English

Fujitsu Software BS2000

POSIX
Loading Shared Objects

User Guide

Valid for:
POSIX A49
BS2000 V21.0B

Edition August 2025

Comments… Suggestions… Corrections…

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to: bs2000.info@fujitsu.com.

Certified documentation according to DIN EN ISO 9001:2015

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2015.

Copyright and Trademarks

Table of Contents

POSIX_sharedObjects_en 2025-08

1 General remarks

This is the interface description for shared objects (SO) used with POSIX in BS2000.

Please consider the following limitations and restrictions:

The sum of the number of loaded program contexts plus the number of loaded object contexts is limited to 192. This is dependent on the number of 200 Link-and-Load systems contexts currently available.

1.1 Definition of a shared object

The following types of elements generated by the compiler itself or from the compiler output have been supported in POSIX hitherto:

`a.out`	executable file (type X)
`file.o`	object generated by the compiler (type O)
`libx.a`	ar library (type AR; normally containing types X and O)

We now introduce another object, called a “shared object”. The name “shared object” is derived from UNIX. Other names for shared objects that we also use in certain contexts are “dynamic library” or “shared library” as in UNIX.

In POSIX, a shared object always has the following structure:

`object.so`	so element (type SO; built from types X, O, AR and SO itself, and from PLAM libraries)

While generating a shared object, libraries (of type SO and AR) and files with .o extensions can be specified. To use executable files (type X) while generating a shared object, these files must be renamed (see page 14).

Supporting shared objects in POSIX

Shared objects can be reloaded in POSIX while running using an overlay technique (program controlled reloading). The functions *dlopen()*, *dlclose()*, *dlsym()*, *dlerror()*, and *dladdr()* are provided for reloading (see "genso - generate a shared object").

In POSIX automatic reloading (like classic BS2000) is only possible for modules of the runtime system CRTE.

Unlike UNIX, POSIX does not support the automatic binding of objects during runtime and the shared coding can not be used by several programs in a shared memory.

Organization (Structure) of a shared object

Physically, a shared object is an ar library which contains object modules that are part of the shared object and a description of the shared object itself. The description of the shared object takes the form of a text file in the ar library. This text file is generated during the generation of the shared object (with the *genso* command).

The processing of this file is processed during *dlopen()* is done in the following way:

- If no dependent libraries are found, the o files are loaded in reverse order as specified in the file.
- If a dependent shared object exists, then a new list is generated. The dependent shared object is appended to the end of this list.
- Another search is then made for the first dependent object in the new list. This may be a dependent object of the first dependent object.

This is repeated until all dependent objects (and dependent objects of these objects, ...) are replaced by listed o files. Checks are made to avoid recursion. Finally, this results in a list containing only o files which will then be loaded as specified in the list. This resolution of an so file is what is described as **dependency ordering** in the description of *dlopen()*.

The handling of ar files is the same as for o files; an ar file is considered as an ordered set of o files.

Example

Here is a typical example of such a description. This is the description of the example 1 of the genso command (see "genso - generate a shared object").

Shared object description for file *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 for file *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 for file *libtest23.so*:

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

Shared object description for file *libtest24.so*:

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

The elements of this shared object have the following appearance when they are processed by *dlopen()*.

Object *libtest21.so*:

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

Object *libtest22.so*:

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

Object *libtest23.so*:

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

Object *libtest24.so*:

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

The reference to *libtest22.so* in the description of *libtest21.so* is replaced by the description of *libtest22.so* in a first step. This produces the following arrangement:

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

This procedure is repeated until finally the following is produced.

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

The elements are then loaded in this order.

1.2 Generation of a shared object

A shared object is generated by a special command: *genso*. The options and parameters are as for *cc*.

-B plam ... -B ar

The options *–L* and *-l* bracketed between these options are interpreted so that PLAM libraries can be processed. The new option *–m* is also interpreted.

-B static
-B dynamic

Specifies, whether ar libraries or dynamic libraries are used to generate shared objects.

Used only at generation time only.

-B symbolic

Indicator of resolution algorithm.

-L directory

Name of the directory.

-L bs2000_user_id

Only valid between *–B plam* and *–B ar*. Specifies the BS2000 user ID:

$

stands for TSOS.

.

(period) stands for the user ID under which the application is running.

%name

the user ID is taken from the environment variable *name*.

-l xxx

Name of the library: libxxx.a or libxxx.so

If *-l xxx* is specified between *–B plam* and *–B ar*, then *xxx* is the name of the BS2000 library.

-m member

Only applies between *–B plam* and *–B ar.* Specifies the name of the PLAM library member (L member).

-o output

Names the output.

-S low | **high**

The contents of the shared object are output on *stdout*. For *low* this is only the current object, for *high* this is also all shared objects still dependent on it.

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

Specifies the programming language of the object to be dynamically loaded. Mixed objects are possible.

Default value is *lang=c*.

`file.o`

.o file used for generation.

Other familiar options from UNIX like *-h name*, *-Kpic* and *-b* are not supported.

The *genso* command generates the so file. If *–B symbolic* is not specified, then at generation time the external links are resolved as far as possible. The objects in the .so file are linked into a link-and-load module (one LLM). When the shared object is loaded, the shared object itself gets the highest priority in resolving the references.

Priority rules to find a library:

Directories for libraries are searched with the following priority:

1. directories contained in the variable LD_LIBRARY_PATH
2. directories defined by *–L* option
3. the default directory */usr/lib*

1.3 Functions on shared objects

Functions that are processed on shared objects are:

Name	Description
`dlopen()`[1]	Open a shared object
`dlclose()`	Close a shared object
`dlsym()`[1]	Get addresses within a shared object
`dladdr()`[1]	Get nearest symbol name for a given address in a shared object
`dlerror()`[1]	Diagnostic information about a preceding function call processed with error

[1] The functions *__dlopen_ascii()*, *__dlsym_ascii()*, *__dladdr_ascii()*, and *__dlerror_ascii()* are provided for the call in an ASCII environment.

1.4 Installation

The components which are necessary for the shared object support are automatically installed during an initial installation or a delta installation of POSIX-BC.

These components are installed in the following default directories:

Component	Installation directory	Type
`libdl.a`	`/usr/lib`	ar library
`genso`	`/usr/bin`	command
`dlfcn.h`	`/usr/include`	header file

Please note, that *libdl.a* is not a shared library, but an ar library.

This library must be statically linked to the program that uses the above interfaces, for example with:

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

2 dll interfaces in BS2000

This chapter describes the *genso* command, the *dlopen()*, *dlclose()*, *dlsym()*, *dlerror()* and *dladdr()* functions and the *dlfcn.h* header file.

2.1 genso - generate a shared object

The *genso* command generates shared objects.

Syntax

```
/usr/bin/genso  [options] [files]
```

options

genso accepts some of the options used for the POSIX C compiler; other options are derived from the ld command used on UNIX systems.

-L dir

Adds *dir* directory to the list of directories in which *genso* searches for libraries. The *-L* option must be specified **before** the *-l* option. The *-L* option is valid until a new *-L* option is specified.

-L bs2000_user_id

Only in connection with *–B plam*: name of the BS2000 user ID.

The following specifications are possible:

$

stands for TSOS

.

(period) stands for the user ID under which the application is running

%name

if a BS2000 user ID is to be explicitly specified. The user ID must then be written in the environment variable *name*.

-l xxx

libxxx.a or *libxxx.so* will be used to generate a shared object. Which type of library is actually used depends on the actual use of *–B static* and *–B dynamic*. If nothing is specified, dynamic libraries have priority over static ones.

-l plam_library

Only in connection with *–B plam*: name of the BS2000 PLAM library.

-m member

Only in connection with *–B plam*: name of the BS2000 PLAM library member to be processed.

-o output

The name of the shared object to be generated. The shared object is stored under this name in the current directory. The extension of the file name should be *.so*; the *.so* file extension is not added automatically to the file name. To avoid misunderstandings, the suffixes *.o* and *.a* are rejected.

-B plam

From now on, PLAM libraries are processed. To switch to ar or so libraries, *–B ar* must be specified.

-B ar

If PLAM libraries have been processed, then ar libraries or so libraries will be processed from now on.

-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.

The contents of the shared object are output to *stdout*.

If the file name is not an absolute or relative path, the file is searched for in the directories specified with the environment variable LD_LIBRARY_PATH or in */usr/lib*. When applied to a file in the current directory, a ./ may need to be prefixed if this directory is not specified in the environmental variable LD_LIBRARY_PATH.

-S low only outputs the contents of the current shared object.

-S high in addition to the contents of the current shared object, outputs the contents of all dependent objects.

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

Specifies the programming language of the object to be loaded. Mixed objects are possible.

Default value is *lang=c* .

-B static

When this option is given in the command line, static libraries (.a) have priority over dynamic libraries (. so).

-B dynamic

When this option is given in the command line, dynamic libraries (.so) have priority over static libraries (.a).

-B symbolic

Address resolution is effected when an object is loaded (via *dlopen*) and the sequence of resolving addresses is:

(1) the loaded program *a.out*

(2) all shared objects loaded before the actual object is loaded (RTLD_GLOBAL)

(3) the currently loaded object

-B symbolic not specified:

The resolution is effected in reverse order (3 - 2 - 1).

files

Only file names with .o file extension can be used (or with the name suffix .so when the *-S* option is used). Executable files must be renamed if necessary.

Files can only be specified **after** all options.

Exit status

The following exit values are returned:

0 Successful generation

>0 An error occurred

File

In the directory in which the output will be generated, a directory is temporarily generated and will be deleted when the command terminates.

Environment

Directories for libraries are searched with the following priority:

1. directories contained in the variable LD_LIBRARY_PATH. If more than one directory is specified, the directories must be separated by colons (without blanks!).
2. directories defined by *–L* option.

Example 1

Four Shared Libraries have to be generated: *libtest21.so, libtest22.so, libtest23.so* and *libtest24.so.*

The o files *test21.o, test22.o, test23.o* and *test24.o* are located in the current directory.

The shared objects *libtest23.so* and *libtest24.so* only comprise the o files *test23.o* and *test24.o* respectively.

The library *libtest22.so* consists of *test22.o* and the dependent shared object *libtest24.so*; the library *libtest21.so* consists of *test21.o* and the dependent shared objects *libtest22.so* and *libtest23.so*.

The following calls of *genso* are required to generate the shared objects:

```
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
```

With the option *–S*, you can view the contents of a shared object.

```
$ 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
```

If you also wish to view the contents of the dependent libraries, *–S high* must be specified.

```
$ 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
```

Example 2

A shared object with the name *libp1.so* must be generated from the following components:

- from the member *UNTEST1.O* from the PLAM library *$BACH.DL.LIB*

- from all members of the ar library *libar.a* and the o file *file1.o*.

The generation command is called up under the ID *$BACH*, and has the following format:

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

Here, *-B plam -L . -lDL.LIB* indicates that *DL.LIB* is a PLAM library located in the user ID under which *genso* is called up.

If you view the contents of *libp1.so*, you will see:

```
$ 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 - gain access to a shared object file

Syntax

```
#include <dlfcn.h>

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

The function *__dlopen_ascii()* with the same parameters must be used for the call in an ASCII environment.

Description

dlopen() makes a shared object file specified by *file* available to the calling program.

A successful *dlopen()* returns a handle which the caller may use on subsequent calls to *dlsym()* and *dlclose()*. The value of this handle should not be interpreted in any way by the caller.

file is used to construct a pathname to the object file:

- If *file* is beginning with a slash character, the *file* argument is used as the complete filename.
- If *file* does not begin with a slash character, the variable LD_LIBRARY_PATH is used to generate the complete filename together with *file*. LD_LIBRARY_PATH contains a list of directories (absolute or also relative path names) separated by a colon. If this list is empty, the current working directory is used.
- If the value of *file* is 0, *dlopen()* provides a handle on a global symbol object. This object provides access to the symbols from an ordered set of objects consisting of the original program image file, together with any objects loaded at program startup and the set of objects loaded using a *dlopen()* operation together with the RTLD_GLOBAL flag. As the latter set of objects can change during execution, the set identified by handle can also change dynamically.

The *mode* parameter describes how *dlopen()* will operate upon file with respect to the processing of relocations and the scope of visibility of the symbols provided within *file*. When an object is brought into the address space of a process, it may contain references to symbols whose addresses are not known until the object is loaded. These references must be resolved before the symbols can be accessed. The *mode* parameter governs when these relocations take place and may have the following values:

RTLD_LAZY

> The same behavior as RTLD_NOW.

RTLD_NOW

> All necessary relocations are performed when the object is first loaded. Each shared object together with its dependent objects is loaded in a link-and-load context of its own. In case of unresolved relocations no warning is sent; *dlopen()* does not end with an error.
>
> Any object loaded by *dlopen()* that requires relocations against global symbols can reference the symbols in the original process image file, any objects loaded at program startup, from the object itself as well as any other object included in the same *dlopen()* invocation, and any objects that were loaded in any *dlopen()* invocation and which specified the RTLD_GLOBAL flag.

To determine the scope of visibility for the symbols loaded with a *dlopen()* invocation, the *mode* parameter should be bitwise or'ed with one of the following values:

RTLD_GLOBAL

The object's symbols are made available for the relocation processing of any other object. In addition, symbol lookup using *dlopen (0, mode)* and an associated *dlsym()* allows objects loaded with this mode to be searched.

RTLD_LOCAL

The object's symbols are **not** made available for the relocation processing of any other object.

If neither RTLD_GLOBAL nor RTLD_LOCAL is specified, then RTLD_LOCAL is default value.

Note that once RTLD_GLOBAL has been specified, the object will maintain the RTLD_GLOBAL status regardless of any previous or future specification of RTLD_LOCAL, so long as the object remains in the address space (see *dlclose()*).

Symbols introduced into a program through calls to *dlopen()* may be used in relocation activities, for example. Symbols so introduced may duplicate symbols already defined by the program or previous *dlopen()* operations.

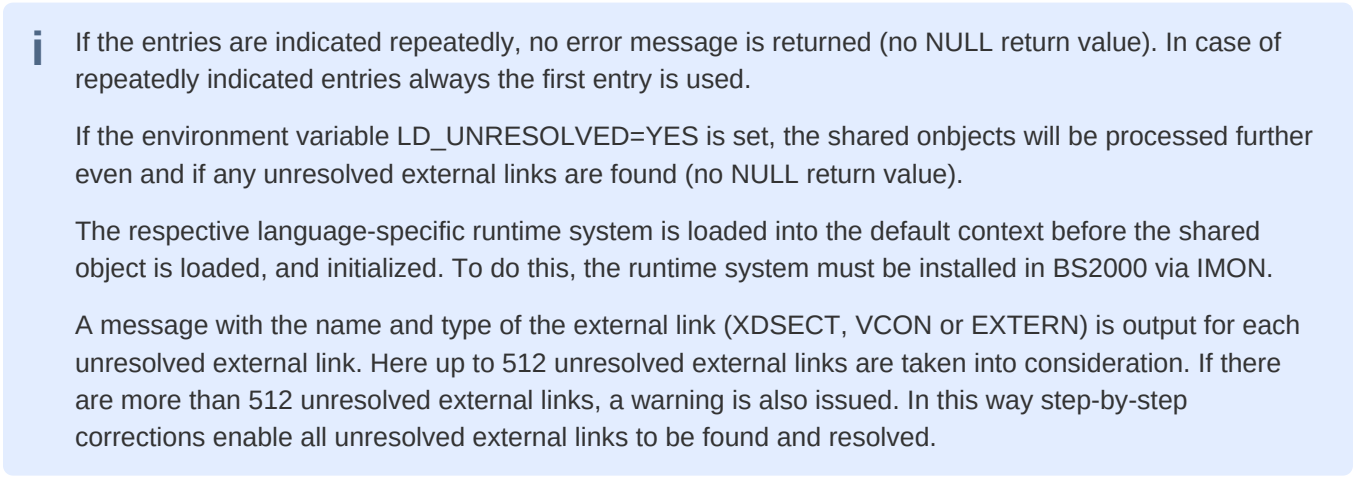
The symbols introduced by *dlopen()* operations, and available through *dlsym()* are those that are of type ENTRY shown by a VSVI call.

Return value

dlopen() returns NULL in the following cases:

- *file* cannot be found.
- *file* cannot be opened for reading.
- The *file* object format is not dedicated (suitable) for processing by *dlopen()*.
- An error occurs during the process of loading file or relocating its symbolic references.
- Unresolved external references were found. In this case, the shared object is not processed further.

The *errno* variable is not set. An error message (diagnostic information) will be available through *dlerror()*.



If the entries are indicated repeatedly, no error message is returned (no NULL return value). In case of repeatedly indicated entries always the first entry is used.

If the environment variable LD_UNRESOLVED=YES is set, the shared onbjects will be processed further even and if any unresolved external links are found (no NULL return value).

The respective language-specific runtime system is loaded into the default context before the shared object is loaded, and initialized. To do this, the runtime system must be installed in BS2000 via IMON.

A message with the name and type of the external link (XDSECT, VCON or EXTERN) is output for each unresolved external link. Here up to 512 unresolved external links are taken into consideration. If there are more than 512 unresolved external links, a warning is also issued. In this way step-by-step corrections enable all unresolved external links to be found and resolved.

Example

The following example illustrates how *dlopen()* can be used.

```
void    *handle;

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

See also

dlclose(), dlerror(), dlsym()

2.3 dlclose - close an object

Syntax

```
#include <dlfcn.h>

int dlclose(void *handle);
```

dlclose() can also be called in an ASCII environment using identical syntax. An own ASCII variant is not necessary since *dlclose()* uses no strings.

Description

dlclose() is used to inform the system that the object referenced by a *handle* returned from a previous *dlopen()* invocation is no longer needed by the application.

Once an object has been closed using *dlclose()*, an application should assume that its symbols are no longer available to *dlsym()*. All objects loaded automatically as a result of invoking *dlopen()* on the referenced object are also closed.

Return value

If the referenced object was successfully closed, *dlclose()* returns 0.

If the object could not be closed, or if *handle* does not refer to an open object, *dlclose()* returns a non-zero value.

The *errno* variable is not set. An error message (diagnostic information) will be available through *dlerror()*.

Application usage

The application should employ a *handle* returned from a *dlopen()* invocation only within a given scope bracketed by the *dlopen()* and *dlclose()* operations. Multiple calls to *dlopen()* referencing the same object may return the same object for *handle*. Applications are also free to re-use a *handle*. For these reasons, the value of a *handle* must be treated as an opaque object by the application, used only in calls to *dlsym()* and *dlclose()*.

For C++, language-specific finalizations are performed when the shared object is closed:

dlclose() marks the addressed shared object as being no longer accessible. The object is only physically unloaded if there are no other shared objects which have or could have references to this object.

Example

The following example illustrates how *dlclose()* can be used.

```
void    *handle;
int ret;

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

See also

dlerror(), dlopen(), dlsym()

2.4 dlsym - get the address of a symbol from a dlopen() object

Syntax

```
#include <dlfcn.h>

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

The function *__dlsym_ascii()* with the same parameters must be used for the call in an ASCII environment.

Description

dlsym() allows a process to obtain the address of a symbol defined within an object made accessible through a *dlopen()* call.

handle determines the search strategy. For *handle* the following entries are possible:

- The value that is returned by a *dlopen()* call and which has not been released since then by a *dlclose()* call.
- RTLD_DEFAULT
  All objects are searched in the chronological sequence of their loading.
- RTLD_NEXT
  This only searches objects that were loaded after the object in which the *dlsym()* call was performed.
- RTLD_SELF
  The object from which the call *dlsym()* is placed is searched first. All objects loaded afterwards are searched next.

name is the name of the symbol as character string.

Return value

If *handle* does not refer to a valid object, or if the named symbol cannot be found within any of the objects associated with *handle*, *dlsym()* will return NULL.

Note, that capitalization and the replacement of '_' through '$' are defined during the compiling of the objects (with corresponding options of the *cc* command).

The *errno* variable is not set. An error message (diagnostic information) will be available through *dlerror()*.

Example

The following example shows how one can use *dlopen()* and *dlsym()* to access either function or data objects. For simplicity, error checking has been omitted.

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

/* Opening the appropriate object*/
handle = dlopen("/usr/home/me/libfoo.so.1",RTLD_LAZY);

/* Searching the address of functions and data objects*/
fptr = (int (*)(int))dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_object");

/* Calling the function and handing over the integer value as a parameter*/
(*fptr)(*iptr);
```

See also

dlclose(), dlerror(), dlopen()

2.5 dlerror - get diagnostic information

Syntax

```
#include <dlfcn.h>

char *dlerror(void);
```

The function *__dlerror_ascii()* with the same parameters must be used for the call in an ASCII environment.

Description

dlerror() returns a null-terminated character string (with no trailing newline) that describes the last error that occurred during dynamic linking processing. If no dynamic linking errors have occurred since the last invocation of *dlerror()*, *dlerror()* returns NULL. Thus, invoking *dlerror()* a second time, immediately following a prior invocation, will result in NULL being returned.

Return value

If successful, *dlerror()* returns a null-terminated character string. Otherwise, NULL is returned.

The *errno* variable is not set.

Application usage

The messages returned by *dlerror()* may reside in a static buffer that is overwritten on each call to *dlerror()*. Application code should not write to this buffer. Programs wishing to preserve an error message should make their own copies of that message.

See also

dlclose(), dlopen(), dlsym()

2.6 dladdr - translate address to symbolic information

Syntax

```
#include <dlfcn.h>

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

The function *__dladdr_ascii()* with the same parameters must be used for the call in an ASCII environment.

Description

dladdr() determines if the specified address is located within one of the mapped objects that make up the current applications address space. An address is deemed to fall within a mapped object when it is between the base address and the end address of that object. If a mapped object fits these criteria, the symbol table made available to the dynamic linker is searched to locate the nearest symbol to the specified address. The nearest symbol is one that has a value less than or equal to the required address.

The *Dl_info* structure must be preallocated by the user. The structure members are filled in by *dladdr()* based on the specified address.

The *Dl_info* structure includes the following members:

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

Descriptions of these members:

`dli_fname`

Contains a pointer to the filename of the containing object.

`dli_fbase`

Contains the base address of the containing object.

`dli_sname`

Contains a pointer to the symbol name nearest to the specified address. This symbol either has the same address or is the nearest symbol with a lower address.

`dli_saddr`

Contains the actual address of the above symbol.

Return value

If the specified address cannot be matched to a mapped object, a 0 is returned. Otherwise, a non-zero return is made and the associated *Dl_info* elements are filled.

The *errno* variable is not set. An error message (diagnostic information) will be available through *dlerror()*.

Note

The *Dl_info* pointer elements point to addresses within the mapped objects. These addresses may become invalid if objects are removed prior to these elements being used (see *dlclose()*). If no symbol is found to describe the specified address, both the *dli_sname* and *dli_saddr* members are set to 0.

The element *dli_fbase* of the structure *Dl_info* is not set and always has the value 0.

Example

For reasons of simplicity error traps have been omitted.

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

/* Open library */
handle = dlopen("mydynlib.so", RTLD_NOW | RTLD_GLOBAL);

/* Determine the address of the entry symbolname */
symboladdr = dlsym(handle, "symbolname");

/* What symbol is located at an offset of 8KB from the symboladdr? */
symboladdr += 8192;
if((ret = dladdr((void *)symboladdr, (struct Dl_info *) &obj_info)) == 0) {
    /* error */
    printf("dladdr() failed for address %08X\n", symboladdr);
    fprintf("dlerror(): %s\n", dlerror());
} else {
    /* success */
    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
    );
}
```

See also

dlclose(), dlerror(), dlopen()

2.7 dlfcn.h - header for dynamic linking

Syntax

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

Description

The *dlfcn.h* header defines macros for use in the *dlopen()* construction of a *mode* argument. In addition, *dlfcn.h* contains structures and prototypes of the function calls.

```
#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 */
```

See also

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