NAME

DynaLoader - Dynamically load C libraries into Perl code

SYNOPSIS

```
package YourPackage;
require DynaLoader;
@ISA = qw(... DynaLoader ...);
bootstrap YourPackage;

# optional method for 'global' loading
sub dl_load_flags { 0x01 }
```

DESCRIPTION

This document defines a standard generic interface to the dynamic linking mechanisms available on many platforms. Its primary purpose is to implement automatic dynamic loading of Perl modules.

This document serves as both a specification for anyone wishing to implement the DynaLoader for a new platform and as a guide for anyone wishing to use the DynaLoader directly in an application.

The DynaLoader is designed to be a very simple high-level interface that is sufficiently general to cover the requirements of SunOS, HP-UX, NeXT, Linux, VMS and other platforms.

It is also hoped that the interface will cover the needs of OS/2, NT etc and also allow pseudo-dynamic linking (using `ld -A` at runtime).

It must be stressed that the DynaLoader, by itself, is practically useless for accessing non-Perl libraries because it provides almost no Perl-to-C 'glue'. There is, for example, no mechanism for calling a C library function or supplying arguments. A C::DynaLib module is available from CPAN sites which performs that function for some common system types. And since the year 2000, there's also Inline::C, a module that allows you to write Perl subroutines in C. Also available from your local CPAN site.

DynaLoader Interface Summary

```
  @dl_library_path
  @dl_resolve_using
  @dl_require_symbols
  $dl_debug
  @dl_librefs
  @dl_modules
  @dl_shared_objects
                                                  Implemented in:
  bootstrap($modulename)                               Perl
  @filepaths = dl_findfile(@names)                     Perl
  $flags = $modulename->dl_load_flags                  Perl
  $symref  = dl_find_symbol_anywhere($symbol)          Perl

  $libref  = dl_load_file($filename, $flags)           C
  $status  = dl_unload_file($libref)                   C
  $symref  = dl_find_symbol($libref, $symbol)          C
  @symbols = dl_undef_symbols()                        C
  dl_install_xsub($name, $symref [, $filename])        C
  $message = dl_error                                  C
```

@dl_library_path

The standard/default list of directories in which dl_findfile() will search for libraries etc.

Directories are searched in order: $dl_library_path[0], [1], ... etc

@dl_library_path is initialised to hold the list of 'normal' directories (*/usr/lib*, etc) determined by **Configure** (`$Config{'libpth'}`). This should ensure portability across a wide range of platforms.

@dl_library_path should also be initialised with any other directories that can be determined from the environment at runtime (such as LD_LIBRARY_PATH for SunOS).

After initialisation @dl_library_path can be manipulated by an application using push and unshift before calling dl_findfile(). Unshift can be used to add directories to the front of the search order either to save search time or to override libraries with the same name in the 'normal' directories.

The load function that dl_load_file() calls may require an absolute pathname. The dl_findfile() function and @dl_library_path can be used to search for and return the absolute pathname for the library/object that you wish to load.

@dl_resolve_using

A list of additional libraries or other shared objects which can be used to resolve any undefined symbols that might be generated by a later call to load_file().

This is only required on some platforms which do not handle dependent libraries automatically. For example the Socket Perl extension library (*auto/Socket/Socket.so*) contains references to many socket functions which need to be resolved when it's loaded. Most platforms will automatically know where to find the 'dependent' library (e.g., */usr/lib/libsocket.so*). A few platforms need to be told the location of the dependent library explicitly. Use @dl_resolve_using for this.

Example usage:

```
@dl_resolve_using = dl_findfile('-lsocket');
```

@dl_require_symbols

A list of one or more symbol names that are in the library/object file to be dynamically loaded. This is only required on some platforms.

@dl_librefs

An array of the handles returned by successful calls to dl_load_file(), made by bootstrap, in the order in which they were loaded. Can be used with dl_find_symbol() to look for a symbol in any of the loaded files.

@dl_modules

An array of module (package) names that have been bootstrap'ed.

@dl_shared_objects

An array of file names for the shared objects that were loaded.

dl_error()

Syntax:

```
$message = dl_error();
```

Error message text from the last failed DynaLoader function. Note that, similar to errno in unix, a successful function call does not reset this message.

Implementations should detect the error as soon as it occurs in any of the other functions and save the corresponding message for later retrieval. This will avoid problems on some platforms (such as SunOS) where the error message is very temporary (e.g., dlerror()).

$dl_debug

Internal debugging messages are enabled when $dl_debug is set true. Currently setting $dl_debug only affects the Perl side of the DynaLoader. These messages should help an application developer to resolve any DynaLoader usage problems.

$dl_debug is set to `$ENV{'PERL_DL_DEBUG'}` if defined.

For the DynaLoader developer/porter there is a similar debugging variable added to the C code (see dlutils.c) and enabled if Perl was built with the **-DDEBUGGING** flag. This can also be set via the PERL_DL_DEBUG environment variable. Set to 1 for minimal information or higher for more.

dl_findfile()

Syntax:

```
@filepaths = dl_findfile(@names)
```

Determine the full paths (including file suffix) of one or more loadable files given their generic names and optionally one or more directories. Searches directories in @dl_library_path by default and returns an empty list if no files were found.

Names can be specified in a variety of platform independent forms. Any names in the form **-lname** are converted into *libname.**, where *.** is an appropriate suffix for the platform.

If a name does not already have a suitable prefix and/or suffix then the corresponding file will be searched for by trying combinations of prefix and suffix appropriate to the platform: "$name.o", "lib$name.*" and "$name".

If any directories are included in @names they are searched before @dl_library_path. Directories may be specified as **-Ldir**. Any other names are treated as filenames to be searched for.

Using arguments of the form `-Ldir` and `-lname` is recommended.

Example:

```
@dl_resolve_using = dl_findfile(qw(-L/usr/5lib -lposix));
```

dl_expandspec()

Syntax:

```
$filepath = dl_expandspec($spec)
```

Some unusual systems, such as VMS, require special filename handling in order to deal with symbolic names for files (i.e., VMS's Logical Names).

To support these systems a dl_expandspec() function can be implemented either in the *dl_*.xs* file or code can be added to the dl_expandspec() function in *DynaLoader.pm*. See *DynaLoader_pm.PL* for more information.

dl_load_file()

Syntax:

```
$libref = dl_load_file($filename, $flags)
```

Dynamically load $filename, which must be the path to a shared object or library. An opaque 'library reference' is returned as a handle for the loaded object. Returns undef on error.

The $flags argument to alters dl_load_file behaviour. Assigned bits:

```
 0x01  make symbols available for linking later dl_load_file's.
       (only known to work on Solaris 2 using dlopen(RTLD_GLOBAL))
       (ignored under VMS; this is a normal part of image linking)
```

(On systems that provide a handle for the loaded object such as SunOS and HPUX, $libref will be that handle. On other systems $libref will typically be $filename or a pointer to a buffer

containing $filename. The application should not examine or alter $libref in any way.)

This is the function that does the real work. It should use the current values of @dl_require_symbols and @dl_resolve_using if required.

```
    SunOS: dlopen($filename)
    HP-UX: shl_load($filename)
    Linux: dld_create_reference(@dl_require_symbols);
dld_link($filename)
    NeXT:  rld_load($filename, @dl_resolve_using)
    VMS:   lib$find_image_symbol($filename,$dl_require_symbols[0])
```

(The dlopen() function is also used by Solaris and some versions of Linux, and is a common choice when providing a "wrapper" on other mechanisms as is done in the OS/2 port.)

dl_unload_file()

Syntax:

```
    $status = dl_unload_file($libref)
```

Dynamically unload $libref, which must be an opaque 'library reference' as returned from dl_load_file. Returns one on success and zero on failure.

This function is optional and may not necessarily be provided on all platforms. If it is defined, it is called automatically when the interpreter exits for every shared object or library loaded by DynaLoader::bootstrap. All such library references are stored in @dl_librefs by DynaLoader::Bootstrap as it loads the libraries. The files are unloaded in last-in, first-out order.

This unloading is usually necessary when embedding a shared-object perl (e.g. one configured with -Duseshrplib) within a larger application, and the perl interpreter is created and destroyed several times within the lifetime of the application. In this case it is possible that the system dynamic linker will unload and then subsequently reload the shared libperl without relocating any references to it from any files DynaLoaded by the previous incarnation of the interpreter. As a result, any shared objects opened by DynaLoader may point to a now invalid 'ghost' of the libperl shared object, causing apparently random memory corruption and crashes. This behaviour is most commonly seen when using Apache and mod_perl built with the APXS mechanism.

```
    SunOS: dlclose($libref)
    HP-UX: ???
    Linux: ???
    NeXT:  ???
    VMS:   ???
```

(The dlclose() function is also used by Solaris and some versions of Linux, and is a common choice when providing a "wrapper" on other mechanisms as is done in the OS/2 port.)

dl_load_flags()

Syntax:

```
    $flags = dl_load_flags $modulename;
```

Designed to be a method call, and to be overridden by a derived class (i.e. a class which has DynaLoader in its @ISA). The definition in DynaLoader itself returns 0, which produces standard behavior from dl_load_file().

dl_find_symbol()

Syntax:

```
    $symref = dl_find_symbol($libref, $symbol)
```

Return the address of the symbol $symbol or `undef` if not found. If the target system has separate functions to search for symbols of different types then dl_find_symbol() should search for function symbols first and then other types.

The exact manner in which the address is returned in $symref is not currently defined. The only initial requirement is that $symref can be passed to, and understood by, dl_install_xsub().

```
SunOS: dlsym($libref, $symbol)
HP-UX: shl_findsym($libref, $symbol)
Linux: dld_get_func($symbol) and/or dld_get_symbol($symbol)
NeXT:  rld_lookup("_$symbol")
VMS:   lib$find_image_symbol($libref,$symbol)
```

dl_find_symbol_anywhere()

Syntax:

```
$symref = dl_find_symbol_anywhere($symbol)
```

Applies dl_find_symbol() to the members of @dl_librefs and returns the first match found.

dl_undef_symbols()

Example

```
@symbols = dl_undef_symbols()
```

Return a list of symbol names which remain undefined after load_file(). Returns `( )` if not known. Don't worry if your platform does not provide a mechanism for this. Most do not need it and hence do not provide it, they just return an empty list.

dl_install_xsub()

Syntax:

```
dl_install_xsub($perl_name, $symref [, $filename])
```

Create a new Perl external subroutine named $perl_name using $symref as a pointer to the function which implements the routine. This is simply a direct call to newXSUB(). Returns a reference to the installed function.

The $filename parameter is used by Perl to identify the source file for the function if required by die(), caller() or the debugger. If $filename is not defined then "DynaLoader" will be used.

bootstrap()

Syntax:

bootstrap($module [...])

This is the normal entry point for automatic dynamic loading in Perl.

It performs the following actions:

- locates an auto/$module directory by searching @INC
- uses dl_findfile() to determine the filename to load
- sets @dl_require_symbols to `("boot_$module")`
- executes an *auto/$module/$module.bs* file if it exists (typically used to add to @dl_resolve_using any files which are required to load the module on the current platform)
- calls dl_load_flags() to determine how to load the file.
- calls dl_load_file() to load the file

- calls dl_undef_symbols() and warns if any symbols are undefined
- calls dl_find_symbol() for "boot_$module"
- calls dl_install_xsub() to install it as "${module}::bootstrap"
- calls &{"${module}::bootstrap"} to bootstrap the module (actually it uses the function reference returned by dl_install_xsub for speed)

All arguments to bootstrap() are passed to the module's bootstrap function. The default code generated by *xsubpp* expects $module [, $version] If the optional $version argument is not given, it defaults to `$XS_VERSION // $VERSION` in the module's symbol table. The default code compares the Perl-space version with the version of the compiled XS code, and croaks with an error if they do not match.

AUTHOR

Tim Bunce, 11 August 1994.

This interface is based on the work and comments of (in no particular order): Larry Wall, Robert Sanders, Dean Roehrich, Jeff Okamoto, Anno Siegel, Thomas Neumann, Paul Marquess, Charles Bailey, myself and others.

Larry Wall designed the elegant inherited bootstrap mechanism and implemented the first Perl 5 dynamic loader using it.

Solaris global loading added by Nick Ing-Simmons with design/coding assistance from Tim Bunce, January 1996.