# Mac OS X ABI Dynamic Loader Reference

**Tools > Compiling & Debugging**

**2005-11-09**

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010



Simultaneously published in the United States and Canada.

# Contents


## Mac OS X ABI Dynamic Loader Reference 7

Overview 7
Functions by Task 7
- Dynamic Loader Compatibility Functions 7
- Object File Image Functions 8
- Library Functions 9
- Section and Segment Accessors 9
- Low-Level Functions 10
- Glue Functions for Indirect Addressing 11

Functions 11
- dladdr 11
- dlclose 12
- dlerror 12
- dlopen 13
- dlsym 14
- dyld_stub_binding_helper 15
- getsectbyname 16
- getsectbynamefromheader 16
- getsectbynamefromheader_64 16
- getsectdata 17
- getsectdatafromFramework 17
- getsectdatafromheader 18
- getsectdatafromheader_64 18
- getsegbyname 19
- NSAddImage 19
- NSAddLibrary 20
- NSAddLibraryWithSearching 21
- NSAddressOfSymbol 21
- NSCreateObjectFileImageFromFile 21
- NSCreateObjectFileImageFromMemory 22
- NSDestroyObjectFileImage 23
- NSIsSymbolNameDefined 23
- NSIsSymbolNameDefinedInImage 23
- NSIsSymbolNameDefinedWithHint 24
- NSLibraryNameForModule 24
- NSLinkModule 25
- NSLookupAndBindSymbol 26
- NSLookupAndBindSymbolWithHint 26
- NSLookupSymbolInImage 27
- NSLookupSymbolInModule 28
- NSModuleForSymbol 28

NSNameOfModule 28
NSNameOfSymbol 29
NSUnLinkModule 29
_dyld_bind_fully_image_containing_address 30
_dyld_bind_objc_module 30
_dyld_func_lookup 30
_dyld_get_image_header 31
_dyld_get_image_name 31
_dyld_get_image_vmaddr_slide 31
_dyld_get_objc_module_sect_for_module 32
_dyld_image_containing_address 32
_dyld_image_count 33
_dyld_launched_prebound 33
_dyld_lookup_and_bind 33
_dyld_lookup_and_bind_fully 34
_dyld_lookup_and_bind_objc 34
_dyld_lookup_and_bind_with_hint 35
_dyld_moninit 35
_dyld_present 36
_dyld_register_func_for_add_image 36
_dyld_register_func_for_link_module 36
_dyld_register_func_for_remove_image 37
Constants 37
Mach-O Image Creation Return Codes 37

**Document Revision History 39**

**Index 41**

# Tables

**Mac OS X ABI Dynamic Loader Reference 7**

Table 1 Values for the handle parameter 15

# Mac OS X ABI Dynamic Loader Reference

## Overview

Most Mac OS X applications need to load and use dynamic shared libraries or bundles at runtime. The dynamic loader, `dyld`, is a shared library that programs use to gain access to other shared libraries. The dynamic loader locates Mach-O files on disk and maps them into the memory space of the current program.

Well-written programs don't load shared libraries until they are needed. This minimizes the launch time and memory footprint of the application. It may also improve the general performance of the system through reduced paging.

This document describes the low-level functions of the Mac OS X application binary interface (ABI) that you can use to load, link, and unload Mach-O files at runtime.

You should read this document if you develop applications that use dynamic shared libraries or libraries that use other libraries.

These documents provide information about the runtime loading of code:

- "Mach-O Programming Topics" describes runtime code-loading concepts and ways to correctly package shared libraries into versioned frameworks.
- "Dynamically Loading Code" describes higher level techniques to load code at runtime.

## Functions by Task

The dynamic linker provides several types of functionality that allow your application to manipulate Mach-O files at runtime.

### Dynamic Loader Compatibility Functions

These are the recommended functions to use to interact with the dynamic loader. These functions work in Mac OS X v10.3 and v10.4. However, in Mac OS X v10.4 they are more efficient than other image-loading functions. These functions are declared in `/usr/include/dlfcn.h`.

`dladdr` (page 11)
: Finds the image and nearest symbol corresponding to an address. Available only in dynamically linked programs.

`dlclose` (page 12)
: Closes a dynamic library or bundle.

`dlerror` (page 12)
: Provides diagnostic information corresponding to problems with calls to `dlopen` (page 13), `dlsym` (page 14), and `dlclose` (page 12) in the same thread.

`dlopen` (page 13)
: Loads and links a dynamic library or bundle.

`dlsym` (page 14)
: Returns the address of a symbol.

## Object File Image Functions

These functions are for loading Mach-O bundle files. They are declared in `/usr/include/mach-o/dyld.h`. The use of these functions is discouraged. You should use the more efficient functions described in "Dynamic Loader Compatibility Functions" (page 7).

`NSCreateObjectFileImageFromFile` (page 21)
: Creates an image reference for a given Mach-O file.

`NSCreateObjectFileImageFromMemory` (page 22)
: Creates an image reference for a Mach-O file currently in memory.

`NSDestroyObjectFileImage` (page 23)
: Releases the given object file image.

`NSLinkModule` (page 25)
: Links the given object file image as a module into the current program.

`NSLookupAndBindSymbol` (page 26)
: Given a symbol name, returns the corresponding symbol from the global symbol table.

`NSLookupAndBindSymbolWithHint` (page 26)
: Given a symbol name, returns the corresponding symbol from the global symbol table.

`NSLookupSymbolInModule` (page 28)
: Given a module reference, returns a reference to the symbol with the given name.

`NSNameOfModule` (page 28)
: Returns the name of the given module.

`NSIsSymbolNameDefined` (page 23)
: Returns `TRUE` if the given symbol is defined in the current program.

`NSIsSymbolNameDefinedInImage` (page 23)
: Returns `TRUE` if the given image contains the named symbol.

`NSIsSymbolNameDefinedWithHint` (page 24)
: Returns `TRUE` if the given symbol is defined in the current program, with a hint specifying the name of the shared library likely to contain the symbol.

`NSModuleForSymbol` (page 28)
: Returns a reference to the module containing the given symbol.

`NSUnLinkModule` (page 29)
: Unlinks the given module from the current program.

## Library Functions

The functions described in this section are declared in `/usr/include/mach-o/dyld.h`. The use of these functions is discouraged. You should use the more efficient functions described in "Dynamic Loader Compatibility Functions" (page 7).

`NSAddImage` (page 19)
: Adds the specified Mach-O image to the currently running process.

`NSAddLibrary` (page 20)
: Adds a dynamic shared library to the search list.

`NSAddLibraryWithSearching` (page 21)
: Adds a dynamic shared library to the search list—using the various `dyld` environment variables—as if the library were linked into the program.

`NSAddressOfSymbol` (page 21)
: Returns the address in the program's address space of the data represented by the given symbol. The data may be a variable, a constant, or the first instruction of a function.

`NSLibraryNameForModule` (page 24)
: Returns the name of the library that contains the given module.

`NSLookupSymbolInImage` (page 27)
: Returns a reference to the specified symbol from the specified image.

`NSNameOfSymbol` (page 29)
: Returns the name of the given symbol.

## Section and Segment Accessors

The functions described in this section are declared in `/usr/include/mach-o/getsect.h`.

`getsectbyname` (page 16)
: Returns a data structure representing a section of the Mach-O file that contains the main executable program of the current process.

`getsectbynamefromheader` (page 16)
: Returns the data structure representing a section of a specified 32-bit Mach-O file.

`getsectbynamefromheader_64` (page 16)
: Returns the data structure representing a section of a specified 64-bit Mach-O file.

`getsectdata` (page 17)
: Returns the data for a section from the Mach-O file of the main executable program of the current process.

`getsectdatafromFramework` (page 17)
: Returns the data for a section of the Mach-O file containing a specified framework.

`getsectdatafromheader` (page 18)
: Returns the data for a section of a specified 32-bit Mach-O file.

`getsectdatafromheader_64` (page 18)
: Returns the data for a section of a specified 64-bit Mach-O file.

`getsegbyname` (page 19)
: Returns a data structure representing a segment of the Mach-O file containing the main executable program of the current process.

## Low-Level Functions

`_dyld_bind_fully_image_containing_address` (page 30)
Finds the image containing the specified address and fully binds all the modules within it.

`_dyld_bind_objc_module` (page 30)
Binds the module that contains a given Objective-C address.

`_dyld_func_lookup` (page 30)
Obtains the address of the implementation of a `dyld` library function.

`_dyld_get_image_header` (page 31)
Returns the data structure for the header of a specified image. The image is specified by index into the list of images maintained by `dyld` for the current process.

`_dyld_get_image_name` (page 31)
Retrieves the name of an image.

`_dyld_get_image_vmaddr_slide` (page 31)
Returns the virtual memory address slide amount of an image.

`_dyld_get_objc_module_sect_for_module` (page 32)
Obtains the size and starting location of an Objective-C module.

`_dyld_image_count` (page 33)
Returns the number of images that `dyld` has mapped into the address space of the current process.

`_dyld_image_containing_address` (page 32)
Returns whether or not a specified address is within any loaded image.

`_dyld_launched_prebound` (page 33)
Returns whether or not the dynamic linker was able to launch the program with the prebinding optimization enabled.

`_dyld_lookup_and_bind` (page 33)
Finds the given symbol name and binds it into the program.

`_dyld_lookup_and_bind_fully` (page 34)
Finds the module containing the specified symbol and fully binds all the symbol references within it.

`_dyld_lookup_and_bind_objc` (page 34)
Obtains and binds the Objective-C module that contains the specified symbol.

`_dyld_lookup_and_bind_with_hint` (page 35)
Finds the given symbol name and binds it into the program, with a hint to allow `dyld` to speed up the symbol search for a prebound program.

`_dyld_moninit` (page 35)
This function is used by the profiling routine `moninit` to allow images other than the main executable to be profiled.

`_dyld_present` (page 36)
Indicates whether or not the dynamic linker is loaded into the current program

`_dyld_register_func_for_add_image` (page 36)
Registers a function to be called by the dynamic linker runtime when an image is added to the program.

`_dyld_register_func_for_link_module` (page 36)
Registers a function to be called by the dynamic linker runtime when a module is linked into the program.

`_dyld_register_func_for_remove_image` (page 37)
Registers a function to be called by the dynamic linker runtime when an image is removed from the program.

### Glue Functions for Indirect Addressing

`dyld_stub_binding_helper` (page 15)
Assembly-language glue code that performs binding for a lazy function symbol.

# Functions

## dladdr

Finds the image and nearest symbol corresponding to an address. Available only in dynamically linked programs.

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

**Parameters**

*addr*

On input, an address within the address space of the program.

*info*

Storage for a `Dl_info` object. On return, the symbolic information found.

**Return Value**

When an image containing the address specified in `addr` cannot be found, this function returns `0`. Otherwise, the result is a value other than `0`.

**Discussion**

This is the declaration for the `Dl_info` structure:

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

The descriptions for the fields in `Dl_info` are:

- `dli_fname`: The pathname of the image containing the address in `addr`.
- `dli_fbase`: The base address (`mach_header`) at which the image is mapped into the address space of the program.
- `dli_sname`: The name of the nearest runtime symbol.
- `dli_saddr`: The value of the symbol specified by `dli_sname`.

If the image containing `addr` is found, but no near symbol is found, the `dli_sname` and `dli_saddr` fields in the `Dl_info` object are set to `NULL`. A near symbol is a symbol within the image whose address is equal to or lower than `addr`.

See also `dlopen` (page 13), `dlsym` (page 14).

**Availability**
Available in Mac OS X v10.3 and later.

## dlclose

Closes a dynamic library or bundle.

```
int dlclose(
    void* handle);
```

**Parameters**

*handle*

Handle obtained through a call to `dlopen` (page 13).

**Return Value**
This function returns `0` when successful and a value other than `0` when unsuccessful.

**Discussion**
This function decreases the reference count of the image referenced by `handle`. When the reference count for `handle` becomes `0`, the termination routines in the image are called, and the image is removed from the address space of the current process. After that point, `handle` is rendered invalid.

If this function is unsuccessful, it sets an error condition that can be queried with `dlerror` (page 12).

See also `dlopen` (page 13), `dlerror` (page 12).

**Availability**
Available in Mac OS X 10.3 and later.

## dlerror

Provides diagnostic information corresponding to problems with calls to `dlopen` (page 13), `dlsym` (page 14), and `dlclose` (page 12) in the same thread.

```
const char* dlerror(
    void);
```

**Return Value**
When there's a problem to report, this function returns a pointer to a null-terminated string describing the problem. Otherwise, this function returns `NULL`.

**Discussion**
Each call to `dlerror` resets its diagnostic buffer. If a program needs to keep a record of past error messages, it must store them itself. Subsequent calls to `dlerror` in the same thread with no calls to `dlopen` (page 13), `dlsym` (page 14), or `dlclose` (page 12), return `NULL`.

See also `dlopen` (page 13), `dlsym` (page 14), `dlclose` (page 12).

**Availability**
Available in Mac OS X 10.3 and later.

## dlopen

Loads and links a dynamic library or bundle.

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

**Parameters**

*path*

Path to the image to open.

*mode*

Specifies when the loaded image's external symbols are bound to their definitions in dependent libraries (lazy or at load time) and the visibility of the image's exported symbols (global or local). The value of this parameter is made up by ORing one binding behavior value with one visibility specification value.

The following values specify the binding behavior:

- `RTLD_LAZY` (default): Each external symbol reference is bound the first time it's used.
- `RTLD_NOW`: All external symbol references are bound immediately.

The following values specify external symbol visibility:

- `RTLD_GLOBAL` (default): The loaded image's exported symbols are available to any images that use a flat namespace or to calls to `dlsym` when using a special handle (see `dlsym` (page 14) for details).
- `RTLD_LOCAL`: The loaded image's exported symbols are generally hidden. They are available only to `dlsym` (page 14) invocations that use the handle returned by this function.

**Return Value**
A handle that can be used with calls to `dlsym` (page 14) and `dlclose` (page 12).

**Discussion**
This function examines the Mach-O file specified by `path`. If the image is compatible with the current process and has not already been loaded into the process, the image is loaded and linked. If the image contains initializer functions, they are executed before this function returns.

Subsequent calls to *`dlopen`* to load the same image return the same handle, but the internal reference count for the handle is incremented. Therefore, all `dlopen` calls must be balanced with `dlclose` (page 12) calls.

For efficiency, the `RTLD_LAZY` binding mode is preferred over `RTLD_NOW`. However, using `RTLD_NOW` ensures that any undefined symbols are discovered during the call to `dlopen`.

The dynamic loader looks in the paths specified by a set of environment variables, and in the process's current directory, when it searches for a library. These paths are called dynamic loader search paths. The environment variables are `LD_LIBRARY_PATH`, `DYLD_LIBRARY_PATH`, and `DYLD_FALLBACK_LIBRARY_PATH`. The default value of `DYLD_FALLBACK_LIBRARY_PATH` (used when this variable is not set), is `$HOME/lib:/usr/local/lib:/usr/lib`.

The order in which the search paths are searched depends on whether `path` is a filename (it does not contain a slash) or a pathname (it contains at least one slash).

When `path` is a filename, the dynamic loader searches for the library in the search paths in the following order:

1. `$LD_LIBRARY_PATH`
2. `$DYLD_LIBRARY_PATH`
3. The process's working directory
4. `$DYLD_FALLBACK_LIBRARY_PATH`

When `path` is a pathname, the dynamic loader searches for the library in the search paths in the following order:

1. `$DYLD_LIBRARY_PATH`
2. The given pathname
3. `$DYLD_FALLBACK_LIBRARY_PATH` using the filename

If this function cannot open an image, it sets an error condition that can be accessed with `dlerror` (page 12).

**Important:** In Mac OS X, libraries can combine object code for both 32-bit and 64-bit PowerPC processors. Therefore, there are no separate 32-bit and 64-bit search paths.

See also `dlsym` (page 14), `dlclose` (page 12), `dlerror` (page 12).

**Availability**
Available in Mac OS X 10.3 and later.

## dlsym

Returns the address of a symbol.

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

**Parameters**

*handle*

Handle obtained by a call to `dlopen` (page 13), or a special handle. If the handle was obtained by a call to `dlopen` (page 13), it must not have been closed with a call to `dlclose` (page 12). These are the possible special-handle values: `RTLD_DEFAULT`, and `RTLD_NEXT`.

*symbol*

Null-terminated character string containing the C name of the symbol being sought.

**Return Value**
When successful, this function returns the address of `symbol`. Otherwise, it returns a null pointer.

**Discussion**
The value of handle specifies what images this function searches for to locate the symbol specified by the `symbol` parameter. Table 1 describes the possible values for the `handle` parameter.

**Table 1** Values for the handle parameter

| Handle value | Search scope |
|---|---|
| `dlopen` handle | Image associated with the `dlopen` (page 13) handle. |
| `RTLD_DEFAULT` | Every dependent library or `RTLD_GLOBAL`–opened library in the current process, in the order they were loaded. |
| *`RTLD_NEXT`* | Dependent libraries that were loaded after the one calling this function. Libraries opened with `dlopen` (page 13) are not searched. |

Unlike in the `NS...` functions, the `symbol` parameter doesn't require a leading underscore to be part of the symbol name.

If this function is unsuccessful, it sets an error condition that can be queried with `dlerror` (page 12).

See also `dlopen` (page 13), `dlerror` (page 12).

**Availability**
Available in Mac OS X 10.3 and later.

## dyld_stub_binding_helper

Assembly-language glue code that performs binding for a lazy function symbol.

```
.private_extern dyld_stub_binding_helper
```

**Parameters**

*`PowerPC: r11 x86: stack-based parameter`*

A pointer to the lazy symbol pointer for the function to be bound.

**Discussion**
The `dyld` stub binding helper is a glue function that assists the dynamic linker in lazily binding an external function. When the compiler sees a call to an external function, it generates a symbol stub and a lazy pointer for the function. At the call site, the compiler generates a call to the symbol stub. The symbol stub is a sequence of code that loads the lazy pointer and jumps to it. Initially, the sequence of code and the contents of the lazy pointer call this function, which calls the dynamic linker to bind the symbol. After the symbol is bound, the lazy pointer is set to the address of the symbol, and the symbol is reached directly by jumping to the lazy pointer.

Thereafter, because the address has been changed to the actual address of the function, all calls to the external function call the external function.

On entry, this function accepts the address of the lazy symbol pointer. On exit, the value of the lazy symbol pointer is set to the address of the external function. The `dyld` stub binding helper is assembly-language based and does not use standard calling conventions, and as such, the location of the parameters are specific to each CPU architecture. On PowerPC, the address of the lazy symbol pointer is expected to be in GPR11. On x86, the address of the lazy symbol pointer should be the pushed on the stack.

This function is located in the runtime startup files that are statically linked into the image. For executables, the file is `/lib/crt1.o`. For bundles, it is `/lib/bundle1.o`, and for shared libraries, it is `/lib/dylib1.o`.

## getsectbyname

Returns a data structure representing a section of the Mach-O file that contains the main executable program of the current process.

```
const struct section* getsectbyname(
        const char* segname,
        const char* sectname);
```

**Parameters**

*segname*

A pointer to a C string. Pass the name of the segment in which the section resides.

*sectname*

A pointer to a C string. Pass the name of the section.

**Return Value**

A pointer to a `section` ("Mach-O File Format Reference") data structure.

## getsectbynamefromheader

Returns the data structure representing a section of a specified 32-bit Mach-O file.

```
const struct section* getsectbynamefromheader(
        const struct mach_header* mhp,
        const char* segname,
        const char* sectname);
```

**Parameters**

*mhp*

A pointer to a `mach_header` data structure. Pass the `mach_header` of the file containing the section data you wish to retrieve.

*segname*

A pointer to a C string. Pass the name of the segment in which the section resides.

*sectname*

A pointer to a C string. Pass the name of the section.

**Return Value**

A pointer to a `section` data structure.

## getsectbynamefromheader_64

Returns the data structure representing a section of a specified 64-bit Mach-O file.

```
const struct section_64* getsectbynamefromheader(
        const struct mach_header_64* mhp,
        const char* segname,
        const char* sectname);
```

**Parameters**

*mhp*

A pointer to a `mach_header_64` data structure. Pass the `mach_header` of the file containing the section data you wish to retrieve.

*segname*

A pointer to a C string. Pass the name of the segment in which the section resides.

*sectname*

A pointer to a C string. Pass the name of the section.

**Return Value**
A pointer to a `section_64` data structure.

## getsectdata

Returns the data for a section from the Mach-O file of the main executable program of the current process.

```
char* getsectdata(
        const char* segname,
        const char* sectname,
        unsigned long* size);
```

**Parameters**

*segname*

A pointer to a C string. Pass the name of the segment in which the section resides.

*sectname*

A pointer to a C string. Pass the name of the section.

*size*

A pointer to a long integer. On output, contains the length (in bytes) of the section.

**Return Value**
A pointer to the data of the section.

## getsectdatafromFramework

Returns the data for a section of the Mach-O file containing a specified framework.

```
char* getsectdatafromFramework(
        const char* FrameworkName,
        const char* segname,
        const char* sectname,
        unsigned long* size);
```

**Parameters**

*FrameworkName*

A pointer to a C string. Pass the name of the framework in which the section resides.

*segname*

A pointer to a C string. Pass the name of the segment in which the section resides.

*sectname*

A pointer to a C string. Pass the name of the section.

*size*

A pointer to a long integer. On output, contains the length (in bytes) of the section.

**Return Value**

A pointer to the data of the section. If the Mach-O file is a dynamic shared library (`MH_DYLIB`), you need to add the virtual memory slide amount to this address to get the true address of the data. See `_dyld_get_image_vmaddr_slide` (page 31) for more information.

## getsectdatafromheader

Returns the data for a section of a specified 32-bit Mach-O file.

```
char* getsectdatafromheader(
        const struct mach_header* mhp,
        const char* segname,
        const char* sectname,
        uint32_t* size);
```

**Parameters**

*mhp*

A pointer to a `mach_header` data structure. Pass the `mach_header` of the file containing the section data you wish to retrieve.

*segname*

A pointer to a C string. Pass the name of the segment in which the section resides.

*sectname*

A pointer to a C string. Pass the name of the section.

*size*

A pointer to a long integer. On output, contains the length (in bytes) of the section.

**Return Value**

A pointer to the data of the section. If the Mach-O file is a dynamic shared library (`MH_DYLIB`), you need to add the virtual memory slide amount to this address to get the true address of the data. See `_dyld_get_image_vmaddr_slide` (page 31) for more information.

## getsectdatafromheader_64

Returns the data for a section of a specified 64-bit Mach-O file.

```
char* getsectdatafromheader(
        const struct mach_header_64* mhp,
        const char* segname,
        const char* sectname,
        uint64_t* size);
```

**Parameters**

*mhp*

A pointer to a `mach_header_64` data structure. Pass the `mach_header` of the file containing the section data you wish to retrieve.

*segname*
A pointer to a C string. Pass the name of the segment in which the section resides.

*sectname*
A pointer to a C string. Pass the name of the section.

*size*
A pointer to a long integer. On output, contains the length (in bytes) of the section.

**Return Value**
A pointer to the data of the section. If the Mach-O file is a dynamic shared library (`MH_DYLIB`), you need to add the virtual memory slide amount to this address to get the true address of the data. See `_dyld_get_image_vmaddr_slide` (page 31) for more information.

## getsegbyname

Returns a data structure representing a segment of the Mach-O file containing the main executable program of the current process.

```
const struct segment_command* getsegbyname(
        const char* segname);
```

**Parameters**

*segname*
A pointer to a C string. Pass the name of the segment.

**Return Value**
A pointer to a `segment_command` (“Mach-O File Format Reference”) data structure.

## NSAddImage

Adds the specified Mach-O image to the currently running process.

```
const struct mach_header* NSAddImage(
        const char* image_name,
        uint32_t options);
```

**Parameters**

*image_name*
A pointer to a C string. Pass the pathname to a shared library on disk. For best performance, specify the full pathname of the shared library—not a symlink.

*options*

A bit mask. Pass one or more of the following options or NSADDIMAGE_OPTION_NONE to specify no options:

NSADDIMAGE_OPTION_RETURN_ON_ERROR

If an error occurs and you have specified this option, this function returns NULL. You can then use the function NSLinkEditError to retrieve information about the error.

If an error occurs, and you have not specified this option, this function calls the linkEdit error handler you have installed using the NSInstallLinkEditErrorHandlers function. If you have not installed a link edit error handler, this function prints an error to stderr and causes a breakpoint trap to end the program.

NSADDIMAGE_OPTION_WITH_SEARCHING

With this option, the image_name passed for the library and all its dependents is affected by the various dyld environment variables as if this library were linked into the program.

NSADDIMAGE_OPTION_RETURN_ONLY_IF_LOADED

With this option, this function returns NULL if the shared library was not loaded prior to the call to this function.

**Return Value**
A pointer to a mach_header ("Mac OS X ABI Mach-O File Format Reference") data structure. This is a pointer to the start of the loaded image.

**Discussion**
This function loads the shared library specified by image_name into the current process, returning a pointer to the mach_header data structure of the loaded image. Any libraries that the specified library depends on are also loaded.

The linkEdit error handler is documented in the NSModule(3) man page.

For portability and efficiency, consider using dlopen (page 13).

**Availability**
Available in Mac OS X v10.1 and later.

## NSAddLibrary

Adds a dynamic shared library to the search list.

```
extern bool NSAddLibrary(
        const char* pathName);
```

**Parameters**

*pathName*

A C string. Pass the name of a dynamic shared library.

**Return Value**
TRUE if the library was successfully added to the search list, FALSE otherwise.

**Discussion**
Deprecated in Mac OS X v10.4. Use NSAddImage (page 19) instead.

**Special Considerations**

Instead of using this function, you should use `NSAddImage` (page 19) with the `NSADDIMAGE_OPTION_NONE` option.

## NSAddLibraryWithSearching

Adds a dynamic shared library to the search list—using the various `dyld` environment variables—as if the library were linked into the program.

```
extern bool NSAddLibraryWithSearching(
        const char* pathName);
```

**Parameters**

*pathName*

A C string. Pass the name of a dynamic shared library.

**Return Value**

`TRUE` if the library was successfully added to the search list, `FALSE` otherwise.

**Discussion**

Deprecated in Mac OS X v10.4. Use `NSAddImage` (page 19) instead.

**Special Considerations**

Instead of using this function, you should use `NSAddImage` (page 19) with the `NSADDIMAGE_OPTION_WITH_SEARCHING` option.

## NSAddressOfSymbol

Returns the address in the program's address space of the data represented by the given symbol. The data may be a variable, a constant, or the first instruction of a function.

```
void* NSAddressOfSymbol(
        NSSymbol symbol);
```

**Parameters**

*symbol*

A symbol reference. Pass the symbol whose address you wish to obtain.

**Return Value**

A pointer to the data represented by the given symbol.

**Discussion**

For portability and efficiency, consider using `dlsym` (page 14).

## NSCreateObjectFileImageFromFile

Creates an image reference for a given Mach-O file.

```
NSObjectFileImageReturnCode NSCreateObjectFileImageFromFile(
        const char* pathName,
        NSObjectFileImage* objectFileImage);
```

**Parameters**

*pathName*

A C string. Pass the pathname to a Mach-O executable file. You must have previously built this file with the `-bundle` linker option; otherwise, this function returns an error.

*objectFileImage*

On output, a pointer to an `NSObjectFileImage` opaque data structure.

**Return Value**

See Mach-O Image Creation Return Codes (page 37).

**Discussion**

Given a pathname to a Mach-O executable, this function creates and returns a `NSObjectFileImage` reference. The current implementation works only with bundles, so you must build the Mach-O executable file using the `-bundle` linker option.

For portability and efficiency, consider using `dlopen` (page 13).

## NSCreateObjectFileImageFromMemory

Creates an image reference for a Mach-O file currently in memory.

```
NSObjectFileImageReturnCode NSCreateObjectFileImageFromMemory(
        const void* address,
        size_t size,
        NSObjectFileImage* objectFileImage);
```

**Parameters**

*address*

A pointer to the memory block containing the Mach-O file contents.

*size*

The size of the memory block, in bytes.

*objectFileImage*

On output, a pointer to an `NSObjectFileImage` opaque data structure.

**Return Value**

See Mach-O Image Creation Return Codes (page 37).

**Discussion**

Given a pointer to a Mach-O file in memory, this function creates and returns an `NSObjectFileImage` reference. The current implementation works only with bundles, so you must build the Mach-O executable file using the `-bundle` linker option.

The memory block that `address` points to, must be allocated with `vm_allocate` (`/usr/include/mach/vm_map.h`).

See also `NSDestroyObjectFileImage` (page 23).

**Availability**

Available in Mac OS X v10.3 and later.

## NSDestroyObjectFileImage

Releases the given object file image.

```
bool NSDestroyObjectFileImage(
        NSObjectFileImage objectFileImage);
```

**Parameters**

*objectFileImage*

A reference to the object file image to destroy.

**Return Value**

TRUE if the image was successfully destroyed, FALSE if not.

**Discussion**

When this function is called, the dynamic loader calls vm_deallocate (/usr/include/mach/vm_map.h) on the memory pointed to by the objectFileImage parameter.

For portability and efficiency, consider using dlopen (page 13) in conjunction with dlclose (page 12).

See also NSCreateObjectFileImageFromMemory (page 22).

## NSIsSymbolNameDefined

Returns TRUE if the given symbol is defined in the current program.

```
enum bool NSIsSymbolNameDefined(
        const char* symbolName);
```

**Parameters**

*symbolName*

A C string. Pass the name of the symbol whose definition status you wish to discover.

**Return Value**

TRUE when the symbol is defined by any image loaded in the current process; FALSE when the symbol cannot be found.

**Discussion**

Deprecated in Mac OS X v10.4. Use NSLookupSymbolInImage (page 27) instead.

If you know the name of the library in which the symbol is likely to be located, you can use the NSIsSymbolNameDefinedWithHint (page 24) function, which may be faster than this function. You should use the NSIsSymbolNameDefinedInImage (page 23) function to perform a two-level namespace lookup.

## NSIsSymbolNameDefinedInImage

Returns TRUE if the given image contains the named symbol.

```
enum bool NSIsSymbolNameDefinedInImage(
        const struct mach_header* image,
        const char* symbolName);
```

**Parameters**

*image*

A pointer to a mach_header (*Mach-O Runtime Architecture*) data structure.

*symbolName*
A C string. Pass the name of the symbol.

**Return Value**
`TRUE` if the image contains a symbol with the given name, `false` otherwise.

**Discussion**
Deprecated in Mac OS X v10.4. Use `NSLookupSymbolInImage` (page 27) instead.

## NSIsSymbolNameDefinedWithHint

Returns `TRUE` if the given symbol is defined in the current program, with a hint specifying the name of the shared library likely to contain the symbol.

```
enum bool NSIsSymbolNameDefinedWithHint(
        const char* symbolName,
        const char* libraryNameHint);
```

**Parameters**
*symbolName*
A C string. Pass the name of the symbol whose definition status you wish to discover.

*libraryNameHint*
A C string. Pass any part of the name of the shared library that is likely to contain the symbol. It searches only the first shared library that matches.

**Return Value**
`TRUE` when the symbol is defined by any image loaded in the current process; `FALSE` when the symbol cannot be found.

**Discussion**
Deprecated in Mac OS X v10.4. Use `NSLookupSymbolInImage` (page 27) instead.

The library name you pass to this function allows it to determine a position in the list of loaded symbols from which to start the search. This can result in a considerably faster lookup search time than is possible using `NSIsSymbolNameDefined` (page 23).

Note that this function performs a flat lookup even if the symbol namespace of the current program has two levels. You should use the `NSIsSymbolNameDefinedInImage` (page 23) function to perform a two-level namespace lookup.

## NSLibraryNameForModule

Returns the name of the library that contains the given module.

```
const char* NSLibraryNameOfModule(
        NSModule module);
```

**Parameters**
*module*
A module reference. Pass the module whose library name you wish to retrieve.

**Return Value**
A C string containing the name of the library that contains the module. The string is owned by the dynamic linker and you should not free it.

**Discussion**
See Building Mach-O Files in *Mach-O Programming Topics* for more information about modules.

## NSLinkModule

Links the given object file image as a module into the current program.

```
NSModule NSLinkModule(
        NSObjectFileImage objectFileImage,
        const char* moduleName,
        uint32_t options);
```

**Parameters**

*objectFileImage*

An object file image reference. Pass a reference created using the `NSCreateObjectFileImageFromFile` (page 21) function.

*moduleName*

A C string. Pass the absolute path to the object file image. GDB uses this path to retrieve debug symbol information from the library.

*options*

An `unsigned long` value. Pass one or more of the following bit masks or `NSLINKMODULE_OPTION_NONE` to specify no options:

`NSLINKMODULE_OPTION_BINDNOW`

The dynamic linker binds all undefined references immediately, rather than waiting until the references are actually used. All dependent libraries are also bound.

`NSLINKMODULE_OPTION_PRIVATE`

Do not add the global symbols from the module to the global symbol list. Instead, you must use the `NSLookupSymbolInModule` (page 28) function to obtain symbols from this module.

`NSLINKMODULE_OPTION_RETURN_ON_ERROR`

If an error occurs while binding the module, return `NULL`. You can then use the function `NSLinkEditError` to retrieve information about the error.

Without this option, this function calls the `linkEdit` error handler you have installed using the `NSInstallLinkEditErrorHandlers` function. If you have not installed a link edit error handler, this function prints a message to the standard error stream and causes a breakpoint trap to end the program.

**Return Value**
A reference to the linked module.

**Discussion**
When you call this function, all libraries referenced by the given module are added to the library search list. Unless you pass the `NSLINKMODULE_OPTION_PRIVATE`, `NSLinkModule` adds all global symbols in the module to the global symbol list.

For portability and efficiency, consider using `dlopen` (page 13).

See “Building Mach-O Files” in “Mach-O Programming Topics” for more information about modules.

## NSLookupAndBindSymbol

Given a symbol name, returns the corresponding symbol from the global symbol table.

```
NSSymbol NSLookupAndBindSymbol(
        const char* symbolName);
```

**Parameters**

*symbolName*

A pointer to a C string. Pass the name of the symbol you wish to find.

**Return Value**

The symbol reference for the requested symbol.

**Discussion**

Deprecated in Mac OS X v10.4. Use `NSLookupSymbolInImage` (page 27) instead.

On error, if you have installed a link edit error handler, it is called; otherwise, this function writes an error message to file descriptor 2 (usually the standard error stream, `stderr`) and causes a breakpoint trap to end the program.

If you know the name of the library in which the symbol is likely to be located, you can use the `NSLookupAndBindSymbolWithHint` (page 26) function, which may be faster than this function. You should use the `NSLookupSymbolInImage` (page 27) function to perform a two-level namespace lookup.

## NSLookupAndBindSymbolWithHint

Given a symbol name, returns the corresponding symbol from the global symbol table.

```
NSSymbol NSLookupAndBindSymbolWithHint(
        const char* symbolName,
        const char* libraryNameHint);
```

**Parameters**

*symbolName*

A pointer to a C string. Pass the name of the symbol you wish to find.

*libraryNameHint*

A pointer to a C string. Pass any part of the name of the library that the symbol is likely to be found in.

**Return Value**

The symbol reference for the requested symbol.

**Discussion**

On error, if you have installed a link edit error handler, it is called; otherwise, this function writes an error message to file descriptor 2 (usually the standard error stream, `stderr`), and causes a breakpoint trap to end the program.

Note that this function performs a flat lookup even if the symbol namespace of the current program has two levels. You should use the `NSLookupSymbolInImage` (page 27) function to perform a two-level namespace lookup.

Deprecated in Mac OS X v10.4. Use `NSLookupSymbolInImage` (page 27) instead.

## NSLookupSymbolInImage

Returns a reference to the specified symbol from the specified image.

```
NSSymbol NSLookupSymbolInImage(
        const struct mach_header* image,
        const char* symbolName
        uint32_t options);
```

### Parameters

*image*

A pointer to a `mach_header` data structure. Pass a pointer to the start of the image that contains the symbol. You can get this pointer from a shared library name using `NSAddImage` (page 19).

If the process does not have a two-level namespace, `NSLookupSymbolInImage` ignores this argument and searches for the symbol in the global symbol table.

*symbolName*

A pointer to a C string. Pass the name of the symbol you wish to find.

*options*

A bit mask. Pass any of the following options:

`NSLOOKUPSYMBOLINIMAGE_OPTION_BIND`

Bind the nonlazy symbols of the module in the image that defines `symbolName` and let all lazy symbols in the module be bound on first call. You should pass this option when you expect the module to bind without errors (for example, a library supplied with the system). If, later, you call a lazy symbol, and the lazy symbol fails to bind, the runtime calls the link edit error handler you have installed using the `NSInstallLinkEditErrorHandlers` function.

If there is no link edit error handler installed, the runtime prints a message to the standard error stream and causes a breakpoint trap to end the program.

`NSLOOKUPSYMBOLINIMAGE_OPTION_BIND_NOW`

Bind all the nonlazy and lazy symbols of the module in the image that defines the symbol name, and bind symbols in the dependent libraries as needed.

Pass this option for a library that might not be expected to bind without errors but that links against only system-supplied libraries that are themselves expected to bind without any errors.

`NSLOOKUPSYMBOLINIMAGE_OPTION_BIND_FULLY`

Bind all the symbols of the module that defines `symbolName` and all the dependent symbols of all needed libraries.

Because it may take a long time to fully bind the image, you should pass this option only for libraries that cannot bind other symbols once executed, such as code that implements signal handlers.

`NSLOOKUPSYMBOLINIMAGE_OPTION_RETURN_ON_ERROR`

Return `NULL` if the symbol cannot be bound.

### Return Value

The symbol reference for the requested symbol, or `NULL` if the symbol cannot be found and you passed the option `NSLOOKUPSYMBOLINIMAGE_OPTION_RETURN_ON_ERROR`.

### Discussion

On error, if you have installed a link edit error handler, it is called; otherwise, this function writes an error message to file descriptor 2 (usually the standard error stream, `stderr`) and causes a breakpoint trap to end the program.

For portability and efficiency, consider using `dlsym` (page 14).

**Availability**
Available in Mac OS X v10.1 and later.

## NSLookupSymbolInModule

Given a module reference, returns a reference to the symbol with the given name.

```
NSSymbol NSLookupSymbolInModule(
        NSModule module,
        const char* symbolName);
```

**Parameters**

*module*

A module reference. Pass the module that contains the symbol.

*symbolname*

A pointer to a C string. Pass the name of the symbol to look up.

**Return Value**
The symbol reference or `NULL` if the symbol cannot be found.

**Discussion**
For portability and efficiency, consider using `dlsym` (page 14).

## NSModuleForSymbol

Returns a reference to the module containing the given symbol.

```
NSModule NSModuleForSymbol(
        NSSymbol symbol);
```

**Parameters**

*symbol*

A symbol reference. Pass the symbol whose module you wish to obtain.

**Return Value**
A reference to the module that contains the given symbol.

## NSNameOfModule

Returns the name of the given module.

```
const char* NSNameOfModule(
        NSModule module);
```

**Parameters**

*module*

A module reference. Pass the module whose name you wish to retrieve.

**Return Value**
A C string containing the name of the module. The string is owned by the dynamic linker and you should not free it.

**Discussion**
See "Building Mach-O Files" in *Mac OS X ABI File Format Reference*.for more information about modules.

## NSNameOfSymbol

Returns the name of the given symbol.

```
const char* NSNameOfSymbol(
        NSSymbol symbol);
```

**Parameters**

*symbol*

A symbol reference. Pass the symbol whose name you wish to obtain.

**Return Value**
A pointer to a C string containing the name of the reference. The dynamic linker owns this string and you should not free it.

## NSUnLinkModule

Unlinks the given module from the current program.

```
bool NSUnLinkModule(
        NSModule module,
        uint32_t options);
```

**Parameters**

*module*

A module reference. Pass a reference to a module that you have previously linked using the `NSLinkModule` (page 25) function.

*options*

An `unsigned long` value. You can specify one or more of the following bit masks:

`NSUNLINKMODULE_OPTION_NONE`

Unlink the module and deallocate the memory it occupies.

`NSUNLINKMODULE_OPTION_KEEP_MEMORY_MAPPED`

Unlink the module, but do not deallocate the memory it occupies. Addresses that reside within the module remain valid. You cannot unmap this memory later; it's released when the process exits or is terminated.

`NSUNLINKMODULE_OPTION_RESET_LAZY_REFERENCES`

Unlink the module and reset lazy references from other modules that are bound to the module. You can then link a new module that implements the same symbols, and the function call references are bound to the new module when accessed.

**Discussion**
For portability and efficiency, consider using `dlopen` (page 13) in conduction with `dlclose` (page 12).

See "Building Mach-O Files" in "Mach-O Programming Topics" for more information about modules.

**Special Considerations**

In Mac OS X v10.2 and later, `NSUNLINKMODULE_OPTION_RESET_LAZY_REFERENCES` can be used only with PowerPC CPU executables.

## _dyld_bind_fully_image_containing_address

Finds the image containing the specified address and fully binds all the modules within it.

```
bool _dyld_bind_fully_image_containing_address(
        const void* address);
```

**Parameters**

*address*

A pointer to an address located somewhere within a loaded image.

**Return Value**

A Boolean value. If `true`, the address resides somewhere within a loaded image, and so `_dyld_bind_fully_image_containing_address` attempted to bind that image. If `false`, the address does not reside within a loaded image, and so `_dyld_bind_fully_image_containing_address` did nothing.

**Discussion**

You can use this function to bind error handing code like signal handlers when you have the address of a function, but not the symbol name. This may bind more symbols than are actually needed.

If the image containing the address is a flat namespace image, multiple-defined errors can occur even if the symbols are not really used. Errors in binding are reported through the normal error reporting mechanisms.

## _dyld_bind_objc_module

Binds the module that contains a given Objective-C address.

```
void _dyld_bind_objc_module(
        const void* objc_module);
```

**Parameters**

*objc_module*

A pointer. Pass any address residing within the `__OBJC,__module` section of a loaded Mach-O file.

**Discussion**

This function is used by the Objective-C runtime library.

## _dyld_func_lookup

Obtains the address of the implementation of a `dyld` library function.

```
int _dyld_func_lookup(
        const char* dyld_func_name,
        void** address);
```

**Parameters**

*dyld_func_name*

A pointer to a C string. Pass the name of a `dyld` library function.

*address*

A pointer to a pointer. On output, points to the address of the function if the function is found, otherwise the value is undefined.

**Return Value**
An integer value. Nonzero if the function was found. Zero if the function was not found.

**Discussion**
This function is used by the library code that implements the `dyld` functions.

## _dyld_get_image_header

Returns the data structure for the header of a specified image. The image is specified by index into the list of images maintained by `dyld` for the current process.

```
const struct mach_header* _dyld_get_image_header(
        uint32_t image_index);
```

**Parameters**

*image_index*

A long integer. Pass a zero-based index indicating the position of the image in the list of images loaded into the address space of the current process.

**Return Value**
A pointer to the `mach_header` data structure of the specified image. If `image_index` is greater than the number of loaded images, this pointer is null.

## _dyld_get_image_name

Retrieves the name of an image.

```
const char* _dyld_get_image_name(
        uint32_t image_index);
```

**Parameters**

*image_index*

A long integer. Pass a zero-based index indicating the position of the image in the list of images loaded into the address space of the current process.

**Return Value**
A pointer to a C string. If `image_index` is greater than the number of loaded images, the string pointer is null.

**Discussion**
Returns the name of the image located at the given index into the global image list.

## _dyld_get_image_vmaddr_slide

Returns the virtual memory address slide amount of an image.

```
intptr_t _dyld_get_image_vmaddr_slide(
        uint32_t image_index);
```

**Parameters**

*image_index*

A long integer. Pass a zero-based index indicating the position of the image in the list of images loaded into the address space of the current process.

**Return Value**

If `image_index` is greater than or equal to the value returned by `_dyld_image_count` (page 33), zero. Otherwise, the `vmaddr_slide` value for the specified image.

**Discussion**

When the dynamic linker loads an image, the image must be mapped into the virtual address space of the process at an unoccupied address. The dynamic linker accomplishes this by adding a value—the virtual memory slide amount—to the base address of the image.

## _dyld_get_objc_module_sect_for_module

Obtains the size and starting location of an Objective-C module.

```
void _dyld_get_objc_module_sect_for_module(
        NSModule module,
        void** objc_module,
        size_t* size);
```

**Parameters**

*module*

A module reference from an image.

*objc_modulet*

A pointer to a pointer. On output, contains a pointer to the start of the `__OBJC,__module` section for the specified module.

*size*

A pointer to a long integer. On output, the long integer contains the a value indicating the size of the output module.

**Discussion**

This function is used by the Objective-C runtime library.

## _dyld_image_containing_address

Returns whether or not a specified address is within any loaded image.

```
bool _dyld_image_containing_address(
        const void* address);
```

**Parameters**

*address*

An unsigned long integer. Pass the address that you wish to obtain status about.

**Return Value**

`TRUE` if the address is located within an image loaded by the dynamic linker, `FALSE` otherwise.

## _dyld_image_count

Returns the number of images that `dyld` has mapped into the address space of the current process.

```
uint32_t _dyld_image_count(void);
```

**Return Value**
A long integer containing the number of images that `dyld` has mapped into the address space of the current process.

**Discussion**
This function provides you with a count of the number of the images in the image list. You can use this number to iterate the images loaded into the address space of the current process, using functions such as `_dyld_get_image_header` (page 31) and `_dyld_get_image_name` (page 31).

## _dyld_launched_prebound

Returns whether or not the dynamic linker was able to launch the program with the prebinding optimization enabled.

```
bool _dyld_launched_prebound(void);
```

**Return Value**
A Boolean value. `TRUE` if the program was launched successfully using the prebound state; `FALSE` if the either the program was not prebound or the prebinding couldn't be used for some reason.

**Discussion**
If the program was not successfully launched with the prebinding optimization, the linker did not prebind the program, the addresses of some images overlapped and so the linker could not use the prebound addresses, or some other problem occurred. In any case, the program continues to launch, but it runs slower than with prebinding enabled.

## _dyld_lookup_and_bind

Finds the given symbol name and binds it into the program.

```
void _dyld_lookup_and_bind(
        const char* symbol_name,
        void ** address
        NSModule* module);
```

**Parameters**

*symbol_name*
A pointer to a C string. Specify the name of the symbol to bind.

*address*
A pointer to a pointer. On output, points to the address of the symbol specified by `symbol_name`. This parameter is optional; pass `NULL` for this pointer on input if you do not want to retrieve this data.

*module*
A pointer to a module pointer. On output, the module pointer contains the module of the symbol specified by `symbol_name`. This parameter is optional; specify `NULL` for this pointer on input if you do not want to retrieve this data.

**Discussion**
Deprecated in Mac OS X v10.4. Use `NSLookupSymbolInImage` (page 27) instead.

You can use `_dyld_lookup_and_bind` to find a given symbol name in the global search list and bind it (and all other defined symbols in the same module) into the program.

If the program is prebound and you know the name of the library that contains the symbol, consider using `_dyld_lookup_and_bind_with_hint` (page 35) instead.

## _dyld_lookup_and_bind_fully

Finds the module containing the specified symbol and fully binds all the symbol references within it.

```
void _dyld_lookup_and_bind_fully(
        const char* symbol_name,
        void** address,
        NSModule* module);
```

**Parameters**

*symbol_name*

A pointer to a C string. Specify the name of the symbol to bind.

*address*

A pointer to a pointer. On output, points to the address of the specified symbol.

*module*

A pointer to a pointer. On output, the pointer is set to the address of the module in which the specified symbol resides.

**Discussion**
You can use this function to bind modules containing signal handlers or other error handling code that cannot be initialized lazily.

Errors in binding are reported through the normal mechanisms.

## _dyld_lookup_and_bind_objc

Obtains and binds the Objective-C module that contains the specified symbol.

```
void _dyld_lookup_and_bind_objc(
        const char* symbol_name,
        void** address,
        NSModule* module);
```

**Parameters**

*symbol_name*

A pointer to a C string. Specify the name of the symbol to bind, such as `.objc_class_name_Foo`.

*address*

A pointer to a pointer. On output, points to the address of the symbol specified by `symbol_name`. This parameter is optional; pass `NULL` for this pointer on input if you do not want to retrieve this data.

*module*

A pointer to a module pointer. On output, the module pointer contains the module of the symbol specified by `symbol_name`. This parameter is optional; specify `NULL` for this pointer on input if you do not want to retrieve this data.

**Discussion**
This routine is used by the Objective-C runtime library. It performs the same function as `_dyld_lookup_and_bind` (page 33) but, for performance reasons, does not update the symbol pointers if the symbol is in a bound module. An Objective-C symbol such as `.objc_class_name_Object` is never used by a symbol pointer, and updating the symbol pointers is a relatively expensive operation; so this provides a way for the Objective-C runtime to avoid that overhead.

## _dyld_lookup_and_bind_with_hint

Finds the given symbol name and binds it into the program, with a hint to allow `dyld` to speed up the symbol search for a prebound program.

```
void _dyld_lookup_and_bind_with_hint(
        const char* symbol_name,
        const char* library_name_hint,
        void** address,
        NSModule* module);
```

**Parameters**

*symbol_name*

A pointer to a C string. Specify the name of the symbol to bind.

*library_name_hint*

A pointer to a C string. Specify the name of the library in which the symbol is probably located. The dynamic linker compares this name with the actual library install names using the standard C library function `strstr`.

*address*

A pointer to a pointer. On output, points to the address of the symbol specified by `symbol_name`. This parameter is optional; pass `NULL` for this pointer on input if you do not want to retrieve this data.

*module*

A pointer to a module pointer. On output, the module pointer contains the module of the symbol specified by `symbol_name`. This parameter is optional; specify `NULL` for this pointer on input if you do not want to retrieve this data.

**Discussion**
Deprecated in Mac OS X v10.4. Use `NSLookupSymbolInImage` (page 27) instead.

You can use `_dyld_lookup_and_bind_with_hint` to quickly find a given symbol name in the global search list of a prebound program and bind the symbol (and all other defined symbols in the same module) into the program.

## _dyld_moninit

This function is used by the profiling routine `moninit` to allow images other than the main executable to be profiled.

```
void _dyld_moninit( void (*monaddition)(
        char* lowpc,
        char* highpc);
```

**Parameters**

*monaddition*

A pointer to a callback function. The callback is called when an image is first mapped in.

**Discussion**
This function is usually called by the profiling runtime (specifically, from the `moninit` function). It is documented here for completeness. See the man page for `moninit` and `monaddtion` for further information.

## _dyld_present

Indicates whether or not the dynamic linker is loaded into the current program

```
bool _dyld_present (void);
```

**Return Value**
A Boolean value indicating the presence of `dyld`. This value is `FALSE` if `dyld` is not loaded in the current process, and `TRUE` if `dyld` is loaded in the current process.

## _dyld_register_func_for_add_image

Registers a function to be called by the dynamic linker runtime when an image is added to the program.

```
void _dyld_register_func_for_add_image(
        void (*func)(struct mach_header* mh,         intptr_t vmaddr_slide));
```

**Parameters**

*func*

A pointer to a callback function that accepts a pointer to a `mach_header` data structure and a virtual memory slide amount. The virtual memory slide amount specifies the difference between the address at which the image was linked and the address at which the image is loaded.

**Discussion**
When you call `_dyld_register_func_for_add_image`, the dynamic linker runtime calls the specified callback (`func`) once for each of the images that are currently loaded into the program. When a new image is added to the program, your callback is called again with the `mach_header` for the new image, and the virtual memory slide amount of the new image.

You might use this, for example, in implementing a runtime system, such as the Objective-C runtime, to discover when new images are added to the program.

## _dyld_register_func_for_link_module

Registers a function to be called by the dynamic linker runtime when a module is linked into the program.

```
void _dyld_register_func_for_link_module(
        void (*func)(NSModule module));
```

**Parameters**

*func*

A pointer to a callback that accepts a module reference.

**Discussion**
When you call `_dyld_register_func_for_link_module`, the dynamic linker runtime calls the specified callback (`func`) once for each module that is currently linked into the program. When a new module is linked into the program, the `func` callback is called again for that module.

### _dyld_register_func_for_remove_image

Registers a function to be called by the dynamic linker runtime when an image is removed from the program.

```
void _dyld_register_func_for_remove_image(
        void (*func) (struct mach_header* mh),
        intptr_t vmaddr_slide) );
```

**Parameters**

*func*

A pointer to a callback function that accepts a pointer to a `mach_header` data structure and a virtual memory slide amount. The virtual memory slide amount specifies the difference between the address at which the image was linked and the address at which the image is loaded.

# Constants

## Mach-O Image Creation Return Codes

Potential return values when creating a Mach-O image.

```
typedef enum {
    NSObjectFileImageFailure,
    NSObjectFileImageSuccess,
    NSObjectFileImageInappropriateFile,
    NSObjectFileImageArch,
    NSObjectFileImageFormat,
    NSObjectFileImageAccess
} NSObjectFileImageReturnCode;
```

**Constants**

`NSObjectFileImageSuccess`

The image creation operation was completed successfully.

`NSObjectFileImageFailure`

The image creation operation was not successfully completed.

When this value is returned, an error message is printed to the standard error stream.

`NSObjectFileImageInappropriateFile`

The Mach-O file is not of a type the called function can operate upon.

`NSObjectFileImageArch`

The specified Mach-O file is for a different CPU architecture.

`NSObjectFileImageFormat`

The specified file or memory block does not appear to point to a Mach-O file.

`NSObjectFileImageAccess`

The access permissions for the specified file do not permit the creation of the image.

**Discussion**

These return values are returned from `NSCreateObjectFileImageFromFile` (page 21) and `NSCreateObjectFileImageFromMemory` (page 22).

**Declared In**

`mach-o/dyld.h`

# Document Revision History

This table describes the changes to *Mac OS X ABI Dynamic Loader Reference*.

| Date | Notes |
| --- | --- |
| 2005-11-09 | Made minor corrections to the "Dynamic Loader Functions" section. |
| | Changed `DL_info` to `Dl_info` in `dladdr` (page 11) prototype. |
| | Added paragraph explaining difference in symbol-name specification between `NS...` functions and `dlsym` (page 14). |
| | Changed title to "Mac OS X ABI Dynamic Loader Reference." |
| | Added the phrase “Mac OS X application binary interface (ABI)” to the introduction to raise this document’s visibility in searches. |
| 2005-08-11 | Clarified terminology for binaries that contain 32-bit and 64-bit object code. |
| 2005-06-04 | Corrected descriptions of dlopen and dlsym. |
| 2005-04-29 | Added information on new dynamic loader functions. Updated for 64-bit support in Mac OS X v10.4. Changed title from "Mach-O Runtime Reference." |
| | Added "Dynamic Loader Compatibility Functions" (page 7) section with information on `dlopen`, `dlsym`, `dlclose`, `dladdr`, and `dlerror`. |
| | Updated function declarations to reflect Mac OS X v10.4 64-bit support. Also updated availability information for deprecated functions. |
| | Moved non–Mach-O bundle functions `NSAddImage`, `NSAddressOfSymbol`, `NSLibraryNameForModule`, `NSLookupSymbolInImage`, `NSNameOfSymbol` from "Object File Image Functions" (page 8) to "Library Functions" (page 9). |
| 2004-08-31 | New document that describes the functions to access Mach-O files at runtime. |
| | This document replaces Mach-O reference information that was published previously in *Mach-O Runtime Architecture*. |
| | Added "Library Functions" (page 9) describing `NSAddLibrary` and `NSAddLibraryWithSearching` functions. |

Document Revision History

# Index

## Symbols


_dyld_bind_fully_image_containing_address function 30
_dyld_bind_objc_module function 30
_dyld_func_lookup function 30
_dyld_get_image_header function 31
_dyld_get_image_name function 31
_dyld_get_image_vmaddr_slide function 31
_dyld_get_objc_module_sect_for_module function 32
_dyld_image_containing_address function 32
_dyld_image_count function 33
_dyld_launched_prebound function 33
_dyld_lookup_and_bind function 33
_dyld_lookup_and_bind_fully function 34
_dyld_lookup_and_bind_objc function 34
_dyld_lookup_and_bind_with_hint function 35
_dyld_moninit function 35
_dyld_present function 36
_dyld_register_func_for_add_image function 36
_dyld_register_func_for_link_module function 36
_dyld_register_func_for_remove_image function 37


## D


dladdr function 11
dlclose function 12
dlerror function 12
dlopen function 13
dlsym function 14
dyld_stub_binding_helper function 15


## G


getsectbyname function 16
getsectbynamefromheader function 16
getsectbynamefromheader_64 function 16
getsectdata function 17
getsectdatafromFramework function 17
getsectdatafromheader function 18
getsectdatafromheader_64 function 18
getsegbyname function 19


## M


Mach-O Image Creation Return Codes 37


## N


NSAddImage function 19
NSAddLibrary function 20
NSAddLibraryWithSearching function 21
NSAddressOfSymbol function 21
NSCreateObjectFileImageFromFile function 21
NSCreateObjectFileImageFromMemory function 22
NSDestroyObjectFileImage function 23
NSIsSymbolNameDefined function 23
NSIsSymbolNameDefinedInImage function 23
NSIsSymbolNameDefinedWithHint function 24
NSLibraryNameForModule function 24
NSLinkModule function 25
NSLookupAndBindSymbol function 26
NSLookupAndBindSymbolWithHint function 26
NSLookupSymbolInImage function 27
NSLookupSymbolInModule function 28
NSModuleForSymbol function 28
NSNameOfModule function 28
NSNameOfSymbol function 29
NSObjectFileImageAccess constant 37
NSObjectFileImageArch constant 37
NSObjectFileImageFailure constant 37
NSObjectFileImageFormat constant 37
NSObjectFileImageInappropriateFile constant 37
NSObjectFileImageSuccess constant 37

NSUnLinkModule function 29