Deutsch

Fujitsu Software BS2000

POSIX
Laden von Shared Objects

Benutzerhandbuch

Stand der Beschreibung:
POSIX A49
BS2000 V21.0B

Ausgabe August 2025

Kritik… Anregungen… Korrekturen…

Die Redaktion ist interessiert an Ihren Kommentaren zu diesem Handbuch. Ihre Rückmeldungen helfen uns, die Dokumentation zu optimieren und auf Ihre Wünsche und Bedürfnisse abzustimmen.

Sie können uns Ihre Kommentare per E-Mail an bs2000.info@fujitsu.com senden.

Zertifizierte Dokumentation nach DIN EN ISO 9001:2015

Um eine gleichbleibend hohe Qualität und Anwenderfreundlichkeit zu gewährleisten, wurde diese Dokumentation nach den Vorgaben eines Qualitätsmanagementsystems erstellt, welches die Forderungen der DIN EN ISO 9001: 2015 erfüllt.

Copyright und Handelsmarken

Inhaltsverzeichnis

POSIX_sharedObjects_de

1 Allgemeine Bemerkungen

Dieses Dokument beschreibt die Schnittstellen für Shared Objects, die in Verbindung mit POSIX im BS2000 verwendet werden. Shared Objects (SO) sind gemeinsam nutzbare Objekte.

> **Beachten Sie bitte die folgende Einschränkung:**
>
> Die Summe aus der Anzahl der Kontexte des geladenen Programms (normalerweise ein Kontext) und der Anzahl der Kontexte der geladenen Objekte ist auf 192 begrenzt. Dies hängt damit zusammen, dass nur 200 Kontexte des Binder-Lader-Systems tatsächlich verfügbar sind.

1.1 Definition eines Shared Object

Bisher wurden von POSIX die Elementarten unterstützt, die vom Compiler oder durch die Compilerausgabe generiert werden:

`a.out`	Ausführbare Datei (Typ X)
`file.o`	Vom Compiler generiertes Objekt (Typ O)
`libx.a`	ar-Bibliothek (Typ AR; enthält normalerweise die Typen X und O)

Nun werden auch so genannte "Shared Objects" unterstützt. Die Bezeichnung "Shared Object" wurde aus UNIX abgeleitet. Alternative Bezeichnungen sind auch "dynamische Bibliothek" oder im Zusammenhang mit UNIX "Shared Library".

Unter einem Shared Object versteht man in POSIX immer folgendes Objekt:

`object.so`	so-Element (Typ SO; erstellt aus den Typen X, O, AR und SO selbst, sowie aus PLAM-Bibliotheken)

Beim Generieren eines Shared Object können Bibliotheken (Typ SO und Typ AR sowie Typ PLAM) sowie Dateien mit der Namens-Endung ".o" angegeben werden. Um ausführbare Dateien bei der Generierung verwenden zu können (Objekttyp X), müssen diese entsprechend umbenannt werden (siehe "genso - Shared Object erzeugen").

Unterstützung von Shared Objects in POSIX

Shared Objects werden in POSIX in der Weise unterstützt, dass Objekte zur Laufzeit mittels Overlaytechnik explizit nachgeladen werden können (programmgesteuertes Nachladen). Dazu stehen die Funktionen *dlopen()*, *dlclose()*, *dlsym()*, *dlerror()* und *dladdr()* zur Verfügung, siehe Kapitel 2.

Ein automatisches Nachladen wie im klassischen BS2000 ist in POSIX nur für Module des Laufzeitsystems CRTE möglich.

Nicht unterstützt wird die in UNIX angebotene Funktionalität, dass Objekte zur Laufzeit automatisch dazugebunden werden können und dass Shared Coding von mehreren Programmen in einem gemeinsamen Speicherbereich verwendet werden kann.

Aufbau eines Shared Object

Physikalisch handelt es sich bei einem Shared Object um eine ar-Bibliothek, die Objektmodule und eine Beschreibung des Shared Object enthält (= Shared object description). Die Beschreibung des Shared Object ist eine Textdatei in der ar-Bibliothek. Diese Textdatei wird beim Erzeugen des Shared Object (per Kommando *genso*) generiert.

Diese Datei wird während des *dlopen()*-Aufrufs wie folgt verarbeitet:

- Wenn keine abhängigen Bibliotheken gefunden werden, werden die o-Dateien wie in der Datei angegeben geladen.
- Wenn ein abhängiges Shared Object vorhanden ist, wird eine neue Liste erzeugt. Das abhängige Shared Object wird an das Ende dieser Liste gehängt.
- Anschließend wird erneut nach dem ersten abhängigen Objekt in der neuen Liste gesucht. Dabei kann es sich um ein abhängiges Objekt des ersten abhängigen Objekts handeln.

Diese Verfahren wird wiederholt, bis alle abhängigen Objekte (und die abhängigen Objekte dieser Objekte usw.) durch o-Dateien ersetzt sind. Überprüfungen vermeiden, dass die Liste unendlich fortgesetzt wird. Die o-Dateien in der Liste werden anschließend wie in der Liste angegeben geladen. Diese Auflösung einer so-Datei wird in der Beschreibung von *dlopen()* als **Dependency ordering** bezeichnet (= Abhängigkeitsreihenfolge).

ar-Dateien werden auf die gleiche Weise verwaltet wie o-Dateien, da eine ar-Datei als eine geordnete Liste von o-Dateien betrachtet wird.

Beispiel

Hier ist ein typisches Beispiel für eine solche Beschreibung. Diese Beschreibung gehört zum Beispiel 1 beim Kommando *genso* (siehe "genso - Shared Object erzeugen").

Shared object description für die Datei *libtest21.so*:

```
DLL? /posix315/bachmann/sharedlib/examples/libtest21.so
ofile_GM_.o
##### /posix315/bachmann/sharedlib/examples/test21.o
###SO### /posix315/bachmann/sharedlib/examples/libtest22.so
libtest22.so
###SO### /posix315/bachmann/sharedlib/examples/libtest23.so
libtest23.so
-X lang=c
```

Shared object description für Datei *libtest22.so*:

```
DLL? /posix315/bachmann/sharedlib/examples/libtest22.so
ofile_GM_.o
##### /posix315/bachmann/sharedlib/examples/test22.o
###SO### /posix315/bachmann/sharedlib/examples/libtest24.so
libtest24.so
-X lang=c
```

Shared object description für Datei *libtest23.so*:

```
DLL? /posix315/bachmann/sharedlib/examples/libtest23.so
ofile_GM_.o
##### /posix315/bachmann/sharedlib/examples/test23.o
-X lang=c
```

Shared object description für Datei *libtest24.so*:

```
DLL? /posix315/bachmann/sharedlib/examples/libtest24.so
ofile_GM_.o
##### /posix315/bachmann/sharedlib/examples/test24.o
-X lang=c
```

Die Elemente dieses Shared Object sehen bei der Bearbeitung durch *dlopen()* wie folgt aus.

Objekt *libtest21.so*:

```
ofile_GM_.o (libtest21.so)
libtest22.so
libtest23.so
```

Objekt *libtest22.so*:

```
ofile_GM_.o (libtest22.so)
libtest24.so
```

Objekt *libtest23.so*:

```
ofile_GM_.o (libtest23.so)
```

Objekt *libtest24.so*:

```
ofile_GM_.o (libtest24.so)
```

In einem ersten Schritt wird in der Beschreibung von *libtest21.so* die Referenz auf *libtest22.so* durch deren Beschreibung ersetzt. Dadurch ergibt sich folgende Anordnung:

```
ofile_GM_.o (libtest21.so)
ofile_GM_.o (libtest22.so)
libtest24.so
libtest23.so
```

Dieses Verfahren wird wiederholt, bis sich am Schluss Folgendes ergibt:

```
ofile_GM_.o (libtest21.so)
ofile_GM_.o (libtest22.so)
ofile_GM_.o (libtest24.so)
ofile_GM_.o (libtest23.so)
```

In dieser Reihenfolge werden die Elemente dann geladen.

1.2 Erzeugen eines Shared Object

Ein Shared Object wird mit Hilfe des Kommandos *genso* erzeugt. Die Optionen und Parameter entsprechen weitgehend denen des Kommandos *cc*.

-B plam ... -B ar

Innerhalb dieser Klammer werden die Optionen *–L* und *-l* so interpretiert, dass PLAM-Bibliotheken verarbeitet werden können, zusätzlich wird die neue Option *-m* interpretiert.

-B static
-B dynamic

Gibt an, ob ar-Bibliotheken oder dynamische Bibliotheken zum Generieren des Shared Object verwendet werden sollen.

Verwendung nur beim Generieren.

-B symbolic

Gibt den Auflösungsalgorithmus an.

-**L** verzeichnis

Gibt das Verzeichnis an.

-**L** bs2000_user_id

Gilt nur zwischen *–B plam* und *–B ar* und gibt die BS2000-Benutzerkennung an:

$

steht für TSOS.

.

(Punkt) steht für die Benutzerkennung, unter der die Anwendung abläuft.

%name

die Benutzerkennung wird der Umgebungsvariablen *name* entnommen.

-l xxx

Gibt die Bibliothek an: libxxx.a oder libxxx.so

Wenn *-l xxx* zwischen *–B plam* und *–B ar* angegeben wird, dann ist *xxx* der Name der BS2000-Bibliothek.

-m member

Gilt nur zwischen *–B plam* und *–B ar* und gibt den Namen des Elements der PLAM-Bibliothek an (L-Element).

-o output

Gibt die Ausgabe an.

-S low | **high**

Der Inhalt des Shared Object wird auf *stdout* ausgegeben; im Falle von *low* ist dies nur das aktuelle Objekt, im Falle von *high* zusätzlich noch alle davon abhängigen Shared Objects.

-X lang=c | **lang=c++** | **lang=cobol**

Gibt die Programmiersprache des nachzuladenden Objekts an. Mischungen sind möglich.

Standardwert ist *lang=c*.

file**.o**

Beim Generieren verwendete o-Datei.

Andere geläufige UNIX-Optionen wie *-h name*, *-Kpic* und *-b* werden nicht unterstützt.

Das Kommando *genso* erzeugt die so-Datei. Wenn die Option *-B symbolic* nicht angegeben wird, dann werden bereits zum Generierungszeitpunkt die Externverweise so weit wie möglich aufgelöst. Die Objekte in der so-Datei werden dabei zu einem Großmodul gebunden. Beim Laden erhält das Shared Object selbst die höchste Priorität beim Auflösen der Verweise.

Prioritätsregeln für die Suche nach einer Bibliothek

Für die Suche nach Bibliotheksverzeichnissen gelten folgende Prioritäten:

1. Verzeichnisse, die in der Variablen LD_LIBRARY_PATH enthalten sind
2. Verzeichnisse, die über die Option *-L* definiert sind
3. das Standard-Verzeichnis */usr/lib*

1.3 Funktionen für Shared Objects

Für Shared Objects können folgende Funktionen ausgeführt werden:

Name der Funktion	**Bedeutung**
`dlopen()`[1]	Ein Shared Object öffnen
`dlclose()`	Ein Shared Object schließen
`dlsym()`[1]	Adressen in einem Shared Object ermitteln
`dladdr()`[1]	"Nächstgelegenen" Symbolnamen für eine bestimmte Adresse in einem Shared Object ermitteln
`dlerror()`[1]	Diagnoseinformationen zu einer vorangegangenen, fehlerhaft ausgeführten Funktion ausgeben

[1] Für den Aufruf in einer ASCII-Umgebung stehen die Funktionen *__dlopen_ascii()*, *__dlsym_ascii()*, *__dladdr_ascii()* und *__dlerror_ascii()* zur Verfügung.

1.4 Installation

Die für die Unterstützung von Shared Objects benötigten Komponenten werden automatisch bei einer Erst-Installation oder einer Delta-Installation von POSIX-BC eingerichtet.

Die einzelnen Komponenten werden dabei in folgenden Verzeichnissen abgelegt:

Komponente	**Installationsverzeichnis**	**Typ**
`libdl.a`	`/usr/lib`	ar-Bibliothek
`genso`	`/usr/bin`	Kommando
`dlfcn.h`	`/usr/include`	Header-Dadtei

Beachten Sie, dass es sich bei *libdl.a* nicht um eine Shared Library, sondern um eine ar-Bibliothek handelt.

Diese Bibliothek muss statisch zu dem Programm gebunden sein, das die oben aufgeführte Schnittstelle verwendet, z.B. durch:

```
cc -o prog prog.c -ldl.
```

2 dll-Schnittstellen in BS2000

Dieses Kapitel enthält die Beschreibung des Kommandos *genso*, der Funktionen *dlopen()*, *dlclose()*, *dlsym()*, *dlerror()* und *dladdr()* sowie des Headers *dlfcn.h*.

2.1 genso - Shared Object erzeugen

Mit Hilfe des Kommandos *genso* werden Shared Objects erzeugt.

Syntax

```
/usr/bin/genso  [optionen] [dateien]
```

Optionen

genso unterstützt einige der für den POSIX-C-Compiler verwendeten Optionen; andere Optionen wurden von dem auf UNIX-Systemen verwendeten Kommando *ld* abgeleitet.

-L dir

Das Verzeichnis *dir* wird in die Liste mit Verzeichnissen aufgenommen, in denen *genso* nach Bibliotheken sucht. Die Option *-L* muss immer **vor** der Option *-l* angegeben werden und gilt bis zur nächsten Angabe von *-L*.

-L bs2000_user_id

Nur in Verbindung mit *–B plam* : Name der BS2000-Benutzerkennung.

Folgende Angabe sind möglich:

$

steht für TSOS

.

(Punkt) steht für die Benutzerkennung, unter der die Anwendung abläuft

%name

wenn explizit eine BS2000-Benutzerkennung angegeben werden soll. Die Benutzerkennung muss dann in die Umgebungsvariable *name* geschrieben werden.

-l xxx

libxxx.a oder *libxxx.so* wird zum Erzeugen eines Shared Object verwendet. Welche Art von Bibliothek verwendet wird, hängt davon ab, ob *-B static* oder *-B dynamic* verwendet wird. Wenn keine Angabe gemacht wird, haben dynamische Bibliotheken Vorrang.

-l plam_bibliothek

Nur in Verbindung mit *–B plam* : Name der BS2000-PLAM-Bibliothek.

-m member

Nur in Verbindung mit *–B plam* : Name des zu verarbeitenden Elements der BS2000-PLAM-Bibliothek.

-o ausgabe

Name des Shared Object, das erzeugt werden soll. Das Shared Object wird unter diesem Namen im aktuellen Verzeichnis abgelegt. Der Dateiname sollte auf *.so* enden; er wird nicht automatisch auf .so ergänzt. Um Missverständnisse zu vermeiden, werden die Endungen *.o* und *.a* abgewiesen.

-B plam

Von nun an werden PLAM-Bibliotheken verarbeitet. Zum Umschalten auf ar- oder so-Bibliotheken muss *–B ar* angegeben werden.

-B ar

Wenn PLAM-Bibliotheken verarbeitet wurden, so wird nun auf ar-Bibliotheken bzw. so-Bibliotheken umgeschaltet.

-S low | **-S high**

Der Inhalt des Shared Object wird auf *stdout* ausgegeben.

Ist der Dateiname kein absoluter oder relativer Pfad, dann wird die Datei in den mit der Umgebungsvariablen LD_LIBRARY_PATH spezifizierten Verzeichnissen oder in */usr/lib* gesucht. Bei der Anwendung auf eine Datei im aktuellen Verzeichnis ist also gegebenenfalls ein ./ voranzustellen, falls dieses Verzeichnis nicht in der Umgebungsvariablen LD_LIBRARY_PATH spezifiziert ist.

-S low gibt nur den Inhalt des aktuellen Shared Object aus.

-S high gibt zusätzlich zum Inhalt des aktuellen Shared Object noch den Inhalt aller abhängigen Objekte aus.

-X lang=c | **lang=c++** | **lang=cobol**

Gibt die Programmiersprache des nachzuladenden Objekts an. Mischungen sind möglich.

Standardwert ist *lang=c* .

-B static

Wenn diese Option in der Kommandozeile angegeben wird, erhalten statische Bibliotheken (.a) Priorität gegenüber dynamischen (.so).

-B dynamic

Wenn diese Option in der Kommandozeile angegeben wird, erhalten dynamische Bibliotheken (.so) Priorität gegenüber statischen (.a).

-B symbolic

Die Adressauflösung findet beim Laden eines Objekts über *dlopen()* statt, und zwar in folgender Reihenfolge:

(1) Geladenes Programm *a.out*

(2) Alle Shared Objects, die vor dem zu ladenden Objekt geladen werden (RTLD_GLOBAL)

(3) Zu ladendes Objekt

`-B symbolic` nicht angegeben:

Die Adressauflösung findet in umgekehrter Reihenfolge statt (3 - 2 - 1).

Dateien

Es können nur Dateinamen mit dem Namenssuffix .o (bzw. mit dem Namenssuffix .so bei Verwendung der *-S* option) angegeben werden. Ausführbare Dateien müssen ggf. umbenannt werden.

Dateien können nur **nach** allen Optionen angegeben werden.

Endestatus

Folgende Exit-Werte werden zurückgegeben:

0 Generierung erfolgreich

>0 Fehler

Datei

In dem Verzeichnis, in dem die Ausgabe generiert wird, wird ein temporäres Verzeichnis eingerichtet. Dieses Verzeichnis wird bei Beendigung des Kommandos wieder gelöscht.

Umgebung

Für die Suche nach Bibliotheksverzeichnissen gelten folgende Prioritäten:

1. Verzeichnisse, die in der Variablen LD_LIBRARY_PATH enthalten sind. Werden dort mehrere Verzeichnisse angegeben, dann müssen diese durch einen Doppelpunkt getrennt sein (ohne Leerzeichen!).
2. Verzeichnisse, die über die Option `-L` definiert sind

Beispiel 1

Es sollen vier Shared Libraries generiert werden: *libtest21.so, libtest22.so, libtest23.so* und *libtest24.so.*

Im aktuellen Verzeichnis befinden sich die o-Dateien *test21.o, test22.o, test23.o* und *test24.o*.

Die Shared Objects *libtest23.so* und *libtest24.so* bestehen jeweils nur aus den o-Dateien *test23.o* und *test24.o*.

Die Bibliothek *libtest22.so* besteht aus *test22.o* und dem abhängigen Shared Object *libtest24.so*; die Bibliothek *libtest21.so* besteht aus *test21.o* und den abhängigen Shared Objects *libtest22.so* und *libtest23.so*.

Für die Generierung der Shared Objects sind folgende Aufrufe von *genso* notwendig:

```
genso -o libtest24.so test24.o
genso -o libtest23.so test23.o
genso -o libtest22.so -l test24 test22.o
genso -o libtest21.so -l test22 -l test23 test21.o
```

Mit der Option *–S* kann man sich den Inhalt eines Shared Object ansehen.

```
$ genso -S low   ./libtest21.so
analysis of shared object ./libtest21.so
shared object ./libtest21.so consists of
         Grossmodul ofile_GM_.o built of
                 objectmodule /home/bach/dll/test/reihentest/test21.o
         dep. shared object libtest22.so (/home/bach/dll/test/reihentest/libtest22.so)
         dep. shared object libtest23.so (/home/bach/dll/test/reihentest/libtest23.so)
         option: -X lang=c
```

Will man auch den Inhalt der abhängigen Bibliotheken sehen, so muss *–S high* angegeben werden.

```
$ genso -S high ./libtest21.so
analysis of shared object ./libtest21.so
shared object ./libtest21.so consists of
         Grossmodul ofile_GM_.o built of
                 objectmodule /home/bach/dll/test/reihentest/test21.o
         dep. shared object libtest22.so (/home/bach/dll/test/reihentest/libtest22.so)
         dep. shared object libtest23.so (/home/bach/dll/test/reihentest/libtest23.so)
         option: -X lang=c
analysis of shared object /home/bach/dll/test/reihentest/libtest22.so
shared object /home/bach/dll/test/reihentest/libtest22.so consists of
         Grossmodul ofile_GM_.o built of
                 objectmodule /home/bach/dll/test/reihentest/test22.o
         dep. shared object libtest24.so (/home/bach/dll/test/reihentest/libtest24.so)
         option: -X lang=c
analysis of shared object /home/bach/dll/test/reihentest/libtest24.so
shared object /home/bach/dll/test/reihentest/libtest24.so consists of
         Grossmodul ofile_GM_.o built of
                 objectmodule /home/bach/dll/test/reihentest/test24.o
         option: -X lang=c
analysis of shared object /home/bach/dll/test/reihentest/libtest23.so
shared object /home/bach/dll/test/reihentest/libtest23.so consists of
         Grossmodul ofile_GM_.o built of
                 objectmodule /home/bach/dll/test/reihentest/test23.o
         option: -X lang=c
```

Beispiel 2

Es soll ein Shared Objekt *libp1.so* aus den folgenden Bestandteilen erzeugt werden:

- aus dem Element *UNTEST1.O* aus der PLAM-Bibliothek *$BACH.DL.LIB*
- aus allen Elementen der ar-Bibliothek *libar.a* und der o-Datei *file1.o*.

Das Kommando zur Generierung wird unter der Kennung *$BACH* aufgerufen und sieht folgendermaßen aus:

```
genso -o libp1.so -B plam -L . -lDL.LIB -m UNTEST1.O -B ar -L . -l ar file1.o
```

Dabei bedeutet *-B plam -L . -lDL.LIB*, dass *DL.LIB* eine PLAM-Bibliothek ist, die sich in Benutzerkennung befindet, unter der *genso* aufgerufen wird.

Sieht man sich den Inhalt von *libp1.so* an, so erhält man:

```
$ genso -S low ./libp1.so
analysis of shared object ./libp1.so
shared object ./libp1.so consists of
         Grossmodul ofile_GM_.o built of
                 objectmodule /home/bach/dll/newcommands/file1.o
         arlibrary /home/bach/dll/newcommands/libar.a with elements
                 objectmodule arfile1.o
                 objectmodule arfile2.o
         plam library dl.lib with elements
                UNTEST1.O
          option: -X lang=c
```

2.2 dlopen - Zugriff auf eine Shared Object Datei ermöglichen

Syntax

```
#include <dlfcn.h>

void *dlopen(const char *file, int mode);
```

Für den Aufruf in einer ASCII-Umgebung müssen Sie die Funktion *__dlopen_ascii()* mit denselben Parametern verwenden.

Beschreibung

dlopen() ermöglicht es dem aufrufenden Programm, auf eine über *file* angegebene Shared Object Datei zuzugreifen.

dlopen() gibt ein Handle zurück, welches das aufrufende Programm für anschließende *dlsym()*- und *dlclose()*-Aufrufe verwenden kann. Der Wert dieses Handle darf vom aufrufenden Programm nicht interpretiert werden.

Mit Hilfe von *file* wird der Pfadname für die Objektdatei wie folgt generiert:

- Wenn *file* mit einem Schrägstrich beginnt, dann wird das Argument von *file* als vollständiger Dateiname interpretiert.
- Beginnt *file* nicht mit einem Schrägstrich, dann wird die Variable LD_LIBRARY_PATH benutzt, um zusammen mit *file* den vollständigen Dateinamen zu erzeugen.
  LD_LIBRARY_PATH enthält eine Liste von Verzeichnissen (absolute oder auch relative Pfadnamen), die durch Doppelpunkt getrennt sind. Wenn diese Liste leer ist, wird das aktuelle Arbeitsverzeichnis verwendet.
- Wenn *file* den Wert 0 hat, dann liefert *dlopen()* ein Handle für ein globales Symbolobjekt. Dieses Objekt ermöglicht den Zugriff auf die Symbole aus der Menge der globalen Objekte des Programms. Diese Menge besteht aus der ursprünglichen Imagedatei des Programms, allen beim Programmstart geladenen Objekten sowie die Menge der Objekte, die bei einem *dlopen()*-Aufruf mit Flag RTLD_GLOBAL geladen wurden. Da sich die zuletzt genannte Menge von Objekten während der Ausführung ändern kann, kann sich auch die über das Handle identifizierte Objektmenge dynamisch ändern.

Der Parameter *mode* beschreibt, wie *dlopen()* die Adressauflösung und die Sichtbarkeit von Symbolen in Bezug auf *file* behandelt. Wenn ein Objekt in den Adressraum eines Prozesses verschoben wird, dann kann es Verweise auf Symbole enthalten, deren Adressen vor dem Laden des Objekts nicht bekannt sind. Solche Verweise müssen aufgelöst werden, damit auf die Symbole zugegriffen werden kann. Über den Parameter *mode* wird gesteuert, wann diese Adressauflösungen stattfinden. *mode* kann folgende Werte annehmen:

RTLD_LAZY

> Gleiches Verhalten wie bei RTLD_NOW.

RTLD_NOW

> Alle notwendigen Adressauflösungen werden beim ersten Laden eines Objekts durchgeführt. Jedes Shared Object wird zusammen mit den zugehörigen abhängigen Objekten in einem eigenen Binde-Lade-Kontext geladen. Bleiben Externverweise offen, so wird keine Warnung ausgegeben; *dlopen()* beendet sich nicht mit Fehler.

Jedes über *dlopen()* geladene Objekt, dessen Verweise auf globale Symbole aufgelöst werden müssen, kann auf die Symbole in der Imagedatei des ursprünglichen Prozesses, auf alle beim Programmstart geladenen Objekte (d. h. auf sich selbst sowie auf alle im selben Aufruf von *dlopen()* enthaltenen Objekte), und auf alle über einen *dlopen()*-Aufruf mit Flag RTLD_GLOBAL geladenen Objekte verweisen.

Die Sichtbarkeit für die über einen *dlopen()*-Aufruf geladenen Symbole kann über folgende Werte von *mode* festgelegt werden (bitweise „Oder“-Verknüpfung):

RTLD_GLOBAL

Die Symbole des Objekts stehen für die Adressauflösung von anderen Objekten zur Verfügung.

Damit können Objekte, die mit dieser *mode*-Einstellung geladen wurden, über Symbole gesucht werden, d.h. mit Hilfe von *dlopen (0, mode)* und zugeordnetem *dlsym()*.

RTLD_LOCAL

Die Symbole des Objekts stehen **nicht** für die Adressauflösung von anderen Objekten zur Verfügung.

Wenn weder RTLD_GLOBAL noch RTLD_LOCAL angegeben werden, dann wird RTLD_LOCAL als Standardwert genommen.

Außerdem bewirkt die Angabe von RTLD_GLOBAL, dass das Objekt unabhängig von einer früheren oder späteren Angabe von RTLD_LOCAL so lange den Status RTLD_GLOBAL behält, wie es im Adressraum bleibt (siehe *dlclose()*).

Über *dlopen()* in ein Programm aufgenommene Symbole können referenziert werden, z.B. für offene Externverweise später geladener Objekte. Bei solchen Symbolen handelt es sich möglicherweise um Duplikate von Symbolen, die bereits vom Programm oder von vorangegangenen *dlopen()*-Aufrufen definiert wurden.

Die Symbole, die über *dlopen()*-Aufrufe aufgenommen wurden und mit Hilfe von *dlsym()* verfügbar sind, sind genau diejenigen, die bei einem VSVI-Aufruf als Typ ENTRY angezeigt werden.

Returnwert

dlopen() gibt NULL zurück, wenn eine der folgenden Bedingungen erfüllt ist:

- Die in *file* angegebene Datei kann nicht gefunden werden.
- Der Lesezugriff auf *file* ist nicht möglich.
- Das Objektformat von *file* eignet sich nicht für die Verarbeitung durch *dlopen()*.
- Während des Ladens von *file* oder der Befriedigung der Symbolverweise ist ein Fehler aufgetreten.
- Es wurden unbefriedigte Externverweise gefunden. Das Shared Object wird in diesem Fall nicht weiterverarbeitet.

Die Variable *errno* wird nicht gesetzt. Ein Fehlertext (Diagnoseinfo) kann über *dlerror()* ermittelt werden.

Wenn Entries mehrfach angegeben werden, dann wird dies nicht als Fehler betrachtet (kein NULL-Returnwert). Bei mehrfach angegebenen Entries wird immer der erste Entry verwendet.

Wenn die Umgebungsvariable LD_UNRESOLVED=YES gesetzt ist, dann werden Shared Objects auch dann weiterverarbeitet, wenn ungelöste Externverweise gefunden wurden (kein NULL-Returnwert!).

Das jeweilige sprachspezifische Laufzeitsystem wird vor dem Laden des Shared Object in den Defaultkontext geladen und initialisiert. Das Laufzeitsystem muss hierzu im BS2000 über IMON installiert sein.

Pro ungelöstem Externverweis wird eine Meldung mit dem Namen und dem Type des Externverweises (XDSECT, VCON oder EXTERN) ausgegeben. Dabei werden maximal 512 ungelöste Externverweise berücksichtigt. Sind mehr als diese 512 ungelösten Externverweise vorhanden, so wird zusätzlich ein Warnungshinweis ausgegeben. Durch schrittweise Korrekturen lassen sich damit alle ungelösten Externverweise auffinden und beheben.

Beispiel

Das folgende Beispiel veranschaulicht, wie *dlopen()* verwendet werden kann.

```
void    *handle;

/* Das Objekt oeffnen*/
handle = dlopen("./mylib.so",RTLD_LAZY + RTLD_GLOBAL);
if (handle == NULL) {
    printf (error during dlopen, dlerror: %s\n”, dlerror());
    exit(EXIT_FAILURE);
}
```

Siehe auch

dlclose(), dlerror(), dlsym()

2.3 dlclose - Objekt schließen

Syntax

```
#include <dlfcn.h>

int dlclose(void *handle);
```

dlclose() kann mit identischer Syntax auch in einer ASCII-Umgebung aufgerufen werden Da *dlclose()* keine Strings verwendet, ist dazu keine eigene ASCII-Variante notwendig.

Beschreibung

Mit Hilfe von *dlclose()* wird dem System mitgeteilt, dass das Objekt, auf das ein von einem vorausgehenden *dlopen()* -Aufruf zurückgegebenes Handle (*handle*) verweist, von der Anwendung nicht mehr benötigt wird.

Sobald ein Objekt mit Hilfe von *dlclose()* geschlossen wurde, kann die Anwendung davon ausgehen, dass die zugehörigen Symbole nicht mehr für *dlsym()* verfügbar sind. Alle Objekte, die infolge des *dlopen()*-Aufrufs für das Objekt, auf das verwiesen wird, automatisch geladen wurden, werden ebenfalls geschlossen.

Returnwert

Wenn das Objekt, auf das verwiesen wird, erfolgreich geschlossen wurde, gibt *dlclose()* den Wert 0 zurück.

Wenn das Objekt nicht geschlossen werden konnte oder *handle* nicht auf ein geöffnetes Objekt verweist, gibt *dlclose()* einen anderen Wert zurück.

Die Variable *errno* wird nicht gesetzt. Ein Fehlertext (Diagnoseinfo) kann über *dlerror()* ermittelt werden.

Hinweis

Das von einem *dlopen()*-Aufruf in *handle* zurückgegebene Handle sollte nur zwischen einem *dlopen()*- und einem *dlclose()*-Aufruf verwendet werden. Bei mehreren Aufrufen von *dlopen()* wird bei Verweisen auf dasselbe Objekt immer dasselbe Objekt in *handle* zurückgegeben. Ebenso kann *handle* auch mehrmals verwendet werden. Daher muss der Wert von *handle* von der Anwendung als transparentes Objekt behandelt werden, das nur in *dlsym()*- und *dlclose()*-Aufrufen benutzt wird.

Bei C++ werden beim Schließen des Shared Object sprachspezifische Finalisierungen durchgeführt.

Durch *dlclose()* wird das adressierte Shared Object als nicht mehr zugänglich gekennzeichnet. Ein physikalisches Entladen des Objekts wird nur dann durchgeführt, falls es keine anderen Shared Objects gibt, die Referenzen auf dieses Objekt haben oder haben könnten.

Beispiel

Das folgende Beispiel veranschaulicht, wie *dlclose()* verwendet werden kann.

```
void    *handle;
int ret;

/* Objekt schliessen */
if ((ret = dlclose(handle)) != 0) {
    printf (error during dlclose, ret: %d dlerror: %s\n”, ret, dlerror());
    exit(EXIT_FAILURE);
}
```

Siehe auch

dlerror(), dlopen(), dlsym()

2.4 dlsym - Adresse eines Symbols aus einem dlopen()-Objekt ermitteln

Syntax

```
#include <dlfcn.h>

void *dlsym(void *handle, const char *name);
```

Für den Aufruf in einer ASCII-Umgebung müssen Sie die Funktion *__dlsym_ascii()* mit denselben Parametern verwenden.

Beschreibung

Mit Hilfe von *dlsym()* können Prozesse die Adresse eines Symbols ermitteln, das in einem per *dlopen()*-Aufruf verfügbar gemachten Objekt definiert wurde.

handle bestimmt die Suchstrategie. Für *handle* sind folgende Angaben möglich:

- Der Wert, der von einem *dlopen()*-Aufruf zurückgegeben und seitdem nicht über einen *dlclose()*-Aufruf freigegeben wurde.
- RTLD_DEFAULT
  Es werden alle Objekte werden in der zeitlichen Reihenfolge ihres Ladens durchsucht.
- RTLD_NEXT
  Es werden nur die Objekte durchsucht, die zeitlich nach dem Objekt geladen wurden, aus dem der Aufruf *dlsym()* abgesetzt wurde.
- RTLD_SELF
  Zuerst wird das Objekt durchsucht, aus dem de Aufruf *dlsym()* abgesetzt wurde.
  Anschließend werden alle danach geladenen Objekte durchsucht.

name ist der Name des Symbols als Zeichenfolge.

Returnwert

Wenn *handle* nicht auf ein gültiges Objekt verweist oder wenn das angegebene Symbol in keinem der *handle* zugeordneten Objekte gefunden werden kann, gibt *dlsym()* der Wert NULL zurück.

Bei den Namen ist zu beachten, dass Großschreibung und das Ersetzen von '_' durch '$' beim Übersetzen der Objekte festgelegt werden (durch entsprechende Optionen beim Kommando *cc*).

Die Variable *errno* wird nicht gesetzt. Ein Fehlertext (Diagnoseinfo) kann über *dlerror()* ermittelt werden.

Beispiel

Das folgende Beispiel veranschaulicht, wie *dlopen()* und *dlsym()* für den Zugriff auf Funktionen oder Datenobjekte verwendet werden können. Zur Vereinfachung wurde die Überprüfung auf Fehler weggelassen.

```
void    *handle;
int     *iptr, (*fptr)(int);

/* Das benoetigte Objekt oeffnen*/
handle = dlopen("/usr/home/me/libfoo.so.1",RTLD_LAZY);

/* Die Adresse von Funktionen und Datenobjekten suchen*/
fptr = (int (*)(int))dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_object");

/* Die Funktion aufrufen und den Wert des Integer als Parameter uebergeben*/
(*fptr)(*iptr);
```

Siehe auch

dlclose(), dlerror(), dlopen()

2.5 dlerror - Diagnoseinformationen abrufen

Syntax

```
#include <dlfcn.h>

char *dlerror(void);
```

Für den Aufruf in einer ASCII-Umgebung müssen Sie die Funktion *__dlerror_ascii()* mit denselben Parametern verwenden.

Beschreibung

dlerror() gibt eine mit binär Null endende Zeichenfolge ohne abschließendes Zeilenvorschubzeichen zurück. Diese Zeichenfolge beschreibt den letzten Fehler, der während der Verarbeitung von dynamischen Bindeaufrufen aufgetreten ist. Wenn seit dem letzten Aufruf von *dlerror()* keine Fehler beim dynamischen Binden aufgetreten sind, dann wird gibt *dlerror()* der Wert NULL zurück. Daher wird bei einem zweiten Aufruf von *dlerror()* direkt nach einem vorangegangenen Aufruf immer der Wert NULL zurückgegeben.

Returnwert

Bei erfolgreicher Ausführung gibt *dlerror()* eine auf binär Null endende Zeichenfolge zurück. Andernfalls wird der Wert NULL zurückgegeben.

Die Variable *errno* wird nicht gesetzt.

Hinweis

Die von *dlerror()* zurückgegebenen Informationen können sich in einem statischen Puffer befinden, der bei jedem Aufruf von *dlerror()* überschrieben wird. Anwendungscode sollte nicht in diesen Puffer geschrieben werden. Programme, die eine Fehlernachricht aufbewahren möchten, sollten eine eigene Kopie der Nachricht erstellen.

Siehe auch

dlclose(), dlopen(), dlsym()

2.6 dladdr - Adresse in symbolische Informationen umwandeln

Syntax

```
#include <dlfcn.h>

int dladdr(void *address, struct Dl_info *dlip);
```

Für den Aufruf in einer ASCII-Umgebung müssen Sie die Funktion *__dladdr_ascii()* mit denselben Parametern verwenden.

Beschreibung

dladdr() ermittelt, ob sich die angegebene Adresse in einem der zugeordneten Objekte befindet, die den Adressraum der derzeit aktiven Anwendungen bilden. Eine Adresse gilt als innerhalb eines zugeordneten Objekts, wenn sie sich zwischen der Basisadresse und der Endadresse des Objekts befindet. Erfüllt ein zugeordnetes Objekt diese Bedingung, dann wird die dem dynamischen Binder zur Verfügung gestellte Symboltabelle nach dem „nächstgelegenen“ Symbol durchsucht, d.h dem Symbol, dessen Wert mit der erforderlichen Adresse übereinstimmt bzw. möglichst knapp darunter liegt.

Die Struktur *Dl_info* muss zuvor vom Benutzer zugeordnet werden. Die einzelnen Elemente der Struktur werden auf Grund der angegebenen Adresse mit Hilfe von *dladdr()* versorgt.

Die Struktur *Dl_info* enthält folgende Elemente:

```
const char *   dli_fname;
void *         dli_fbase;
const char *   dli_sname;
void *         dli_saddr;
```

Bedeutung dieser Elemente:

`dli_fname`

enthält einen Zeiger, der auf den Dateinamen des enthaltenden Objekts verweist.

`dli_fbase`

enthält die Basisadresse des enthaltenden Objekts.

`dli_sname`

enthält einen Zeiger, der auf den Namen des nächstgelegenen Symbols verweist. Dessen Wert stimmt mit der angegebenen Adresse übereinstimmt bzw. liegt möglichst knapp darunter.

`dli_saddr`

enthält die tatsächliche Adresse des Symbols.

Returnwert

Wenn die angegebene Adresse keinem zugeordneten Objekt zugeordnet werden kann, wird der Wert 0 zurückgegeben. Andernfalls wird ein anderer Wert zurückgegeben und die zugeordneten *Dl_info*-Elemente werden versorgt.

Die Variable *errno* wird nicht gesetzt. Ein Fehlertext (Diagnoseinfo) kann über *dlerror()* ermittelt werden.

Hinweis

Die *Dl_info*-Zeigerelemente verweisen auf Adressen innerhalb der zugeordneten Objekte. Diese Adressen können ungültig werden, wenn die entsprechenden Objekte entfernt werden (siehe *dlclose()*). Wenn kein Symbol zum Beschreiben der angegebenen Adresse gefunden wird, werden die beiden Elemente *dli_sname* und *dli_saddr* auf den Wert 0 gesetzt.

Das Element *dli_fbase* der Struktur *Dl_info* wird nicht gesetzt und hat stets den Wert 0.

Beispiel

Der Einfachheit halber sind die Fehlerabfragen weggelassen.

```
void *handle;
int symboladdr;
int ret;
struct Dl_info obj_info;

/* Oeffnen der Bibliothek */
handle = dlopen("mydynlib.so", RTLD_NOW | RTLD_GLOBAL);

/* Ermitteln Adresse zum Entry symbolname */
symboladdr = dlsym(handle, "symbolname");

/* welches Symbol ist 8KB hinter symboladdr zu finden? */
symboladdr += 8192;
if((ret = dladdr((void *)symboladdr, (struct Dl_info *) &obj_info)) == 0) {
    /* Fehler */
    printf("dladdr() schlug fehl fuer Adresse %08X\n", symboladdr);
    fprintf("dlerror(): %s\n", dlerror());
} else {
    /* erfolgreich */
    printf("dladdr:\n
        \tdli_fname %s\n
        \tdli_fbase %08X\n
        \tdli_sname %s\n
        \tdli_saddr %08X\n",
        obj_info.dli_fname,
        (int)obj_info.dli_fbase,
        obj_info.dli_sname,
        (int)obj_info.dli_saddr
    );
}
```

Siehe auch

dlclose(), dlerror(), dlopen()

2.7 dlfcn.h - Header für dynamisches Binden

Syntax

```
#include <dlfcn.h>
```

Beschreibung

Über den Header *dlfcn.h* werden u.a. Makros definiert, die für das Argument *mode* des *dlopen()*-Aufrufs verwendet werden können. Außerdem enthält *dlfcn.h* Strukturen sowie Prototypen der Funktionsaufrufe.

```
#ifndef _DLFCN_H
#define _DLFCN_H


#if defined(_LITERAL_ENCODING_ASCII)
#    if (_LITERAL_ENCODING_ASCII - 0 == 1) && !defined(_ASCII_SOURCE)
#       define _ASCII_SOURCE 1  /*automatische Umsetzung*/
#    endif
#endif
#if defined(_ASCII_SOURCE)
#   if (_ASCII_SOURCE - 0 != 0) && (_ASCII_SOURCE - 0!= 1)
#       error unsupported _ASCII_SOURCE
#   endif
#else
#   define _ASCII_SOURCE 0
#endif


struct Dl_info {
    const char *    dli_fname;
    void *          dli_fbase;
    const char *    dli_sname;
    void *          dli_saddr;
};
```

```
extern void *__dlopen_ascii(const char *, int);
extern void *__dlsym_ascii(void *, const char *);
extern char *__dlerror_ascii(void);
extern int   __dladdr_ascii(void *, struct Dl_info *);
extern void *dlopen(const char *, int );
extern void *dlsym(void *, const char *);
extern int   dlclose(void *);
extern char *dlerror(void);
extern int   dladdr(void *, struct Dl_info *);
#if (_ASCII_SOURCE - 0 == 1)
#   ifdef _MAP_NAME
#       define dlopen  __dlopen_ascii
#       define dlsym   __dlsym_ascii
#       define dlerror __dlerror_ascii
#       define dladdr  __dladdr_ascii
#   else
#       define dlopen(_n, _f) __dlopen_ascii(_n, _f)
#       define dlsym(_h, _s)  __dlsym_ascii(_h, _s)
#       define dlerror()      __dlerror_ascii()
#       define dladdr(_v, _i) __dladdr_ascii(_v, _i)
#   endif
#endif /* _ASCII_SOURCE == 1 */


/* valid values for mode argument to dlopen */
#define RTLD_LAZY       1       /* lazy function call binding */
#define RTLD_NOW        2       /* immediate function call binding */
#define RTLD_GLOBAL     4       /* symbols in this dlopen'ed obj are visible */
                                /* to other dlopen'ed objs */
#define RTLD_LOCAL      8       /* symbols in this dlopen'ed obj are */
                                /* invisible to other dlopen'ed objs */


#define RTLD_MAIN_UPPERCASE     0x10    /* uppercase names in main program */
#define RTLD_MAIN_DOLLAR        0x20    /* dollar for underscore in names */


/*
 * defines for dlsym
 * RTLD_DEFAULT searches all objects loaded
 * RTLD_NEXT    searches all objects loaded after the object the call comes from
 * RTLD_SELF    searches all objects loaded after the object the call comes from
 *              including this object as the first one
 */
```

```
#define RTLD_DEFAULT    (void *)(-2)
#define RTLD_NEXT       (void *)(-1)
#define RTLD_SELF       (void *)(-3)


#endif  /* _DLFCN_H */
```

Siehe auch

dlopen(), dlclose(), dlsym(), dlerror()