

1 Der Sprachumfang

1.1 Datentypen

1.1.1 Zahlen

INTEGER sind positiv im Bereich von $0 \dots 2^{16} - 1$. Hexadezimalzahlen (vierstellig) können direkt mit einem vorgestellten 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.

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

1.1.2 Bitweise Integeroperationen

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 Integerwerte 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 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 Realationen

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

1.1.5 RECORDs

Sind in üblicher Weise implementiert. Zuweisungen zwischen diesen sind nur komponentenweise möglich. Die tatsächliche Adresse der jeweiligen Komponente wird vom Compiler vorab berechnet und entsprechend eingesetzt, so dass kein zusätzlicher Overhead im Programm entsteht.

1.1.6 ARRAYs

Sind global und für lange Prozeduren implementiert aber noch wenig getestet. Auch ein ARRAY OF RECORD sollte funktionieren, das muss aber auch noch getestet werden.

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.

Als Parameter können diese nicht übergeben werden, da die Adressauflösung hierfür sehr kompliziert wird. Der Compiler sollte bei einem solchen Versuch eine Fehlermeldung ausgeben. Ich überlege aber diese Beschränkung aufzuheben. Hierzu müsste vom Parser an `Index` die vorhandene Variable `ProcPar` übergeben und dort entsprechend ausgewertet werden.

Auch für Arrays sind Zuweisungen nur elementweise zulässig. Ist der Index 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.

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. Sie sollte daher im Anwendungsprogramm zumindest in der Testphase erfolgen, was unter Oberon ja sehr leicht möglich ist. Stellt sich später heraus, dass nie ein ungültiger Index verwendet wird, kann diese Überprüfung entfernt und damit der erzeugte Code verkürzt werden.

1.1.7 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 am Ende hinter den Programmtext eingefügt und mit "String" + Zahl gelabelt. Die Zahl wird von 1 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.

Strings können praktisch nur als Parameter für Prozeduren verwendet werden, insbesondere für `write`. Ansonsten gibt es (bisher) keinerlei Verarbeitungsroutinen für diese. Die Deklaration einer Variablen vom Typ `STRING` ist zwar möglich, es ist aber noch nicht einmal eine Zuweisung an diese Variable implementiert, womit die Deklaration natürlich sinnlos wird.

In Prozeduren können Strings mittels Assembler-routinen beliebig verarbeitet werden. Als Parameter wird die (gerade) Adresse des ersten Zeichens (Byte!) übergeben. Das Ende ist durch ein Nullzeichen markiert.

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 Seite 10.

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 Variablen als Bedingung

Diese habe ich im Parser in `SimpleExpression` nachgerüstet. Damit im Generator die Prozedur `Relation` nicht noch unübersichtlicher wird, erzeugt die neue Prozedur `TestTrue` den nötigen Vergleich.

Damit sind keine logischen Verknüpfungen als Prozedurparameter möglich. Ist der einzige Parameter beim Prozeduraufruf vom Typ `BOOLEAN`, erkennt der Parser diesen sonst als logische Verknüpfung. Um diesen Fehler zu vermeiden habe ich die Parservariable `ProcPar` eingeführt, die diese Fehlerkennung unterdrückt. Das sollte noch getestet und verbessert werden. Eventuell könnte durch `&` bzw. `OR` eine globale Parservariable gesetzt werden, da das Problem nur am Ende des Ausdrucks auftauchen dürfte. Damit könnten die Einschränkungen für Parameter entfallen.

Soweit getestet funktionieren diese auch mit Invertierung mittels `~`. Auch Kombinationen mit `&` und `OR` sind möglich, die verknüpften boolschen Variablen müssen hierzu aber in Klammern gesetzt werden, wie z.B.:

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

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 verschwendet. Der Output der obigen Anweisung wird zu (immer noch Fehler drin?):

```

        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 Integervariable für Flags anzulegen und mit dieser bis zu 16 boolsche Variablen zu ersetzen. Z.B. so:

```

CONST Bit0 = $0001; Bit8 = $0100;
VAR Flags: INTEGER;

```

```

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

Der Zugriff auf diese ist möglich, wenn die benötigten SFRs dem Namen nach vor den Konstanten im Programm deklariert werden. Die Namen müssen dabei den von AS verwendeten (richtigen) entsprechen. Z.B.:

```
SFR P4; P5; P6;
```

```
VAR i, j: INTEGER;
```

Um ein SFR zu ändern, muss dieses in eine Variable geladen und anschließend mitsamt der Änderung ins SFR zurück zugewiesen werden. Z.B.:

```
    i := P4;
    P4 := i ANDB ($FFFF - 8)    ; $
```

Das klingt umständlich, aber da die SFRs als *mem* angesprochen werden, muss der Quelloperand sowieso in einem Register liegen.

Aus den obigen Anweisungen erzeugt der Compiler folgenden Code:

```
    mov     0E000h, P4
    mov     R2, 0E000h
    and     R2, #65527    ; #
    mov     P4, R2
```

Ich glaube, da kann man nicht viel optimieren. Lediglich die Zuweisung einer Konstanten in ein SFR kann ohne Register erfolgen und demnach noch verkürzt werden. Dies ist mittlerweile so implementiert.

Für die SFR-Bezeichner sollten wie im Datenblatt auch Großbuchstaben verwendet werden. Zwar unterscheidet der Assembler nicht zwischen Groß- und Kleinschreibung, jedoch kann es in Spezialfällen, wie z.B. im nächsten Abschnitt beschrieben, zu Problemen kommen.

1.3.1 Extended SFRs

Meist werden die SFRs vom **AS** als *mem* angesprochen, so dass die extended SFRs genauso behandelt werden wie normale SFRs. Die Ausnahme liegt bei der Zuweisung von Konstanten an die SFRs. Dazu müssen diese als *reg* angesprochen werden, was mit den extended erstmal nicht geht. Der C167 stellt hierfür den Befehl **EXTR op** bereit, mit dem *op* Befehle auf die extended SFRs umgeleitet werden.

Da sonst zuviel Overhead entsteht, wird dieses auch genutzt. Hierfür muss dem ESFR-Namen ein **e** vorangestellt werden, DP1L wird also als SFR eDP1L deklariert. Dann nutzt der Compiler den **EXTR**-Befehl und der **AS** versteht dies auch korrekt. Das kleine **e** entfernt der Compiler bei der Codeausgabe, so dass wieder der korrekte Registername im Assemblertext zu finden ist.

Alternative, vor allem in kleinen Prozeduren: Wenn der Assembler eine ungültige Adressierungsart meldet, einfach den Wert erst einer Variablen zuweisen und diese dann dem ESFR. Dabei wird das Register wieder als *mem* angesprochen und Alles hat seine Richtigkeit.

1.3.2 Aliasnamen für SFRs

Assemblerprogramme kann man leichter portabel halten indem man die Direktive **Equate** zur Benennung von Ports oder ähnlichem benutzt. Dieselbe Möglichkeit habe ich in den Compiler mittels Aliasnamen eingebaut.

Hierzu werden dem jeweiligen SFR bei der Deklaration der Aliasname und der Originalname zugewiesen. Hängen z.B. zur Anzeige 8 LEDs an Port P1L, kann hierzu die Deklaration folgendermaßen aussehen:

```
SFR PLed = P1L; DPLed = eDP1L;
```

In *declarations* im Parser wird für eine solche Deklaration eine Liste angelegt, in der jeweils beide Namen enthalten sind. Im Generator durchsucht *MakeItem* diese Liste für jeden Aufruf eines SFRs und setzt gegebenenfalls den Originalnamen ein. Dreht man versehentlich die Deklaration um, so findet der Assembler das SFR nicht und erzeugt eine Fehlermeldung.

Der Vorteil der Aliasnamen liegt darin, dass im Modul selbst nun alle Zugriffe über den Aliasnamen erfolgen, z.B.: `PLed := 255;` Möchte man später einmal das Programm auf einer anderen Hardware verwenden, die einen anderen Port für die LEDs verwendet, so muss lediglich der Originalport in der Deklaration geändert werden.

Es stört den Compiler nicht, wenn der Originalport zusätzlich direkt deklariert ist. So etwas kann leicht passieren, wenn man eine Prozedur aus einem anderen Projekt in das aktuelle Modul kopiert.

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
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 Prozeduren

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 `expression` 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 (dieses Vorgehen ist in `op2` momentan auskommentiert, könnte also reaktiviert werden).

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 durchgeführt, 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 kann dieses Register eigentlich wieder frei gegeben werden, was in g_Parameter im Generator erfolgen müsste. Dies ist noch nicht implementiert, würde aber nur ein `nextreg := 1;` erfordern.

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 werden (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.

1.4.4 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. Seite 10).

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

```

VAR tick: INTEGER;

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

```

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

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 ausdrücklich 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 Standardprozeduren

Achtung: Die Ein- und Ausgaberroutinen sollten nicht in Interrupts verwendet werden. Dazu sind sie schlicht zu zeitaufwendig. Eventuell werde ich diese Möglichkeit 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, wobei bisher nur das kleine “h” für hexadezimale Ausgabe implementiert ist. Hexadezimale Ausgaben werden genauso im Speicher angelegt wie dezimale jedoch inklusive der führenden Nullen versendet.

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. Seite 15) und das Drumherum mit dem Prozeduraufruf für eine einzelne Assembleranweisung genervt hat.

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;                                dummi
BEGIN                                           ; verbatim
VERBATIM                                       ob der AS
    Ob der AS                                  das versteht?
    das versteht?                              ; end_verbatim
END_VERBATIM;                                  sub    CP, #32
END dummi;                                     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 L und einer fünfstelligen Zahl $< 2^{15}$. Eine solche Konstruktion sollte daher im Assemblercode vermieden werden. Beispiel:

```
PROCEDURE* Fstep(VAR step:INTEGER; cur:INTEGER);
VAR bf:INTEGER;
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, [R11]          ; 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 Assemblerrouinen 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

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 inkludiert. 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 `$FE00`, diese muss im ersten Speichersegment liegen.

StackSize: (-ss"n") Länge des Stacks in Bytes, diese wird nur benutzt, um den Wächter STKOV zu setzen. Default ist 1024.

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 2.2.1). Default ist 0, d.h. keine Optimierungen aktiv.

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.

2.2.1 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.

2.3 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 und die Standarddefinitionen für diesen Prozessortyp als **reg166.inc** eingebunden. Für neuere Prozessoren sollte dieses Includefile

entsprechend angepasst werden. Eventuell sind noch nicht alle SFR des C167 in diesem Includefile definiert, einzelne habe ich schon hinzugefügt, was sehr einfach möglich ist.

Momentan muss dieses File, das zum **AS** gehört, in das aktuelle Verzeichnis kopiert werden. Eventuell wäre es sinnvoll für solche Sachen ein Defaultverzeichnis anzulegen und dieses in den Compiler zu übernehmen, da aber auch das Assemblerfile als Text problemlos editiert werden kann, ist dieses nicht vordringlich.

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 Dateieindung 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 zugehöriger 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 Bootstrapladers, 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.S. 17).

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 ($x = 0 \dots 4$) wird ein Register namens BUSCON x verwendet, das die jeweiligen Hardwareeinstellungen für diesen Bereich enthält. Hinzu kommt für die Bereiche mit $x > 0$ jeweils ein Register mit Namen ADDRSEL x , 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 Waitstates 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
bset    SYSCON.2      ; enable XRAM
or      BUSCON0, #13  ; set waitstates
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 Jumpfern 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 oligatorisch, 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 Procedures genutzt werden kann, festgelegt. Dieser Frame wird über den FSP, der immer im GPR R0 verfügbar ist, verwaltet.

```

mov     STKUN, #0FE00h
mov     STKOV, #0FA00h
mov     SP, #0FE00h
mov     CP, #0F600h
nop
mov     R0, #0E7FEh      ; FSP = top of softstack
jmp     mainstart
; 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.