. Das Ende ist durch ein Nullzeichen markiert. Der Zugriff auf sie erfolgt dann über ihre Adresse, die einer Variablen vom Typ WORD mittels `ADR` zugewiesen werden kann und `GET`.

Beispiele:

```
WRITELN("Hallo Welt");  ShowLCD("Hallo Welt");
```

Strings können praktisch nur als Parameter für Prozeduren verwendet werden, insbesondere für `WRITE`. Ansonsten gibt es keinerlei Verarbeitungsroutinen für diese. Die Deklaration einer Variablen vom Typ STRING ist zwar möglich, diese kann aber nur die Startadresse eines Strings enthalten (ein Register bzw. Word). Eine Zuweisung an eine solche Variable ist also nur möglich, wenn der String ein variabler Prozedurparameter ist, aber auch diese ist unsinnig, weil mit der Variablen anschließend nichts anzufangen ist.

1.1.9 TABLEs

Wertetabellen (Lookup tables) werden bei der Mikrocontrollerprogrammierung häufig benötigt, z.B. zur Linearisierung von Sensorkennlinien oder zur Näherungsberechnung transzendenter Funktionen. Programminitialisierte Tabellen können leicht in Arrays angelegt werden, für die genannten Beispiele bevorzugt man jedoch meist Tabellen im Programmspeicher um RAM zu sparen.

Solche Tabellen sind in Oberon nicht vorgesehen, so dass ich mir selbst eine passende Sprachkonstruktion überlegen musste. Aus naheliegenden Gründen, habe ich mich entschieden diese der Konstantendeklaration hinzuzufügen, die Deklaration ist an die der Arrays angelehnt. Beispiel:

```
CONST tab = TABLE OF WORD = 11, 12, 13, 14, 15;
```

Der Zugriff auf die Tabellenwerte erfolgt dann ebenso wie bei den Arrays, wobei auch hier der erste Index 1 ist. Z.B.:

```
w := tab[3];
```

Als Basistyp der Werte sind WORD oder INTEGER möglich, die max. Zahl an Werten habe ich willkürlich auf 512 begrenzt. Lange Tabellen sind mühselig einzugeben, so dass man sich hierfür andere Speichermöglichkeiten überlegen würde.

Bei der Übersetzung werden auch die Tabellen zunächst in einer Pointerliste gespeichert, nach Abschluss der jew. Prozedur hinter den Strings in den Programmspeicher geschrieben. Im Assembleroutput werden sie mit einem Label, gebildet aus "Table" + Modulname + Zahl versehen, die Zahl wird von 0 ausgehend nach jeder Tabelle um eins erhöht.

Zur Übergabe von Tabellenwerten wird die Prozedur TabIndex im Generator benutzt, da die Werte im Gegensatz zu den Arrays aus dem Programmspeicher gelesen werden müssen. Die Werte der Datenpointer DPPx sind hierbei unbekannt, sie könnten ja auf erweitertes RAM umgesetzt sein. Aus diesem Grund wird dem eigentlichen Laden des Wertes immer eine EXTS–Anweisung vorangesetzt, wobei das aktuelle Codesegment Verwendung findet.

Ist der Index konstant, wird sein Wert auf Gültigkeit überprüft. Für einen berechneten Index, der in einer Variablen steht, wird diese Prüfung für aktive Debugoption eingefügt.

1.1.10 POINTER

Mit Hilfe der Pointer werden in Oberon dynamische Variablen realisiert. Diese können nur Records referenzieren, da andere Konstruktionen schlicht sinnlos sind.

Die Pointer selbst enthalten nur die Adresse des Objektes, das sie referenzieren, oder NIL, das als 1 und damit ungerade Adresse realisiert ist. Sie können daher sowohl global als auch lokal als normale Variablen der Klasse g_Ptr realisiert werden. Im Modul sind sie zunächst uninitialisiert, sie sollten daher vom Nutzer mit NIL initialisiert werden.

Pointer müssen als TYPE deklariert werden, wobei eine Vorwärts–Deklaration entsteht weil der Typ des referenzierten Records erst später deklariert werden darf. Bei dieser Deklaration wird ein Objekt angelegt, dessen Eigenschaft `dsc` später auf den referenzierten Typ zeigt. Da dieser noch unbekannt ist, wird `dsc` auf NIL initialisiert und ein neuer Eintrag in die Liste `Listptr` innerhalb des Parsers angelegt. Der Eintrag enthält die Namen des Pointertyps sowie des referenzierten Typs. Wird später der Pointer benötigt, z.B. weil eine Variable des Typs angelegt wird, merkt `find` dass diese Eigenschaft NIL ist, löst mit Hilfe von `Listptr` die Zuordnung auf und setzt `dsc` des Pointerobjektes korrekt.

Ein Beispiel zur Deklaration;

```
TYPE Plist = POINTER TO Tlist;
     Tlist = RECORD
```

```
      a: WORD;
      next: Plist;
    END;

VAR Hlist: Plist;(* globaler Listenkopf *)
    p: Plist;
```

Wie gewohnt, enthält die Variable `Hlist` nur eine Ramadresse. Sie ist zum Modulstart aber uninitialisiert, d.h. ihr Wert ist völlig zufällig. Da der Compiler, genauer das Laufzeitsystem, die Initialisierung nicht übernimmt, sollte sie vom Benutzer als erstes auf NIL initialisiert werden, wodurch sie den Wert 1 erhält, was eine ungerade Adresse ergibt. Greift man nun versehentlich auf `Hlist` zu, obwohl sie noch keinen gültigen Wert enthält, löst dies einen Hardwaretrap–B aus, also einen Interrupt mit dem Vektor $0A. Im zugehörigen Trap–Flag–Register `TFR` ist dann das Bit2 namens `ILLOPA` gesetzt, so dass im Modul dieser Fehler leicht abgefangen werden kann. Man muss nur eine ISR für diesen Fall implemntieren.

Eine Variable des Typs wird mit der Standardprozedur NEW angelegt, also z.B. durch `NEW(p);`. Dadurch wird `p` der Wert der Systemvariablen _HeapTop zugewiesen und _HeapTop um die Länge des Records `Tlist` erhöht, so dass nun Speicher auf dem Heap für einen Record geschaffen wurde. Auf diese Variable kann anschließend ganz normal zugegriffen werden, z.B. mit: `p.a := 0; p.next := NIL;`.

Eine Garbage–Collection existiert nicht, der Platz auf dem Heap kann also nicht mehr freigegeben werden. Für eine effektive Nutzung des Heaps ist daher allein der Nutzer verantwortlich.

Die Dereferenzierung der Pointer übernimmt die Prozedur `PtrField` im Generator. Sie addiert den Offset des jeweiligen Recordelementes zum Pointer, so dass ein Zeiger auf das gewünschte Element resultiert. Die Eigenschaft `mode` des jeweiligen Items wird durch diese Prozedur auf `g_Ptr` gesetzt, woran die späteren Generatorprozeduren Pointerelemente erkennen. Dies funktioniert für lokale Variablen als Parameter nicht, da die Offsetberechnung schon im neuen Kontext erfolgen würde, die ursprünglichen Register dann aber nicht mehr verfügbar sind. Deshalb wird dieser Prozedur der Parameter `procpar` übergeben, ist dieser TRUE wird von `PtrField` kein Output erzeugt, sondern nur der Eigenschaft `b` des Items der berechnete Offset zugewiesen. Die Prozedur `g_Parameter` kümmert sich dann um den Rest. Sowohl Pointer als auch Pointerreferenzen können als variable oder Wert–Parameter an Prozeduren übergeben werden. Dies golt jedoch nur für direkte Referenzen, sonst wird der Aufwand zur Dereferenzierung für eine ziemlich sinnbefreite Aktion zu groß. Es funktioniert also ein Aufruf:

```
Tuwas(p.next, p.a);
```

nicht jedoch:

```
Tuwas(p.next.a);
```

1.2 Kontrollstrukturen

1.2.1 WHILE–DO–END

Ist aus dem Standardumfang von Oberon–0 und implementiert.

1.2.2 FOR–TO–DO–END

Ist implementiert, vorerst ohne BY, das aber als Konstante leicht ergänzt werden könnte. Hierzu müsste g_Inc im Generator so erweitert werden, dass beliebige Inkremente möglich werden.

Die Laufvariable und deren Endmarke werden aus Effizienzgründen immer in Registern gehalten. Deshalb ist etwas Vorsicht bei Berechnungen mit diesen, also bei Ausdrücken, die diese enthalten, geboten. Siehe hierzu die Hinweise zu syntaktischen Einschränkungen auf Abschnitt 1.4.1.

1.2.3 REPEAT–UNTIL

Ist implementiert.

1.2.4 IF–THEN–ELSIF–ELSE–END

Ist aus dem Standardumfang von Oberon–0 und implementiert.

1.2.5 Boolsche Ausdrücke als Bedingung

Für Boolsche Variablen werden diese im Generator in der Prozedur `TestTrue` ausgewertet, damit die Prozedur `Relation` nicht noch unübersichtlicher wird.

Damit sind keine logischen Verknüpfungen als Prozedurparameter möglich, der Compiler erkennt den Versuch nicht und liefert unsinnigen Code.

Soweit getestet funktionieren diese auch mit Invertierung mittels ~ . Auch Kombinationen mit & und OR sind möglich. Beispiel:

```
IF (j) & (k) THEN ...
```

Werden statt boolscher Werte Ausdrücke verwendet, müssen diese geklammert werden. Ohne die Klammern erkennt der Parser die Konstruktion nicht und erzeugt ohne Fehlermeldung unsinnigen Code (wenn das Symbol & auftaucht, ist die schließende Klammer nicht mehr zugänglich, so dass die Syntax nicht geprüft werden kann).

Diese kombinierten Ausdrücke erzeugen jedoch viel Code und da die boolschen Variablen jeweils ein Word belegen wird auch viel Speicher verschenkt. Der Output der obigen Anweisung wird zu:

```
        mov     R2, 0E004h
        cmp     R2, #1
        jmp     NE, L00005
        jmp     L00006                  ; AND passed
L00005
        jmp     L00007                  ; AND failed
L00006                  ;  AND next
        mov     R2, 0E006h
        cmp     R2, #1
        jmp     NE, L00007
        mov     R2, #0
        jmp     L00008
L00007                  ; AND failed
        mov     R2, #1
L00008                  ;  AND done
        jmp     NE, L00003
        mov     R2, #5
        mov     0E002h, R2
```

Besser ist es, eine Wordvariable für Flags anzulegen und mit dieser bis zu 16 boolsche Variablen zu ersetzen. Z.B. so:

```
CONST Bit0 = $0001; Bit8 = $0100;
VAR Flags: WORD;

IF (Flags ANDB Bit8) # 0 THEN (* Flag gesetzt *)

IF Flags ANDB (Bit0 + Bit8) = (Bit0 + Bit8) THEN
(* wenn beide Flags gesetzt sind *)
```

Die konstanten Ausdrücke berechnet hierbei der Compiler und setzt sie immediate in den Code ein. Für das zweite Beispiel:

```
mov     R2, 0E008h
and     R2, #257
cmp     R2, #257
jmp     NE, L00003
```

1.2.6 Kommentare

Alles, was zwischen `(*` und `*)` steht, wird schon vom Scanner ignoriert. Dies sollte auch mit geschachtelten Konstruktionen funktionieren, ich bin aber noch nicht sicher ob das immer fehlerfrei hinhaut.

1.3 SFRs

Deren Adressen sind jetzt im Compiler integriert, so dass der **ASL** keine Definitionsdatei mehr benötigt und sie auch nicht mehr deklariert werden müssen. Bitdeklarationen sind hierbei nicht vorgesehen und ich werde diese vermutlich auch nicht nachrüsten. Der Aufwand für die Bereichsüberprüfung wird einfach zu groß.

Als Standardsatz sind die Definitionen des C167CR in Form der Unit `C167CR.pas` in das Steuerprogramm `obcc` eingebunden. Dieser kann mit der Compileroption `-irFileName` ersetzt werden, wenn das zugehörige File gefunden wird. Das File ist eine Textdatei, die zeilenweise jeweils den Namen des Registers sowie, durch ein Komma getrennt, dessen Speicheradresse enthält. Beispiele hierzu lege ich dem Compiler bei.

Eine direkte Manipulation der SFR ist nicht möglich, es sind ja keine Variablen. Es können ihnen lediglich der Wert einer Word–Variablen oder eine Konstanten zugewiesen werden. Um z.B. Bit 0 einer solchen zu setzen ist daher folgende Sequenz nötig:

`w`:= DP1L; DP1L := w ORB 1;

Mit der Deklaration entfällt auch die Möglichkeit der Aliasnamen, vermutlich kann man leicht darauf verzichten.

Das Steuerprogramm `obcc` legt für die SFR die Pointerliste `Listsfr`, die im Generator deklariert ist, an und trägt Namen und Adresse der Register hierein. Im Parser ruft `find` ggf. `findsfr` auf, welche bei Erfolg ein Objekt des Namens anlegt und dessen Eigenschaft `val` die Adresse zuweist. Beim Zugriff auf das Objekt legt `MakeItem` ein Item an, kopiert die Adresse in dessen Eigenschaft `a` und berechnet für dessen Eigenschaft `b` die Registernummer. SFR, die nicht über eine Registernummer angesprochen werden können (z.B. die über den XBUS adressierten) erhalten die Registernummer -1.

Für die Benutzung von SFR des Compilers selbst gibt es in `SProcs`, das auch im Generator eingebunden ist, die Funktion Fsfr vom Typ `Tsfrreg`, die als Argument den

Namen des Registers erhält und im Erfolgsfall die Adresse und Registernummer als Eigenschaft `adr` bzw. `reg` zurückliefert. Wird das Register nicht gefunden, werden beide Eigenschaften auf -1 gesetzt.

1.4 Prozeduren

Bei Aufruf einer Prozedur wird der Kontextpointer des C167 (CP) um 32 erhöht, so dass jede Prozedur (wie auch jeder Interrupt) zunächst über 16 Register verfügt. Der FSP wird in R0 übergeben, so dass die Prozedur auch Platz auf dem Frame nutzen könnte.

```
push    R0          ; R0 = FSP
add     CP,#32
nop                 ; wg. Pipelining
pop     R0          ; R0 = FSP
```

Bei der Rückkehr aus einer Prozedur wird dann nur der Kontextpointer um 32 dekrementiert, wobei der FSP automatisch restauriert wird.

Es gibt lange und kurze Prozeduren, wobei diese sich zunächst in der Ablage der Parameter und lokalen Variablen unterscheiden. In kurzen Prozeduren werden alle Parameter und lokalen Variablen direkt in Registern gespeichert und verarbeitet, in langen Prozeduren sind diese auf dem Frame angelegt.

Globale Variablen und variable Parameter werden bei Änderung jeweils sofort zurückgeschrieben. Dies bringt natürlich einen größeren Overhead mit sich, ist aber nötig, da sich diese Werte in einem Interrupt ändern können und eine Prozedur das mitbekommen muss. Wenn dieser Effekt z.B. aus Performancegründen unerwünscht ist, muss der Wert dieser Größen am Anfang der Prozedur einer lokalen Variablen zugewiesen und am Ende zurückgeschrieben werden. Das sollte natürlich nur mit solchen Werten erfolgen, die sich während der Prozedur außerhalb sicher nicht ändern können.

1.4.1 Kurze Proceduren

So nenne ich der Kürze halber die Prozeduren mit Registervariablen. Da diese für die Syntax einige Einschränkungen mitbringen (s.u.), habe ich den Automatismus zu ihrer Erkennung verworfen und verwende lieber N. Wirths Leaf–Konzept: Um eine solche Prozedur zu deklarieren, muss dem Schlüsselwort PROCEDURE ein Sternchen (*) folgen. Die Anzahl an Parametern und lokalen Variablen wird hierbei überwacht und bei einer Überschreitung des Wertes von `maxvarcnt` eine Fehlermeldung (“too many variables”) ausgegeben.

Der erste Parameter wird im höchsten Register R15 abgelegt. Absteigend landet dann die letzte deklarierte Variable im niedrigsten verwendeten Register, momentan maximal in R8. Bei variablen Parametern wird wie gewöhnlich die Adresse der tatsächlichen Variablen im jew. Register abgelegt, der Mode dieser Parameter ist dann g_Par anstelle g_Var. Die Registernummer mal 2 (!) jeder Größe steckt in ihrer Werteigenschaft *val* und wird von *load* entsprechend umgerechnet in der Eigenschaft r des jeweiligen Items abgelegt. Die maximale Anzahl an Parametern und Variablen kann später sicher noch deutlich erhöht werden,

Der FSP belegt Register R0.

Syntaktische Einschränkungen

Da alle Variablen in Registern stehen, können sie ohne Laden verarbeitet werden, wodurch natürlich die Codeeffektivität stark verbessert wird. Gleichzeitig entstehen hierdurch jedoch unerwünschte Nebenwirkungen, deren sich der Programmierer bewusst sein muss. Er soll hieran erinnert werden, indem er der Prozedurdeklaration explizit ein Sternchen anhängt.

Alle Aktionen werden direkt in den jeweiligen Registern ausgeführt. Dies bedeutet, dass für

```
x := y + z;
```

zunächst zum y repräsentierenden Register der Wert von z addiert wird und das Register x dann durch das Register y überschrieben wird. Dass hierbei y verändert wird, ist natürlich unerwünscht. Es lässt sich aber nicht einfach vermeiden, da der Parser zur Auswertung dieser Zuordnung die Schritte

```
MakeItem(x); expression(y); store(x,y);
```

ausführt.

Der in `epression` auszuwertende Ausdruck y und das hierbei im Generator aufgerufene `op2` wissen also nichts von x und können dieses Zielregister nicht zur Berechnung verwenden. Die einzige Möglichkeit das Ändern von y zu vermeiden wäre, in `op2` ein neues Register anzufordern, die Rechnung hierin auszuführen und das Ergebnis x zuzuweisen.

Dann würde aber z.B. auch für

```
x := x DIV 2;
```

derselbe Mechanismus greifen, wodurch die Nutzung der Registervariablen sinnlos wird.

Es muss daher im Quellcode schon an dieses Problem gedacht und entsprechend umcodiert werden:

```
x := y; x := x + z;
```

bringt das gewünschte Ergebnis, ohne dass y verändert wird und ohne zusätzlichen Overhead.

Besonders tückisch ist dieses Problem bei der Auswertung bedingter Anweisungen, da auch hierbei der Quelloperand verändert wird. In

```
IF (x ANDB 64) # 0 THEN ...
```

wird die Und–Verknüpfung im Register x duchgeführt, wodurch dessen ursprünglicher Inhalt zerstört wird. Hier muss also eine temporäre Variable benutzt werden, der jeweils vor der Anweisung der Wert von x zugeschrieben wird, sofern der Wert von `x` nach der Operation noch benötigt wird.

1.4.2 Lange Prozeduren

Alle Parameter und lokalen Variablen werden auf dem Frame angelegt. Hierzu wird FSP zunächst um *locblksize* der Prozedur (aus ihrer Werteigenschaft *val*) dekrementiert.

Der erste Parameter liegt dann bei (FSP), die nächste Größe bei (FSP + 2) und die letzte bei (FSP + locblksize - 2), (FSP + locblksize) gehört ja noch der aufrufenden Ebene. Der jeweilige Offset steckt in der Eigenschaft *val* des Objektes und wird von $MakeItem$ in die Eigenschaft a des Items kopiert. Die Eigenschaft r des Items wird für diese Variablen auf FSP gesetzt.

Da zur Parameterübergabe der ursprüngliche FSP noch benötigt wird, wird er bei Aufruf der Prozedur in den FP (R1) kopiert. Nach Abschluss der Parameterübergabe wird dieses Register wieder frei gegeben.

Für die langen Prozeduren muss gegenüber den kurzen kein großer Performancenachteil entstehen, wenn die flexible indizierte Adressierung des C167 eingesetzt wird, etwa wie in

```
mov Rwn, [Rwm + #data16].
```

Achtung: Damit der **AS** dies versteht, muss die Syntax exakt eingehalten werde (keine Leerzeichen), z.B.:

```
mov R2, [R1+#2]
```

Letztendlich sieht der Overhead doch recht groß aus, insbesondere wegen der vielen Ladeoperationen in Register.

1.4.3 Parameter

Wertparameter:

Diese werden exakt wie lokale Variablen realisiert und vom Parser auch so (als g_Var) markiert. Dementsprechend können sie innerhalb der Prozedur auch als Variablen verwendet werden, wenn ihr Ursprungswert nicht mehr benötigt wird. Das spart bei kurzen Prozeduren Register, bei langen Prozeduren immerhin Speicherplatz auf dem Frame.

Variable Parameter:

Werden diese in Zuweisungen (z.B. Berechnungen) benutzt, müssen sie jeweils in ein freies Register geladen und anschließend zurückgespeichert werden. Hierdurch steigt natürlich der Aufwand, was sich aber nicht vermeiden lässt, da die entsprechenden Befehle beim C167 ein Register als Zieloperand voraussetzen.

In zeitkritischen Prozeduren sollten sie daher am Anfang in eine lokale Variable geladen und erst am Ende zurückgespeichert werden. Das Register wird sowieso benötigt!

Bei Ausdrücken, die mit variablen Parametern ausgeführt werden, wird der Modus des Resultats von `MakeItem` entsprechend der Quelle als g_Par markiert. In diesem Fall liegt die Quelle aber sicherlich in einem Register vor, da sie vorher von `Op2` geladen wurde. In `store` wird für diesen Fall der Modus des Quelloperanden schlicht auf g_Reg gesetzt, wodurch das gewünschte Ergebnis resultiert.

In `load` wurde bisher der Modus für var. Parameter auf g_Par gelassen, was zu einem Fehler in `Op2` führt. Jetzt setzt `load` den Modus auf g_Reg. Sollte es dadurch Probleme geben, muss `Op2` so geändert werden, dass für das Y–Item g_Par wie g_Reg behandelt wird.

Variable Parameter einer Prozedur können nicht als Parameter beim Aufruf einer weiteren Prozedur genutzt werden. Dieses wird zu kompliziert umzusetzen. Der Compiler erkennt den Versuch und gibt eine entsprechende Fehlermeldung aus. Für strukturierte Parameter (Records und Arrays) ist diese Beschränkung aufgehoben und ich sollte sie gelegentlich ganz entfernen.

1.4.4 Funktionsprozeduren

Dies sind Prozeduren, die einen skalaren Wert zurückliefern. Der Typ des Rückgabewertes wird bei der Prozedurdeklaration angegeben, die Zuweisung des Wertes erfolgt mit RETURN. Dies sieht z.B. folgendermaßen aus:

```
PROCEDURE* Odd(v: WORD): BOOLEAN;
BEGIN
  IF v MOD 2 = 1 THEN RETURN TRUE
  ELSE RETURN FALSE; END;
END Odd;
```

Der Aufruf einer solchen Prozedur erfolgt als Ausdruck (`expression`) und wird in Unterfall Faktor (`factor`) behandelt. Er kann also in Zuweisungen oder als Bedingung in bedingten Anweisungen erfolgen, z.B. so:

```
IF Odd(i) THEN WRITELN("Ungerade")
ELSE WRITELN("Gerade"); END;
```

Für den Rückgabewert legt der Compiler gleich nach seinem Aufruf die (erste) globale Variable `_FRESULT` zunächst vom Typ INTEGER an. Damit wird für ihn kein Register verschwendet und unnötige Stackarithmetik vermieden. Bei der Übersetzung einer Funktionsprozedur wird der Typ dieser Variablen auf den Ergebnistyp der Prozedur gesetzt, was problemlos möglich ist, da alle skalaren Typen 2 Bytes im Speicher belegen. Durch dieses Vorgehen erfolgt sowohl bei der Return–Anweisung als auch beim Aufruf eine Typüberprüfung. Im obigen Beispiel würde ein falscher Typ im IF–Zweig allerdings mit der folgenden Zeilennummer angegezeigt, ein Problem das öfter auftritt. Die Prozedur `expression` und ihre Kollegen haben halt vorauseilend schon das nächste Symbol angefordert.

Der Compiler überwacht, ob eine Return–Anweisung in der Funktionsprozedur vorhanden ist. Er kann dabei allerdings nicht hellsehen. Fehlt diese z.B. im obigen Beispiel im ELSE–Zweig, merkt er das nicht und liefert keine Fehlermeldung. In diesem Fall wird ein unsinniger Wert zurückgeliefert, es ist der zuletzt an `_FRESULT` zugewiesene Wert.

1.4.5 Interrupts

Das sind immer kurze Prozeduren und sollten daher mit dem Sternchen deklariert werden. Sie haben keinen aktuellen Zugriff zum Frame, dafür sind alle 16 Register verfügbar, wobei wie bei kurzen Prozeduren die lokalen Variablen (Parameter gibt es natürlich keine) die Register von oben füllen. Für sie gelten die syntaktischen Einschränkungen für kurze Prozeduren (s. Abschnitt 1.4.1).

Der Prozedurname lautet `ISR` und der Interruptvektor wird in eckiger Klammer übergeben. Zum Beispiel:

```
VAR tick: WORD;

     PROCEDURE* ISR[35]; (* Timer3-Interrupt *)
     BEGIN tick := tick + 1;
     END ISR;
```

Alternativ kann auch der Name des Interruptvektors in der eckigen Klammer angegeben werden, dann muss man die Vektornummer nicht nachschlagen: Z.B.:

```
     PROCEDURE* ISR[T3INT]; (* Timer3-Interrupt *)
```

Der zugehörige Vektor wird automatisch gesetzt, das Unterprogramm erhält als Label den Vektornamen laut Datenblatt. Die Interruptfreigabe auch über **IEN** muss im Programm erfolgen.

Als Standard sind die Interruptvektoren des C167CR im Compiler integriert, Definitionen für andere Derivate können an das Registerdefinitionsfile angehängt werden. Hierzu wird das Schlüssewort `Vectors:` und anschließend die Vektordefinitionen angegeben. Die Definitionen enthalten jeweils eine Zeile pro Vektor, der (in dieser Reihenfolge) die Vektornummer, den Vektornamen und die Vektoradresse angibt. Trennzeichen ist das Komma, Beispiele liegen den Quellen bei. Um Definitionen für andere Derivate zu erstellen, kann als Ausgangspunkt die Datei C167CR.def aus den Beispielen dienen. Eingebunden werden diese mit der Compileroption `-irFilname`.

Ein direkter Aufruf der ISR aus dem Programm heraus ist nicht möglich, der Compiler wirft bei dem Versuch Fehlermeldungen. Dieser ist unsinnig und wäre fehlerträchtig, da bei einem Interrupt im Gegensatz zum normalen Prozeduraufruf das PSW auf den Stack gepusht und dies von `RETI` automatisch wieder zurückgeschrieben wird. Sollte solch ein Aufruf nötig sein, kann einfach das zugehörige Interruptflag direkt gesetzt werden. Dies ist beim C167 auzsdrücklich zulässig und führt zum gewünschten Ergebnis. Das Flag wird nach Ausführung der ISR vom C167 automatisch zurückgesetzt (im Gegensatz zu den nichtmaskierbaren Interrupts).

Aus einem Interrupt heraus können keine Prozeduren aufgerufen werden. Dies würde, soweit ich das bisher überblicke, mit dem Kontextpointerkonzept der XC16x und ev. auch der Nachfolger kollidieren. Der Parser bemerkt den Versuch eines solchen Aufrufs und erzeugt eine Fehlermeldung.

Wenn ein Interrupt Multiplikation oder Division nutzt, müssen `MDH, MDL` und `MDC` auf den Stack gepusht werden, da diese Operationen von Interrupts unterbrochen werden können. Hier fehlt noch ein Automatismus, der aber leicht zu ergänzen sein sollte.

1.5 Inlineprozeduren

Achtung: Die Ein– und Ausgaberoutinen sollten nicht in Interrupts verwendet werden. Dazu sind sie schlicht zu zeitaufwendig. Eventuell werde ich diese Möglichkit im Generator sperren und eine Fehlermeldung ausgeben lassen.

1.5.1 WRITE und WRITELN

`WRITE` und `WRITELN` erwarten einen oder mehrere, durch Kommas separierte, Parameter, der ein String, eingeschlossen in die doppelten Anführungszeichen (Shift + 2), eine Konstante oder eine Variable sein kann. Die Ausgabe erfolgt sofort auf den UART0 im Pollingbetrieb. Während der Durchführung werden die **Interrupts** gesperrt (IEN := 0), hinterher durch `pop` des `PSW` ggf. wieder freigegeben. Gegenüber `WRITE` hängt `WRITELN` ein CR und ein LF an die Ausgabe an.

Die serielle Schnittstelle muss natürlich vorher passend konfiguriert sein. Die Ausgabe erfolgt aus Effizienzgründen in Unterprogrammen, die bei Bedarf automatisch hinter dem Hauptprogramm und seinen Strings in den Code eingefügt werden.

Zahlen, d.h. Konstanten und Variablenwerte, werden als ASCII–Zeichen am Ende des von globalen Variablen belegten Speicherbereichs (*vartop*) angelegt und anschließend versendet. Eventuelle führende Nullen werden hierbei unterdrückt.

Durch einen Doppelpunkt kann an Variablen und Konstanten eine optionale Formatanweisungen angehängt werden. Bisher sind die folgenden Optionen implementiert:

:h , hexadezimale Ausgabe. Hexadezimale Ausgaben werden genauso im Speicher angelegt wie dezimale jedoch inklusive der führenden Nullen versendet.

:c , Zeichenausgabe. Der Wert wird in der Folge Highbyte, Lowbyte als Zeichen (Character) ausgegeben. Hierzu werden einfach die Werte der Bytes übertragen und vom Empfänger als ASCII–Zeichen interpretiert.

Für die Ausgabe wird jeweils ein neuer Kontext eröffnet und anschließend sofort wieder beendet. Da für `WRITELN` ohne Parameter keine Register benötigt werden, wird hierfür auch kein Kontext eröffnet, nur das PSW wird gepusht, Interrupts disabled und hinterher durch das Rückladen des PSW ggf. wieder enabled. Dieser Fall wird im Generator noch in IOCall behandelt, für die beiden anderen Fälle wurden hier eigene Prozeduren angelegt.

1.5.2 READ

Wartet bis ein Zeichen im UART angekommen ist und übergibt der als Argument übergebenen Variablen das empfangene Zeichen. In der Regel ist dies der ASCII–Code der am Terminal gedrückten Taste.

Kommt kein Zeichen an wird ewig gewartet. Es könnte daher sinnvoll sein, vor dem `READ` zunächst zu prüfen, ob ein Zeichen angekommen ist, also ob das Flag `SORIR` gesetzt ist. Dieses wird durch `READ` zurückgesetzt.

Im Monitor von Keil wird kurz nach Aufruf von `READ` ein Zeichen mit Code 17 oder 255 empfangen. Keine Ahnung, ob hier ein Timeout verwendet wird. Um dieses Problem zu umgehen, kann man in einer Schleife einlesen, bis der Code einem druckbaren Zeichen entspricht.

In Prozeduren und insbesondere bei Verwendung variabler Parameter als Argument ist `READ` praktisch noch ungetestet!

1.5.3 SLEEP

Versetzt den Kontroller in den Idle–Modus, woraus er nur durch einen Interrupt wieder geweckt wird. Bei 20–MHz–Takt läuft der Kontroller hierdurch merklich kühler. Die habe ich nur implementiert, weil mich die direkte Assemblereingabe (s. Abschnitt 1.6) und das Drumherum mit dem Prozeduraufruf für eine einzelne Assembleranweisung genervt hat.

1.5.4 INC und DEC

Zum Inkrementieren bzw. Dekrementieren einer Variablen. Die vollständige Syntax lautet:

`INC(x[,`y]); bzw. DEC(x[, y]);

Der erste Prameter x ist die Variable, der Zweite y der Wert der addiert bzw. subtrahiert werden soll. Fehlt der zweite Parameter, wird 1 für ihn angenommen. Der zweite Parameter kann ein beliebiger Integerausdruck sein.

Diese Inline–Prozeduren sollen eigentlich nur die Tipparbeit abkürzen, da das häufig benutzte `i:= i + 1;` mit strukturierten Variablen doch sehr länglich wird. Gerade mit Arrayelementen erzeugen sie aber auch etwas kompakteren Code. Der Parser weiß ja, dass hierbei Ziel und Quelle identisch sind und muss dann die zugehörige Adresse nur einmal dekodieren und sie in eine temporäre Variable (Register) kopieren.

1.5.5 SWAP

Vertauscht die Bytes eines Words. Dies ist recht häufig nötig, da die Struktur der C16x auf Little–Endianess beruht. Dies bedeutet, dass im Speicher das niedrigwertige Byte zuerst abgelegt wird, das hochwertige Byte folgt. Solange man mit Registern arbeitet ist dies unrelevant. Es wird erst entscheidend, wenn man mit Peripheriebauteilen kommuniziert. Selbst der CS8900A, der mit einem 16–Bit–Bus an den C167 angebunden ist, erfordert diese Bytevertauschung, da er, wie die meisten Peripherieeinheiten, auf einer Big–Endian–Struktur basiert.

Natürlich kann diese Vertauschung leicht in Oberon erfolgen, z.B. mit folgendem Code:

```
y := x; y := (y MOD 256) * 256; x := x DIV 256; x := x + y;
```

Damit kann leicht eine Prozedur zu diesem Zweck erstellt werden. Oft ist für diese Operation jedoch lediglich die Assembleranweisung `rol` Rx, #8 notwendig und es wäre übertrieben hierfür einen Kontextswitch einzufügen. Deshalb ist diese Operation im Compiler besser aufgehoben.

1.5.6 GET und PUT, GETB und PUTB

Lädt ein Word aus dem Speicher (GET), bzw. schreibt ein Word in den Speicher (PUT). Syntax:

`GET(adr,`segment: WORD):WORD;

`PUT(adr,`segment, wert: WORD);

Die Adresse **muss gerade sein** und liegt im Bereich 0 bis $FFFE, das Speichersegment im Bereich 0 bis 255. Damit wird der Zugriff auf den gesamten Speicher des Controllers möglich, was natürlich mit externem Bus interessant ist, aber auch bei den neueren Derivaten z.B. für den Zugriff auf das PSRAM.

Die Speicherseite wird nicht kontrolliert, für Adresse und Wert sind sowieso keine Tests möglich. Der Anwender muss vor allem bei PUT selbst sicherstellen, dass er auf die richtige Speicherseite zugreift. Wild im Speicher rumzupoken ist halt riskant.

GETB und PUTB wirken genau wie GET und PUT, mit dem Unterschied, dass ein Bytetransfer durchgeführt wird. Dies ist nur in krassen Ausnahmen nötig und sollte auch nur dann verwendet werden. Nötig wird das z.B. wenn Peripherie mit einem 8–Bit–Bus angekoppelt ist oder Flash an einem solchen Bus beschrieben werden soll. Argumente und Rückgabewert sind vom Typ WORD, wobei das niedrigwertige Byte übertragen wird. Bei GETB wird das hochwertige Byte mit Null gefüllt, bei PUTB ignoriert.

1.5.7 INTSEN und INTSDIS

`INTSEN` setzt Bit IEN im PSW und gibt damit maskierbare Interrupts frei. `INTSDIS` löscht dieses Bit und sperrt damit Interrupts. Diese habe ich nur zur Abkürzung eingeführt, andere Lösungen waren davor schon möglich.

1.5.8 HALT

Ist der Debuglevel beim Compilieren 1 oder 3, wird an dieser Stelle ein illegaler Opcode in den Assembleroutput eingefügt. Dies führt zum Aufruf des Debugmonitors, der bei diesen Leveln an die Anwendung angehängt wird.

Bei anderen Debugleveln wird diese Anweisung schlicht ignoriert.

1.5.9 ADR

Normalerweise benötigt man nie die Adresse einer Variablen, da sich ja der Compiler um die Zuordnung kümmert.. Zwei Ausnahmem sind mir hierzu aber aufgefallen:

Zum Einen benötigt man für PEC–Transfers Puffer, deren Startadresse dem jeweiligen PEC–Pointer übergeben wird. Für diesen Zweck kommen nur globale Variablen vom Typ ARRAY OF WORD in Frage, da lokale Variablen nach dem Ende einer Prozedur nicht mehr existieren.

Zum Anderen sollten Strings nicht nur durch WRITE(LN) auszugeben, sondern z.B. auch an LCD–Routinen als variable Parameter übergebbar sein.

Genau für diese beiden Fälle liefert die funktionale Systemprozedur `ADR` die Startadresse des jeweiligen Objektes an eine Variable vom Typ WORD.

Beispiele:

```
w1 := ADR(meinglobaleswordarray[1]); w2 := ADR(s);
```

Im ersten Fall muss dem Arraynamen der erste zu berücksichtigende Index wie gewohnt in eckiger Klammer übergeben werden. So können neben dem gesamten Array auch wechselnde Teile desselben als Puffer eingesetzt werden. Auch Records werden hierbei korrekt ausgewertet (soweit ich das überprüft habe).

Achtung: Bei der Verwendung von externem RAM: Dieses kann nicht als Puffer für PEC–Transfers verwendet werden. In diesem Fall wird das interne RAM in Segment 0 jedoch nicht vom Compiler verwendet und kann daher in eigener Verwaltung auch für solche Zwecke eingesetzt werden.

1.5.10 TRUNC und ABS

Dies sind die Transferfunktionen zwischen Daten der Typen INTEGER und WORD. Formal sind sie folgendermaßen deklariert:

```
PROCEDURE TRUNC(w: WORD): INTEGER;

PROCEDURE ABS(i: INTEGER): WORD;
```

TRUNC löscht einfach Bit 15 des Parameters, ABS berechnet das Zweierkomplement einer negativen Zahl, lässt sie unverändert wenn sie positiv ist. Der Wert des Parameters `w` bzw. `i` wird hierbei nicht verändert, da er zur Umwandlung in ein Register geladen wird.

1.5.11 ODD

Liefert TRUE, wenn ihr Parameter ungerade ist, sonst FALSE. Der Parameter kann vom Typ INTEGER oder WORD sein.

1.5.12 EINIT, SRST, DUMMIJMP

Dies sind Spezialitäten, die nur für den Bootstraploader benötigt werden. Sie tun im Prinzip genau das, was ihr Name ausdrückt. Wer das braucht, weiß wozu es gut ist.

1.6 Direkte Assemblereingabe

Da der Output des Compilers vom **AS** verarbeitet wird, ist diese leicht möglich, wobei natürlich die Regeln von **AS** eingehalten werden müssen.

Der Assemblertext wird zwischen `VERBATIM` und `END_VERBATIM` gesetzt, wobei darauf zu achten ist, dass dieser im Editor ganz am linken Rand eingegeben wird, damit der **AS** die Einrückungen, insbesondere garkeine für Labels, versteht. Beispiel für In– und Output:

```
PROCEDURE dummi;
BEGIN
VERBATIM
        Ob der AS
        das versteht?
END_VERBATIM;
END dummi;
```

```
dummi
; verbatim
        ob der AS
        das versteht?
; end_verbatim
        sub     CP, #32
        ret             ;dummi
```

Auf die strengen Vorschriften für die Einrückung kann verzichtet werden, wenn Label im Assemblertext mit einem Doppelpunkt abgeschlossen werden. Dann erkennt der **AS** diese auch, wenn sie nicht in der ersten Spalte beginnen.

Assemblercode lässt sich am besten in kurzen Prozeduren nutzen, wobei es sich anbietet, dass die ganze Prozedur nur aus Assembler besteht. Dann stehen alle 16 Register zur Verfügung, abzüglich der von Parametern belegten.

Label im Assemblercode müssen im ganzen Modul, sowie in ev. importierten Modulen eindeutig sein. Die vom Compiler erzeugten Label bestehen aus dem Modulnamen und einer angehängten fünstelligen Zahl $< 2^{15}$. Eine solche Konstruktion sollte daher im Assemblercode vermieden werden. Beispiel:

```
        PROCEDURE* Fstep(VAR step:WORD; cur:WORD);
        VAR bf:WORD;
BEGIN bf := step * 2; (* bf = R13, mal 2 wg. Word *)
        VERBATIM
        jmp Fstepnext
SHsteptable
        DW 224, 255, 7, 127, 96, 125, 5, 253
Fstepnext
        mov R12, #SHsteptable
        add R12, R13
        mov R13, [R12]        ; Wert nach bf laden
        END_VERBATIM
        IF cur = 2 THEN (* mittlerer Strom *)
          bf := bf ANDB %10111110;
        ELSIF cur = 1 THEN (* niedriger Strom *)
          bf := bf ANDB %11011011;
        ELSIF cur = 0 THEN (* Strom aus *)
          bf := bf ANDB %10011010;
        END;
        step := bf;
        END Fstep;
```

Sollen längere Assemblerroutinen eingebunden werden, bietet es sich an, dass die Verbatim–Umgebung nur eine Include–Direktive enthält.

Vorsicht ist beim Zugriff auf die SFR in dieser Umgebung geboten, da diese mit Hilfe des Datenseitenpointers DPP3 adressiert werden könnten. Für solche Zugriffe muss dieser 3 sein, was sich im Assemblercode durch Voransetzen des Befehls `extp` #3, #1 erreichen lässt. Mit dieser Anweisung werden die DPPx für einen Befehl ausgeblendet und stattdessen der nächste Befehl direkt auf die Speicherseite 3 ausgeführt.

2 Der Compiler

2.1 Aufruf

Dieser erfolgt über das Steuerprogramm obcc mit folgender Syntax:

```
obcc [ optionen ] datei
```

Zum Compilieren wird beim Aufruf nur der Name des Hauptmoduls ohne Suffix angegeben. Das .mod hängt die Initialisierung im Scanner automatisch an.

Momentan mögliche Optionen sind: `-psn, -rsn, -ren, -spn, -ssn, -cpn, -opn, -wsn, -wdn`. Ein Aufruf von obcc ohne Parameter oder mit `-h` listet die möglichen Optionen auf. Das "n" ist jeweils eine ganze Zahl, ist diese hexadezimal, muss unter Linux das Dollarzeichen mit dem Backslash escaped werden.

2.2 Kommandozeilenoptionen

IncludeRegister: (-ir"FileName") Lädt die Definitionen des konkreten Controllers aus der Datei "FileName". Ohne diese Option werden die Standarddefinitionen des C167CR verwendet (s. Abschnitt 2.2.1).

ProgStart: (-ps"n") Wohin das Programm geladen wird. Default ist 0, dann wird auch die Vektortabelle und die Grundkonfiguration in das erzeugte Programm eingefügt.

Weicht die Startadresse von 0 ab, wird weder die Konfiguration noch Vektoren in das Programm includiert. Die Konfiguration muss dann in der Anwendung erfolgen, sofern sie nicht durch z.B. einen Monitor durchgeführt wird. Dies ist vor allem bei Verwendung des Bootstraploaders zu beachten. Die angegebene Startadresse wird per `org` in den Assembleroutput eingefügt.

Bei Verwendung des BSL ist die abweichende Vorbelegung der Systemkonfiguration zu beachten, insbesondere `CP` und `SP`. Wird diese nicht angepasst, stehen nur sehr begrenzte Ressourcen zur Verfügung.

RamStart: (-rs"n") Beginn des Speicherbereichs, der für Variablen benutzt wird. Default ist $E000, der Anfang des XRAMS

RamEnd: (-re"n") Ende des für Variablen benutzten Speicherbereichs. Default ist $E7FE, das letzte Word im XRAM.

StackPointer: (-sp"n") Anfangsposition des Stacks, Default ist $FCE0, diese muss im ersten Speichersegment liegen.

ContextPointer: (-cp"n") Anfangsposition dieses Pointers, Default ist $F600, auch dieser muss im ersten Speichersegment liegen.

OPtimierungslevel: (-op"n") Hiermit können Optimierungen ein– und ausgeschaltet werden (s. Abschnitt 2.2.2). Default ist 0, d.h. keine Optimierungen aktiv.

DeBuglevel: (-db"n") Setzt ebendiesen fest, Default ist 0, d.h. kein Debugging (s. Abschnitt 2.2.3 und 3 ff.).

Schnittstelle für read und write (-as"n") Default ist 0, d.h. ASC0. Da der Debugger diese Schnitstelle benötigt, können diese Systemprozeduren auf eine bei manchen Derivaten vorhandene andere Schnittstelle (eg. ASC1) umgelenkt werden.

WaitStates: (-ws"n") Die Anzahl ebendieser für den externen Bus. Der C167 ist nach dem Reset auf 15 Waitstates vorkonfiguriert, diese sollten sich mit 70–ns–Speicher auf 1 reduzieren lassen. Die Wartezeit beträgt bei 20 MHz je 50 ns. Das C164–Board läuft problemlos mit 0 WS, ein Buszyklus dauert damit ca. 250 ns. Default ist 2 Waitstates.

WatchDog: (-wd"n") Default ist FALSE (n = 0), dann wird der Watchdog in der Grundkonfiguration disabled. Mit n = 1 kann er enabled werden.

UseUpper(-uu"n") Wandelt Buchstaben des Inputs in Großschreibung um. Gedacht für Anfänger, die sich nicht an die Unterscheidung von Groß– und Kleinschreibung in Oberon gewöhnen wollen. Default ist n=0, d.h. aus, mit n=1 wird diese aktiviert.

2.2.1 Definitionsdatei

Diese sind reine Textdateien und enthalten bisher die Definitionen für SFRs und Interrupts. Eventuell werde ich sie noch erweitern, z.B. um die verwendbaren Speichersegmente.

Die SFR–Definitionen habe ich nach ihrer Speicheradresse sortiert. Die Einträge enthalten je Zeile den Namen des Registers laut Datenblatt, ein Komma zur Abtrennung und die Speicheradresse. Insgesamt gibt es 4 Arten an SFRs: Die Standard–SFRs im Adressbereich von $0FE00 bis $0FFFF und die erweiterten SFRs im Bereich $0F000 ... $0F1FF sind der Standard des C167CR. Hinzu kommen die XBUS–Register im Bereich $0E800 ... $0EFFF, zu denen auch die CAN–Register des C167CR gehören, sowie die Flashregister, die außerhalb des ersten Speichersegmentes liegen.

Die ersten beiden Arten werden vom Compiler korrekt verwendet. Die XBUS–Register hält er dagegen noch für E–Register, was bei der Zuweisung von konstanten Werten

Probleme bereitet. Auf diese sollte daher vorerst nur über Variablen zugegriffen werden. Die Flashregister mit ihrer 16–MB–Adresse kann der Compiler noch nicht ansprechen, das muss also vorerst mit PUT bzw. GET erldigt werden. Natürlich werde ich diese Einschränkung gelegentlich aufheben und dann diesen Absatz löschen.

Das Ende der Registerdefinitionen erkennt der Compiler an der Zeile mit dem Eintrag "Vectors:". Anschließend folgen die Interruptdefinitionen, wobei wiederum pro Interrupt eine Zeile belegt ist. Diese enthält die Trapnummer, den Vektorname und die Vektoradresse getrennt durch Kommas. Das Ende dieser Liste wird am EOF erkannt.

2.2.2 Optimierungen

Diese sind zunächst einmal als experimentell zu betrachten, da ich nicht ausschließen kann, dass sie an manchen Stellen noch fehlerhaften Code erzeugen, Deshalb müssen sie über die Option **op** aktiviert werden, damit sie beim Verdacht, dass aus ihnen Fehler resultieren, leicht ausgeschaltet werden können.

Registeroptimierung I: Hiermit soll verhindert werden, dass die Werte von Variablen unnötig in Register geladen werden, wenn sie noch in einem Register vorhanden sind. Sie wird mit $n > 0$ aktiviert und ist standardmäßig ausgeschaltet. In langen Prozeduren reduziert sie den Code ganz schön, ist also einen Versuch wert.

Registeroptimierung II: Zur Unterstützung der 1. Registeroptimierung soll diese dafür sorgen, dass möglichst viele Register verwendet werden. Hierdurch wird seltener ein alter Wert überschrieben und die Chance, dass ein wiederbenötigter Wert noch in einem Register vorliegt, steigt. Diese wird mit $n > 1$ aktiviert, eine echte Optimierung hierdurch konnte ich aber bisher nicht feststellen. Ev. ist die Implementierung noch fehlerhaft, oder diese Optimierung macht sich nur in sehr umfangreichen Prozeduren bemerkbar.

2.2.3 Debugging

Levels:

0: kein Debugging

1: der Monitor wird an die Anwendung angehängt und aktiviert

2: zusätzliche Tests werden in den Code eingebaut

3: Kombination aus 1 und 2

2.3 Assemblierung

Der Output des Compilers wird mit dem Assembler **AS** von Alfred Arnold assembliert, wobei der Aufruf unter Linux **asl**, der unter Windows **asw** lautet. Als Prozessortyp wird `80C167` in den Compileroutput eingefügt, die Registerdefinitionen sind jetzt im Compiler integriert.

Als Standardoptionen für den **AS** verwende ich:

`-L -E -x`.

Damit wird ein Listing erzeugt, die Fehlermeldungen werden in ein Logfile geschrieben und sind etwas ausführlicher.

Nach der Assemblierung wird das erzeugte .p–File in das INTEL-HEX-Format übertragen, wobei die Tools des **AS** benutzt werden. Der entsprechende Aufruf lautet:

`p2hex -r \$-\$ name.p`.

2.4 Dateieinbindung

Solange der Modulimport noch fehlt, soll diese Einbindung auf Textbasis eine primitive Aufteilung des Codes ermöglichen. Getestete Prozeduren können in eine Extradatei ausgelagert und mit `INSERT` *Dateiname* ins Hauptmodul eingebunden werden. Die Einbindung ist auf einzelne Dateien beschränkt, die ihrerseits keine weiteren Dateien einbinden können. Die Dateiendung ist hierbei beliebig, der Dateiname wird wie eingegeben übernommen. Für die Syntaxhervorhebung in Kate habe ich die Endung `.obn` neben `.mod` zur Erkennung von Oberon–Dateien vorgesehen. Die Datei muss im aktuellen Verzeichnis liegen, alternativ kann der vollständige Pfad mitangegeben werden.

2.5 Vektoren

Diese werden vom Compiler automatisch nach Bedarf eingefügt. Da erst am Ende eines Moduls feststeht welche Interrupts benutzt werden, wird die Vektortabelle hinter dem Programmcode eingefügt. Ein entsprechendes `org` teilt dem **AS** mit wohin was gehört.

Der Reset springt zum Label `mainstart`, die anderen nichtmaskierbaren Interrupts löschen die zugehörigen Flags und kehren sofort mit `reti` zurück.

Unbenutzte maskierbare Interrupts springen nur mit `reti` zurück, jeweils ein `nop` bringt sie auf die erforderliche Länge von 4 Bytes. Das zugehörige Flag wird bei diesen automatisch gelöscht.

Wird ein Interrupt genutzt, so existiert ein zughöriger Handler, was beim Compilieren festgestellt wird. Für den zugehörigen Vektor wird ein `jmp` zum Label "Vektorname" eingefügt, die ISR hatte vorher schon dieses Label erhalten. Da nicht vorhersehbar ist, ob der **AS** einen 2– oder 4–Byte–Sprungbefehl verwendet, wird hinter den Sprung ein `org` eingefügt und alles hintendran hat wieder seine Ordnung.

2.6 Konfiguration des Controllers

Vor dem Start der Anwendung muss der Controller konfiguriert werden, wozu einige Register entsprechend beschrieben werden müssen. Dies ist der großen Flexibilität dieser Controller–Familie geschuldet.

Nur für Programme die an der Adresse 0 beginnen greift der Compiler in diese Konfiguration ein. Ansonsten, das betrifft eigentlich nur Anwendungen des Bootstraploaders, muss dies das Programm selbst übernehmen.

Einige dieser Einstellungen muss und kann der Compiler übernehmen, wenn über Kommandozeilenoptionen bei seinem Aufruf die eventuell zu ändernden Parameter angegeben werden. Andere Einstellungen müssen vor allem für die zweite Controllergeneration auf die verwendete Hardware angepasst sein, so dass eine Änderungsmöglichkeit für den Nutzer nötig ist

Vor allem diejenigen Einstellungen, die die verwendete Hardware betreffen, müssen innerhalb des Resets des Controllers erfolgen, die für den Compiler wichtigen Softwareeinstellungen können auch noch später vorgenommen werden. Der Reset des Controllers wird mit dem Befehl `EINIT` beendet. Für maximale Flexibilität habe ich eine Abfrage in das Steuerprogramm `obcc.pas` eingebaut, ob im aktuellen Verzeichnis eine Datei namens `Startup.asm` vorliegt und diese dann mittels `include` in den Compileroutput übernehmen lässt. Diese Assemblerdatei kann die gewünschte Konfiguration enthalten, auch das `EINIT` muss dann in ihr vorhanden sein. Fehlt diese Datei, wird die Konfiguration komplett vom Compiler erstellt, mit ihr können die Einstellungen des Compilers überschrieben werden.

Wie diese Konfiguration vom Compiler erstellt wird, möchte ich getrennt für Hard– und Softwareeinstellungen beschreiben. Zum Teil kann diese Konfiguration mit Optionen beim Aufruf des Compilers beeinflusst werden (s. Abschnitt 2.2).

2.6.1 Hardwareeinstellungen

Hier gibt es bisher drei Einstellungen, die der Compiler für Chips der zweiten Generation vornimmt.

Der Watchdogtimer kann nur im Reset ausgeschaltet werden, was der Compiler standardmäßig erledigt. Falls erwünscht ist, dass der Watchdog aktiv bleibt, kann dies dem Compiler mit der Option `-wd1` mitgeteilt werden, so dass hier wohl keine Aktivitäten am Compiler vorbei nötig sind.

Die Systemkonfiguration erfolgt über das Register SYSCON, das nur im Reset geändert werden kann. Der Wert dieses Registers wird in den wesentlichen Punkten hardwaremäßig durch entsprechende Pulldownwiderstände an den zugehörigen Pins des Port P0 vorgenommen. Hier muss der Compiler wenig ändern, lediglich die Aktivierung des internen XRAMs des Controllers nimmt er vor. Eine Möglichkeit zur Änderung wäre der Wunsch das Taktsignal des Controllers an Portpin P3.15 auszugeben, was durch Setzen des Bits 8 (ClkEn) im Register SYSCON erledigt werden kann. So etwas ist dann in `Startup.asm` möglich, da dieses nach der Initialisierung durch den Compiler eingebunden wird und daher dessen Einstellungen überschreiben kann.

Für Controller der 4. und 5. Generation sind hiermit weitere Einstellungen in mehreren Registern möglich, die ich bei Gelegenheit einbinden werde. Diese beinhalten z.B. die Konfiguration des Taktgenerators, die bei den früheren Chips noch per Hardware eingestellt werden muss.

Die Buskonfiguration ist nur für Controller mit externem Programmspeicher nötig, d.h. für Controller der 1. und 2, Generation. Die Wahl des jeweiligen Speichers am externen Bus können die Controller mit bis zu 5 Chipselectsignalen vornehmen, wobei die Anzahl der erforderlichen Selectsignale per Hardware (Pulldownwiderstände an P0) festgelegt wird. Für jedes benutzte Chipselectsignal x ($x = 0 \ldots 4$) wird ein Register namens BUSCONx verwendet, das die jeweiligen Hardwareeinstellungen für diesen Bereich enthält. Hinzu kommt für die Bereiche mit $x > 0$ jeweils ein Register mit Namen ADDRSELx, das die physikalische Adresslage des jeweiligen Speicherbereichs definiert. Solange keine anderen ADDRSEL–Register definiert sind, d.h. direkt nach dem Reset, bestimmt das Register BUSCON0 den Buszugriff, dessen Grundeinstellung daher per Hardware vorgenommen werden muss. Ist dabei etwas fehlerhaft, kann der Controller nicht auf seinen Programmspeicher zugreifen und dann geht garnichts mehr.

Die Buskonfiguration kann auch außerhalb des Resets geändert werden. Damit sie jedoch für BUSCON0 in den wichtigsten Punkten durch `Startup.asm` beeinflusst werden kann, erfolgt auch dessen Konfiguration durch den Compiler vor dem EINIT–Befehl.

Es gibt in BUSCON0 und seinen Kollegen Bits, die nicht per Hardware eingestellt werden und trotzdem an das jeweilige System angepasst werden sollten. Diese betreffen zum Einen die Waitstates, die beim Zugriff auf den Bus eingefügt werden, zusätzlich gibt es weitere die Performance betreffende Einstellungen. Die Waistates können vom Compiler eingestellt werden, ohne Änderung der Einstellung wird die maximale Anzahl derselben verwendet. Als Standard reduziert der Compiler die

Waitstates von 15 auf 2, die gewünschte Anzahl ist mit der Option `-ws` einstellbar, wobei die Zugriffsgeschwindigkeit der auf der jeweiligen Hardware eingesetzten Speicherbauteile die benötigte Anzahl festlegt.

Um die anderen Bits in BUSCON0 kümmert sich der Compiler bisher nicht. Da könnten, bei entsprechend schneller Hardware, sicher noch Optimierungen sinnvoll sein, die dann per `Startup.asm` eingefügt werden können .

Der Beginn der Konfiguration durch den Compiler sieht dementsprechend folgendermaßen aus, wobei das Label RESET durch den zugehörigen Vektor angesprungen wird:

```
RESET:
        diswdt                  ; watchdog off
; Set waitstates for BUSCON9:
        or      0FF0Ch, #13
        bset    0FF12h.2        ; SYSCON: enable XRAM
        or      0FF12h, #0E000h ; SYSCON: 1024 Bytes Stack
        einit
```

Die Buskonfiguration ist bei diesen Controllern enorm flexibel. Bei dem von mir zum Test verwendeten Minimodul–167 wird durch /CS0 = 0 die erste Flashspeicherbank aktiviert, was auch in Ordnung ist, da letztendlich das Anwendungsprogramm genau dort liegen soll. Während der Entwicklung geht es aber bedeutend schneller das Programm in RAM zu laden und von dort zu starten. Da das Testprogramm mittels Bootstraploader geladen wird, kann die Buskonfiguration in diesem entsprechend geändert werden, so dass keinerlei Hardwareänderungen, wie Umstecken von Jumpern o.ä., nötig sind. Hierzu dient die folgende Konfiguration im Bootstraploader:

```
        mov     SYSCON, #0004h          ; enable XRAM
        jmps    next                    ; dummi jump
next:   mov     DPP0, #0
        mov     DPP1, #1
        mov     DPP2, #2
        mov     DPP3, #3
; RAM mit /CS1 aktiv schalten, 256 kB ab Adresse 0000:
        mov     ADDRSEL1,#0006
; aktiv schalten mit 1 Waitstate und ReadWriteDelay:
        mov     BUSCON1, #04AEh         ;

        jmps    0,0                     ; Start der Anwendung
```

Es wird hierbei die erste Ramspeicherbank als Programmspeicher genutzt, die mittels /CS1 = 0 aktiviert wird. Für /CS1 sind die beiden Register BUSCON1 und ADDRSEL1

zuständig und müssen im BSL entsprechend konfiguriert werden. Zunächst wird jedoch SYSCON konfiguriert, wobei das interne ROM des Controllers abgeschaltet und gleichzeitig das XRAM eingeschaltet wird. Der Dummi–Sprung mittels der JMPS–Anweisung ist obligatorisch, erst durch ihn wird die Abschaltung des internen ROMs tatsächlich aktiv. Anschließend werden die Datenspeicherzeiger DP0 ... DP3 so initialisiert, dass für Datentransfers ein 64 kByte langer Block ab Adresse $000000 zur Verfügung steht. Die letzten beiden Anweisungen bewirken das Ummappen des Speichers. Zuerst wird ADDRSEL1 konfiguriert. In seinem höherwertigen Teil steht der Anfang des Speicherbereichs für den dieses Register zuständig sein soll, in diesem Fall ist das 0. Seine niederwertigsten 4 Bits geben die Länge des Speicherbereiches an, die 6 hierin bedeutet 256 kByte. Damit ist der Controller so konfiguriert, dass er für jeden Speicherzugriff im Bereich der ersten 256 kBytes /CS1 aktiviert, d.h. auf 0 setzt. Zusätzlich muss dieser Speicher mittels BUSCON1 aktiviert und die für ihn sinnvollen Hardwareeinstellungen, wie z.B. die Zahl der Waitstates usw., hierin eingestellt werden.

Im Anschluss an dieses Ummappen darf das Anwendungsprogramm nach seinem Laden nicht durch einen Softwarereset gestartet werden, da dieser die Konfiguration wieder überschreiben würde. Sein Start wird daher durch einen Sprung auf Adresse $000000 vorgenommen.

2.6.2 Softwareeinstellungen

Diese betreffen die Benutzung des RAMs, wofür vier Bereiche benötigt werden.

Als Erstes ist der Anfangswert für den Stackpointer SP festzulegen, wobei beachtet werden muss, dass die seine Grenzen überwachenden Funktionsregister STKUN, für die obere, und STKOV für die untere Grenze vorher gesetzt werden. Sonst wird bei der Änderung des SP sofort ein Trap der Klasse A ausgelöst.

Auch für die Lage der allgemein verwendbaren Register, der GPRs, muss ein Startwert angegeben werden, welcher in das Funktionsregister CP, den Kontextpointer, geschrieben wird.

Weiter geht es mit dem Speicherplatz für globale Variablen, der hier jedoch nicht berücksichtigt werden muss, da er nur über die Compilervariable `ramstart` definiert wird.

Als Letztes wird der Startwert für den Frame, den Softstack der für Proceduren genutzt werden kann, festgelegt. Dieser Frame wird über den FSP, der immer im GPR R0 verfügbar ist, verwaltet.

```
mov     0FE16h, #0FCE0h ; STKUN
mov     0FE14h, #0F8E0h ; STKOV
mov     0FE12h, #0FCE0h ; SP
mov     0FE10h, #0F600h ; CP
nop
```

```
        mov     R0, #0E7FEh     ; FSP = top of softstack
; End Of INIT
```

Für Controller der 2. Generation (C16X) wird für diese vier Bereiche bisher das interne IRAM, für Stack und GPRs, sowie das XRAM, für globale Variablen und den Frame, mit einer Größe von je 2 kBytes genutzt. Zusätzlich besteht die Möglichkeit anstelle des XRAMs einen größeren externen Speicher zu verwenden, so dass mehr Variablenspeicher zur Verfügung steht. Dies ist mit den Optionen **rs** und **re** möglich.

Die Controller der 1. Generation (80C166) verfügen lediglich über das IRAM in Größe von einem Kilobyte, das für Stack und GPRs genutzt werden muss. Als Variablenspeicher muss bei diesen externes RAM verwendet werden, was genauso erfolgen kann wie bei den C16X.

Bei den jüngeren Generationen (XC16X und XE16X) tritt das DPRAM anstelle des IRAMs und das DSRAM anstelle des XRAMS. Außer den Namen und der Zugriffsgeschwindigkeit ändert sich dadurch zunächst nichts, so dass diesbezüglich kaum Änderungen durch den Compiler nötig werden. Lediglich die Adressen des Variablenspeichers müssen angepasst werden, wobei gleichzeitig die Größe dieses Bereiches (4...16 kBytes) eingestellt wird. Diese Controller bringen zusätzlich einen dritten Speicherbereich, das PSRAM mit, dessen Größe typabhängig zwischen 2 kB und 16 kB, neuerdings bis zu 64kB, reicht. Dieser Bereich könnte alternativ als Variablenspeicher verwendet werden, wobei dann der Stack in das DSRAM wandern könnte. Auch diese Änderung ist mit den Optionen **rs**, **re** und **sp** möglich.

Grundsätzlich bin ich der Meinung, dass für diese Konfiguration der Compiler zuständig bleiben sollte. Mithilfe der Optionen können alle Einstellungen durch den Anwender beliebig manipuliert werden, so dass zusätzliche Eingriffsmöglichkeiten unnötig sind.

2.7 Die Speicherverwendung

Die meisten Einstellungen hierzu erfolgen bei der Grundkonfiguration des Kontrollers und können daher leicht angepasst werden. Wie solche Änderungen vorgenommen werden können, ist in den Abschnitten 2.6 und 2.2 beschrieben.

2.7.1 Der Programmspeicher

Der Compiler erzeugt seinen Code zunächst einmal im Codesegment 0, wobei jedes dieser Segmente bei der C16X–Familie einen Umfang von 64 kByte aufweist. Beim C167 ist dieses Segment 0 jedoch um das interne RAM reduziert. Hierfür sind 16 kBytes reserviert, was eine maximale Codegröße von 48 kByte ermöglicht. Neuere Derivate begrenzen den

ersten Sektor auf 32 kBytes, das wäre also z.B. für die ST10F* die momentane Codegrenze. Das ist erstmal nicht schlecht, da der Compiler sehr kompakten Code generiert. Die meisten Systeme mit dem C167 sind jedoch mit deutlich größerem Programmspeicher ausgestattet, der damit nicht genutzt werden kann. Es wäre relativ einfach möglich, über einen Kommandozeilenparameter eine Umbelegung des Codes nach der Vektortabelle in das Codesegment 1 zu ermöglichen, womit die maximale Codelänge erstmal auf 64 kBytes erweitert wird.

Bei den XC16X und ihren Nachfolgern liegt der interne Programmspeicher (Flash) im Codesegment 0C0h und folgenden. Hier gibt es keine Überschneidung mit dem internen RAM, so dass hierfür 64 kBytes verfügbar sind.

Über das erste Codesegment hinaus kann der Compiler momentan keinen Code erzeugen. Es wäre hierzu notwendig ab einer Grenze die folgenden Prozeduren in ein neues Segment zu schreiben. Hierzu wurde im Generator der Codelängenzähler `cl` implementiert, der für jeden erzeugten Assemblerbefehl entsprechend inkrementiert wird. Sein Endwert liegt typischerweise ca. 5 % über der tatsächlich erzeugten Codelänge, was im Wesentlichen den nichtberücksichtigten Optimierungen des Assemblers AS geschuldet ist. Allerdings will ich auch nicht ausschließen, dass einige Summandan noch schlicht falsch sind. So exakt muss dieser Zähler allerdings garnicht sein, da ich zu seiner Benutzung ja von Systemen mit sehr viel Flash ausgehe. Zum Einsatz müsste nun eine Segmentgrenze festgelegt werden, die zum Ende jeder Prozedur überprüft wird, und bei deren Überschreitung in das folgende Segment gewechselt wird. Dies ist mit einer ORG–Direktive für den Assembler leicht zu realisieren, allerdings wird hierzu gerade für die ST10F*–Derivate eine kleine Segmentverwaltung nötig, da bei diesen der Flash sehr seltsam aufgeteilt ist.

Zusätzlich müssen alle CALL–Befehle für Prozeduren, die Einträge der Interrupttabelle und die Rücksprünge auf Segmentbefehle geändert werden, was aber lediglich drei Stellen im Generator betrifft. Die Codelänge würde hierdurch kaum steigen und um den Rest kümmert sich freundlicherweise der Assembler AS.

2.7.2 Registerbänke und Stack

Beim C167 müssen der Stack und die GPRs im internen RAM, dem sog. IRAM, mit einer Größe von 2 kByte untergebracht werden. Bei den Nachfolgern tritt das sog. DPRAM an diese Stelle, das ebenfalls 2 kByte groß ist. Die Startadresse dieses Bereiches liegt in beiden Fällen bei 0F600h, sein Ende dann bei 0FDFEh. Am Ende dieses Bereiches liegen 256 Bytes bitadressierbares RAM, das ich bisher frei gehalten habe. Direkt darunter sind 32 Bytes ab 0FCE0h für die PEC–Vektoren reserviert. Diese müssen freigehalten werden um PEC–Transfers zu ermöglichen. Damit fehlen vom IRAM 288 Bytes, wobei der bitadressierbare Bereich bisher völlig ungenutzt bleibt. Eventuell wird dieser später einmal für Rückgabewerte von Funktionsprozeduren o.ä. genutzt. Es bleiben also 1760 Bytes im IRAM, die für den Stack und die Registerbänke genutzt werden können.

Von unten wird dieser Speicher von den GPRs genutzt, indem der Kontextpointer CP beim Programmstart auf 0F600h gesetzt wird. Mit jedem Prozeduraufruf, auch Interrupts zählen hierzu, wird dieser CP um 32 inkrementiert.

Gleichzeitig wird dieser Bereich von oben für den Stack benutzt, indem der Stackpointer SP zum Programmstart auf das Ende dieses Bereiches, also auf 0FCE0h, initialisiert und vor jeder Benutzung um 2 dekrementiert wird.

Während des Programmablaufs laufen diese beiden Zeiger CP und SP aufeinander zu, ein Crash ist hierbei nicht auszuschließen und hätte fatale Folgen.

1760 Bytes klingen erstmal nicht üppig, aber dieser Speicher wird vom Compiler recht sparsam verwendet. Pro Prozeduraufruf wird jeweils ein Registersatz per CP benötigt, auf dem Stack wird die Rückkehradresse und der Codesegmentpointer CSP, bei Interrupts zusätzlich das PSW, abgelegt. Insgesamt werden also maximal 38 Bytes pro Prozeduraufruf benötigt, so dass 46 verschachtelte Prozeduraufrufe bzw. Interrupts möglich sind. Es sind daher nur bei unbedachten rekursiven Prozeduraufrufen Probleme zu erwarten.

Mit Debuglevel größer 1 wird u.a. ein Test bezüglich einer Kollision bei jedem Unterprgrammaufruf in den Code integriert. Dieser Test erzeugt im Falle einer Kollision einen Stacküberlaufinterrupt (s. Abschnitt 3 ff.).

2.7.3 Der Variablenspeicher

Beim C167 wird hierfür das interne XRAM von 0E000h bis 0E7FFh mit einer Länge von 2 kByte genutzt. Für die XC16X sollte hierfür das DSRAM ab 0C000h geutzt werden, das heute meist 4 kByte bietet, ebenso für die XE16X, bei denen das DSRAM ab Adresse 0A000h zu finden ist Auch dieser Speicher wird mehrfach genutzt.

Global definierte Variablen werden vom Anfang dieses Speicherbereichs aufsteigend angelegt. Der Compiler kennt ihre absolute Adresse, wodurch der Zugriff auf sie sehr schnell erfolgen kann. Das erste Word des globalen Variablenspeichers belegt grundsätzlich die Ergebnisvariable _FRESULT für Funktionsprozeduren, das zweite der Zeiger auf den freien Heapbereich _HEAPTOP.

Vom Ende her absteigend wird derselbe Speicherbereich für den Frame benutzt, auf dem bisher nur die lokalen Variablen langer Prozeduren angelegt werden. Der Frame ist ein Softstack, d.h. die einzelnen Variablen werden relativ zum in R0 verwalteten Framestartpointer FSP adressiert. Hierfür ist natürlich ein größerer Aufwand nötig, weswegen die langen Prozeduren langsamer und umfangreicher sind als kurze. Bei den XE16X könnte hierfür auch das PSRAM ab 0E00000h genutzt werden, das heute meist mehr Speicher bietet.

Zwischen diese beiden Bereiche schiebt sich ein Puffer direkt hinter den globalen Variablen. Dieser wird bei der Ausgabe von Zahlen mittels Write oder Writeln genutzt, wozu

nur wenige Bytes benötigt werden. Momentan habe ich hierfür 16 Bytes reserviert, was recht üppig ist. Der Rest des Rams bis zum Frame wird als Heap für Pointervariablen genutzt. Eine Kollision von Heap und Frame wird bisher nicht erkannt, ich werde aber einen Test für die relevanten Debuglevel hinzufügen bzw. diesen entsprechend ändern.

Im kleinsten Fall bietet dieser Speicher Platz für gut 1000 Variablen, was in den meisten Fällen ausreichen sollte. Es gibt aber keinerlei Kontrolle, ob dieser Speicherbereich vom Programm überschritten wird, darauf muss man also selbst achten. Eine diesbezügliche Kontrolle müsste ja zur Laufzeit erfolgen und würde damit die Performance beeinträchtigen. In der Praxis dürften Probleme hiermit jedoch nur auftreten, wenn größere ARRAYs angelegt werden, in diesem Fall sollte man schon nachrechnen.

Wie im Abschnitt 2.2 schon erläutert wurde, ist es möglich für größere Datenmengen externes RAM nutzbar zu machen, was aus Sicht des Compilers genauso erfolgt wie die Verwendung von PSRAM. Um das Vorgehen hierbei verständlich und die hierdurch eventuell auftretenden Probleme deutlich zu machen, muss man sich etwas genauer anschauen, wie diese Controller den Speicher adressieren.

Mit seinen 16–Bit–Registern kann der Controller zunächst nur Adressen innerhalb eines Blockes von 64 kByte ansprechen. Um diese Grenze zu knacken, haben die Entwickler einen Pagingmechanismus für den Datenspeicher eingeführt. Der gesamte Speicher wird hierzu in Seiten (Pages) von je 16 kByte aufgeteilt. Für jede Seite innerhalb eines 64–kB–Blockes ist ein Speicherseitenzeiger DPPx zuständig und die beiden höchstwertigen Bits der Speicheradresse entscheiden darüber, welcher Seitenzeiger zuständig ist. Für Adressen unterhalb 04000h sind diese beiden Bits 0, so dass DPP0 benutzt wird, für Adressen oberhalb 0BFFFh sind diese Bits 1, was bedeutet hier muss DPP3 verwendet werden. Dazwischen liegen noch die Bereiche für DPP1 und und DPP2.

Diese Zeiger werden in den gleichnamigen Funktionsregistern gehalten, dessen jeweiliger Inhalt vor die verbleibenden 14 Bits der Adressangabe gesetzt wird, wodurch die physikalische Adresse des Speichers gebildet wird. Dieser Mechanismus greift sowohl für die direkte Adressierung, als auch für die indirekte durch Register.

Nach dem Reset des Controllers sind diese Seitenzeiger so initialisiert, dass die ersten 64 kByte im Gesamtspeicher adressiert werden, d.h. DPP0 = 0 . . . DPP3 = 3. Der Compiler spricht standardmäßig nur internen Speicher des Controllers an, der bei den C16X im Bereich 0E000h . . . 0FFFEh liegt. Zur Adressierung dieses internen Speichers wird also lediglich der Seitenzeiger DPP3 mit seinem Initialisierungswert 3 benötigt und dies gilt sowohl für den Zugriff auf Variablenspeicher wie auch den auf Funktionsregister SFR. Hierfür sind also nach dem Reset keinerlei Änderungen nötig.

Wird mit den Compileroptionen **rs** und **re** der Rambereich verschoben, setzt das Steuerprogramm `obcc` die Generatorvariable `rammapped`, worauf im Generator die Prozedur `mapram` aufgerufen wird. Diese berechnet aus der Start– und Endadresse des angegebenen Bereichs die zur Adressierung nötigen Seitenzeiger und schreibt in den Assemblercode die Anweisungen um alle vier Seitenzeiger entsprechend zu setzen. Zusätzlich wird hierbei mittels der ASSUME-Direktive auch dem Assembler diese Belegung mitgeteilt. Die

Adressen verarbeitet der Compiler in voller Länge und gibt sie so an den Assembler weiter. Dieser kürzt die Adressangaben entsprechend der Vorgabe auf 14 Bit und kontrolliert, ob die zugehörigen Seitenzeiger passend eingerichtet sind. Ist dies nicht der Fall, erzeugt er eine Warnung. Damit ist der Zugriff auf den Speicher für Variablenzugriffe eindeutig und fehlerfrei gewährleistet.

Normalerweise beginnt der Speicherbereich für RAM immer auf einer 64–kB–Segmentgrenze, so dass der Wert des DPP0 ohne Rest durch 4 teilbar ist. Dies ist für externes RAM sehr wahrscheinlich, da normalerweise 64–kB–Bauteile oder solche, die ganzzahlige Vielfache davon bieten, eingesetzt werden. Auch für das PSRAM der 4. und 5. Controllergeneration ist diese Annahme zutreffend. Damit werden von Anfang bis zum Ende des RAMs die Seitenzeiger DPP0 bis DPP3 in aufsteigender Folge zur Adressierung des Variablenspeichers verwendet.

Aber auch der Zugriff auf die Funktionsregister SFR erfolgt meist über ihre Speicheradresse. Diese liegen immer auf der Speicherseite 3 und für den Zugriff auf sie wird dementsprechend DPP3 benutzt. Ist dieses jedoch durch Umbelegung des Rambereichs umgebogen, würden Zugriffe auf die SFR ins Leere laufen. Der Compiler erkennt Zugriffe auf die SFR und kann dabei überprüfen, ob DPP3 $\neq$ 3 ist. Ist dies der Fall, muss er eingreifen um dieses Problem zu umgehen. Hierzu fügt er den Befehl "`EXTP #3, #1`" vor Zugriffe auf die Funktionsregister ein. Mit diesem Befehl werden die Seitenzeigerregister umgangen und für eine geringe Anzahl an Anweisungen, die Anzahl wird durch den 2. Parameter angegeben, die konstante Seitennummer, die als erstes Argument aufgeführt ist, also 3, verwendet.

Damit sollte eigentlich alles erwartungsgemäß funktionieren. allerdings blähen die extp–Anweisungen den Code nicht unerheblich auf. Da können leicht bis zu 10 Prozent an zusätzlichem Code zustande kommen, der meistens aber völlig unnötig ist. Benötigt man nicht die vollen 64 kByte an RAM, kann man einfach dessen obere Grenze mit der Compileroption **re** auf 48 kByte festlegen. Damit wird DPP3 für den RAM–Zugriff nicht benötigt und bleibt daher auf dem Wert 3, wodurch für Zugriffe auf die SFR keine zusätzlichen Maßnahmen nötig werden, der zusätzliche Code also vermieden wird.

Ein Beispiel soll dieses verdeutlichen. Auf meinem C164–Testboard möchte ich das externe RAM im 4. Segment als Variablenspeicher nutzen, davon jedoch nur 48 kBytes, da ich mir keine EXTP–Strafbefehle einhandeln möchte. Dieses erreiche ich mit dem Compileraufruf für das Programm `tincg`:

`obcc` -rs\$30000 -re\$3BFFE tincg

Der Compiler erzeugt damit folgende Initialisierung:

```
RESET:
        diswdt                  ; watchdog off
        or      BUSCON0, #13    ; set waitstates
        bset    SYSCON.2        ; enable XRAM
```

```
        einit                   ; end of reset
        mov     STKUN, #0FCE0h
        mov     STKOV, #0F8E0h
        mov     SP, #0FCE0h
        mov     CP, #0F600h
        nop
        mov     R0, #0BFFEh     ; FSP = top of softstack
        mov     DPP0, #12
        mov     DPP1, #13
        mov     DPP2, #14
        mov     DPP3, #3
        assume DPP0:12, DPP1:13, DPP2:14, DPP3:3
        jmp     mainstart
; End Of Init
```

Mehr als 64 kByte als Variablenspeicher wären nur sehr schwer zu realisieren, da hierfür im laufenden Programm die Seitenzeiger umdefiniert werden müssten. Weitere RAM–Bereiche können jedoch für selbstverwaltete Buffer verwendet werden, auf die mit `GET` und `PUT` zugegriffen wird. Diesen Inlineprozeduren wird auch die Segmentnummer als Parameter übergeben und in der Umsetzung wird immer eine EXTS–Instruktion eingefügt.

3 Der Debugger obcdbg

3.1 Überblick

Das PC–Programm **obcdbg** dient einerseits dazu das erstellte Programm auf den Controller zu übertragen, andererseits wurden auch Debuggingmöglichkeiten implemetierrt. Die Funktion beruht auf dem Bootstraploader (BSL) der auf allen C16x–Controllern implementiert ist und über die serielle Schnittstelle bedient wird.

Um die Anwendung auf das Controllerboard überspielen zu können, muss man natürlich die jeweilige Hardwareausstattung des Boards berücksichtigen, gerade in dieser Beziehung sind die C16x–Derivate ja sehr flexibel. Dies ist jedoch leicht möglich, da die PC–Anwendung nur die Daten im korrekten Protokoll übertragen muss, die zweite Stufe des BSL, eine kleine Anwendung die mit übertragen wird, kümmert sich auf Controllerseite um das Speichern der Daten. Diese zweite Stufe des BSL muss daher jeweils exakt für das jeweilige Board angepasst werden, wofür ich natürlich nur Beispiele liefern kann.

Die Datenübertragung vom PC zum Cotroller erfolgt mit 38400 Baud (8N1), der für den C167 maximalen Rate. Für neuere Derivate muss vermutlich eine Baudrateumschaltung im Verlauf der Übertragung erfolgen, da diese vom Start weg wohl nur 19200 Baud vertragen, nach Start der PLL jedoch mehr. Die gesamte Datenübertragung erfolgt binär, um die Übertragungszeit möglichst kurz zu halten.

Die PC–Anwendung wurde mit Lazarus erstellt und sollte damit auf allen von dieser Umgebung unterstützten Plattformen übersetzbar sein, wenn die jeweilige Debuggschnittstelle darunter verwendbar ist. Im Moment ist diese sicher nur als Machbarkeitsstudie zu betrachten, die ich nach und nach erweitern werde. Es macht aber schon Spaß, damit einer Anwendung unter die Haube zu schauen.

3.2 Debugging

Die Funktion des Debuggers beruht darauf, dass an das Ende der zu debuggenden Anwendung ein Monitorprogramm mit einer Länge von ca. 1 kByte angehängt wird. Dieser Monitor wird nach dem Reset des Kontrollers aufgerufen und die Interruptvektoren für den NMI (NMITRAP), für den Empfangsinterrupt der ser. Schnittstelle ASC0 (S0RINT) und den Interrupt für undefinierten Opcode (BTRAP) werden vom Compiler auf den Anfang dieses Monitors umgelegt. Da der Monitor hinter die eigentlichr

Anwendung geladen wird, ist sichergestellt, dass diese mit oder ohne Monitor im selben Speicherbereich ausgeführt wird.

Der Monitor kann dann die eigentliche Anwendung starten, Haltepunkte setzen oder löschen und ähnliche Aktionen ausführen. Das PC–Programm steuert diesen Monitor über wenige, sehr primitive Anweisungen, wobei die Kommunikation zwischen diesen beiden bisher über die ser. Schnittstelle erfolgt.

Der Monitor wurde natürlich in Oberon geschrieben, wobei relativ viel Assembler in der Verbatim–Umgebung benötigt wurde.

3.3 Debuglevel

Mittels der Option `-dbn` kann der Level n des Debugging eingestellt werden. Die Werte von n haben folgende Konsequenz:

1: Der erzeugte Code bleibt (fast) unverändert, es wird ein Monitorprogramm an den Code angehängt und nach dem Reset aufgerufen. Die Anwendung kann unter diesem Monitor gestartet werden.

2: Zusätzliche Tests werden in den Code eingebaut, die bei Versagen einen Interrupt erzeugen. Diese betreffen:

Stackkollision: Bei jedem Aufruf eines Unterprogramms oder Interrupts wird geprüft, ob der Kontextpointer (CP) mit dem Stackpointer (SP) zu kollidieren droht. Wenn der Abstand beider unter 46 Bytes beträgt wird Flag `STKOF` gesetzt und damit ein `STOTRAP` ausgelöst.

Arrayindex ungültig: Jeder Zugriff auf ein Arrayelement wird auf einen ungültigen Index überprüft. Ist dieses gegeben, wird Flag `ILLOPA` gesetzt und damit ein `BTRAP` ausgelöst.

Arithmetischer Überlauf: Bei allen arithmetischen Operationen wird das Ergebnis auf Überlauf o.ä. geprüft. Ist ein solcher gegeben, wird das Flag `ILLINA` gesetzt und damit ein `BTRAP` ausgelöst.

In diesem Level kann der Mikrocontroller ohne Monitorprogramm, also z.B. direkt in der Anwendung, überwacht werden. Es müssen lediglich Serviceroutinen für die jeweiligen Interrupts im Programm vorgesehen werden, die jeweils die Fehlerursache anzeigen (z.B. auf einm LCD) und den Kontroller in den IDLE–Modus versetzen.

3: Die Zusatztests und der Monitor werden in das Programm integriert, der Monitor bedient die Interrupts.

Zusätzlich wird bei allen Leveln größer 0 die Startadresse des Hauptmoduls als Trapvektor an Adresse $01FE eingefügt. Dies ist der letzte Usertrap und er wird von keinem Controller als Interrupt verwendet. Von dieser Adresse kann die Startadresse leicht vom Monitor ausgelesen werden.

3.4 Einschränkungen

Da der C167 und andere Kontroller dieser 2. Generation der Familie über keine echte Debug–Schnittstelle verfügen, muss zum Debuggen deren serielle Schnittstelle ASC0 verwendet werden. Dies bringt zwei Einschränkungen mit sich:

1. Diese ser. Schnittstelle kann in der zu debuggenden Anwendung nicht verwendet werden. Nachrichten, die über sie gesendet werden, laufen ins Leere und die zugehörigen Antworten können nicht übermittelt werden. Auch eine Initialisierung dieser Schnittstelle kann das Debuggen erschweren, wird z.B. die Baudrate darin geändert, versteht der Kontroller das Break des Debuggers nicht mehr und kann daher nicht darauf reagieren. Dann hilft nur noch ein NMI zum Aufruf des Monitors.

   Konsequenz: Alle Zugriffe auf diese ser. Schnittstelle müssen auskommentiert werden, ebenso die Initialisierung dieser Schnittstelle, sofern diese nicht ebenso konfiguriert wird, wie im Debugger (38400 8N1).

   Modernere Mitglieder dieser Kontrollerfamilie verfügen über eine zusätzliche serielle Schnittstelle ASC1 und diese kann natürlich uneingeschränkt genutzt werden, z.B. mit einem Terminalprogramm. Hierzu wurde die Compileroption `-asn` eingeführt, die Ein– und Ausgabe entsprechend umleitet. Der Debugger selbst kann nicht auf eine andere Schnittstelle umgelegt werden, da der Bootstraploader nur mit ASC0 funktioniert.

2. Um ein Programm mittels Haltepunkten unterbrechen zu können, muss der Debugger "ungültige Opcodes" in das Programm einfügen. Dies funktioniert natürlich nur wenn das Programm in RAM liegt, einzelne Zellen des Flash–Programmspeichers können nicht überschrieben werden.

   Um auch Programme im Flash rudimentär debuggen zu können, wurde die Möglichkeit implementiert im Quelltext Breakpoints einzufügen. Für diese fügt der Compiler bei entsprechendem Debugglevel ungültige Opcodes in den Assembleroutput ein, wodurch der Monitor aufgerufen wird. In diesem Fall muss der Monitor durch Stackmanipulation eine Weiterführung der Anwendung ermöglichen. Auch eine aus dem Flash laufende Anwendung kann über die serielle Schnittstelle unterbrochen werden.

Wer angesichts dieser Einschränkung die Nase rümpft sollte bedenken, dass auch die Profis von Keil und Tasking mit diesem Problem konfrontiert waren und keine bessere Lösung gefunden haben. Diese wird erst mit den Nachfolgern der Familie, den XC16x

und XE16x möglich, die über eine JTAG–kompatible Schnittstelle zum Debuggen verfügen. Die Kommunikation des Monitors mit dem Debugger wird auf dem PC über eine Zwischenschicht realisiert, die zukünftige Verbesserungen in dieser Beziehung zulässt.

3.5 Vorbereitungen

Bootstraploader: Dieser lädt die Anwendung in drei Schritten in den Programmspeicher. Die erste Stufe des Laders, ein exakt 32 Byte langes Assemblerprogramm, ist als String in **obcdbg** integriert. Es lädt die zweite Stufe des BSL in das XRAM, so dass diese max. 2 kBytes lang sein kann. Diese Stufe habe ich in Assembler geschrieben und sie sieht folgendermaßen aus:

```
; Flash-Loader ins XRAM laden:
; Das Ende von Stage2 an Byteposition 19 (LowByte) und 20
;  (Higbyte)  muss entsprechend ihrer Laenge angepasst werden.
        cpu 80c167
        include reg166.inc
        org     0FA40h          ;
;
        bset    SYSCON.2        ; enable XRAM
        mov     R0, #0E000h     ; Start of XRAM
loop:
        jnb     S0RIR, loop
        bclr    S0RIR
        movb    [R0], S0RBUF
        cmpi1   R0, #0E1A3h     ; Endwert muss angepasst werden!
        jmp     NE, loop
        nop                     ; 32 Bytes auffüllen
        nop
        nop
        jmp     stage2

        org     0E000h          ; Start of XRAM
stage2:
END
```

Die Filelänge der zweiten Stufe fragt **obcdbg** beim Betriebssystem nach und passt die Endadresse im String der 1. Stufe entsprechend an.

Die zweite Stufe des Laders muss die Buskonfiguration vornehmen und deshalb auf die jeweilige Hardware angepasst werden. Ein Beispiel mit Buildskript lege ich den Compilerquellen bei. Diese wird als **stage2.bin** kompiliert und assembliert, wobei auf die nötigen Compileroptionen zu achten ist (s. Buildskript).

Monitor: Das File monitor.mod wird mit der Option `-im1` kompiliert. Dabei entsteht ein Assemblerfile **monitor.asm** ohne Startupcode, Interruptvektoren und solche Sachen. Zusätzlich wird durch die Option eine Symboldatei dbgmonitor.sym erzeugt, die hier jedoch nicht benötigt wird und gelöscht werden kann.

Für Programme im RAM und im Flash wird je ein eigener Monitor benötigt, da die Behandlung von Haltepunkten unterschiedlich ist. Es muss also die jweilige Version als **monitor.asm** in das aktuelle Arbeitsverzeichnis kopiert werden.

Anwendung: Die hat natürlich ihren eigenen Namen, ich nenne sie einfach anwendung. Sie muss mit der Compileroption `-db1/3` kompiliert werden, wodurch der Monitor eingebunden wird. Zusätzlich werden die Zeilennummern des Quelltextes und die Startadresse der Anwendung als Kommentare in die erzeugte Assemblerdatei geschrieben. Bei der Assemblierung erstellt der Assembler ein Listing, das dann diese Zeilennummern und die zugehörigen Codeadressen beinhaltet. Nach der Assemblierung muss die erzeugte Ausgabedatei in das binäre Format umgewandelt werden, damit der Bootstraploader sie übertragen kann. Die für den ganzen Vorgang nötigen Schritte und alle Optionen können den Buildskripten meiner Beispiele entnommen werden. In verkürzter Version:

```
obcc -db1 anwendung
asl -L -E -x anwendung.asm
p2bin -r \$-\$ anwendung.p
```

Unter Windows heißt der Assembler asw statt asl und beim p2hex müssen die Backslashes vor den Dollarzeichen weggelassen werden. Sonst geht das genauso.

Letztendlich resultieren drei Dateien, die vom Debugger benötigt werden: **anwendung.mod**, **anwendung.bin** und **anwendung.lst**.

3.6 Das Transferprotokoll

Die Übertragung erfolgt in sechs Schritten:

1. Die PC–Anwendung sendet ein Nullbyte, an dessen Länge erkennt der Controller die verwendete Baudrate und sendet ein Byte als Controller–Kennung zurück. Die Kennung wird von meiner Anwendung nicht ausgewertet, sie wird nur zur Information in der Statuszeile angezeigt.
2. Die PC–Anwendung sendet die erste Stufe des BSL, die exakt 32 Bytes lang ist. Diese wird vom Controller an vorgegebener Stelle des IRAMs abgelegt und angesprungen. Sie enabled das XRAM, lädt die zweite Stufe des BSL ab dessen Anfang und springt diese an.
3. Die zweite Stufe des BSL sendet das Zeichen “R” zum Start, erledigt die Buskonfiguration u.ä. und startet dann den eigentlichen Download.

4. Die PC–Anwendung sendet die Sektornummer als ein Byte für den 64–kByte–Block in den die Anwendung geladen werden soll. Ist der verwendete Sektor als Flash implementiert, muss die zweite Stufe zunächst diesen Sektor löschen, was bis zu 1 s dauern kann. Ist das Löschen erfolgreich, meldet die 2. Stufe dies mit dem Zeichen “D”, falls nicht sendet sie “F”.
5. Die PC–Anwendung sendet die Endadresse der zu ladeneden Anwendung mit dem Highbyte zuerst, gefolgt vom Lowbyte. Als Startadresse wird 0 im jeweiligen Sektor vorausgesetzt.
6. Die zweite Stufe des BSL lädt die Anwendung in den Speicher des Cotrollers und springt diesen an. Liegt die Anwendung so im Flash, dass sie per Hardwarekonfiguration automatisch gestartet wird, sollte die 2. Stufe zum Start einen “srst” durchführen. wird die durch die 2. Stufe des BSL eingestellte Buskonfiguration benötigt, weil z.B. die Anwendung im umgemapptem RAM liegt, verwendet man ein “jmps 0, 0”.

3.7 Der Debugger obcdbg

Die Steuerung des Debuggers erfolgt über die sieben Speedbuttons oben, die eigentlich selbsterklärend sind. Verweilt man kurz mit dem Mauszeiger über diesen, wird ein “Hint” zur Bedeutung angezeigt. Von links nach rechts haben wir:

1. open: Öffnet ein Projekt.
2. connect to target/disconnect: Stellt die Verbindung zum Target her bzw. trennt diese.
3. run: Startet die Anwendung auf dem Target.
4. step: Führt einen Einzelschritt der Anwendung an der aktuellen Adresse aus.
5. break: Unterbricht die Anwendung auf dem Target, es wird in den Monitor zurückgekehrt.
6. reset application: Neustart der Anwendung und des Monitors.
7. exit debugger: habe ich vergessen, was war das nochmal?

Ich habe mich bemüht, immer nur die sinvollen Buttons zu aktivieren, so dass eine Fehlbedienung erschwert wird, dies ist aber sicher noch nicht perfekt.

Mit dem Öffnen wird eine Fileselectbox aufgerufen, in der man das zu debuggende Projekt in Form der .mod–Datei auswählt. Dieser Quelltext wird in das Stringgitter geladen und aus dem zugehörigen Listing die Speicheradressen der einzelnen Zeilen extrahiert und mitangezeigt. Auch die Startadresse wird aus dem Listing gelesen, diese unten als IP angezeigt und im Stringgitter an die Quelltextposition dieser Adresse gesprungen.

Der Scrollbalken springt hierbei leider nicht mit, da bin ich mal auf die Windowsversion gespannt. Das Stringgitter ist in diesem Moment für den Nutzer noch gesperrt.

Hat man rechts oben das Device eingetragen, über das die Verbindung zum Target hergestellt wird, kann man "verbinden" anklicken. Hierbei wird der Bootstraploader angesprochen und nacheinander die 3 Dateien zum Target übertragen. Im Statusbalken sieht man momentan rechts noch den Erfolg. Anschließend ist das Stringgitter freigegeben, man kann darin herumscrollen und mit einem Doppelklick auf eine Zeile einen Haltepunkt setzen bzw. genauso wieder löschen. Die Haltepunkte werden durch ein "B" in der linken Spalte angezeigt.

Die Anwendung kann dann gestartet, gestoppt oder in Einzelschritten durchlaufen werden. Im gestoppten Zustand kann man sich Programmdaten anschauen wie links unten den Inhalt der Register oder daneben Speicherinhalte. Durch die von–Neumann-Architektur der C16x kann damit wirklich alles besichtigt werden. 4 verschiedene Speicherbereiche können mit den Reitern angewählt werden, im Editierfeld über der Anzeige kann die Anfangsadresse des gewünschten Bereichs eingegeben werden. Hat man diese geändert, bringt ein Klick auf "Load" die neuen Werte zur Anzeige, sonst werden bei jedem Halt des Targets die gewählten Werte aktualisiert.

Wenn man sich mal völlig verirrt hat, kann das gestoppte Programm zurückgesetzt und die Session erneut gestartet werden.

4 Debuggerkonzept

4.1 Stacklayout

Nach einem Interrupt sieht dies folgendermaßen aus, wobei der letzte Eintrag von der Initialisierung des Debuggers (dbgmonitor) hinzugefügt wird.:

SP	Wert
+6	PSW
+4	CSP
+2	IP
+0	CP

Nach dem RESET, der am gesetzten Flag USR0 im PSW erkennbar ist, muss ein solcher Stack erst aufgebaut werden.

In den Prozeduren des Monitors kommt jeweils noch deren Returnadresse dazu. Die muss beachtet und darf nicht verändert werden.

Startadresse der Anwendung: Diese wird vom Compiler als letzter Eintrag der Vektortabelle an Adresse $01FC übergeben. Dort wird sie als `01FC jmps seg, adr` eingefügt, so dass an Adresse $1FC `FA seg` steht, im folgenden Word die Startadresse. Der letzte Vektor wird vom Controller selbst nicht genutzt, er steht als Usertrap zur Verfügung. Selbst bei (modernsten) XC22xx–Controllern werden nur die Vektoren bis zur Nummer $01BC benutzt.

4.2 Nachrichten vom C16x zum PC

Die Übertragung erfolgt in Vielfachen von 4 Byte, erstmal je 4 Bytes pro Nachricht, ev. können Nachfolgebytes einfach übertragen werden, wenn das PC–Programm weiß, wieviele Bytes noch folgen. Das erste Byte gibt den Zweck der Übertragung an:

R (82): Aufruf durch Reset, restliche Bytes sind unrelevant.

N (78): Aufruf durch NMI, der CSP und der IP vor dem NMI werden gesendet.

I (73): Aufruf durch IllegalInstruction d.h. Breakpoint, der CSP und IP werden mitgesendet.

S (83): Aufruf durch ASC0 d.h. das Kontrollprogramm, der CSP und IP werden mitgesendet.

A* (65): Arrayindex übergelaufen (ILLOPA in BTRAP)

M* (77): Überlauf in Arithmetik (ILLINA in BTRAP)

C* (67): Stack–Kollision (STKOF in STOTRAP)

B (66): Codewert am Breakpoint als Antwort auf "B" im 2. Word.

P (80): PSW senden, das Statusword ist das 2. Word (wird nach einer Unterbrechung vom Monitor automatisch gesendet, also ohne Anforderung).

4.3 Nachrichten vom PC zum C16x

A (65): Adresse des Ausführungsstarts, CSP und Adresse werden mitgesendet (4 Bytes). **obsolet, s.o.**

S (83): Starte ab Ausführungsstart (1 Byte).

B (66): Setze Breakpoint, der CSP und die Adresse werden mitgesendet (4 Bytes).

E (69): Ersetze Breakpoint durch Wert, CSP, Adresse und Wert (6 Bytes).

R (82): Reset Target (jmps 0,0)

V (86): Sende Werte, je 32 Bytes. Folgebyte:

R (82): Registerwerte (2 Bytes)

M (77): Speicherwerte, CSP und Startadresse werden mitgesendet (5 Bytes)

4.4 Aktionen vom PC–Programm aus

(SetBreakpoint): Implizit im StringGrid1

- "B", CSP, Adresse
- warte auf "B", speichere 2. Word für komplette Adresse in Liste BreakP

RunToBreakpoint:

- "A", CSP, Adresse,
- "S"
- warte auf (I), (N), (S) oder (R)
- "E", CSP, Adresse, Wert (aus der Liste BreakP)

SingleStep:

-

Break: 1 Byte senden

ResetTarget:

- “R” senden
- warte auf (R)