

**NAME**

`dladdr`, `dladdr1` – translate address to symbolic information

**SYNOPSIS**

```
#define _GNU_SOURCE
```

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

```
int dladdr(void *addr, Dl_info *info);
```

```
int dladdr1(void *addr, Dl_info *info, void **extra_info, int flags);
```

Link with `-ldl`.

**DESCRIPTION**

The function **dladdr()** determines whether the address specified in *addr* is located in one of the shared objects loaded by the calling application. If it is, then **dladdr()** returns information about the shared object and symbol that overlaps *addr*. This information is returned in a *Dl\_info* structure:

```
typedef struct {
    const char *dli_fname; /* Pathname of shared object that
                           contains address */
    void *dli_fbase; /* Base address at which shared
                     object is loaded */
    const char *dli_sname; /* Name of symbol whose definition
                           overlaps addr */
    void *dli_saddr; /* Exact address of symbol named
                     in dli_sname */
} Dl_info;
```

If no symbol matching *addr* could be found, then *dli\_sname* and *dli\_saddr* are set to NULL.

The function **dladdr1()** is like **dladdr()**, but returns additional information via the argument *extra\_info*. The information returned depends on the value specified in *flags*, which can have one of the following values:

**RTLD\_DL\_LINKMAP**

Obtain a pointer to the link map for the matched file. The *extra\_info* argument points to a pointer to a *link\_map* structure (i.e., *struct link\_map \*\**), defined in *<link.h>* as:

```
struct link_map {
    ElfW(Addr) l_addr; /* Difference between the
                      address in the ELF file and
                      the address in memory */
    char *l_name; /* Absolute pathname where
                  object was found */
    ElfW(Dyn) *l_ld; /* Dynamic section of the
                     shared object */
    struct link_map *l_next, *l_prev;
                      /* Chain of loaded objects */

    /* Plus additional fields private to the
       implementation */
};
```

**RTLD\_DL\_SYMENT**

Obtain a pointer to the ELF symbol table entry of the matching symbol. The *extra\_info* argument is a pointer to a symbol pointer: *const ElfW(Sym) \*\**. The *ElfW()* macro definition turns its argument into the name of an ELF data type suitable for the hardware architecture. For example, on a 64-bit platform, *ElfW(Sym)* yields the data type name *Elf64\_Sym*, which is defined in *<elf.h>* as:

```
typedef struct {
    Elf64_Word st_name; /* Symbol name */
```

```

        unsigned char st_info;      /* Symbol type and binding */
        unsigned char st_other;     /* Symbol visibility */
        Elf64_Section st_shndx;     /* Section index */
        Elf64_Addr    st_value;     /* Symbol value */
        Elf64_Xword   st_size;     /* Symbol size */
    } Elf64_Sym;

```

The *st\_name* field is an index into the string table.

The *st\_info* field encodes the symbol's type and binding. The type can be extracted using the macro **ELF64\_ST\_TYPE(st\_info)** (or **ELF32\_ST\_TYPE()** on 32-bit platforms), which yields one of the following values:

Value	Description
<b>STT_NOTYPE</b>	Symbol type is unspecified
<b>STT_OBJECT</b>	Symbol is a data object
<b>STT_FUNC</b>	Symbol is a code object
<b>STT_SECTION</b>	Symbol associated with a section
<b>STT_FILE</b>	Symbol's name is file name
<b>STT_COMMON</b>	Symbol is a common data object
<b>STT_TLS</b>	Symbol is thread-local data object
<b>STT_GNU_IFUNC</b>	Symbol is indirect code object

The symbol binding can be extracted from the *st\_info* field using the macro **ELF64\_ST\_BIND(st\_info)** (or **ELF32\_ST\_BIND()** on 32-bit platforms), which yields one of the following values:

Value	Description
<b>STB_LOCAL</b>	Local symbol
<b>STB_GLOBAL</b>	Global symbol
<b>STB_WEAK</b>	Weak symbol
<b>STB_GNU_UNIQUE</b>	Unique symbol

The *st\_other* field contains the symbol's visibility, which can be extracted using the macro **ELF64\_ST\_VISIBILITY(st\_info)** (or **ELF32\_ST\_VISIBILITY()** on 32-bit platforms), which yields one of the following values:

Value	Description
<b>STV_DEFAULT</b>	Default symbol visibility rules
<b>STV_INTERNAL</b>	Processor-specific hidden class
<b>STV_HIDDEN</b>	Symbol unavailable in other modules
<b>STV_PROTECTED</b>	Not preemptible, not exported

## RETURN VALUE

On success, these functions return a nonzero value. If the address specified in *addr* could be matched to a shared object, but not to a symbol in the shared object, then the *info->dli\_sname* and *info->dli\_saddr* fields are set to NULL.

If the address specified in *addr* could not be matched to a shared object, then these functions return 0. In this case, an error message is *not* available via **dlderror(3)**.

## VERSIONS

**dladdr()** is present in glibc 2.0 and later. **dladdr1()** first appeared in glibc 2.3.3.

## ATTRIBUTES

For an explanation of the terms used in this section, see **attributes(7)**.

Interface	Attribute	Value
<b>dladdr()</b> , <b>dladdr1()</b>	Thread safety	MT-Safe

## CONFORMING TO

These functions are nonstandard GNU extensions that are also present on Solaris.

## BUGS

Sometimes, the function pointers you pass to **dladdr()** may surprise you. On some architectures (notably i386 and x86-64), *dli\_fname* and *dli\_fbase* may end up pointing back at the object from which you called **dladdr()**, even if the function used as an argument should come from a dynamically linked library.

The problem is that the function pointer will still be resolved at compile time, but merely point to the *plt* (Procedure Linkage Table) section of the original object (which dispatches the call after asking the dynamic linker to resolve the symbol). To work around this, you can try to compile the code to be position-independent: then, the compiler cannot prepare the pointer at compile time any more and **gcc(1)** will generate code that just loads the final symbol address from the *got* (Global Offset Table) at run time before passing it to **dladdr()**.

## SEE ALSO

**dl\_iterate\_phdr(3)**, **dlinfo(3)**, **dlopen(3)**, **dlsym(3)**, **ld.so(8)**

## COLOPHON

This page is part of release 5.02 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at <https://www.kernel.org/doc/man-pages/>.