KiezSoft

R.-E. Ebert, S. Jensen

info@kiezsoft.de

http://www.kiezsoft.de

DOS-DLL
Dynamische Bibliotheken für MS-DOS

Benutzungsanleitung

Einleitung

Dynamisch ladbare Bibliotheken sind nicht neu. Sie sind seit langem fester Bestandteil vieler Betriebssysteme, wie MS-Windows, OS/2 oder verschiedener UNIX-Derivate. Unter MS-DOS (und kompatiblen Systemen) existiert ein entsprechendes Konzept nicht. Deshalb gibt es auch keine Entwicklungswerkzeuge, die dynamische Bibliotheken unter MS-DOS direkt unterstützen[1].

Wenn man sich heute (im Herbst 2007) mit dynamischen Bibliotheken für MS-DOS beschäftigt, sind zunächst zwei Fragen zu beantworten:

- Welchen Sinn hat es, überhaupt Software für MS-DOS zu entwickeln?
- Welchen Nutzen können dynamische Bibliotheken für MS-DOS bringen?

MS-DOS wird heute vor allem auf Systemen eingesetzt, die direkten Zugriff auf Hardware-Ressourcen benötigen und/oder zeitkritische Aufgaben erfüllen müssen. Darüber hinaus kommt es in Embedded Systems sowie als Start-Betriebssystem für Diagnoseprogramme zum Einsatz[2].

Gemeinsam ist all diesen Einsatzfällen, dass die verfügbaren Ressourcen – sowohl hinsichtlich des vorhandenen Haupt- als auch des Massenspeichers – zumeist eng begrenzt sind. Andererseits kann durchaus der Wunsch bestehen, größere oder auch mehrere Anwendungen auf solch einem System einzusetzen. Dann liegt es nahe, Programmcode, der entweder von mehreren Anwendungen genutzt werden kann oder der während des Programmlaufs nicht ständig benötigt wird, in separate Bibliotheken auszulagern und diese bei Bedarf *dynamisch* in die jeweilige Anwendung einzubinden.

Damit ist auch die Frage nach dem Nutzen solcher Bibliotheken zu beantworten:

- Wiederverwendung nur einmal abgelegter Bibliotheken in mehreren Programmen,
- Möglichkeit des Austauschs von Funktionalität zur Laufzeit, etwa zur dynamischen Anpassung einer Anwendung an unterschiedliche Geräte,
- verbesserte Nutzung des verfügbaren Hauptspeichers, indem Anwendungs-Code nur für die Zeit geladen wird, in der er tatsächlich benötigt wird.

Der in diesem Text beschriebene Weg zur Bereitstellung dynamischer Bibliotheken unter MS-DOS geht auf einen Lösungsvorschlag von Jean-Paul Mikkers aus dem Jahre 1995 zurück[3], aus dem das Laden von Modulen über die Overlay-Schnittstelle von MS-DOS (Interrupt 21H, Funktion 4B03H) sowie der Mechanismus zum Auffinden des Eintrittspunktes im geladenen Modul übernommen wurden. Er geht jedoch weiter, indem ein Verwaltungskonzept für dynamische Bibliotheken analog dem von MS-Windows (bis hin zu den Schnittstellen) bereitgestellt wird, was insbesondere die Entwicklung von Bibliotheken erleichtern soll, die sich sowohl für

[1] Es gibt GNU-C++-Implementierungen für MS-DOS, die Shared Libraries im UNIX-Stil unterstützen, indem sie in ihrem Laufzeitsystem Teile eines UNIX-Kernels mitbringen.. Sie setzen jedoch Protected-Mode-Erweiterungen und mindestens einen i386-Prozessor voraus.

[2] Außerdem und nicht zuletzt auch auf dem berühmt-berüchtigten Atari-Portfolio, der der Ausgangspunkt für die Überlegungen war.

[3] J.-P. Mikkers' Lösung kann z.B. über http://www.programmersheaven.com/download/6424/download.aspx heruntergeladen werden.

MS-DOS als auch für MS-Windows übersetzen lassen. Außerdem erlaubt der hier vorgestellte Ansatz die direkte Verwendung von Funktionen der C/C++-Laufzeitbibliothek in dynamischen Bibliotheken.

Die in diesem Text beschriebene Implementierung bezieht sich auf Turbo- bzw. Borland C++ in den Versionen 1.01 bis 3.1. Eine Verwendung mit anderen Compilern ist prinzipiell möglich, jedoch unter Verzicht auf die Nutzung der C++-Laufzeitbibliothek in den dynamischen Bibliotheken.

Funktionsprinzip

Eine dynamische Bibliothek (DLL) ist ihrer Struktur nach eine ausführbare Datei im DOS-EXE-Format. Im Unterschied zu einem „normalen“ Programm ist jedoch der Einsprungpunkt für die reguläre Programmausführung (die main-Funktion unter C/C++) ohne Bedeutung[4]. Dafür hat eine DLL eine interne Tabelle mit Zuordnungen zwischen den exportierten Objekten (Zeiger auf Funktionen und/oder statische Daten) und Namen, über die diese Objekte von außen ansprechbar sind. Darüber hinaus besitzt sie eine spezielle Einsprungfunktion zur Initialisierung sowie eine Registrierungsstruktur folgender Gestalt (C++-Notation):

```
struct DllRegStruct
{
      char _regTag[DLLREGTAGSIZE+1]; // Tag zur Identifizierung
      DllEntryFunc pEntryFunc;       // Zeiger auf Einsprungfunktion
      DllExpEntry* entryArray;       // Zeiger auf die Exporttabelle
      unsigned entryCount;           // Anzahl der Einträge in der Tabelle
};
```

Ein Programm, das DLLs verwendet, besitzt eine Liste von Registrierungseinträgen geladener DLLs, in der jede DLL mit ihrem vollständigen Dateipfad und einem Zähler für die aktuelle Anzahl der Referenzen auf die Bibliothek verzeichnet ist.

Die Anforderung einer DLL innerhalb des Programms erfolgt durch den Aufruf der Funktion

```
HDLL LoadLibrary(const char* lpFileName);
```

die als Resultat ein Handle auf die geladene DLL zurück liefert.

Fordert das Programm die DLL erneut an, so wird lediglich der Referenzzähler erhöht, gibt es sie frei (mit Funktion `FreeLibrary`), wird der Zähler vermindert. Erreicht ein Referenzzähler den Wert Null, so wird die Bibliothek entladen und der Eintrag entfernt.

Fordert ein Programm eine DLL zum ersten Mal an, so wird ein neuer Listeneintrag erzeugt, der absolute Pfad der DLL bestimmt (sofern nicht bereits angegeben) und die DLL mit Hilfe der DOS-Funktion INT21H/4B03H (load overlay) in den Prozessraum geladen. Anschließend wird mittels Zeichenkettensuche nachdem Tag „D_O_S_L_I_B0“[5] die Adresse der Registrierungsstruktur der DLL bestimmt und deren Einsprungfunktion aufgerufen. Die Einsprungfunktion ist dafür verantwortlich, den Zeiger auf die Exporttabelle[6] sowie deren Größe in die Registrierungsstruktur

[4] Sie ist trotzdem vorhanden und kann verwendet werden, um einen Hinweis der Art „Dies ist eine Bibliothek“ auszugeben oder ein Modul zu realisieren, das sowohl als Bibliothek als auch als eigenständiges Programm lauffähig ist.
[5] Zweifellos ist dies keine besonders elegante Lösung – eine bessere Idee ist jederzeit wilkommen!
[6] Sofern sie nicht statisch initialisiert wurde.

einzutragen sowie ggf. weitere Initialisierungen – z.B. die der Laufzeitbibliothek – vorzunehmen.

War die Anforderung der DLL durch das Programm erfolgreich (d.h. der Rückgabewert von `LoadLibrary` nicht NULL), so kann es nun über eine der Funktionen

```
PDLLPROC GetProcAddress(HDLL hModule, const char* lpProcName); bzw.
PDLLPROC GetEntryAt(HDLL hModule, int index);
```

die Adresse eines exportierten Objekts aus der DLL ermitteln.

Möglichkeiten, Grenzen und Probleme

Export

DLLs, die mit dieser Bibliothek erstellt wurden, können – wie auch die DLLs des Windows-API – zwei Dinge exportieren:

- Zeiger auf Funktionen und
- Zeiger auf statische Daten.

Nicht möglich ist dagegen der direkte Export von Objekten (im Sinne von Klasseninstanzen) oder deren Methoden. Hier bleibt nur die Möglichkeit des Wrappings mit einem Satz ganz normaler C-Funktionen.

Benutzung von Laufzeitbibliotheken

Mit einem Compiler gelieferte Standard-Laufzeitbibliotheken benutzen meist eine Reihe globaler Variablen, für deren Initialisierung der Startup-Code eines Programms sorgt. Im Falle der DLLs wird dieser Code jedoch nicht ausgeführt – es findet ja kein „normaler“ Programmstart statt. Folglich muss eine DLL, die Laufzeitbibliotheken nutzen will, selbst für die entsprechenden Initialisierungen sorgen. Das Problem besteht hierbei darin, dass diese Initialisierungen vom jeweils verwendeten Compiler abhängen und im Allgemeinen nicht dokumentiert sind.

Die DOSDLL-Bibliothek versucht diesem Problem Rechnung zu tragen, indem sie in einem separaten Header (dllglob.h) eine Compiler-abhängige Struktur für die Übertragung globaler Initialisierungsdaten zwischen Hauptprogramm und DLL sowie Makros zum Füllen und Auslesen dieser Struktur bereitstellt. Eine konkrete Implementierung existiert derzeit nur für Turbo-/BorlandC++ in den Versionen 1.01 bis 3.1. Für andere Compiler wird nur eine leere Dummy-Struktur erzeugt, so dass die Benutzung der Laufzeitbibliotheken dort nicht möglich ist.

Ein weiteres Problem besteht in der dynamischen Belegung von Speicher innerhalb einer DLL. Üblicherweise verwenden C/C++-Compiler eine eigene Speicherverwaltung zur Bereitstellung der Funktionalität von malloc/free (und verwandten Funktionen), die auf eine Reihe interner Variablen zurückgreift. Eine DLL besitzt einen eigenen Satz dieser Variablen, so dass das Hauptprogramm von Speicheranforderungen und –freigaben innerhalb der DLL nichts mitbekommt. Die Folge davon ist, dass entsprechende Aufrufe die Speicherverwaltung des Hauptprogramms durcheinanderbringen können.

Um hier Abhilfe zu schaffen, stellt DOS-DLL drei zusätzliche Funktionen bereit, die (sofern man eines der Standard-Makros für die Implementierung der DLL-Entrypoint-Funktion benutzt) automatisch in jeder DLL bereitgestellt werden[7]:

- GetMem : fordert einen Speicherblock an
- FreeMem : gibt einen Speicherblock frei
- Spawn : führt ein anderes Programm als Kindprozess aus

Die letzte dieser drei Funktionen ist erforderlich, da die spawn/exec-Funktionen der C-Bibliothek implizit Speicheranforderungen und –freigaben durchführen.

Eine weitere häufig vorhandene (aber nicht standardisierte) Bibliotheksfunktion, die implizit Speicher anfordert, ist `strdup`. Sie sollte in DLLs niemals verwendet werden!

Ebenfalls innerhalb einer DLL nicht benutzt sollten direkte Aufrufe von DOS-Funktionen zur Speicheranforderung werden. Wird durch einen solchen Aufruf ein Speicherblock direkt oberhalb des Heaps des Hauptprogramms belegt, so kann dessen Heap-Block nicht mehr vergrößert werden, was zu Fehlern bei der Speicheranforderung im Hauptprogramm führen kann.

Speichermodell

Hauptprogramm und DLLs müssen immer im EXE-Format vorliegen und mit dem LARGE-Speichermodell übersetzt werden. Außerdem darf in aufgerufenen Funktionen nicht von der Annahme DS=SS ausgegangen werden[8].

DLLs dürfen nicht komprimiert werden!

Die gerade bei knappem Platz gern praktizierte Kompression ausführbarer Dateien (z.B. mit LZEXE oder PKLITE) ist für DLLs nicht möglich. Der Grund ist einfach: Es wird kein normaler Programmstart ausgeführt und deshalb findet auch keine Dekompression statt.

Overhead

Erstellt man (z.B. mit Turbo-C++) eine DLL, so weiß der Compiler natürlich nichts davon und fügt den Startup-Code einer normalen Anwendung ein – auch wenn dieser niemals ausgeführt wird. Daraus ergibt sich ein gewisser Overhead – bei Turbo-C++ sind es etwa 4KB – der ohne größere Tricks nicht vermeidbar ist.

Gleitpunkt-Arithmetik

Benutzt eine Anwendung Gleitpunkt-Arithmetik (also die Datentypen float bzw. double), so müssen normalerweise sowohl das Hauptprogramm als auch alle verwendeten DLLs mit identischen Einstellungen für die Gleitpunkt-Unterstützung übersetzt werden. Dies führt zu zusätzlichem Overhead, insbesondere, wenn die Koprozessor-Emulation verwendet wird (hier sind es knapp 20KB je Modul). Um dieses Problem zu lindern, besteht die Möglichkeit, einzelne DLLs ohne Gleitpunkt-Unterstützung zu erstellen. In diesem Fall muss bei der Übersetzung das Symbol _NOFLOAT_ definiert sein.

[7] Siehe Refrenz
[8] Bitte ggf. bei den Compiler-Einstellungen beachten.

Ein einfaches Anwendungsbeispiel

In diesem Abschnitt wird ein triviales Beispiel für die Verwendung der Bibliothek gezeigt. Es hat keinen praktische Nutzen, sondern soll lediglich die Implementierung in C++ veranschaulichen.

Die DLL

Die hier implementierte DLL exportiert lediglich zwei einfache „Sag-Hallo"-Funktionen, von denen eine einen Parameter übernimmt[9]:

```
// TESTDLL.CPP

#define _NOFLOAT_

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "dosdll.h"

void huge sayHello()
{
      char* s = "Hello from testdll\n";
      puts(s);
}

void huge sayHello1(const char* name)
{
   printf("Hello from testdll to %s!\n",name);
}

BEGIN_EXPTABLE
   EXPENTRY(sayHello),
   EXPENTRY2("sayHello(const char*)",sayHello1)
END_EXPTABLE

IMPLEMENT_DEFAULT_ENTRYFUNC

REGISTERENTRYFUNC

int main(int argc, char* argv[])
{
      puts("Can't run standalone.\n");
      return -1;
}
```

Wie man sieht, unterscheidet sich die Implementierung nur wenig von der eines „normalen" Programms.

Die huge-Definition der Funktionen legt deren Adressen als FAR-Zeiger mit Segment-Normalisierung fest (letzeres ist eher eine Vorsichtsmaßnahme - vgl. Turbo-C++-Hilfe).

Die verwendeten Makros sind in der importierten Header-Datei „dosdll.h" definiert und verbergen die Implementierung der für die DLL nötigen Hilfskonstruktionen[10].

Die Makros `BEGIN_EXPTABLE` und `END_EXPTABLE` implementieren den Rahmen der Exporttabelle. Mit den Makros `EXPENTRY` bzw. `EXPENTRY2` werden die einzelnen

[9] Sie ist eine „abgespeckte" Variante des mitgelieferten Beispiels.
[10] Vgl. hierzu die Referenz

Exporte in die Tabelle eingetragen. Bei der einfachen Variante entspricht der Name in der Exporttabelle dem internen Namen, während die erweiterte Variante die Angabe eines beliebigen Bezeichners erlaubt. Dies lässt sich beispielsweise – wie hier gezeigt – nutzen, um die vollständige Signatur einer Funktion nach außen sichtbar zu machen.

Die beiden verbleibenden Makros tun das, was ihr Name vermuten lässt. Das erste implementiert eine (wohl fast immer ausreichende) Standardvariante der Eintrittsfunktion der DLL und das zweite registriert diese Funktion.

Für die Implementierung einer DLL ist wichtig, dass die Makros in der hier angegebenen Reihenfolge verwendet werden.

Die `main`-Funktion signalisiert (im Fall des versehentlichen direkten Aufrufs) lediglich, dass es sich hier um kein richtiges Programm handelt.

Das Hauptprogramm

... zeigt die Verwendung der DLL:

```
#include <stdio.h>
#include "usedll.h"

typedef void (* FUNC0)();
typedef void (* FUNC1)(const char*);

int huge enumExpFunc(const char* name, PDLLPROC pExpEntry)
{
      printf("%s\n",name);
      return 1;
}

int main(int argc, char* argv[])
{
    HDLL hDll = LoadLibrary("testdll.exe");
    if (hDll == 0)
    {
       printf("error loding library\n");
       return 1;
    }
    FUNC0 sayHello = (FUNC0) GetProcAddress(hDll,"sayHello");
    if (sayHello == 0)
      printf("sayHello not found\n");
    else
      sayHello();
    FUNC1 sayHello2 = (FUNC1) GetProcAddress(hDll,"sayHello(const char*)");
    if (sayHello2 == 0)
       printf("sayHello(const char*) not found\n");
    else
       sayHello2("DllApp");
    printf("\nNow enumerate exports:\n");
    EnumExports(hDll,&enumExpFunc);
    if (FreeLibrary(hDll) < 0)
    {
       printf("Unloading library failed!\n");
       return 2;
    }
    return 0;
}
```

Hier wird zunächst die Header-Datei „`usedll.h`“ importiert, die die Datentypen und Funktionen der DLL-Schnittstelle deklariert – siehe Referenz.

Danach folgen die Definitionen der Prototypen für die aufzurufenden DLL-Funktionen. Dies ist für einen korrekten Aufruf erforderlich.

Die anschließend definierte Funktion `enumExpFunc` ist ein Callback, der weiter unten verwendet wird, um die von der DLL exportierten Funktionen aufzulisten[11].

Die in der `main`-Funktion verwendeten Aufrufe folgen dem selben Muster, wie man es von Windows kennt:

- Eine DLL wird mit der Funktion `LoadLibrary` geladen, die als Argument den Namen der DLL (mit Erweiterung!) übergeben bekommt. Die Bibliothek wird zunächst im aktuellen Verzeichnis und dann entlang der Pfade der PATH-Variablen gesucht. War das Laden erfolgreich, wird ein Handle auf die Bibliothek zurückgegeben, das für weitere Aufrufe als Argument benötigt wird.
- Mit der Funktion GetProcAddress bekommt man einen Zeiger auf die exportierte Funktion, der explizit auf den korrekten Typ „gecastet“ werden muss. Danach kann die DLL-Funktion „ganz normal“ aufgerufen werden.
- Die Freigabe der DLL erfolgt durch den Aufruf der Funktion `FreeLibrary`.

Die Funktion `EnumExports` hat keine Entsprechung im Windows-API. Sie ruft einen Callback für alle exportierten Einträge einer DLL auf. Dies lässt sich – wie im Beispiel – zu Informationszwecken verwenden, erlaubt es aber auch, in einem einzigen Aufruf, die Adressen aller exportierten Objekte zu bestimmen.

[11] Das ist das gleiche Prinzip, wie es auch bei den EnumXXX-Funktionen des Windows-API verwendet wird.

Hinweise zur Implementierung

Nachfolgend werden zusammenfassend einige Hinweise gegeben, die helfen sollen, Fallen bei der Implementierung von DLL-Anwendungen zu umgehen. Die Zusammenstellung erhebt allerdings keinen Anspruch auf Vollständigkeit.

- Die Implementierung der DLL-Schnittstelle (Datei „usedll.cpp“) wird vom Hauptprogramm benötigt und muss als C++ übersetzt werden. Alle Header-Dateien sind Standard-C-kompatibel. Das bedeutet, dass es prinzipiell möglich ist, die „usedll.cpp“ mit Turbo-C++ in eine Objektdatei zu kompilieren und gegen eine in Turbo-C 2.0 geschriebene Anwendung zu linken.
- Programme, die DLLs verwenden, sollten defensiv angelegt sein, d.h. mögliche Fehler einkalkulieren und entsprechende Prüfungen durchführen.
- Alle zwischen Hauptprogramm und DLL ausgetauschten Zeiger sind FAR-Zeiger, alle beteiligten Module verwenden das LARGE-Speichermodell.
- Alle Funktionsaufrufe erfolgen nach C-Konvention. Dies ist insbesondere zu berücksichtigen, wenn DLLs in anderen Sprachen implementiert werden sollen und heißt konkret:
    - Funktionsargumente werden in umgekehrter Reihenfolge auf den Stack gelegt, das bedeutet, das erste Argument liegt zuoberst, also direkt unter der Rücksprungadresse.
    - Der Aufrufer ist für das Abräumen des Stacks zuständig.
    - Funktionsergebnisse werden in AL (8 Bit), AX (16 Bit) bzw. DX:AX (32 Bit, Zeiger) zurückgegeben.
- Funktionen in DLLs benutzen den Stack des aufrufenden Programms. Sie sollten keine Annahmen über die Belegung anderer Segment-Register machen.
- Funktionen in DLLs können die Register AX, BX, CX, DX, DI und SI frei verwenden. Alle anderen Register sollten gesichert und vor dem Rücksprung wieder hergestellt werden.
- In Turbo-/Borland-C/C++ geschriebene DLLs, die über Bibliotheksfunktionen (z.B. `getenv`, `searchpath`) auf Umgebungsvariablen zugreifen wollen, müssen in ihrer Implementierung statt des Makros IMPLEMENT_DEFAULT_ENTRYFUNC dessen erweiterte Form IMPLEMENT_DEFAULT_ENTRYFUNC_ENV verwenden. Dadurch wird für die korrekte Initialisierung der globalen Variablen `environ` gesorgt (und etwas mehr Speicherplatz benötigt).

Erstellen von DLLs mit Assembler

Bei knappen Ressourcen, besonders zeitkritischen oder hardwarenahen Funktionen bietet sich oft die Nutzung der Assembler-Sprache an. Selbstverständlich ist es auch möglich, DLLs mit Assembler zu implementieren – sofern sie die vom Aufrufer geforderten Konventionen erfüllen. Naturgemäß ist hierfür etwas mehr Handarbeit erforderlich.

In diesem Abschnitt wird die Implementierung einer DLL in Assembler vorgestellt, die die aus dem obigen Beispiel ersetzt, d.h. direkt an deren Stelle verwendet werden kann. Der hier folgende Quelltext wurde mit A86 (Version 3.22) übersetzt und mit TLINK gebunden[12].

```
DSEG SEGMENT 'DATA'
; some messages
msg DB "Can't run standalone.",0DH,0AH,'$'
msg1 DB "Hello from testdll.asm.",0DH,0AH,'$'
msg2 DB "testdll.asm says hello to ",'$'
msg2end DB ".",0DH,0AH,'$'

;names of exported functions
fname0 DB 'sayHello',0
fname1 DB 'sayHello(const char*)',0

;export table
;first entry
funcEntry0 DW OFFSET fname0
           DW SEG DSEG
           DW func0
           DW SEG CSEG
;second entry
funcEntry1 DW OFFSET fname1
           DW SEG DSEG
           DW func1
           DW SEG CSEG

;DllRegStruct - the registration structure
regTag DB 'D_O_S_L_I_B0'    ; tag searched from main program
oEntryFunc DW OFFSET entryFunc  ; offset of entry point function
sEntryFunc DW SEG CSEG          ; segment of entry point function
oEntryArray DW funcEntry0       ; address of export table
sEntryArray DW SEG DSEG
entryCnt    DW 2                ; number of entries in export table
DSEG ENDS

CSEG SEGMENT 'CODE'
; dummy entry function if started from command line
DUMMY PROC FAR
     MOV AX, DSEG
     MOV DS, AX
     XOR AX, AX
     LEA DX, msg
     CALL PUTS
     MOV AX, 4c00h
     INT 21h
     RET
DUMMY ENDP
```

[12] ... und erhebt keinen Anspruch darauf, die höchste Kunst der Assembler-Programmierung zu sein

```
; Displays the message (DX-address)
PUTS    PROC    NEAR
        PUSH    AX
        MOV     AH, 09h
        INT     21h
        POP     AX
        RET
PUTS    ENDP

; Entry point function
; It does nothing because all initialization was made static
entryFunc PROC FAR
       RET
entryFunc ENDP

; implementation of first exported function sayHello
func0 PROC FAR
     PUSH DS
     MOV AX, DSEG
     MOV DS, AX
     LEA DX, msg1
     CALL PUTS
     POP DS
     RET
func0 ENDP

; implementation of second exported function sayHello(const char*)
func1 PROC FAR
     PUSH DS
     MOV AX, DSEG
     MOV DS, AX
     LEA DX, msg2
     CALL PUTS
     ; get address of C-string from stack
     MOV SI, SP
     MOV AX, SS:[SI+8] ; Segment of string parameter
     MOV ES, AX
     MOV DI, SS:[SI+6] ; Offset of string parameter
     MOV AH, 2 ; DOS Display output
  printchar:
     MOV DL, ES:[DI]
     CMP DL, 0
     JE endprint ; C-string ends with a zero byte.
     INT 21H
     INC DI, 1
     JMP printchar
  endprint:
     LEA DX, msg2end
     CALL PUTS
     POP DS
     RET
func1 ENDP


CSEG ENDS
     END DUMMY
```

Zu erklären ist hier die notwendige Handarbeit.

Im Datensegment (hinter dem `DllRegStruct`-Kommentar) steht die Registrierungsstruktur. Sie beginnt mit dem Tag „D_O_S_L_I_B0“, das beim Laden zum Auffinden der Struktur benötigt wird. Ihm folgen die (FAR !) Adressen der Einsprungfunktion und der Exporttabelle sowie das Feld mit der Anzahl der Einträge in der Exporttabelle. Die Initialisierung der Struktur und der Exporttabelle erfolgt statisch. Deshalb ist die Einsprungfunktion `entryFunc` leer. Sie könnte verwendet werden, um ggf. notwendige weitere Initialisierungen vorzunehmen.

Wichtig ist außerdem, dass alle extern aufrufbaren Funktionen als FAR deklariert werden. Für nur intern verwendete (siehe PUTS) spielt das keine Rolle.

Interessant ist vielleicht noch der Größenvergleich mit der C++-Variante. Während eine vergleichbare DLL in C++ etwa 8 KB benötigt, kommt die Assembler-Variante mit knapp 800 Bytes aus – sie benutzt eben keine Laufzeitbibliotheken und schleppt auch sonst keinen zusätzlichen Code mit sich herum. Man sollte mit der Bewertung aber vorsichtig sein: Bei größeren Programmen/Bibliotheken relativiert sich der Unterschied in dem Maße, wie der Anteil des eigenen Codes gegenüber dem der Laufzeitbibliotheken wächst. Außerdem bleibt es einem auch mit Assembler nicht erspart, die eine oder andere Hilfsfunktion zu schreiben – oder eben auch hier auf Bibliotheken zurückzugreifen. Ob es sinnvoller ist, eine DLL mit Assembler oder in einer Hochsprache zu schreiben, hängt also vom konkreten Anwendungsfall ab.