9 Strukturierte Programmierung

Die "strukturierte Programmierung" ist entstanden aus dem Wunsch, die Korrektheit von Programmen beweisen zu können. Dies wäre ein ziemlich hoffnungsloses Unterfangen, wenn jede beliebige Programmstruktur zugelassen wäre. Die strukturierte Programmierung verlangt deshalb, ein Programm aus den drei Grundstukturen

- Sequenz (Folge),
- Alternative,
- Schleife

aufzubauen.

9.1 Darstellung der drei Grundstrukturen

Im Struktogramm, im Programmablaufplan (PAP, früher üblich) und in der Programmiersprache C stellen sich die Grundstrukturen Sequenz, Alternative und Schleife wie folgt dar (Bilder 9-1, 9-2 und 9-3)

Sequenz

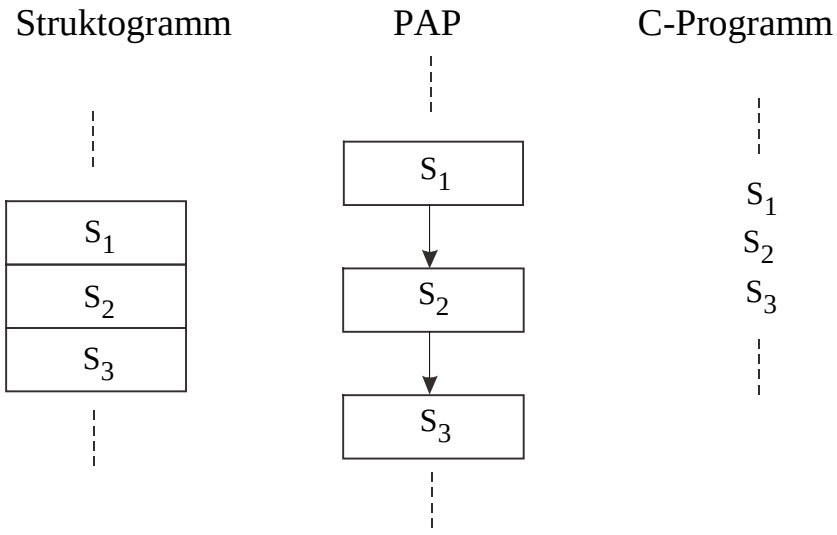
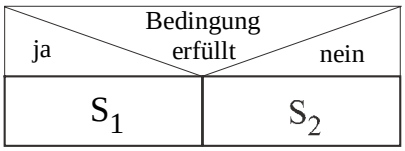


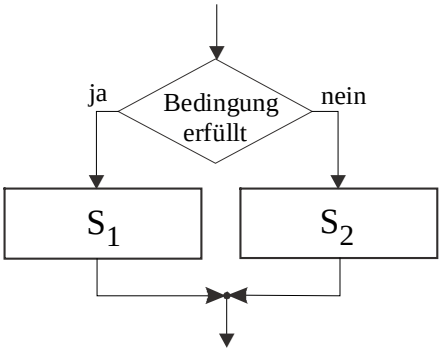
Bild 9-1: Struktogramm, Programmablaufplan nach DIN 66001 und C-Programm einer Sequenz. Ist S_i im C-Programm ein einzelnes Statement, wird es mit Semikolon abgeschlossen.;

Dabei wird angenommen, daß jedes S_i ein Block ist, der selbst wieder aus einer oder mehreren der Elemente Sequenz, Alternative, Schleife zusammengesetzt ist. Zerlegt man in diesem Sinne immer weiter, kommt man schließlich zu den Instruktionen, in Hochsprachen meist Statements genannt. Diese Zerlegung eines Programms von oben nach unten heißt auch Top-Down-Programmierung.

Alternative

Struktogramm PAP C-Programm



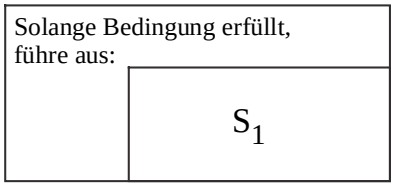


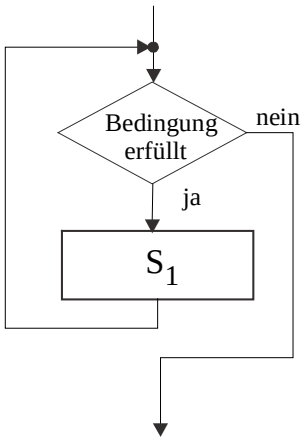
if (Bedingung) {S_1}
else {S_2}

Bild 9-2: Alternative

Schleife (While-Schleife)

Struktogramm PAP C-Programm





while (Bedingung) {S_1}

Bild 9-3: (While-)Schleife

Man kann beweisen, daß alle durch einen Programmablaufplan darstellbaren Programme ohne Änderung ihrer Funktion so abgeändert werden können, daß sie nur noch aus den obigen drei Grundtypen der Sequenz, Alternative und Schleife bestehen. Da in den Struktogrammen ebenso wie in den entsprechenden C-Programmen keine Sprungbefehle sichtbar sind, spricht man auch von Programmierung ohne "GO TO" und von der Vermeidung eines Programmierstils, der "Spaghetti-Code" erzeugt. Diese Forderung, in höheren Programmiersprachen auf "GO TO" zu verzichten, wurde schon 1968 von Dijkstra aufgestellt.

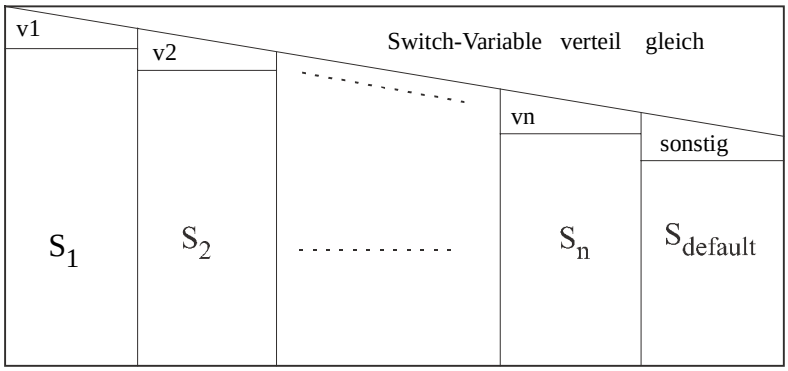
9.2 Varianten der Grundstrukturen Alternative und Schleife

In der Praxis wäre es oft umständlich, ein Programm nur aus den drei Grundtypen von Abschnitt 9.1 zusammenzusetzen. Deshalb werden Varianten eingesetzt, was aber an der Intention der strukturierten Programmierung nichts ändert. Zu nennen sind hier:

Case-Struktur

Jede Mehrfach-Alternative läßt sich als Folge ineinandergeschachtelter Alternativen gemäß Bild 9.2 schreiben. Bequemer ist die Darstellung als Case-Struktur (auf die Darstellung im PAP wird von jetzt an verzichtet):

Struktogramm



C-Programm

```
switch(verteil)
{
 case v1: {S1}
          break;
 case v2: {S2}
          break;
       :
 case vn: {Sn}
          break;
 default: {Sdefault}
}
```

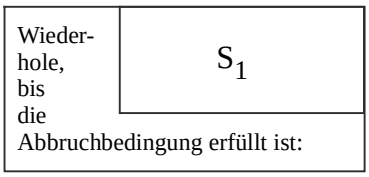
Bild 9-4: Mehrfach-Alternative oder Case-Struktur

Wenn bekannt ist, daß einer der Fälle v1, v2, ..., vn mit Sicherheit eintreten muß, kann man auf den Default-Teil auch verzichten.

Repeat-Until-Schleife

Eine While-Schleife wird möglicherweise kein einziges Mal durchlaufen, nämlich dann, wenn die Bedingung am Anfang bereits nicht erfüllt ist. Demgegenüber wird die Repeat-Until-Schleife mindestens einmal durchlaufen, weil hierdie Bedingung erst am Ende geprüft wird. In C wird sie mit der Struktur do .... while realisiert.

Struktogramm



C-Programm

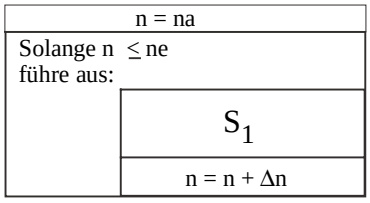
```
do {S1} while (Nicht-Abbruchbedingung)
```

Bild 9-5: Repeat-Until-Schleife

Zähl-Schleife

Sehr häufig wird die Zählschleife oder FOR-Schleife verwendet. Hier handelt es sich um eine Sequenz aus einer Initialisierungsanweisung und einer While-Schleife, wie Bild 9-6 zeigt.

Struktogramm



C-Programm

```
for (n = na; n <= ne; n+=Δn) {S1}
```

oder einfacher:

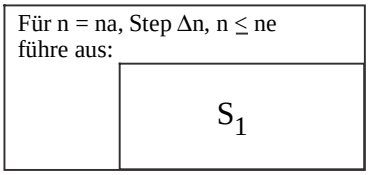
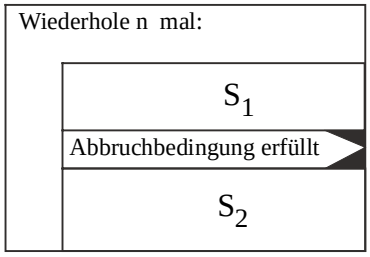


Bild 9-6: FOR-Schleife

Schleife mit Mittenausgang

Bei mathematischen Iterationen, z.B. der Berechnung von Nullstellen nach dem Newton-Verfahren wird im eine Schleife sooft wiederholt, bis sich das Rechenergebnis vom Rechenergebnis des vorherigen Schleifendurchlaufs um weniger als eine kleine Zahl ε unterscheidet. Außerdem wird zur Sicherheit für den Fall, daß keine Konvergenz erfolgt, die Zahl der Schleifendurchläufe begrenzt. Es gibt also zwei Abbruchkriterien, die man gern an zwei verschiedenen Stellen der Schleife plaziert. Man kommt so zu der in Bild 9-7 dargestellten Schleife mit Mittenausgang.

Struktogramm



C-Programm

```
for (i = 0; i < n; i++)
{
   {S1}
   if (Abruchbedingung) break;
   {S2}
}
```

Bild 9-7: Schleife mit Mittenausgang

Ein Spezialfall der Schleife mit Mittenausgang besteht darin, daß man n gleich ∞ setzt, also im C-Programm statt der FOR-Schleife eine While-Schleife der Form

```
WHILE (TRUE)
```

verwendet.

Aufgabe 9.1: Zeichnen Sie das Struktogramm und den PAP für die Endlosschleife mit Mittenausgang. Formulieren Sie auch das entsprechende C-Programm.

9.3 Realisierung der Strukturelemente mit dem 68000-Assembler

Der MC 68000 ist in Hinblick auf Anforderungen der strukturierten Programmierung entwickelt worden. Nachstehend sind einige Methoden zusammengestellt, wie man die Elemente Sequenz, Alternative und Schleife im Assemblercode effektiv verwirklichen kann.

Sequenz

Die Module S_1, S_2, ... sind entweder bereits einzelne Assemblerbefehle oder können in Durchführung des TOP-DOWN-Prinzip in solche aufgelöst werden. Die Befehlsfolgen selbst können einfach untereinander geschrieben werden. Es empfiehlt sich, dabei folgende Regeln zu beachten:

- Kleine Blöcke S_1, S_2, ... bilden,
- die Blöcke durch Kommentarzeilen mit kurzer Funktionsbeschreibung zu trennen,
- (weitgehend) alle Befehle in der gleichen Zeile zu kommentieren. Regel: Der Kommentar soll mindestens soviele Zeichen enthalten wie der Befehlscode.

Diese Empfehlung provoziert den Einwand, man vergeude Zeit. In Wirklichkeit spart man Zeit, wenn man nicht nur an die Erstellung des Programms, sondern an seine Lebenzeit mit Wartung, Ergänzungen usw. denkt. Denn Programme ohne Kommentar werden schon nach wenigen Wochen vom Autor selbst nicht mehr verstanden, geschweige denn von Fremden.

Alternative

Die Bedingung zur Wahl des einen oder anderen Zweiges der Alternative wird durch das Setzen von Flags im CCR gesteuert. Die Flags können gesetzt werden

- durch arithmetische oder logische Operationen wie ADD, SUB usw.,
- durch beliebige MOVE-Instruktionen mit Registern oder Speicherplätzen als Ziel. Hier unterscheidet sich der MC 68000 von vielen anderen Prozessoren wie z.B. dem 80535, wo MOVE-Befehle keine Auswirkungen auf die Flags haben.
- durch künstliche Manipulation des CCR-Registers (siehe Beispiel Abschnitt 6.3).

Der Einfluß der verschiedenen Befehle auf die Flags X, N, Z, V, C des CCR wurde bereits in von Abschnitt 5.2 in Tabelle 5-7 zusammengestellt. Dort findet man auch die verschiedenen Conditioncodes cc für den bedingten Sprungbefehl Bcc. Im Assemblercode hat die Alternative in Bild 9-8 dargestellten Aufbau.

```
                bcc    S1          ; Sprung S1, wenn Bedingung cc erfuellt
        S2:        .               ; Beginn Block S2
                   .               ;          .
                   .               ;          .
                   .               ; Ende   Block S2
                bra    Endalt      ; Sprung zum ersten Befehl nach S1
        S1:        .               ; Beginn Block S1
                   .               ;          .
                   .               ;          .
                   .               ; Ende   Block S1
        Endalt:    ....            ; Programmfortsetzung nach
                                   ; Ende der Alternative
```

Bild 9-8: Prinzipieller Aufbau der Alternative im Assemblercode (kleinere Module)

Module sollten übersichtlich klein bleiben. Sie haben dann sicher eine Länge kleiner als 64 K und können folglich mit bra- bzw. bcc-Befehlen auskommen. Ist das ausnahmsweise nicht der Fall, muß man entsprechend Bild 9-9 programmieren.

```
                bcc/   S2          ; Sprung S2, wenn cc nicht (!) erfuellt
                jmp    S1          ; (cc/ soll hier bedeuten:  n i c h t  cc)
        S2:        .               ; Beginn Block S2
                   .               ;          .
                   .               ; Ende   Block S2
                jmp    Endalt      ; Sprung zum ersten Befehl nach S1
        S1:        .               ; Beginn Block S1
                   .               ;          .
                   .               ; Ende   Block S1
        Endalt:    ....            ; Programmfortsetzung nach
                                   ; Ende der Alternative
```

Bild 9-9: Prinzipieller Aufbau der Alternative im Assemblercode (große Module)

Case-Struktur

Bei der Case-Struktur von Bild 9-4 kann man auch das ´Break´ zwischen zwei Blöcken, z.B. zwischen {S_1} und {S_2} weglassen. Dann wird im Falle von verteil = v1 nicht nur {S_1}, sondern anschließend auch {S_2} ausgeführt.

Aufgabe 9.2 Versuchen Sie, die Case-Struktur von Bild 9-4 in allgemeiner und übersichtlicher Weise im 68000-Assemblercode darzustellen.

While-Schleife

```
anf:                .       ; Auswerten der Bedingung cc, was
                    .       ; zum Setzen oder Rücksetzen
                    .       ; eines oder mehrerer Flags führt.
              bcc/   Whend  ; Sprung zum Ausgang, falls cc nicht
                            ; erfüllt
                    .       ; Beginn Block S1
                    .       ;          .
                    .       ;          .
                    .       ; Ende   Block S1
              bra    anf    ; Rücksprung zum Schleifenanfang
Whend:        ....          ; Programmfortsetzung nach
                            ; Ende der While-Schleife
```

Bild 9-10: Prinzipieller Aufbau der While-Schleife im Assemblercode

Beispiel: Zeichenweise Bildschirmausgabe aus einem Puffer, bis das ASCII-Zeichen für <CR> gefunden wird. <CR> selbst wird nicht ausgegeben (Bild 9-11).

```
   Debug       equ     $00F8179A
   OutRaw      equ     $00F809AA
               org     $3000
   puffer:     ds.b    256           ; 256-Byte-Puffer
   start:                .           ; Programmteil, der Puffer füllt
                         .
               lea     puffer,a0     ; Vorbereitung While-Schleife:
                                     ; Puffer-Anfangsadresse -> a0
   anf:
               cmpi.b  #$0d,(a0)     ; Auswerten Bedingung: <CR> nicht im Puffer
               beq     Whend         ; Sprung zum Schleifenende
; Beginn S1 --------------------
               move.b  (a0)+,d0      ; Normales Zeichen ungleich <CR> -> d0
               jsr     OutRaw        ; Zeichen -> Bildschirm
; Ende   S1 --------------------
               bra     anf           ; Rücksprung zum Schleifenanfang
   Whend:      ....
               end     start
```

Bild 9-11: Beispiel einer While-Schleife im Assemblercode

Repeat-Until-Schleife

```
      anf:          .             ; Beginn Block S1
                    .             ;          .
                    .             ;          .
                    .             ; Ende   Block S1
   ; --------------------
                          .       ; Auswerten der Abbruchbedingung cc, was
                    .             ; zum Setzen oder Rücksetzen
                    .             ; eines oder mehrerer Flags führt.
                 bcc/   anf       ; Rücksprung zum Schleifenanfang, wenn
                                   ; Abbruchbedingung nicht erfüllt.
                 ....              ; Programmfortsetzung nach
                                   ; Ende der Repeat-Until-Schleife
```

Bild 9-12: Prinzipieller Aufbau der Repeat-Until-Schleife im Assemblercode

Diese Schleifenart ist also im Assemblecode noch einfacher zu programmieren als die While-Schleife. Der einzige Nachteil: Geringere Allgemeinheit, da die Repeat-Until- Schleife mindestens einmal durchlaufen wird.

Aufgabe 9.3 Programmieren Sie das Beispiel aus Bild 9-12 unter der Annahme, daß mindestens ein Zeichen ungleich <CR> im Puffer steht, als Repeat-Until-Schleife im 68000-Assemblercode.

Aufgabe 9.4 Notieren Sie das Prinzip für die Schleife mit Mittenausgang im Assemblercode.

Zähl-Schleife

```
                 move.l  #na,d1    ; Zaehler (hier d1) initialisieren
      abfr:      cmpi.l  #ne,d1    ; Abfrage auf Zaehler-Endwert          .
                 bgt     endzaehl  ; Wenn <d1> groesser ne, Ende der Zaehl
                                   ; schleife
   ; --------------------------
                    .              ; Beginn Block S1
                    .              ;        ...
                    .              ; Ende   Block S1
   ; --------------------------
                 add.l   #deltn,d1 ; Zaehler erhöhen um Delta-n und
                 bra     abfr      ; Rueecksprung zur Abfrage.
      endzaehl:  ....              ; Programmfortsetzung nach
                                   ; Ende der Zaehl-Schleife
```

Bild 9-13: Prinzipieller Aufbau der Zähl-Schleife im Assemblercode

DBcc-Befehl beim MC 68000

Programmiert man eine Zählschleife entsprechend dem Muster von Bild 9-13, kann man die maximal mögliche Zahl von Schleifendurchläufen (in einer Einfach-Schleife) erzielen. Doch wird das meist gar nicht benötigt. Ferner arbeitet man meistens mit Delta-n = 1. Auch ist es nur selten zwingend erforderlich, vorwärts zu zählen, es geht auch mit Dekrementieren. Zur Programmierung dieses Normalsfalls hat Motorola den dbcc-Befehl eingeführt, der Abfrage und Sprung vereinigt und auch wesentlich schneller ausgeführt wird. Seine Syntax lautet

DBcc (Decrement and branch on Condition cc)

```
[<label>]   DBcc    Di,<marke>          [;Kommentar]
```

Dieser Befehl ist besonders für Zählschleifen mit zusätzlicher Abbruchbedingung geeignet. Für die einfache Zählschleife genügt die einfachere Sonderform

DBF (Decrement and branch on (Condition, which is always) False)

```
[<label>]   DBF     Di,<marke>          [;Kommentar]
```

Bild 9-14 zeigt die Wirkungsweise des DBcc-Befehls. Dabei ist zu beachten, daß beide Befehle nur mit Wort-Operanden arbeiten.

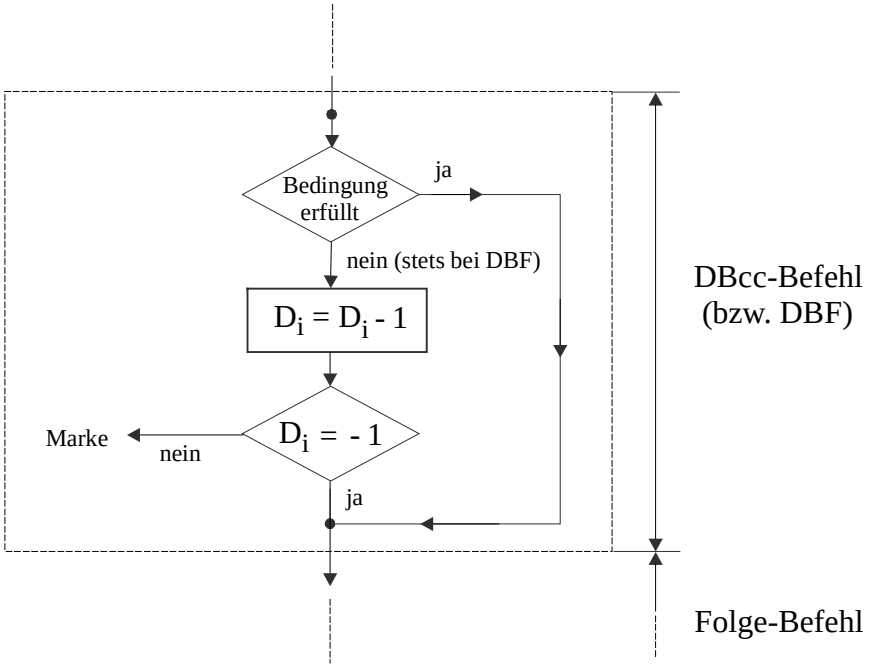


Bild 9-14: Wirkungsweise des DBcc-Befehls. Achtung: Der Befehl arbeitet ausschließlich mit Wort-Operanden!

Die Anwendung des DBF-Befehls auf eine gewöhnliche Zählschleife mit festem n für die Zahl der Wiederholungen führt zu dem in Bild 9-15 dargestellten prinzipiellen Aufbau.

```
                clr.w   d1          ; Zaehler (hier d1) löschen
                move.w  #(n-1),d1   ; und mit n-1 (!) initialisieren
    ; --------------------------
      S1:           .               ; Beginn Block S1
                    .               ;        ...
                    .               ; Ende   Block S1
    ; --------------------------
                dbf     d1,S1       ; Rueecksprung zu S1, falls Inhalt
                    .               ; von d1 noch > -1 ist.
                    .               ; Fortsetzung Programm nach Schleifenende
```

Bild 9-15: Zählschleife mit festem n < 65536 bei Verwendung von DBF

Es ist etwas unpraktisch, daß man bei einer Zählschleife, die z.B. 50 mal durchlaufen werden soll, den Zähler mit 49 = 50 - 1 zu laden hat. Das hängt zusammen mit der hardwaremäßig verdrahteten Abfrag auf -1 im MC 68000. man kann diese Unschönheit aber umgehen durch einen Programmaufbau gemäß Bild 9-16.

```
                clr.w   d1          ; Zaehler (hier d1) löschen
                move.w  #n,d1       ; und mit n initialisieren
                bra     abfrage
    ; --------------------------
      S1:           .               ; Beginn Block S1
                    .               ;        ...
                    .               ; Ende   Block S1
    ; --------------------------
      abfrage:  dbf     d1,S1       ; Rueecksprung zu S1, falls Inhalt
                    .               ; von d1 noch > -1 ist.
                    .               ; Fortsetzung Programm nach Schleifenende
```

Bild 9-15: Zählschleife mit festem n < 65536 bei Verwendung von DBF. Sofortiger Sprung zur Abfrage. Vorteil: Zählerinhalt ist = n und Schleife wird n mal durchlaufen.

Beispiel: 20 Zeichen aus einem Puffer am Bildschirm ausgeben ohne Rücksicht auf Art des Zeichens (vergleiche mit dem Beispiel von Bild 9-11).

```
puffer:     ds.b    256             ; 256-Byte-Puffer
start:                  .           ; Puffer füllen
                        .
            lea     puffer,a0       ; Vorbereitung Schleife:
            clear.w d1
            move.w  #20,d1
            bra     abfrage         ; Sprung zur Abfrage
S1:         move.b  (a0)+,d0        ; Zeichen aus Puffer
            jsr     OutRaw          ; -> Bildschirm
abfrage:    dbf     d1,S1           ; Dekrementieren und abfragen
```

Bild 9-16: Beispiel einer Zähl-Schleife mit festem n unter Verwendung von DBF

Den DBcc-Befehl wendet man gern an bei einer Zählschleife mit zusätzlicher Abbruchbedingung. Das allgemeine Schema ist Bild 9-17 zu entnehmen.

```
                clr.w   d1          ; Zaehler (hier d1) löschen
                move.w  #(n-1),d1   ; und mit n-1 initialisieren
    ; --------------------------
      S1:           .               ; Beginn Block S1
                    .               ;        ...
                    .               ; Ende   Block S1
    ; --------------------------
                    .               ; Abbruchbedingung  cc  auswerten, dabei
                                    ; Flags verändern
    ; --------------------------
                dbcc    d1,S2       ; Sprung zu S2, falls cc nicht erfüllt
                bra     endschl     ; und d1 noch > -1 ist. Sonst Ausgang
    ; --------------------------
      S2:           .               ; Beginn Block S2
                    .               ;        ...
                    .               ; Ende   Block S2
                bra     S1          ; Ruecksprung zu Block S1
    ; --------------------------
      endschl:      .               ;.Fortsetzung Programm nach Schleifenende
```

Bild 9-17: Zählschleife mit zusätzlicher Abbruchbedingung (Schleife mit Mittenausgang)

Beispiel: 20 Zeichen aus einem Puffer am Bildschirm ausgeben, aber vorher aufhören, wenn <CR> gefunden wird.

```
                                .
                                .
      puffer:     ds.b    256           ; 256-Byte-Puffer
      start:                  .         ; Puffer füllen
                              .
                  lea     puffer,a0     ; Vorbereitung Schleife:
                  clear.w d1            ; Zaehler initialisieren
                  move.w  #(20-1),d1    ; mit 20–1 = 19
    ; -------------------------------
      S1:         move.b  (a0)+,d0      ; Block S1: Zeichen aus Puffer -> d0
    ; -------------------------------
                  cmpi.b  #$0d,d0       ; kombinierte Abfrage
                  dbeq    d1,S2         ; Wenn Zeichen = <CR> oder wenn
                  bra     Endschl       ; 20 Zeichen ausgegeben, Schleifenausgang
    ; -------------------------------
      S2:         jsr     OutRaw        ; Block S2: Zeichen -> Bildschirm
                  bra     S1            ; Ruecksprung -> S1
    ; -------------------------------
      Endschl:      .                   ; Fortsetzung Programm
```

Bild 9-18: Beispiel einer Zähl-Schleife mit Mittenausgang