

Package ‘rdyncall’

February 15, 2012

Version 0.7.4

Date 2011-12-13

Title Improved Foreign Function Interface (FFI) and Dynamic Bindings to C Libraries (e.g. OpenGL)

Author Daniel Adler <dadler@uni-goettingen.de>

Maintainer Daniel Adler <dadler@uni-goettingen.de>

Depends R (>= 2.4)

Description The package provides a cross-platform framework for dynamic binding of C libraries using a flexible Foreign Function Interface (FFI). The FFI supports almost all fundamental C types, multiple calling conventions, symbolic access to foreign C struct/union data types and wrapping of R functions as C callback function pointers. Dynamic bindings to shared C libraries are data-driven by cross-platform binding specification using a compact plain text format ; an initial repository of bindings to a couple of common C libraries (OpenGL, SDL, Expat, glew, CUDA, OpenCL, ODE, R) comes with the package. The package includes a variety of technology demos and OS-specific notes for installation of shared libraries.

License BSD

URL <http://dyncall.org>

Repository CRAN

Date/Publication 2011-12-13 16:50:20

R topics documented:

| | |
|----------|---|
| callback | 2 |
| dynbind | 4 |
| dyncall | 6 |
| dynfind | 9 |

| | |
|----------------|----|
| dynload | 11 |
| dynport | 14 |
| packing | 16 |
| rdyncall | 18 |
| rdyncall-demos | 20 |
| struct | 25 |
| typeinfo | 27 |
| utils | 28 |

| | |
|--------------|-----------|
| Index | 30 |
|--------------|-----------|

| | |
|----------|---|
| callback | <i>Dynamic wrapping of R functions as C callbacks</i> |
|----------|---|

Description

Function to wrap R functions as C function pointers.

Usage

```
new.callback( signature, fun, envir = new.env() )
```

Arguments

| | |
|-----------|---|
| signature | character string specifying the call signature of the C function callback type. |
| fun | R function to be wrapped as a C function pointer. |
| envir | the environment in which to evaluate the call to fun. |

Details

Callbacks are user-defined functions that are registered in a foreign library and that are executed at a later time from within that library. Examples include user-interface event handlers that are registered in GUI toolkits, and, comparison functions for custom data types to be passed to generic sort algorithm.

The function `new.callback` wraps an R function `fun` as a C function pointer and returns an external pointer. The foreign C function type of the wrapped R function is specified by a [call signature](#) given by `signature`.

When the C function pointer is called, a global callback handler (implemented in C) is executed first, that dynamically creates an R call expression to `fun` using the arguments, passed from C and converted to R, according to the *argument types signature* within the [call signature](#) specified. See [.dyncall](#) for details on the format. Finally, the handler evaluates the R call expression within the environment given by `envir`. On return, the R return value of `fun` is coerced to the C value, according to the return type signature specified in `signature`. If an error occurs during the evaluation, the callback will be disabled for further invocations. (This behaviour might change in the future.)

Value

`new.callback` returns an external pointer to a synthetically generated C function.

Portability

The implementation is based on the *dyncallback* library (part of the DynCall project).

The following processor architectures are supported: X86, X64, ARM (including Thumb) and partial stable support for PowerPC 32-bit; The library has been built and tested to work on various OSs: Linux, Mac OS X, Windows 32/64-bit, BSDs, Haiku, Nexenta/Open Solaris, Minix and Plan9, as well as embedded platforms such as Linux/ARM (OpenMoko, Beagleboard, Gumstix), Nintendo DS (ARM), Sony Playstation Portable (MIPS 32-bit/eabi) and iOS (ARM - armv6 mode ok, armv7 unstable). Special notes for PowerPC 32-Bit: Callbacks for System V (Linux/BSD) are unstable in this release; MacOS X/Darwin works fine. In the context of R, dyncallback has currently no support for callbacks on MIPS, SPARC and PowerPC 64-Bit. Using dyncallback to implement non-default calling conventions is not supported yet. (e.g. Window Procedures on Win32/X86).

Note

The call signature **MUST** match the foreign C callback function type, otherwise an activated callback call from C can lead to a **fatal R process crash**.

A small amount of memory is allocated with each wrapper. A finalizer function that frees the allocated memory is registered at the external pointer. If the external callback function pointer is registered in a C library, a reference should also be held in R as long as the callback can be activated from a foreign C run-time context, otherwise the garbage collector might call the finalizer and the next invocation of the callback could lead to a **fatal R process crash** as well.

References

Adler, D., Philipp, T. (2008) *DynCall Project*. <http://dyncall.org>

See Also

See [signature](#) for details on call signatures, [reg.finalizer](#) for details on finalizers.

Examples

```
# Create a function, wrap it to a callback and call it via .dyncall:
f <- function(x,y) x+y
cb <- new.callback("ii)i", f)
r <- .dyncall(cb, "ii)i", 20, 3)

# Sort vectors directly via 'qsort' C library function using an R callback:
dynbind(c("msvcrt","c","c.so.6"), "qsort(piip)v;")
cb <- new.callback("pp)i",function(px,py){
  x <- .unpack(px, 0, "d")
  y <- .unpack(py, 0, "d")
  if (x > y) return(1) else if (x == y) return(0) else return(-1)
})
x <- rnorm(100)
qsort(x,length(x),8,cb)
```

 dynbind

Binding C library functions via thin call wrappers

Description

Function to bind several foreign functions of a C library via installation of thin R call wrappers.

Usage

```
dynbind(libnames, signature,
        envir=parent.frame(), callmode="default",
        pat=NULL, replace=NULL, funcptr=FALSE)
```

Arguments

| | |
|-----------|--|
| libnames | vector of character strings giving short library names of the shared library to be loaded. See dynfind for details. |
| signature | character string specifying the <i>library signature</i> that determines the set of foreign function names and types. See details. |
| envir | the environment to use for installation of call wrappers. |
| callmode | character string specifying the calling convention, see details. |
| pat | NULL or regular expression character string applied to symbolic names. |
| replace | NULL or replacement character string applied to pat part of symbolic names. |
| funcptr | logical, that indicates whether foreign objects refer to functions (FALSE, default) or to function pointer variables (TRUE rarely needed). |

Details

dynbind makes a set of C functions available to R through installation of thin call wrappers. The set of functions, including the symbolic name and function type, is specified by *signature* ; a character string that encodes a library signature:

The **library signature** is a compact plain-text format to specify a set of function bindings. It consists of function names and corresponding [call signatures](#). Function bindings are separated by ‘;’ (semicolon); white spaces (including tab and new line) are allowed before and after semicolon.

```
‘function-name’ ‘(’ ‘call-signature’ ‘;’ ...
```

Here is an example that specifies three function bindings to the OpenGL library:

```
"glAccum(If)v ; glClear(I)v ; glClearColor(ffff)v ;"
```

Symbolic names are resolved using the library specified by libnames using [dynfind](#) for loading. For each function, a thin call wrapper function is created using the following template:

```
function(...) .dyncall.<MODE> ( <TARGET>, <SIGNATURE>, ... )
```

<MODE> is replaced by `callmode` argument, see [.dyncall](#) for details on calling conventions. <TARGET> is replaced by the external pointer, resolved by the ‘function-name’. <SIGNATURE> is replaced by the call signature string contained in `signature`.

The call wrapper is installed in the environment given by `envir`. The assignment name is obtained from the function signature. If `pat` and `replace` is given, a text replacement is applied to the name before assignment, useful for basic C name space mangling such as exchanging the prefix.

As a special case, `dynbind` supports binding of pointer-to-function variables, indicated by setting `funcptr` to `TRUE`, in which case <TARGET> is replaced with the expression `.unpack(<TARGET>, "p", 0)` in order to dereference <TARGET> as a pointer-to-function variable at call-time.

Value

The function returns a list with two fields:

```
libhandle      External pointer returned by .dynload.
unresolved.symbols
                vector of character strings, the names of unresolved symbols.
```

As a side effect, for each wrapper, `dynbind` assigns the ‘function-name’ to the corresponding call wrapper function in the environment given by `envir`.

If no shared library is found, an error is reported.

See Also

[.dyncall](#) for details on call signatures and calling conventions, [dynfind](#) for details on short library names, [.unpack](#) for details on reading low-level memory (e.g. dereferencing of (function) pointer variables).

Examples

```
# Install two wrappers to functions of the R shared C library.
info <- dynbind("R",
R_ShowMessage(Z)v;
R_rsort(pi)v;
")
R_ShowMessage("hello")
```

dyncall

*Foreign Function Interface with support for almost all C types***Description**

Functions to call pre-compiled code with support for most C argument and return types.

Usage

```
.dyncall( address, signature, ... , callmode = "default" )
.dyncall.default      ( address, signature, ... )
.dyncall.cdecl        ( address, signature, ... )
.dyncall.stdcall      ( address, signature, ... )
.dyncall.thiscall     ( address, signature, ... )
.dyncall.thiscall.msvc( address, signature, ... )
.dyncall.thiscall.gcc ( address, signature, ... )
.dyncall.fastcall.msvc( address, signature, ... )
.dyncall.fastcall.gcc ( address, signature, ... )
```

Arguments

| | |
|-----------|--|
| address | external pointer to foreign function. |
| signature | character string specifying the <i>call signature</i> that describes the foreign function type. See details. |
| callmode | character string specifying the <i>calling convention</i> . This argument has no effect on most platforms, but on Microsoft Windows 32-Bit Intel/x86 platforms. See details. |
| ... | arguments to be passed to the foreign function. Arguments are converted from R to C values according to the <i>call signature</i> . See details. |

Details

.dyncall offers a flexible Foreign Function Interface (FFI) for the C language with support for calls to arbitrary pre-compiled C function types at run-time. Almost all C fundamental argument- and return types are supported including extended support for pointers. No limitations is given for arity as well. In addition, on the Microsoft Windows 32-Bit Intel/x86 platform, it supports multiple calling conventions to interoperate with System DLLs. Foreign C function types are specified via plain text *type signatures*. The foreign C function type of the target function is known to the FFI in advance, before preparation of the foreign call via plain text *type signature* information. This has several advantages: R arguments do not need to match exactly. Although R lacks some fundamental C value types, they are supported via coercion at this interface (e.g. C float and 64-bit integer). Arity and argument type checks help make this interface type-safe to a certain degree and encourage end-users to use interface from the interpreter prompt for rapid application development.

The foreign function to be called is specified by address, which is an external pointer that is obtained from [.dynsym](#) or [getNativeSymbolInfo](#).

signature is a character string that specifies the formal argument-and-return types of the foreign function using a *call signature* string. It should match the function type of the foreign function given by address, otherwise this can lead to a **fatal R process crash**.

The calling convention is specified *explicitly* via function `.dyncall` using the `callmode` argument or *implicitly* by using `.dyncall.*` functions. See details below.

Arguments passed via `. . .` are converted to C according to `signature`; see below for details.

Given that the signature matches the foreign function type, the FFI provides a certain level of type-safety to users, when exposing foreign functions via call wrappers such as done in `dynbind` and `dynport`. Several basic argument type-safety checks are done during preparation of the foreign function call: The arity of formals and actual arguments must match and they must be compatible as well. Otherwise, the foreign function call is aborted with an error before risking a fatal system crash.

Value

Functions return the received C return value converted to an R value. See section ‘Call Signature’ below for details.

Type Signature

Type signatures are used by almost all other signature formats (call, library, structure and union signature) and also by the low-level (un)-[packing](#) functions.

The following table gives a list of valid type signatures for all supported C types.

| Type Signature | C type | valid R argument types | R return type |
|--------------------------|----------------------------|-------------------------------------|-------------------|
| 'B' | bool | raw,logical,integer,double | logical |
| 'c' | char | raw,logical,integer,double | integer |
| 'C' | unsigned char | raw,logical,integer,double | integer |
| 's' | short | raw,logical,integer,double | integer |
| 'S' | unsigned short | raw,logical,integer,double | integer |
| 'i' | int | raw,logical,integer,double | integer |
| 'I' | unsigned int | raw,logical,integer,double | double |
| 'j' | long | raw,logical,integer,double | double |
| 'J' | unsigned long | raw,logical,integer,double | double |
| 'l' | long long | raw,logical,integer,double | double |
| 'L' | unsigned long long | raw,logical,integer,double | double |
| 'f' | float | raw,logical,integer,double | double |
| 'd' | double | raw,logical,integer,double | double |
| 'p' | <i>C pointer</i> | <i>any vector</i> ,externalptr,NULL | externalptr |
| 'Z' | char* | character,NULL | character or NULL |
| 'x' | SEXP | <i>any</i> | <i>any</i> |
| 'v' | void | <i>invalid</i> | NULL |
| '*' . . . | <i>C type*</i> (pointer) | <i>any vector</i> ,externalptr,NULL | externalptr |
| "*<" <i>typename</i> '>' | <i>typename*</i> (pointer) | raw,externalptr | externalptr |

The last two rows of the table the above refer to *typed pointer* signatures. If they appear as a return type signature, the external pointer returned is a S3 struct object. See [new.struct](#) for details.

Call Signatures

Call Signatures are used by `.dyncall` and `new.callback` to describe foreign C function types. The general form of a call signature is as following:

```
'argument-types' ')' 'return-type'
```

The calling sequence given by the **argument types signature** is specified in direct *left-to-right* order of the formal argument types defined in C. The type signatures are put in sequence without any white space in between. A closing bracket character `)'` marks the end of argument types, followed by a single **return type signature**.

Here are some examples of C library functions and corresponding call signatures:

```
double sqrt(double);           "d)d"
double dnorm(double,double,double,int); "dddi)d"
void R_isort(int*,int);        "pi)v"
void revsort(double*,int*,int); "ppi)v"
```

Calling convention

Calling Conventions specify 'how' sub-routine calls are performed, and, 'how' arguments and results are passed, on machine-level. They differ significantly among families of CPU Architectures as well as OS and Compiler implementations.

On most platforms, a single "default" C Calling Convention is used. As an exception, on the Microsoft Windows 32-Bit Intel/x86 platform several calling conventions are common. Most of the C libraries still use a "default" C (also known as "cdecl") calling convention, but when working with Microsoft System APIs and DLLs, the "stdcall" calling convention must be used.

It follows a description of supported Win32 Calling Conventions:

"cdecl" Dummy alias to *default*

"stdcall" C functions with *stdcall* calling convention. Useful for all Microsoft Windows System Libraries (e.g. KERNEL32.DLL, USER32.DLL, OPENGL32.DLL ...). Third-party libraries usually prefer the default C *cdecl* calling convention.

"fastcall.msvc" C functions with *fastcall* calling convention compiled with Microsoft Visual C++ Compiler. Very rare usage.

"fastcall.gcc" C functions with *fastcall* calling convention compiled with GNU C Compiler. Very rare usage.

"thiscall" C++ member functions.

"thiscall.gcc" C++ member functions compiled with GNU C Compiler.

"thiscall.msvc" C++ member functions compiled with Microsoft Visual C++ Compiler.

As of the current version of this package and for practical reasons, the `callmode` argument does not have an effect on almost all platforms, except that if R is running on Microsoft Windows 32-Bit Intel/x86 platform, `.dyncall` uses the specified calling convention. For example, when loading

OpenGL across platforms, "stdcall" should be used instead of "default", because on Windows, OpenGL is a System DLL. This is very exceptional, as in most other cases, "default" (or "cdecl", the alias) need to be used for normal C shared libraries on Windows.

At this stage of development, support for C++ calls should be considered experimental. Support for Fortran is planed but not yet implemented in dyncall.

Portability

The implementation is based on the *dyncall* library (part of the DynCall project).

The following processor architectures are supported: X86, X64, ARM (including Thumb), PowerPC 32-bit, MIPS 32- and 64-Bit, SPARC 32- and 64-bit; The library has been built and tested to work on various OSs: Linux, Mac OS X, Windows 32/64-bit, BSDs, Haiku, Nexenta/Open Solaris, Minix and Plan9, as well as embedded platforms such as Linux/ARM (OpenMoko, Beagleboard, Gumstix), Nintendo DS (ARM), Sony Playstation Portable (MIPS 32-bit/eabi) and iOS (ARM - armv6 mode ok, armv7 unstable). In the context of R, dyncall has currently no support for PowerPC 64-Bit.

Note

The target address, calling convention and call signature **MUST** match foreign function type, otherwise the invocation could lead to a **fatal R process crash**.

References

Adler, D., Philipp, T. (2008) *DynCall Project*. <http://dyncall.org>

See Also

`.dynam` and `getNativeSymbolInfo` for resolving symbols, `dynbind` for binding several foreign functions via thin call wrappers, `.C` for the traditional FFI to C.

Examples

```
mathlib <- dynfind(c("msvcrt", "m", "m.so.6"))
x <- .dynam(mathlib, "sqrt")
.dyncall(x, "d", 144L)
```

dynfind

Portable searching and loading of shared libraries

Description

Function to load shared libraries using a platform-portable interface.

Usage

```
dynfind(libnames, auto.unload=TRUE)
```

Arguments

`libnames` vector of character strings specifying several short library names.

`auto.unload` logical: if TRUE then a finalizer is registered that closes the library on garbage collection. See [.dynload](#) for details.

Details

`dynfind` offers a platform-portable naming interface for loading a specific shared library.

The naming scheme and standard locations of shared libraries are OS-specific. When loading a shared library dynamically at run-time across platforms via standard interfaces such as [.dynload](#) or [dyn.load](#), a platform-test is usually needed to specify the OS-dependant library file path.

This *library name problem* is encountered via breaking up the library file path into several abstract components:

‘location’ ‘prefix’ ‘libname’ ‘suffix’

By permutation of values in each component and concatenation, a list of possible file paths can be derived. `dynfind` goes through this list to try opening a library. On the first success, the search is stopped and the function returns.

Given that the three components ‘location’, ‘prefix’ and ‘suffix’ are set up properly on a per OS basis, the unique identification of a library is given by ‘libname’ - the short library name.

For some libraries, multiple ‘short library name’ are needed to make this mechanism work across all major platforms. For example, to load the Standard C Library across major R platforms:

```
lib <- dynfind(c("msvcrt", "c", "c.so.6"))
```

On Windows `MSVCRT.dll` would be loaded; `libc.dylib` on Mac OS X; `libc.so.6` on Linux and `libc.so` on BSD.

Here is a sample list of values for the three other components:

- ‘location’: “/usr/local/lib/”, “/Windows/System32/”.
- ‘prefix’: “lib” (common), “” (empty - common on Windows).
- ‘suffix’: “.dll” (Windows), “.so” (ELF), “.dylib” (Mac OS X) and “” (empty - useful for all platforms).

The vector of ‘locations’ is initialized by environment variables such as ‘PATH’ on Windows and `LD_LIBRARY_PATH` on Unix-flavour systems in addition to some hardcoded locations: ‘/opt/local/lib’, ‘/usr/local/lib’, ‘/usr/lib’ and ‘/lib’. (The set of hardcoded locations might expand and change within the next minor releases).

The file extension depends on the OS: ‘.dll’ (Windows), ‘.dylib’ (Mac OS X), ‘.so’ (all others).

On Mac OS X, the search for a library includes the ‘Frameworks’ folders as well. This happens before the normal library search procedure and uses a slightly different naming pattern in a separate search phase:

```
‘frameworksLocation’ “Frameworks/” ‘libname’ “.framework/” ‘libname’
```

The ‘frameworksLocation’ is a vector of locations such as /System/Library/ and /Library/.
dynfind loads a library via `.dynload` passing over the parameter `auto.unload`.

Value

dynfind returns an external pointer (library handle), if search was successful. Otherwise, if no library is located, a NULL is returned.

See Also

See `.dynload` for details on the loader interface to the OS-specific dynamic linker.

| | |
|---------|---|
| dynload | <i>Loading of shared libraries and resolving of symbols (Alternative Framework)</i> |
|---------|---|

Description

Alternative framework for loading of shared libraries and resolving of symbols. The framework offers *automatic unload management* of shared libraries and provides a direct interface to the dynamic linker of the OS.

Usage

```
.dynload(libname, auto.unload=TRUE)
.dynsym(libhandle, symname, protect.lib=TRUE)
.dynunload(libhandle)
```

Arguments

| | |
|-------------|---|
| libname | character string giving the pathname to a shared library in OS-specific notation. |
| libhandle | external pointer representing a handle to an opened library. |
| symname | character string specifying a symbolic name to be resolved. |
| auto.unload | logical, if TRUE a finalizer will be registered that will automatically unload the library. |
| protect.lib | logical, if TRUE resolved external pointers protect library handles from finalization. |

Details

`.dynload` loads a shared library into the current R process using the OS-specific dynamic linker interface. The `libname` is passed *as-is* directly to the dynamic linker and thus is given in OS-specific notation - see below for details. On success, a handle to the library represented as an external pointer R objects is returned, otherwise NULL. If `auto.unload` is TRUE, a finalizer function is registered that will unload the library on garbage collection via `.dynunload`.

`.dynam` looks up symbol names in loaded libraries and resolves them to memory addresses returned as external pointer R objects. Otherwise NULL is returned. If `protect.lib` is TRUE, the library handle is *protected* by resolved address external pointers from unloading.

`.dynunload` explicitly unreferences the loaded library specified by `libhandle`.

Setting both `auto.unload` and `protect.lib` to TRUE, libraries remain loaded as long as resolved symbols are in use, and they get automatic unloaded when no resolved symbols remain.

Dynamic linkers usually hold an internal link count, such that a library can be opened multiple times via `.dynload` - with a balanced number of calls to `.dynunload` that decreases the link count to unload the library again.

Similar functionality is available via `dyn.load` and `getNativeSymbolInfo`, except that path names are filtered and no automatic unloading of libraries is supported.

Value

`.dynload` returns an external pointer `libhandle` on success. Otherwise NULL is returned, if the library is not found or the linkage failed.

`.dynam` returns an external pointer `address` on success. Otherwise NULL is returned, if the address was invalid or the symbol has not been found.

`.dynunload` always returns NULL.

Shared library

Shared libraries are single files that contain compiled code, data and meta-information. The code and data can be loaded and mapped to a process at run-time once. Operating system platforms have slightly different schemes for naming, searching and linking options.

| Platform | Binary format | File Extension |
|--|---------------|----------------|
| Linux, BSD derivatives and Sun Solaris | ELF format | so |
| Darwin / Apple Mac OS X | Mach-O format | dylib |
| Microsoft Windows | PE format | dll |

Library search on Posix platforms (Linux,BSD,Sun Solaris)

The following text is taken from the Linux `dlopen` manual page:

These search rules will only be applied to path names that do not contain an embedded '/'.

- If the `LD_LIBRARY_PATH` environment variable is defined to contain a colon-separated list of directories, then these are searched.
- The cache file `/etc/ld.so.cache` is checked to see whether it contains an entry for filename.

- The directories `/lib` and `/usr/lib` are searched (in that order).

If the library has dependencies on other shared libraries, then these are also automatically loaded by the dynamic linker using the same rules.

Library search on Darwin (Mac OS X) platforms

The following text is taken from the Mac OS X `dlopen` manual page:

`dlopen()` searches for a compatible Mach-O file in the directories specified by a set of environment variables and the process's current working directory. When set, the environment variables must contain a colon-separated list of directory paths, which can be absolute or relative to the current working directory. The environment variables are `$LD_LIBRARY_PATH`, `$DYLD_LIBRARY_PATH`, and `$DYLD_FALLBACK_LIBRARY_PATH`. The first two variables have no default value. The default value of `$DYLD_FALLBACK_LIBRARY_PATH` is `$HOME/lib;/usr/local/lib;/usr/lib`. `dlopen()` searches the directories specified in the environment variables in the order they are listed.

When path doesn't contain a slash character (i.e. it is just a leaf name), `dlopen()` searches the following until it finds a compatible Mach-O file: `$LD_LIBRARY_PATH`, `$DYLD_LIBRARY_PATH`, current working directory, `$DYLD_FALLBACK_LIBRARY_PATH`.

When path contains a slash (i.e. a full path or a partial path) `dlopen()` searches the following the following until it finds a compatible Mach-O file: `$DYLD_LIBRARY_PATH` (with leaf name from path), current working directory (for partial paths), `$DYLD_FALLBACK_LIBRARY_PATH` (with leaf name from path).

Library search on Microsoft Windows platforms

The following text is taken from the Window SDK Documentation:

If no file name extension is specified [...], the default library extension `.dll` is appended. However, the file name string can include a trailing point character (`.`) to indicate that the [shared library] module name has no extension. When no path is specified, the function searches for loaded modules whose base name matches the base name of the module to be loaded. If the name matches, the load succeeds. Otherwise, the function searches for the file in the following sequence:

- The directory from which the application loaded.
- The current directory.
- The system directory. Use the `GetSystemDirectory` Win32 API function to get the path of this directory.
- The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched. Windows Me/98/95: This directory does not exist.
- The Windows directory. Use the `GetWindowsDirectory` Win32 API function to get the path of this directory.
- The directories that are listed in the `PATH` environment variable.

Windows Server 2003, Windows XP SP1: The default value of

`HKLM\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode`

is 1 (current directory is searched after the system and Windows directories).

Windows XP: If

HKLM\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode

is 1, the current directory is searched after the system and Windows directories, but before the directories in the PATH environment variable. The default value is 0 (current directory is searched before the system and Windows directories).

The first directory searched is the one directory containing the image file used to create the calling process. Doing this allows private dynamic-link library (DLL) files associated with a process to be found without adding the process's installed directory to the PATH environment variable.

The search path can be altered using the SetDllDirectory function. This solution is recommended instead of using SetCurrentDirectory or hard-coding the full path to the DLL.

If a path is specified and there is a redirection file for the application, the function searches for the module in the application's directory. If the module exists in the application's directory, the LoadLibrary function ignores the specified path and loads the module from the application's directory. If the module does not exist in the application's directory, LoadLibrary loads the module from the specified directory. For more information, see Dynamic Link Library Redirection from the Windows SDK Documentation.

Portability

The implementation is based on the *dynload* library (part of the DynCall project) which has been ported to all major R platforms (ELF (Linux,BSD,Solaris), Mach-O (Mac OS X) and Portable Executable (Win32/64)).

See Also

This facility is used by [dynfind](#) and [dynbind](#). Similar functionality is available from [dyn.load](#) and [getNativeSymbolInfo](#).

dynport

Dynamic R Bindings to standard and common C libraries

Description

Function to bind APIs of standard and common C libraries to R via dynamically created interface packages comprising R wrappers for C functions, object-like macros, enums and data types.

Usage

```
dynport(portname, portfile=NULL,
        repo=system.file("dynports",package="rdyncall") )
```

Arguments

| | |
|----------|--|
| portname | the name of a dynport, given as a literal or character string. |
| portfile | NULL or character string giving a script file to parse ; portname and repo are . |
| repo | character string giving the path to the root of the <i>dynport</i> repository. |

Details

dynport offers a convenient method for binding entire C libraries to R. This mechanism runs cross-platform and uses dynamic linkage. This means, the run-time library of a chosen binding need to be installed in the system. Depending on the OS, some libraries, supported by dynport, are already installed, but others might need manual intervention. See [rdyncall-demos](#) for OS-specific installation notes.

The binding method is data-driven using platform-portable specifications, named *DynPort* files. *DynPort* files are stored in a repository that is delivered and installed as part of the package installation. When dynport processes a *DynPort* file given by portname, an R name space object is created and populated with R wrapper and helper objects that implement the interface between R and C.

The following gives a list of currently available *DynPorts*:

| DynPort name/C Library | Description |
|-------------------------------|---|
| expat | Expat XML Parser Library |
| GL | OpenGL 1.1 API |
| GLU | OpenGL Utility Library |
| SDL | Simple DirectMedia Layer library |
| SDL_image | Loading of image files (png,jpeg..) |
| SDL_mixer | Loading/Playing of ogg/mp3/mod music files. |
| SDL_ttf | Loading/Rendering of True Type Fonts. |
| glew | OpenGL Extension Wrangler (includes OpenGL 3.0) |
| gl3 | strict OpenGL 3 (untested) |
| R | R shared library |
| ode | Open Dynamics (Physics-) Engine (untested) |
| cuda | NVIDIA Cuda (untested) |
| opencl | OpenCL (untested) |
| stdio | C Standard Library I/O Functions |

As of the current implement, *DynPort* files are actually R scripts that perform basically three things:

- Functions (and pointer-to-function variables) are mapped via [dynbind](#) and a description of the C library using a *library signatures*.
- Symbolic names are assigned to its values for object-like macro defines and C enum types.
- Run-time type-information objects for aggregate C data types (struct and union) are registered via [parseStructInfos](#) and [parseUnionInfos](#).

The file path to the *DynPort* file is derived from portname per default. This would refer to "<repo>/<portname>.R" where repo usually refers to the initial *DynPort* repository located at the sub-folder "dynports/" of the package. If portfile is given, then this value is taken as file path (usually for testing purpose).

A tool suite, comprising boost wave, GCC-XML, and XSLT, was used to generate the available *DynPort* files automatically by extracting type information from C library header files.

In a future release, the DynPort format will be changed to a language-neutral document. For the interested reader: A first prototyp is currently available in an FFI extension to the Lua programming language (see `luadyncall` subversion sub-tree). A third revision (including function types in call signatures, bitfields, arrays, etc..) is currently in development.

References

- Adler, D., Philipp, T. (2008) *DynCall Project*.
<http://dyncall.org>
- Clark, J. (1998). expat - XML Parser Toolkit.
<http://expat.sourceforge.net>
- Ikits, M. and Magallon, M. (2002). The OpenGL Extension Wrangler Library.
<http://glew.sourceforge.net>
- Latinga, S. (1998). The Simple DirectMedia Layer Library.
<http://www.libsdl.org>
- Segal, M. and Akeley, K. (1992). The OpenGL Graphics System. A Specification, Version 1.0.
<http://www.opengl.org>
- Smith, R. (2001). Open Dynamics Engine.
<http://www.ode.org>

Examples

```
# Using SDL and OpenGL in R
dynport(SDL)
dynport(GL)
# Initialize Video Sub-system
SDL_Init(SDL_INIT_VIDEO)
# Initialize Screen with OpenGL Context and Double Buffering
SDL_SetVideoMode(320,256,32,SDL_OPENGL+SDL_DOUBLEBUF)
# Clear Color and Clear Screen
glClearColor(0,0,1,0) # blue
glClear(GL_COLOR_BUFFER_BIT)
# Flip Double-Buffer
SDL_GL_SwapBuffers()
```

packing

Handling of foreign C fundamental data types

Description

Functions to `unpack/pack` (read/write) foreign C data types from/to R atomic vectors and C data objects such as arrays and pointers to structures.

Usage

```
.pack(x, offset, sigchar, value)
.unpack(x, offset, sigchar)
```

Arguments

| | |
|---------|---|
| x | atomic vector (logical, raw, integer or double) or external pointer. |
| offset | integer specifying <i>byte offset</i> starting at 0. |
| sigchar | character string specifying the C data type by a type signature . |
| value | R object value to be coerced and packed to a foreign C data type. |

Details

The function `.pack` converts an R value into a C data type specified by the [signature](#) sigchar and it writes the raw C foreign data value at byte position offset into the object x. The function `.unpack` extracts a C data type according to the [signature](#) sigchar at byte position offset from the object x and converts the C value to an R value and returns it.

Byte offset calculations start at 0 relative to the first byte in an atomic vectors data area.

If x is an atomic vector, a bound check is carried out before read/write access. Otherwise, if x is an external pointer, there is only a C NULL pointer check.

Value

`.unpack` returns a read C data type coerced to an R value.

See Also

[.dyncall](#) for details on type signatures.

Examples

```
# transfer double to array of floats and back, compare precision:
n <- 6
input <- rnorm(n)
buf <- raw(n*4)
for (i in 1:n) {
  .pack(buf, 4*(i-1), "f", input[i])
}
output <- numeric(n)
for (i in 1:n) {
  output[i] <- .unpack(buf, 4*(i-1), "f")
}
# difference between double and float
difference <- output-input
print( cbind(input,output,difference) )
```

rdyncall

Improved Foreign Function Interface (FFI) and Dynamic Bindings to C Libraries (e.g. OpenGL)

Description

The package provides a cross-platform framework for dynamic binding of C libraries using a flexible Foreign Function Interface (FFI). The FFI supports almost all fundamental C types, multiple calling conventions, symbolic access to foreign C struct/union data types and wrapping of R functions as C callback function pointers. Dynamic bindings to shared C libraries are data-driven by cross-platform binding specification using a compact plain text format ; an initial repository of bindings to a couple of common C libraries (OpenGL, SDL, Expat, glew, CUDA, OpenCL, ODE, R) comes with the package. The package includes a variety of technology demos and OS-specific notes for installation of shared libraries.

Details

rdyncall offers a stack of interoperability technologies for working with foreign compiled languages using cross-platform portable abstraction methods.

For R application development, the package facilitates direct access from R to the C Application Programming Interface (API) of common libraries. This enables a new style of development: R applications can use low-level services of portable C libraries. System-level code can be implemented in R without leaving the language. C APIs can be explored from within the R interpreter. Moving the R code from one platform to the other does not involve recompilation. Ofcourse, the run-time libraries need to be installed using a standard procedure of the target Operating-System Distribution. See [rdyncall-demos](#) for details on this.

For R core development and research, the package provides an improved Foreign Function Interface (FFI) that can be used to call arbitrary foreign precompiled C code without the need for additional compilation of wrapper code. The back-end library is extendable with new calling conventions (such as Fortran,Pascal,COM,etc.. - which has not been the focus as of this release, but might be supported officially in the near future). Basic type-safety checks for argument passing and framework support for working with foreign C data types such as pointers, arrays, structs and wrapping of R functions into first-level C callback function pointers round up this framework.

Overview

- Flexible FFI with support for almost all C types, type-safety checks and multiple calling conventions. See [.dyncall](#).
- Loading of shared libraries with *automatic unload management* and using direct access to OS linker. See [.dynload](#).
- Cross-platform naming and loading of shared libraries. See [dynfind](#).
- Binding C library functions via thin call wrappers. See [dynbind](#).
- Handling of foreign C pointer, array and struct/union data types. See [packing](#) and [struct](#).
- Dynamic wrapping of R functions as C function pointers to be used in C callbacks. See [new.callback](#).

- Dynamic bindings to standard and common C libraries and APIs (functions, variables, macro constants, enums, struct and union types). See [dynport](#).

Getting Started

Several demos ranging from simple FFI calls to the C standard math library up to more complex 3D OpenGL/SDL Applications are available. See `demos(package="rdyncall")` for an overview. Some demos require shared C libraries to be installed in the system. Please read [rdyncall-demos](#) for details.

Supported Platforms

The low-level implementation is mainly based on libraries from the DynCall Project (<http://dyncall.org>). The library suite is distributed as part of the package source tree.

The dyncall and dyncallback libraries implement generic low-level services with the help of a small amount of hand-written assembly code and careful modeling of the target machine's calling sequence for each platform to support.

As of version 0.6, the following processor architectures are supported:

- Intel i386 32-bit and AMD 64-bit Platforms
- ARM 32-bit (with support for Thumb)
- PowerPC 32-bit (support for callbacks not implemented for Linux/BSD)
- MIPS 32- and 64-bit (support for callbacks not yet implemented)

The DynCall libraries are tested on Linux, Mac OS X, Windows, BSD derivatives and more exotic platforms such as game consoles and Plan9. Please see the details on portability for [dyncall](#), [dyncallback](#) and [dynload](#) and the official DynCall manual for full details of the back-end. The R Package has been tested on several major R platforms. The following gives a list of comments on platforms about the status of this package.

Linux Debian 4/ppc32, R-2.4.0 : ok, but no callbacks.

Linux Debian 5/arm , R-2.7.0 : ok, SDL not tested.

Linux Debian 6/x86 , R-2.12.2: ok.

Linux Debian 6/x64 , R-2.12.2: ok.

Fedora 14/x86: ok.

Mac OS X 10.4/ppc, R-2.10.0: ok.

Mac OS X 10.6/x86, R-2.12.2: ok.

Mac OS X 10.6/x64, R-2.12.2: ok.

NetBSD 5.0/x86: ok.

NetBSD 5.1/x64: ok.

OpenBSD 4.8/x64, R-2.7.0: SDL failed.

Windows XP/x86, R-2.12.2: ok.

Windows 7/x86, R-2.12.2: ok.

Windows 7/x64, R-2.12.2: ok, use correct 64-bit SDL DLL, SDL extension not tested - see [rdyncall-demos](#))

FreeBSD 8.2/x86: build ok, no tests made for X11.

References

Adler, D., Philipp, T. (2008) *DynCall Project*. <http://dynccall.org>

Examples

```
# multimedia example
# load dynports for OpenGL, Simple DirectMedia library
# globals:
surface <- NULL
# init SDL and OpenGL
init <- function()
{
  dynport(SDL)
  dynport(GL)
  if ( SDL_Init(SDL_INIT_VIDEO) != 0 ) stop("SDL_Init failed")
  surface <- SDL_SetVideoMode(320,240,32,SDL_DOUBLEBUF+SDL_OPENGL)
  cat("surface dimension:", surface$w, "x", surface$h, sep="")
}
# draw blue screen
updateSurface <- function(t)
{
  glClearColor(0,0,t %% 1,0)
  glClear(GL_COLOR_BUFFER_BIT+GL_DEPTH_BUFFER_BIT)
  SDL_GL_SwapBuffers()
}
# wait till close
mainloop <- function()
{
  quit <- FALSE
  evt <- new.struct(SDL_Event)
  base <- SDL_GetTicks() / 1000
  t <- 0
  while(!quit) {
    updateSurface(t)
    while(SDL_PollEvent(evt)) {
      if ( evt$type == SDL_QUIT ) quit <- TRUE
    }
    now <- SDL_GetTicks() / 1000
    t <- now - base
  }
}
init()
mainloop()
```

Description

The demos of the **rdyncall** package (see `demo(package="rdyncall")`) use shared libraries such as SDL, OpenGL and Expat via **dynports** - a dynamic binding approach which requires, that prebuilt binary shared library files are already installed.

Depending on the host system, some libraries are officially a part of the OS or Distribution, some others need to be installed to get the demos running.

As of the current version of this package, the installation of additional shared C libraries need to be done manually. It follows an overview of the required libraries and installation notes for various operating-systems and distributions.

Overview of Libraries

The following Libraries are used as 'run-time' pre-compiled binaries for the particular target OS and Hardware platform. Some notes on installation of additional run-time libraries required for some rdyncall demos:

| Lib | Description | URL |
|-----------|-----------------------------------|---|
| expat | XML Parser | http://www.libexpat.org |
| GL | Open Graphics Library | http://opengl.org , http://www.mesa3d.org |
| GLU | OpenGL Utility Library | see links above |
| glew | OpenGL Extension Wrangler Library | http://glew.sourceforge.net/ |
| SDL | Multimedia Framework | http://libsdl.org/ |
| SDL_mixer | Music Format playing | http://www.libsdl.org/projects/SDL_mixer/ |
| SDL_image | Image Format loading | http://www.libsdl.org/projects/SDL_image/ |
| SDL_ttf | True Type Font rendering | http://www.libsdl.org/projects/SDL_ttf/ |
| SDL_net | Network I/O | http://www.libsdl.org/projects/SDL_net/ |

In short: Place the shared libraries (*.DLL, *.so or *.dylib) in a *standard location* or modify LD_LIBRARY_PATH(unix) or PATH(windows) so that **dynfind** can find the libraries.

On Mac OS X framework folders are supported as well. Place the *.framework folder at /Library/Frameworks.

Detailed platform-specific installation instructions follow up.

Windows Installation Notes

Download the *.zip files, unpack them and place the *.DLL files to a place within PATH.

32-Bit versions:

| Lib | Download Link |
|-----------|---|
| expat | http://expat.sourceforge.net (TODO:test installer) |
| GL | pre-installed |
| GLU | pre-installed |
| glew | http://sourceforge.net/projects/glew/files/glew/1.7.0/glew-1.7.0-win32.zip/download |
| SDL | http://www.libsdl.org/release/SDL-1.2.14-win32.zip |
| SDL_image | http://www.libsdl.org/projects/SDL_image/release/SDL_image-1.2.10-win32.zip |
| SDL_mixer | http://www.libsdl.org/projects/SDL_mixer/release/SDL_mixer-1.2.11-win32.zip |
| SDL_ttf | http://www.libsdl.org/projects/SDL_ttf/release/SDL_ttf-2.0.10-win32.zip |

SDL_net http://www.libsdl.org/projects/SDL_net/release/SDL_net-1.2.7-win32.zip

64-Bit version:

| Lib | Download Link |
|------------|---|
| expat | no prebuilt found (TODO: build) |
| GL | pre-installed |
| GLU | pre-installed |
| glew | http://sourceforge.net/projects/glew/files/glew/1.7.0/glew-1.7.0-win64.zip/download |
| SDL | http://mamedev.org/tools/20100102/sdl-1.2.14-r5428-w64.zip |
| SDL_image | pre-built n/a |
| SDL_mixer | pre-built n/a |
| SDL_ttf | pre-built n/a |
| SDL_net | pre-built n/a |

The prebuilt version of SDL from <http://www.drangon.org/mingw> did not work (exiting with OpenGL errors). If you know of other resources for prebuilt 64-bit packages for SDL and expat, please report.

Mac OS X Installation Notes

Download the *.dmg files, mount them (by double-click) and copy *.framework folders to /Library/Frameworks.

| Lib | Download link |
|------------|---|
| expat | pre-installed |
| GL | pre-installed |
| GLU | pre-installed |
| glew | port install glew |
| SDL | http://www.libsdl.org/release/SDL-1.2.14.dmg |
| SDL_image | http://www.libsdl.org/projects/SDL_image/release/SDL_image-1.2.10.dmg |
| SDL_mixer | http://www.libsdl.org/projects/SDL_mixer/release/SDL_mixer-1.2.11.dmg |
| SDL_ttf | http://www.libsdl.org/projects/SDL_ttf/release/SDL_ttf-2.0.10.dmg |
| SDL_net | http://www.libsdl.org/projects/SDL_net/release/SDL_net-1.2.7.dmg |

Linux/Debian Installation Notes

Debian Package installation via aptitude

```
aptitude install <pkg-names>..
```

| Lib | Debian Package name(s) |
|------------|--|
| expat | libexpat1 (version 1.5.2 - already installed?) |
| GL | libgl1-mesa-glx and libgl1-mesa-dri |
| GLU | libglu1-mesa |

```

glew      libglew1.5
SDL       libsd11.2debian and libsd11.2debian-<SOUNDSYS>
SDL_image libsd1-image1.2
SDL_mixer libsd1-mixer1.2
SDL_ttf   libsd1-ttf2.0
SDL_net   libsd1-net1.2

```

Depending on your sound system, <SOUNDSYS> should be replaced with one of the following: alsa, all, esd, arts, oss, nas or pulseaudio. Tested with Debian 5 and 6 (lenny and squeeze).

Linux/Fedora Installation Notes

```
pkcon install <pkgname>..
```

| Lib | RPM Package name |
|------------|-------------------------|
| expat | expat |
| GL | mesa-libGL |
| GLU | mesa-libGLU |
| glew | glew |
| SDL | SDL |
| SDL_image | SDL_image |
| SDL_mixer | SDL_mixer |
| SDL_ttf | SDL_ttf |
| SDL_net | SDL_net |

Tested with Fedora 13 and 14 on x86 and x86_64.

NetBSD Installation Notes

Installation via pkgsrc:

```
pkg_add <pkgname>..
```

| Lib | pkgsrc name |
|------------|--------------------|
| expat | expat |
| GL | Mesa |
| GLU | glu |
| glew | glew |
| SDL | SDL |
| SDL_image | SDL_image |
| SDL_mixer | SDL_mixer |
| SDL_ttf | SDL_ttf |
| SDL_net | SDL_net |

OpenBSD Installation Notes

Using packages:

```
pkg_add <pkgname>..
```

| Lib | port name |
|------------|------------------|
| expat | expat |
| SDL | SDL |
| SDL_image | sdl-image |
| SDL_mixer | sdl-mixer |
| SDL_ttf | not available |
| SDL_net | sdl-net |

The SDL dynport failed on OpenBSD 4.8 - so no multimedia demos here - using the R 2.7 from the ports tree. This must have been something to do with pthread discrepancies between SDL and R.

FreeBSD Installation Notes

Using packages:

```
pkg_add -r <pkgname>..
```

| Lib | pkgname |
|------------|----------------|
| expat | expat2 |
| GL | xorg |
| glew | glew |
| SDL | sdl |
| SDL_image | sdl_image |
| SDL_mixer | sdl_mixer |
| SDL_ttf | sdl_ttf |
| SDL_net | sdl_net |

Solaris Installation Notes

OpenCSW offers prebuilt binaries for Solaris. The installation of OpenCSW packages is done via pkgutil.

```
pkgutil -i <pkgname>..
```

See <http://www.opencsw.org> for details on the OpenCSW project.

| Lib | pkgname |
|------------|----------------|
| expat | expat |
| GL | mesalibs |

| | |
|-----------|----------|
| GLU | mesalibs |
| glew | glew |
| SDL | libsdl |
| SDL_image | sdlimage |
| SDL_mixer | sdlmixer |
| SDL_net | sdlnet |
| SDL_ttf | sdlttf |

 struct

Allocation and handling of foreign C aggregate data types

Description

Functions for allocation, access and registration of foreign C struct and union data type.

Usage

```

new.struct(type)
as.struct(x, type)
parseStructInfos(sigs, envir=parent.frame())
parseUnionInfos(sigs, envir=parent.frame())
## S3 method for class 'struct'
x$index
## S3 replacement method for class 'struct'
x$index <- value
## S3 method for class 'struct'
print(x, indent=0, ...)

```

Arguments

| | |
|--------|--|
| x | external pointer or atomic raw vector of S3 class 'struct'. |
| type | S3 TypeInfo Object or character string that names the structure type. |
| sigs | character string that specifies several C struct/union type signatures . |
| envir | the environment to install S3 type information object(s). |
| index | character string specifying the field name. |
| indent | indentation level for pretty printing structures. |
| value | value to be converted according to struct/union field type given by field index. |
| ... | additional arguments to be passed to print method. |

Details

References to foreign C data objects are represented by objects of class 'struct'.

Two reference types are supported:

- *External pointers* returned by `.dyncall` using a call signature with a *typed pointer* return type signature and pointers extracted as a result of `.unpack` and S3 struct `$`-operators.
- *Internal objects*, memory-managed by R, are allocated by `new.struct`: An atomic raw storage object is returned, initialized with length equal to the byte size of the foreign C data type.

In order to access and manipulate the data fields of foreign C aggregate data objects, the “\$” and “\$<” S3 operator methods can be used.

S3 objects of class `struct` have an attribute `struct` set to the name of a `TypeInfo` object, which provides the run-time type information of a particular foreign C type.

The run-time type information for foreign C `struct` and union types need to be registered once via `parseStructInfos` and `parseUnionInfos` functions. The C data types are specified by `sigs`, a signature character string. The formats for both types are described next:

Structure type signatures describe the layout of aggregate `struct` C data types. Type Signatures are used within the 'field-types'. 'field-names' consists of space separated identifier names and should match the number of fields.

```
'struct-name' '{' 'field-types' '}' 'field-names' '');
```

Here is an example of a C struct type:

```
struct Rect {
  signed short x, y;
  unsigned short w, h;
};
```

The corresponding structure type signature is:

```
"Rect{ssSS}x y w h;"
```

Union type signatures describe the components of the union C data type. Type signatures are used within the 'field-types'. 'field-names' consists of space separated identifier names and should match the number of fields.

```
'union-name' '|' 'field-types' '}' 'field-names' '');
```

Here is an example of a C union type,

```
union Value {
  int anInt;
```

```
float aFloat;
struct LongValue aStruct
};
```

The corresponding union type signature is:

```
"Value|if<LongValue>}anInt aFloat aStruct;"
```

as .struct can be used to *cast* a foreign C data reference to a different type. When using an external pointer reference, this can lead quickly to a **fatal R process crash** - like in C.

See Also

[.dyncall](#) for type signatures and [TypeInfo](#) for details on run-time type information S3 objects.

Examples

```
# Specify the following foreign type:
# struct Rect {
#   short x, y;
#   unsigned short w, h;
# }
parseStructInfos("Rect{ssSS}x y w h;")
r <- new.struct(Rect)
print(r)
r$x <- 40
r$y <- 60
r$w <- 10
r$h <- 15
print(r)
str(r)
```

typeinfo

S3 class for run-time type information of foreign C data types

Description

S3 class for run-time type information of foreign C data types.

Usage

```
TypeInfo(name, type = c("base", "pointer", "struct", "union"),
  size = NA, align = NA, basetype = NA, fields = NA,
  signature = NA)
getTypeInfo(name, envir = parent.frame())
```

Arguments

| | |
|-----------|--|
| name | character string specifying the type name. |
| type | character string specifying the type. |
| size | integer, size of type in bytes. |
| align | integer, alignment of type in bytes. |
| basetype | character string, base type of 'pointer' types. |
| signature | character string specifying the struct/union type signature . |
| envir | the environment to look for type object. |
| fields | data frame with type and offset information that specifies aggregate struct and union types. |

Details

Type information objects are created at run-time to describe the concrete layout of foreign C data types on the host machine. While [type signatures](#) give an abstract information on e.g. the field types and names of aggregate structure types, these objects store concrete memory size, alignment and layout information about C data types.

Value

List object tagged as S3 class 'TypeInfo' with the following named entries

| | | | | | |
|--------|--|------|-----------|--------|-------------------------------------|
| type | Type name. | | | | |
| size | Size in bytes. | | | | |
| align | Alignment in bytes. | | | | |
| fields | Data frame for field information with the following columns: | | | | |
| | <table> <tr> <td>type</td> <td>type name</td> </tr> <tr> <td>offset</td> <td>byte offset (starts counted from 0)</td> </tr> </table> | type | type name | offset | byte offset (starts counted from 0) |
| type | type name | | | | |
| offset | byte offset (starts counted from 0) | | | | |

See Also

[parseStructInfos](#) for details on the framework for handling foreign C data types.

 utils

Utility functions for working with foreign C data types

Description

Functions for low-level operations on C pointers as well as helper functions and objects to handle C float arrays and strings.

Usage

```

is.nullptr(x)
as.extptr(x)
is.externalptr(x)
floatraw(n)
as.floatraw(x)
floatraw2numeric(x)
ptr2str(x)
strarrayptr(x)
strptr(x)
offsetPtr(x, offset)

```

Arguments

| | |
|--------|---------------------------------|
| x | an R object. |
| n | number of elements to allocate. |
| offset | a offset given in bytes. |

Details

`is.nullptr` tests if the external pointer given by `x` represents a C NULL pointer.

`as.extptr` returns an external pointer to the data area of atomic vector given by `x`. The external pointer holds an additional reference to the `x` R object to prevent it from garbage collection.

`is.externalptr` tests if the object given by `x` is an external pointer.

`floatraw` creates an array with a capacity to store `n` single-precision C float values. The array is implemented via a [raw](#) vector.

`as.floatraw` coerces a numeric vector into a single-precision C float vector. Values given by `x` are converted to C float values and stored in the R raw vector via `.pack`. This function is useful when calling foreign functions that expect a C float pointer via `.dyncall`.

`floatraw2numeric` coerces a C float (raw) vector to a numeric vector.

`ptr2str`, `strarrayptr`, `strptr` are currently experimental.

`offsetPtr` creates a new external pointer pointing to `x` plus the byte `offset`. If `x` is given as an external pointer, the address is increased by the `offset`, or, if `x` is given as a atomic vector, the address of the data (pointing to `offset` zero) is taken as basis and increased by the `offset`. The returned external pointer is protected (as offered by the C function `R_MakeExternalPtr`) by the external pointer `x`.

Value

A logical value is returned by `is.nullptr` and `is.externalptr`. `as.extptr` returns an external pointer value. `floatraw` and `as.floatraw` return an atomic vector of type `raw` tagged with class `'floatraw'`. `floatraw2numeric` returns a numeric atomic vector.

Index

*Topic **interface**

- callback, [2](#)
- dynbind, [4](#)
- dyncall, [6](#)
- dynfind, [9](#)
- dynload, [11](#)
- dynport, [14](#)
- rdyncall, [18](#)
- struct, [25](#)
- utils, [28](#)

*Topic **programming**

- callback, [2](#)
- dynbind, [4](#)
- dyncall, [6](#)
- dynfind, [9](#)
- dynload, [11](#)
- dynport, [14](#)
- rdyncall, [18](#)
- struct, [25](#)
- utils, [28](#)

- .C, [9](#)
- .dyncall, [2](#), [5](#), [8](#), [17](#), [18](#), [26](#), [27](#), [29](#)
- .dyncall (dyncall), [6](#)
- .dynload, [5](#), [10](#), [11](#), [18](#)
- .dynload (dynload), [11](#)
- .dynsym, [6](#), [9](#)
- .dynsym (dynload), [11](#)
- .dynunload (dynload), [11](#)
- .pack (packing), [16](#)
- .unpack, [5](#), [26](#)
- .unpack (packing), [16](#)
- \$.struct (struct), [25](#)
- \$<-.struct (struct), [25](#)
- as.extptr (utils), [28](#)
- as.floatraw (utils), [28](#)
- as.struct (struct), [25](#)
- call signature, [2](#)
- call signature (dyncall), [6](#)

- call signatures, [4](#)
- callback, [2](#)
- dyn.load, [10](#), [12](#), [14](#)
- dynbind, [4](#), [7](#), [9](#), [14](#), [15](#), [18](#)
- dyncall, [6](#), [19](#)
- dyncallback, [19](#)
- dyncallback (callback), [2](#)
- dynfind, [4](#), [5](#), [9](#), [14](#), [18](#), [21](#)
- dynload, [11](#), [19](#)
- dynport, [7](#), [14](#), [19](#), [21](#)
- floatraw (utils), [28](#)
- floatraw2numeric (utils), [28](#)
- getNativeSymbolInfo, [6](#), [9](#), [14](#)
- getTypeInfo (typeinfo), [27](#)
- is.externalptr (utils), [28](#)
- is.nullptr (utils), [28](#)
- loadDynportNamespace (dynport), [14](#)
- new.callback, [8](#), [18](#)
- new.callback (callback), [2](#)
- new.struct, [7](#)
- new.struct (struct), [25](#)
- offsetPtr (utils), [28](#)
- packing, [7](#), [16](#), [18](#)
- parseStructInfos, [15](#), [28](#)
- parseStructInfos (struct), [25](#)
- parseUnionInfos, [15](#)
- parseUnionInfos (struct), [25](#)
- print, [25](#)
- print.struct (struct), [25](#)
- ptr2str (utils), [28](#)
- raw, [29](#)
- rdyncall, [18](#)

rdyncall-demos, [15](#), [18](#), [19](#), [20](#)
rdyncall-package (rdyncall), [18](#)
reg.finalizer, [3](#)

signature, [3](#), [17](#), [25](#), [28](#)
signature (dyncall), [6](#)
strarrayptr (utils), [28](#)
strptr (utils), [28](#)
struct, [18](#), [25](#)

type information (typeinfo), [27](#)
type signature, [17](#), [28](#)
type signature (dyncall), [6](#)
TypeInfo, [25](#)–[27](#)
TypeInfo (typeinfo), [27](#)
typeinfo, [27](#)

utils, [28](#)