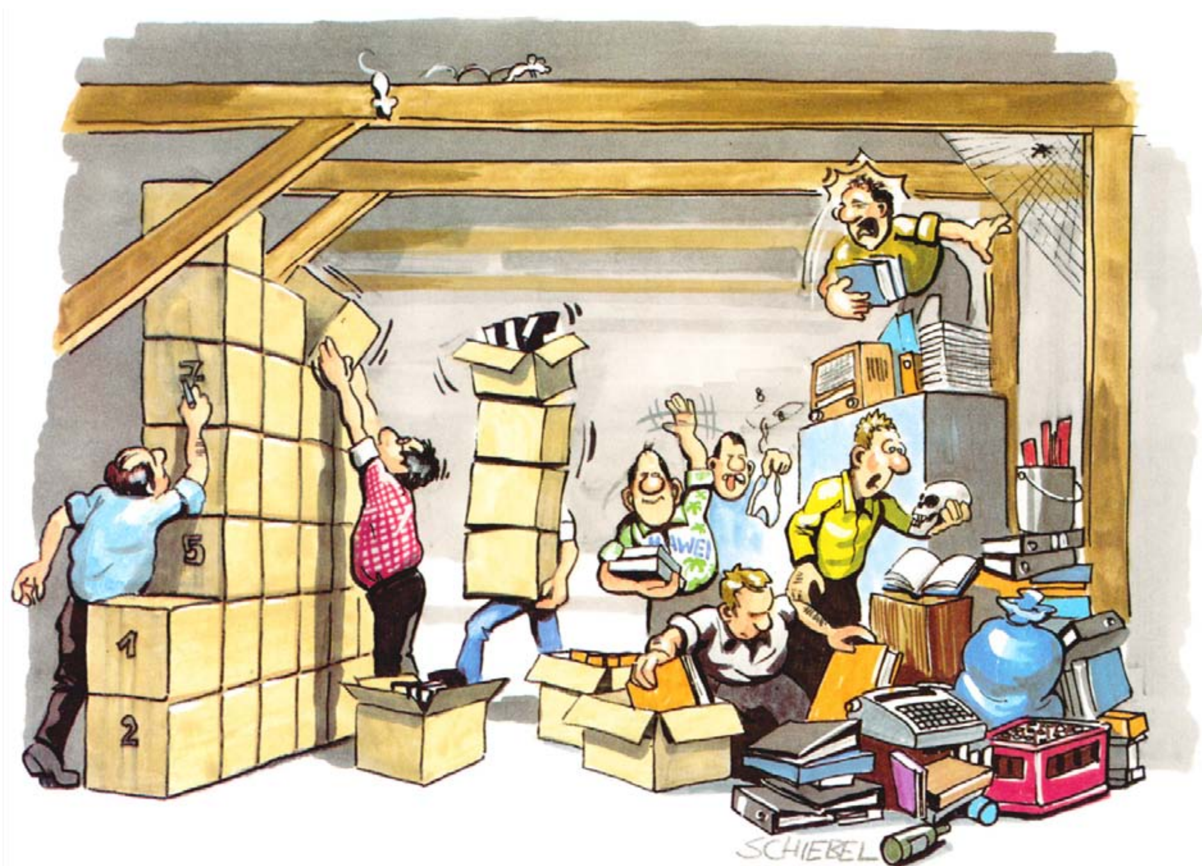
Platz schaffen durch Überlagern

Overlay-Strukturen in Turbo Pascal

Große Programme, oft benötigte Unterstrukturen und bedingungsabhängiges Zuladen weiterer Programmteile lassen sich nur durch Overlays realisieren. Den mit der Version 3.0 einmal begangenen, mit 4.0 wieder verlassenen Weg nimmt Borland mit Turbo Pascal 5.0 und 5.5 wieder auf.

Die Arbeitsweise der Overlay-Technik ab der Version 5.0 ist völlig anders, als aus der Version 3.0 bekannt: Überlagert werden nun nicht mehr einzelne Routinen, sondern ganze Units. Auch reserviert Turbo nun nur noch einen Overlay-Puffer für alle Überlagerungsroutinen, und nicht mehr einen Bereich für jede Overlay-Gruppe. Der Overlay-Puffer wird bereits durch die Initialisierung der Unit System angelegt und entspricht der Größe der längsten überlagerbaren Unit, er kann (und sollte) vom Programmierer vergrößert werden.
Die Codeteile aller Overlay-Units werden nicht mehr in der EXE-Datei gespeichert, sondern in eine Datei mit der Extension OVR geschrieben. Aus dieser Datei wird bei Bedarf der Code in den Arbeitsspeicher nachgeladen. Die globalen Daten der Units werden wie gewohnt im Datensegment gehalten und sind von der Overlay-Technik nicht betroffen.
Beim Schreiben einer Unit braucht noch nicht festgelegt zu werden, ob diese Unit später überlagerbar sein soll. Lediglich die beiden Optionen des Turbo-Pascal-Compilers $F+ und $O+ müssen auf jeden Fall gesetzt sein. Durch die Option $F+ werden auch alle lokalen und nicht exportieren Routinen *far* erzeugt. Bei ihrem Aufruf wird die Return-Adresse in 4 Bytes (Segment und Offset) auf dem Stack abgelegt (*Bild 1*). Dieses Format braucht das Laufzeitsystem, um den gesamten Stack bei Ein- und Auslagerungen zu überprüfen. Die Option muß bei allen Units und beim Hauptprogramm gesetzt sein, wenn Overlays verwendet werden sollen. String- und Set-Konstanten werden von Turbo Pascal im Code-Segment abgelegt.



Daraus ergibt sich ein Problem, wenn solche Konstanten als Parameter übergeben werden: Das aufrufende Programm – und damit auch die Konstanten – sind eventuell nicht mehr im Speicher. Hier greift die Option $O+; es ist die einzige Wirkung dieser Option: Bei jedem Aufruf, der in eine andere Unit führt und der String- oder Set-Konstanten als Parameter übergibt, generiert der Compiler vorab den Aufruf einer Kopier-Routine. Diese Routine kopiert die Konstante auf den Stack und es wird die Adresse der Kopie als Parameter übergeben. Mit dieser Option sollte man allerdings sparsam umgehen, da der zusätzliche Kopiervorgang Laufzeit und Platz erfordert. Im Hauptprogramm hat diese Option nichts zu suchen.
Erst bei der Übersetzung des Hauptprogramms wird die Overlay-Struktur definiert. Hier sind alle Units, die überlagerbar sein sollen, mit der Option $O <name> aufzuführen. Dadurch wird Turbo veranlaßt, den Code in die OVR-Datei auszulagern und statt dessen einen Steuerblock (*Bild 2*) anzulegen.
Zu Programm-Beginn muß dann die Routine *OvrInit* mit dem Namen der OVR-Datei als Parameter aufgerufen werden. Dieser Aufruf eröffnet die OVR-Datei und initialisiert das Overlay-System.

Der Overlay-Puffer

Bereits bei Programmstart wird ein Overlay-Puffer angelegt. Dazu verschiebt der Initialisierungsteil der System-Unit den Beginn des Heap-Speichers *(HeapOrg)* um soviel, daß in den nun freigewordenen Platz die größte Overlay-Unit hineinpaßt. Die Variablen *OvrHeapOrg* und *OvrHeapEnd* werden mit dem Anfang und dem Ende des Puffers besetzt. Die Puffergröße ist in der Konstanten *OvrHeapSize* gespeichert. Allerdings enthält diese Konstante auch nach Vergrößerung des Overlay-Puffers immer noch den Start-Wert; die tatsächliche Größe des Puffers kann nur mit *OvrGetBuf* oder Subtraktion der Werte *OvrHeapEnd – OvrHeapOrg* ermittelt werden. Der Pointer *OvrHeapPtr* zeigt immer auf die Stelle im Puffer, an der das nächste Overlay geladen werden soll, er ist mit *OvrHeapOrg* initialisiert.
Der Puffer kann mit der Routine *OvrSetBuf* vergrößert werden. Dies ist aber nur möglich, wenn der Puffer leer ist – das läßt sich durch Aufruf von *OvrClearBuf* erreichen – und noch kein Heap-Speicher verwendet wurde. Die zweite Einschränkung ist wesentlich: Da durch die Vergrößerung der Beginn des Heap verschoben wird, würden bereits angelegte Heap-Variable überschrieben.
Bei einem Einlagerungsvorgang prüft das Laufzeitsystem, ob oberhalb des Zeigers *OvrHeapPtr* genügend Platz für die neue Unit vorhanden ist. In diesem Fall wird die Unit in den Puffer kopiert. Andernfalls werden die ältesten Units aus dem Puffer entfernt und die verbleibenden an das Puffer-Ende verschoben (*Bild 3*).
Ab der Version 5.5 kann ein Bewährungsbereich definiert werden, dieser Bereich liegt immer oberhalb des Zeigers *OvrHeapPtr*. Standardmäßig ist der Bereich mit 0 Byte angelegt, das Overlay-System verhält sich dann wie in der Version 5.0. Das Laufzeit-System vermerkt alle Aufrufe der Units im Bewährungsbereich in den Steuersegmenten. Muß nun durch eine Einlagerung eine Unit aus dem Puffer entfernt werden, so wird vorab geprüft, ob die Unit „sich bewährt“ hat – in diesem Fall wird sie wieder an den Beginn des Puffers verschoben und eine andere Unit ausgelagert (*Bild 3*). Durch dieses Verfahren bleiben oft referierte Units im Puffer erhalten, die Effizienz des Programms wird erhöht. Die Größe des Bewährungsbereiches wird mit der Prozedur *OvrSetRetry* eingestellt und kann mit *OvrGetRetry* abgefragt werden.
Besitzt der Rechner Expanded Memory (EMS), kann die gesamte OVR-Datei durch *OvrInitEMS* in diesen Speicherbereich übertragen werden. Bei Einlagerungen werden die Units dann von dort statt aus der OVR-Datei gelesen – das Verfahren wird dadurch wesentlich beschleunigt. Der Code wird aber nicht im EMS-Speicher ausgeführt, sondern weiterhin in den normalen Arbeitsspeicher kopiert. Durch eigene Laderoutinen kann dieses Verfahren ab der Version 5.5 auch auf Extended Memory bei AT-Rechnern angewendet werden (siehe *Listing 3*).

Die interne Struktur

Für jede Overlay-Unit erzeugt Turbo Pascal in der EXE-Datei einen Steuerblock. Dieses Steuersegment enthält in einem festen Teil von 32 Byte (2 Paragraphen) die wichtigsten Informationen über die Unit: Position und Länge des Codes in der OVR-Datei, die Ladeadresse, falls der Code geladen ist und eine Verkettung aller geladenen Units sowie aller Overlay-Steuerblöcke. An diesen festen Teil schließt sich eine variable Vektor-Tabelle an. Hier ist für jede Routine der Unit ein 5-Byte-Eintrag vorhanden. Jeder Vektor enthält den Befehl INT 3Fh (CD 3F) sowie die relative Adresse der Routine innerhalb des Unit-Codes. Das fünfte Byte bleibt vorerst frei.
Der Linker setzt alle Verweise auf Routinen in Overlay-Units auf den entsprechenden Vektor. Der Aufruf einer solchen Routine führt nun zu einem Interrupt 3Fh. Der Interrupt-Handler 3Fh wird durch die Prozedur *OvrInit* eingekettet. Auch die System-Unit kettet bereits einen Interrupt-Handler 3Fh ein – dieser erzeugt lediglich den Laufzeit-Fehler 208: Overlay-System nicht initialisiert. Der Interrupt-Handler lädt nun das Overlay-Coding in den Puffer, setzt die Lade-Adresse im Steuersegment und verkettet die Unit mit den übrigen geladenen Units. Außerdem werden alle Vektoren in der Tabelle in Befehle JMP <segment>:<offset> umgesetzt, jetzt wird das 5. Byte benötigt. Jeder Aufruf einer bereits geladenen Routine führt über diesen JMP dann direkt zur richtigen Prozedur.
Durch die Einlagerung müssen eventuell andere Units aus dem Puffer entfernt werden. In den Steuerblöcken dieser Units wird die Lade-Adresse gelöscht und sie werden aus der Lade-Kette entfernt. Damit ist es aber nicht getan: Es könnten Prozeduren aktiv sein, also Return-Adressen auf dem Stack auf diese Unit verweisen. Damit

der spätere Return nicht in die ewigen Jagdgründe führt, untersucht der Interrupt-Handler bei jedem Entfernen einer Unit den gesamten Stack. Auf der Adresse SS:BP-4 ist der Segment-Anteil der Return-Adresse gespeichert. Stimmt dieser mit der alten Ladeadresse der Unit überein, wird der Segmentanteil durch die Adresse des Steuerblockes ersetzt. Zusätzlich wird beim ersten Treffer der Offset-Anteil gesichert und anschließend gelöscht. Die Position dieses Treffers wird ebenfalls im Steuerblock vermerkt. Über das BP-Register wird dann die nächste Verschachtelungsebene untersucht. Dieser Algorithmus macht die Option $F+ notwendig.
Kehrt nun das Programm über einen *retf* zu einem nicht mehr geladenen Overlay zurück, so verweist die Return-Adresse auf die Adresse 0 innerhalb des Steuerblockes. Da dort ebenfalls ein INT 3Fh-Befehl steht, wird der Interrupt-Handler aktiviert und sorgt für die Einlagerung der unterbrochenen Unit.
Jeder Einlagerungsvorgang untersucht ebenfalls den Stack. Findet die Lade-Routine ein Return-Segment, das auf den Steuerblock zeigt, so wird dieses durch den Segment-Anteil der neuen Ladeadresse ersetzt. Ebenso werden natürlich bei Verschiebungen innerhalb des Overlay-Puffers die Return-Segmente entsprechend angepaßt.
Der Bewährungs-Mechanismus funktioniert auf die gleiche Weise: Sobald eine Unit in diesen Bereich verschoben wird, werden die Vektoren auf INT 3Fh gesetzt, die Lade-Adresse im Steuerblock bleibt jedoch erhalten. Der Interrupt-Handler prüft ab der Version 5.5 zuerst, ob noch eine Lade-Adresse vorhanden ist. In diesem Fall werden lediglich die Vektoren wieder auf JMP-Befehle umgesetzt und die Bewährung im Steuerblock vermerkt. Turbo Pascal stellt dazu noch die Zähler *OvrTrapCount* und *OvrLoadCount* zur Verfügung. Im ersten werden alle Interrupts 3Fh gezählt, der zweite gibt Auskunft über die tatsächlichen Ladevorgänge. Die Differenz beider Zähler ergibt die Anzahl Bewährungsaufrufe.
Die Steuersegmente sind mit zwei Zeigerketten untereinander verbunden. Der Anker der ersten Kette ist die Variable *OvrLoadList*, hier sind die Steuersegmente aller geladenen Units verkettet. Die Fortsetzung der Kette befindet sich in den Bytes 14/15 der Steuerblöcke. Die Zeiger haben das Format *word* und enthalten den Segmentanteil der Adressen (Offset ist 0). Die zweite Kette verbindet alle Steuerblöcke über den Anker *OvrCodeList* und die Bytes 0E/0F der Steuersegmente. Die Zeiger haben ebenfalls *word*-Format und enthalten den Segmentanteil der Adressen. Jedoch sind die Zeiger noch nicht relokiert – es muß also das Programmbeginn-Segment (PrefixSeg+$10) aufaddiert werden.

Eigene Laderoutinen

Zum Laden einer Overlay-Unit ruft der Interrupt-Handler die Laderoutine auf. Die Laufzeitbibliothek enthält zwei verschiedene Laderoutinen. Die eine wird von *OvrInit* eingekettet und liest den Code aus der OVR-Datei, die zweite wird durch *OvrInitEMS* aktiviert und sorgt für den Transport aus dem EMS-Speicher. Ab der Version 5.5 ist der Zeiger auf die Laderoutine unter dem Namen *OvrReadFunc* exportiert, dadurch wurde die Möglichkeit geschaffen, eigene Laderoutinen zu realisieren.
Die beiden Beispiele greifen auf die Steuerblöcke zu, deshalb ist die Typ-Vereinbarung als Include-Datei extrahiert (*Listing 1*).
Das erste Beispiel (*Listing 2*) realisiert eine Lade-Statistik für alle Overlay-Units, es ist ein gutes Hilfsmittel für die Programmentwicklung. Anhand dieser Information kann entschieden werden, welche Units sinnvoll überlagerbar gemacht werden und welche besser im festen Teil unterzubringen sind. Die Unit zählt alle Ladeaufrufe in einem freien Feld im Steuerblock und gibt bei Programmende alle Zähler auf dem Bildschirm aus. Zusätzlich exportiert die Unit eine Funktion *ShowLoadCount*, die dem rufenden Overlay den aktuellen Zählerstand meldet; erfolgt der Aufruf nicht aus einer Overlay-Unit, so wird als Fehlermerker der Wert -1 zurückgegeben.
Beachten Sie den Trick, mit dem das Code-Segment der rufenden Routine gelesen wird: Die Funktion definiert eine Tabelle als lokale Variable, diese Tabelle liegt auf Adresse SS:BP-2. Durch den – eigentlich unzulässigen – Index +4 kann auf den Segment-Anteil der Return-Adresse zugegriffen werden. Diese Adressierung kann nicht durch untypisierte Konstanten erfolgen (wie dummy [4]), da der Compiler diesen „Fehler“ bereits bemerkt und die Übersetzung abbricht. Notwendig ist auch der Compilerschalter $R-, um einen Laufzeitfehler bei diesem Zugriff zu unterbinden. Viele AT-Rechner sind heute mit mindestens 1 MByte RAM bestückt, besitzen also extended Memory von 384 KByte oder mehr. Dieser Speicher wird mit dem zweiten Beispiel (*Listing 3*) als Overlay-Zwischenpuffer benutzt und arbeitet dabei ähnlich wie die Standardroutine zur EMS-Unterstützung.
Der Initialisierungsteil arbeitet zunächst die Kette aller Steuerblöcke ab, beginnend bei dem Anker *OvrCodeList* Die Adressen müssen erst noch durch Addition des Prefix-Segmentes aus *PrefixSeg* und 100h Byte (10h Paragraphen) relokiert werden. Die Codelängen aller Overlay-Units werden addiert, um zu sehen, ob das Extended Memory ausreicht.
Danach wird über die gleiche Kette jede Unit einmal geladen und in den erweiterten Speicher gebracht. Die Ladeadresse wird in vier der freien Bytes vermerkt.
Die Laderoutine *OwnOvrRead* ersetzt in diesem Beispiel das Original vollständig. Über den Parameter *OvrSeg* wird vom Laufzeitsystem die Adresse des Steuerblockes übergeben, ebenso hat der Interrupthandler bereits die Ladeadresse eingetragen. Es braucht nur noch der Bereich aus dem erweiterten Speicher hierher kopiert werden. Auch eine Relokierung ist nicht mehr notwendig, da sie bereits während der Initialisierung erfolgte.
Durch dieses Verfahren ergeben sich nur wenige Einschränkungen. Insbesondere funktionieren Sprünge von einem Overlay in ein anderes – im Gegensatz zur Implementierung in der Version 3. Overlay-Units mit Exit-Coding werden korrekt behandelt und auch mit Zeigern auf Prozeduren gibt es keine Probleme.

Von den Standard-Units ist nur die Unit DOS überlagerungsfähig, da die anderen Units nicht mit der Option $O+ übersetzt sind.
Wegen der Manipulationen am Stack müssen alle Routinen des Programms, die direkt oder indirekt überlagerbare Prozeduren aufrufen, mit der Option $F+ (erzeuge far Calls) übersetzt werden. Andernfalls kann die Nachladeroutine Unterbrechungen nicht korrekt erkennen und das Programm verläuft sich im Walde.
Die Prozedur *InitGraph* definiert Variable auf dem Heap und darf deshalb erst nach einer eventuell notwendigen Vergrößerung des Puffers durch *OvrSetBuf* aufgerufen werden. Dies gilt nicht für *OvrInit*, da der Overlay-Puffer bereits von der System-Unit angelegt wird.
Die Overlay-Datei kann nur erzeugt werden, wenn die Option Compile/Destination Disk gesetzt ist; ein Aufruf des Programms aus der integrierten Entwicklungsumgebung heraus ist daher nur möglich, wenn der Schalter gesetzt ist oder das Programm vorab bereits mit Schalter übersetzt wurde.
Units, die eigene Interrupt-Behandlungen machen, dürfen keinesfalls überlagert werden, da sich ihre Adresse laufend verändern könnte, oder sie zum Zeitpunkt des Interrupts gar nicht geladen sind.
Units mit Initialisierungs-Coding müssen schon vor dem ersten Befehl des Hauptprogramms geladen werden. Dieses Problem läßt sich aber umgehen, indem der Aufruf von *OvrInit* selbst in die Initialisierungsroutine einer Unit verlegt wird. Diese Unit darf nicht überlagerbar sein und muß vom Hauptprogramm vor allen Overlay-Units in der Uses-Klausel definiert werden. *Norbert Dohmen*

Literatur:

[1] *Dohmen, N.:* Turbo-Pascal Enzyklopädie, Franzis Verlag, 1990.

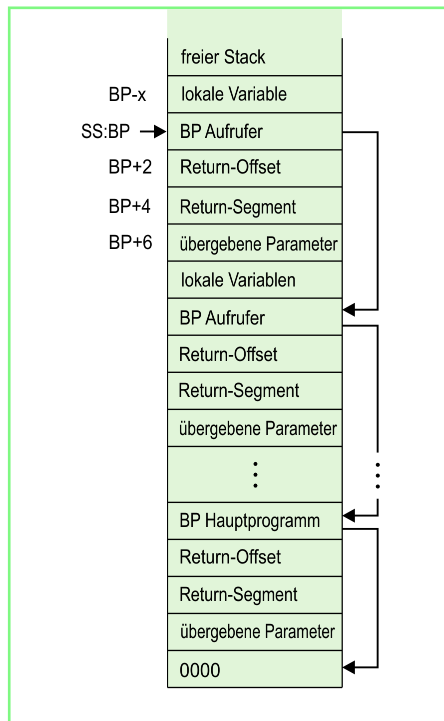


Bild 1. Der Aufbau des Pascal-Stacks

Offset	Inhalt
00-01	CD 3F (Int 3Fh)
02-03	Unterbrechungsadresse
04-07	Offset des Codes in der OVR-Datei
08-09	Länge des Codes
0A-0B	Länge Relokation-Table
0C-0D	Anzahl Vektoren (ab 20h)
0E-0F	CodeList-Kette
10-11	Ladeadresse der Unit
12-13	Stack-Position/Bewährung
14-15	Lade-Kette
16-17	EMS-Seitennummer
18-19	Offset in EMS-Seite
1A-1F	6 Bytes frei
20-25	Vektor Init-Routine (Int 3Fh / Offset / 0 oder JMP FAR / Offset / Segment)
26-2A	Vektor 1. Routine (Int 3Fh / Offset / 0 oder JMP FAR / Offset / Segment)
...	...

Bild 2. Das Overlay-Steuersegment steht anstelle der Unit, die sich in der OVR-Datei befindet, im Programmcode

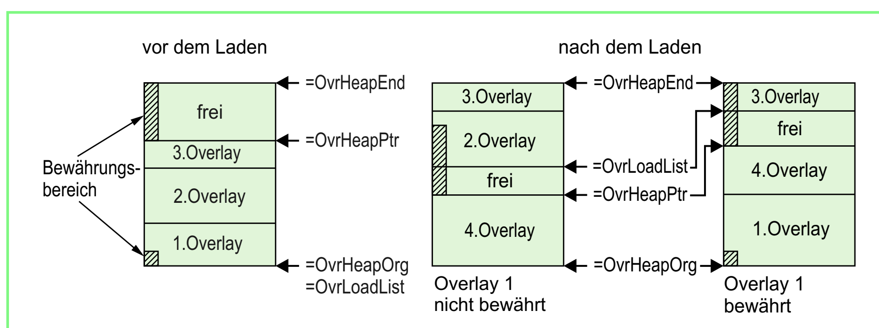


Bild 3. Für das Nachladen einer Overlay-Unit wird, wenn nötig, die älteste Unit entfernt, es sei denn, diese steht im Bewährungsbereich

Listing 1. In einer Include-Datei befinden die Typvereinbarungen für die beiden Pascal-Programme, die Steuerblöcke zugreifen

```
type OvrSteu = record                    {Overlay-Steuer-Block}
     int3f   : integer;                    {"INT 3F"}
     Unterbr : integer;                    {Return-Offset einer Unterbrechung}
     Codeptr : longint;                    {Position in der OVR-Datei}
     Codelen : integer;                    {Länge des Codings}
     relolen : integer;                    {Länge der Relokationstabelle}
     jmpcnt  : integer;                    {Anzahl Sprungvektoren}
     nextcode: word;                       {Kette aller Steuerblöcke}
     loadadr : word;                       {Lade-Segment der Unit}
     stackpos: word;                       {Stackposition einer Unterbrechung}
     nextload: word;                       {Kette Steuerblöcke der geladenen Units}
     emsnr   : integer;                    {EMS-Seitennummer}
     emsoffs : word;                       {Offset in EMS-Seite}
     {die 6 freien Bytes können vom Programm genutzt werden}
     frei1   : integer;
     frei2   : integer;
     frei3   : integer;
     {Sprungvektoren}
     Vektor  : array[0..1] of array [1..4] of byte;
end;
```

Listing 2. Mit diesem Programm werden alle Ladevorgänge erfaßt

```
unit ovstati;

interface

function ShowLoadCount : integer;

implementation

uses overlay;

{$I ovsteu.inc}

var SteuPtr : ^OvrSteu;                  {Steuerblock-Adresse}
    OrgExit : Pointer;                   {Original-Exitroutine}
    OrgLoad : OvrReadFunc;               {Original-Laderoutine}
    OvlName : string[63];                {Name der OVR-Datei
                                          - muß noch individuell gesetzt werden!}
{$F+,R-}

function OwnOvrRead(OvrSeg : Word) : integer;
begin
  SteuPtr := Ptr(OvrSeg, 0);             {Zeiger auf Steuersegment}
  inc(SteuPtr^.frei1);                   {Lade-Vorgang zählen}
  OwnOvrRead := OrgLoad(OvrSeg);         {Original-Laderoutine aufrufen}
end;

function ShowLoadCount : integer;        {Melde Lade-Zähler an Overlay}
var dummy : array[0..0] of word;         {zur Adressierung der Return-Adresse:
                                         {Index -1: function-Return}
                                         {Index -2: BP Aufrufer}
                                         {Index -3: Return-Offset}
                                         {Index -4: Return-Segment}
const i : integer = 4;                   {typisierte Konstante für Tabellenzugriff}
                                         {mit untypisierter Konstanten ist kein Zugriff}
                                         { außerhalb der Tabellengrenzen möglich!}
begin
  dummy[0] := OvrLoadList;               {suche nach rufendem Overlay}
  while dummy[0] <> 0 do
  begin
    SteuPtr := Ptr(dummy[0], 0);         {Zeiger auf Steuersegment}
    if SteuPtr^.LoadAdr = dummy[i] then
    begin                                {Overlay gefunden}
      ShowLoadCount := SteuPtr^.frei1;   {Zähler übergeben}
      exit;                              {und Funktion abbrechen}
    end;
    dummy[0] := SteuPtr^.nextload;       {nächstes geladenes Overlay}
  end;
  ShowLoadCount := -1;                   {Fehler: Aufruf nicht von einem Overlay!}
end;

procedure MeldLoad;
var i : integer;
    w : word;
begin
  ExitProc := OrgExit;
  i := 0;
  w := OvrCodeList;                      {Anker der Overlay-Steuerblöcke}
  while w<>0 do
  begin
    inc(i);
```

```
    SteuPtr := Ptr(w+PrefixSeg+$10, 0); {Zeiger auf Steuerblock}
    writeln('Overlay Nr. ', i:4, ' wurde ', SteuPtr^.frei1:5, ' mal geladen.');
    w := SteuPtr^.nextcode;
  end;
end;

begin
  ovrinit(OvlName);
  OrgLoad := OvrReadBuf;
  OvrReadBuf := OwnOvrRead;
  OrgExit := ExitProc;
  ExitProc := @MeldLoad;
end.
```

Listing 3. Turbo Pascal benutzt von sich aus kein Extended Memory für den Overlay-Puffer, mit diesem Programm ist es möglich

```
unit ovextm;

interface

implementation

uses overlay, dos;

{$I ovsteu.inc}

const Rights = $93000000;               {Recht: lesen und schreiben (Descriptor)}

var MemDesc : record                    {Descriptor-Table für INT15h}
      Dummy   : array[0..7] of byte;
      GDT     : array[0..7] of byte;
      SRClen  : integer;                {Segmentgröße}
      SRCAdr  : longint;                {24-Bit-Adresse Source + Rechte}
      SRCrest : integer;
      DSTlen  : integer;                {Segmentgröße}
      DSTAdr  : longint;                {24-Bit-Adresse Destination + Rechte}
      DSTrest : integer;
      BiosCS  : array[0..7] of byte;
      SS      : array[0..7] of byte;
    end;
var SteuPtr   : ^OvrSteu;               {Steuerblock-Adresse}
    Regs      : Registers;              {Register für BIOS-Aufruf}
    OvlName   : string[63];             {Name der OVR-Datei
                                         - muß noch individuell gesetzt werden!}
{$F+,R-}

function OwnOvrRead(OvrSeg:Word) : integer;
begin
  SteuPtr := Ptr(OvrSeg,0);             {Zeiger auf Steuersegment}
  move(SteuPtr^.frei2,MemDesc.SrcAdr,4); {24-Bit-Adresse im Extended Memory}
  MemDesc.DstAdr :=                     {24-Bit-Adresse im Overlay-Puffer}
    Rights or (longint(SteuPtr^.loadadr) shl 4);
  Regs.CX := SteuPtr^.Codelen;          {zu verschiebende Bytes}
  Regs.ES := Seg(MemDesc);
  Regs.SI := Ofs(MemDesc);
  Regs.AH := $87;
  Intr($15, Regs);                      {INT15/87: Block übertragen}
  OwnOvrRead := Regs.AX;                {Fehlercode als Return-Wert}
end;

procedure InitExtMem;
var w     : word;                       {Zwischenspeicher Steuersegment}
    f     : integer;                    {Ergebnis Ladefunktion}
    OSize : Longint;                    {Größe aller Overlays}
    LAdr  : Longint;                    {Ladeadresse (24 Bit)}
begin
  FillChar(MemDesc, SizeOf(MemDesc), 0); {Descriptor-Table initialisieren}
  MemDesc.SRClen := $7FFF;                {Segment-Größen: 64 KB}
  MemDesc.DSTlen := $7FFF;
  OSize := 0;                           {Berechnung des notwendigen Speichers}
  w := OvrCodeList;                       {Anker der Overlay-Steuerblöcke}
  while w<>0 do
  begin
    SteuPtr := Ptr(w+PrefixSeg+$10, 0);   {Zeiger auf Steuerblock}
    inc(OSize, SteuPtr^.CodeLen);         {Code-Größe addieren}
    w := SteuPtr^.nextcode;               {nächste Unit}
  end;
  OSize := (OSize+$3ff) shr 10;          {Umrechnung in KByte}
  Regs.AH := $88;
  Intr($15,Regs);                         {INT15/88: Größe Extended Memory lesen}
  if Regs.AX < OSize then
  begin                                   {nicht genügend Speicher}
    OvrResult := -6;
    exit;
  end;
```

```
                                        {Overlays in Extended Memory übertragen}
  LAdr := $100000 or Rights;              {Adresse: 1MB plus Recht lesen/schreiben}
  w := OvrCodeList;                       {Anker der Overlay-Steuerblöcke}
  while w<>0 do
  begin
    inc(w, PrefixSeg+$10);                {Umrechnung in Segment-Adresse}
    SteuPtr := Ptr(w, 0);                 {Zeiger auf Steuerblock}
    SteuPtr^.LoadAdr := OvrHeapOrg;       {Ladeadresse eintragen}
    move(LAdr, SteuPtr^.frei2, 4);        {24-Bit-Adresse im Extended Memory merken}
    MemDesc.DstAdr := LAdr;
    MemDesc.SrcAdr :=                     {24-Bit-Adresse im Overlay-Puffer}
      Rights or (longint(OvrHeapOrg) shl 4);
    f := OvrReadBuf(w);                   {Unit laden}
    SteuPtr^.LoadAdr := 0;                {Ladeadresse löschen}
    if f<>0 then
    begin                                 {Lese-Fehler}
      OvrResult := -4;
      exit;
    end;
    Regs.CX := SteuPtr^.CodeLen;          {Unit-Größe}
    Regs.ES := Seg(MemDesc);
    Regs.SI := Ofs(MemDesc);
    Regs.AH := $87;
    Intr($15,Regs);                       {INT15/87: Block übertragen}
    if Regs.AX <> 0 then
    begin                                 {Fehler bei INT15}
      OvrResult := -4;
      exit;
    end;
    inc(LAdr, SteuPtr^.CodeLen);          {Adresse weiterrechnen}
    w := SteuPtr^.nextcode;               {nächste Unit}
  end;
  OvrReadBuf := OwnOvrRead;               {eigene Laderoutine einketten}
end;

begin
  ovrinit(OvlName);
  InitExtMem;
end.
```

aus mc 12/90 – Seite 124-130