

Foreign Function Interface

This document contains the following sections:

[1.0 Foreign functions introduction](#)

[1.1 SWIG: Support for automatic generation of foreign function definitions 1](#)

[1.2 CBIND: Support for automatic generation of foreign function definitions 2](#)

[1.3 Some notation](#)

[1.4 Different versions of Allegro CL load foreign code differently](#)

[1.5 Package information](#)

[1.6 Load foreign code with cl:load](#)

[1.6.1 Foreign File Processing](#)

[1.6.2 Externals must be resolved when the .so/.sl/.dylib/.dll file is created](#)

[1.6.3 One library file cannot depend on an already loaded library file](#)

[1.6.4 Entry points are updated automatically](#)

[1.6.5 Duplicate entry points are never a problem](#)

[1.6.6 If you modify a so/sl/dylib/dll file that has been loaded, you must load the modified file!](#)

[1.7 Foreign code cannot easily be included at installation time](#)

[1.8 Foreign functions and multiprocessing](#)

[1.9 Creating Shared Objects that refer to Allegro CL Functionality](#)

[1.9.1 Linking to Allegro CL shared library on Mac OS X](#)

[1.9.2 Delaying linking the Allegro CL shared-library until runtime](#)

[1.10 Releasing the heap when calling foreign functions](#)

[2.0 The 4.3.x UNIX model for foreign functions](#)

[3.0 The foreign function interface specification](#)

[3.1 Foreign function interface function summary](#)

[3.1.1 A note on foreign addresses](#)

[3.2 def-foreign-call](#)

[3.2.1 def-foreign-call syntax](#)

[3.2.2 def-foreign-call: the :returning keyword argument](#)

[3.2.3 def-foreign-call examples](#)

[3.3 def-foreign-variable](#)

[3.3.1 def-foreign-variable syntax](#)

[3.3.2 def-foreign-variable examples](#)

[4.0 Conventions for passing arguments](#)

[4.1 Modifying arguments called by address: use arrays](#)

[4.2 Lisp may not see a modification of an argument passed by address](#)

[4.3 Lisp unexpectedly sees a change to an argument passed by address](#)

[4.4 Passing fixnums, bignums, and integers](#)

[4.5 Another example using arrays to pass values](#)

[5.0 Passing strings between Lisp and C](#)

[5.1 Passing strings from Lisp to C](#)

[5.2 Special Case: Passing an array of strings from Lisp to C](#)

[6.0 Handling signals in foreign code](#)

[7.0 Input/output in foreign code](#)

[8.0 Using Lisp functions and values from C](#)

[8.1 Accessing Lisp values from C: `lisp_value\(\)`](#)

[8.2 Calling Lisp functions from C: `lisp_call_address\(\)` and `lisp_call\(\)`](#)

[8.3 Calling foreign callables from Lisp](#)

[Appendix A. Foreign Functions on Windows](#)

[Appendix A.1. Making a .dll](#)

[Appendix A.2. Making a Fortran .dll](#)

[Appendix A.3. The Lisp side of foreign functions](#)

[Appendix A.4. A complete example](#)

[Appendix B. Building shared libraries on Solaris](#)

[Appendix C. Building shared libraries on HP-UX 11.0](#)

[Appendix D. Building shared libraries on Compaq Tru64 4.0 or later](#)

[Appendix E. Building shared libraries on AIX 4.2 or later](#)

[Appendix F. Building shared libraries on Linux](#)

[Appendix G. Building shared libraries on FreeBSD](#)

[Appendix H. Building shared libraries on Mac OS X](#)

The description of Foreign Types is not in this document. It can be found in *ftype.htm*.

1.0 Foreign functions introduction

The foreign-function interface allows one to link compiled foreign code dynamically into a running Lisp. *Foreign code* is defined to be code not written in Lisp. For example, code written in C or Fortran is foreign code. The foreign-function interface allows users to load compiled code written in a foreign programming language into a running Lisp, execute it from within Lisp, call Lisp functions from within the foreign code, return to Lisp and pass data back and forth between Lisp and the foreign code.

This mechanism is very powerful, as programs need not be recoded into Lisp to use them. Another advantage arises during program development. For example, a large graphics library can be linked into Lisp and all the functions will be accessible interactively. This enables rapid prototyping of systems that use the library functions, since the powerful Lisp debugging and development environment is now available.

We use the word link because all foreign code should be in a shared object (typically *.so* or *.sl* or *.dylib* on Unix) or dynamic library (*.dll* on Windows) file which is mapped into a running Lisp process. The function that causes this linking is **load**, which has been extended to accept and do the right thing with *.so/.sl/.dylib/.dll* files.

Because **load** is used, we sometimes speak of foreign code being *loaded* into Lisp. Please understand that foreign code is not truly made part of the image. See *Using the load function* in *loading.htm* for details of the Allegro CL implementation to **load**.

1.1 SWIG: Support for automatic generation of foreign function definitions 1

SWIG is a software development tool that reads C/C++ header files and generates the wrapper code needed to make C and C++ code accessible from other languages. See <http://www.swig.org>. An interface for Allegro CL has been added to SWIG. See http://www.franz.com/support/tech_corner/swig042804.lhtml for specific information on the interface to Allegro CL and information on downloading the software. (The SWIG software is not included with the distribution because it is regularly updated. Users should always get the latest update.)

1.2 CBIND: Support for automatic generation of foreign function definitions 2

The **cbind** facility provides tools for automatically generating Lisp code for calling foreign functions using information obtained by scanning C header files. This facility is only available for Solaris and Windows. Look at *cbind-intro.htm*. That file contains pointers to more documentation.

1.3 Some notation

In this chapter, we discuss C or FORTRAN routines and the Lisp functions that call them. These often have the same names. In order to distinguish them, names ending with () are foreign routines and names without () are Lisp functions. Thus we might say:

The foreign function **bar()** is loaded into Lisp. We use **def-foreign-call** to define the Lisp function **bar** which calls **bar()**.

1.4 Different versions of Allegro CL load foreign code differently

The differences are not that significant since all platforms use some form of dynamic linking of shared objects. However, the internal mechanisms are different and the differences may sometimes be important. The type of loading is identified by a feature on the **features** list. The following table lists the relevant features:

Feature	Meaning
:dynload	Foreign code is dynamically linked.
:dlfcn	Loading is done using dlopen() . :dynload will be present if this feature is present. The type of loadable files is <i>.so</i> .
:dlhp	Loading is done using shl_load . :dynload will be present if this feature is present. HP machines only. The type of loadable files is <i>.sl</i> .
:dlwin	Loading is done using LoadLibrary . :dynload will be present if this feature is present. Windows only. The type of loadable files is <i>.dll</i> .
:dlmac	Loading on Mac OS X ports using the system dynamic loader NSLoadModule . :dynload will be present if this feature is present. Mac OS X only. The type of loadable files is <i>.dylib</i> .
:dlld	Foreign code is directly loaded into image. :dynload will not be present. No platforms currently use this method of foreign loading so none have this feature.

The following appendices describe how to create files suitable for loading on various platforms.

- [Appendix A Foreign Functions on Windows](#)
- [Appendix A.1 Making a .dll](#)
- [Appendix A.2 Making a Fortran .dll](#)
- [Appendix A.3 The Lisp side of foreign functions](#)
- [Appendix A.4 A complete example](#)
- [Appendix B Building shared libraries on Solaris](#)
- [Appendix C Building shared libraries on HP-UX 11.0](#)
- [Appendix D Building shared libraries on Compaq Tru64 4.0 or later](#)
- [Appendix E Building shared libraries on AIX 4.2 or later](#)
- [Appendix F Building shared libraries on Linux](#)
- [Appendix G Building shared libraries on FreeBSD](#)
- [Appendix H Building shared libraries on Mac OS X](#)

1.5 Package information

The foreign-function interface in Lisp is in the package `foreign-functions`, nicknamed `ff`. Users must

either use the qualifier `ff :` on these symbols or evaluate

```
(use-package :ff)
```

before using the interface.

The code for the foreign-function interface may not be contained in the basic Allegro CL image. It is loaded only when needed. Executing certain of the interface functions will cause the correct module to be loaded (**def-foreign-call** for instance), but we recommend that you ensure that the code is loaded by evaluating the following form before using the foreign-functions interface:

```
(require :foreign)
```

This will cause the *foreign.fasl* module to be loaded from the Lisp library. The form should be included in any source file using functions in the interface. (It is not an error to call `require` when a module is already loaded.)

Note that the foreign-function interface was designed for the C and Fortran compilers on the system at the time of the release of this version of Allegro CL. New versions of the C or Fortran compilers from the hardware manufacturers may, for purposes of using the foreign-function interface, be incompatible with the version current when the interface was written. In that case, it is possible that already written and compiled Lisp code may cease to work, and that, for a time, the interface may fail altogether. We will maintain the foreign-function interface, and make it compatible with each new release of the system compilers. We cannot guarantee, however, that already compiled code will continue to work in the presence of changes in the C or Fortran compilers.

1.6 Load foreign code with `cl:load`

Foreign code in a `.so/.sl/.dylib/.dll` file is loaded into Lisp with **load** (or the top-level command **:ld**). **load** is, of course, also used to load Lisp source and compiled (*fasl*) files. Since **load** may just be presented with a filename as a single argument, it should be able to determine based on that argument alone whether Lisp code or a foreign library is being loaded. Sometimes other arguments are provided. In that case, those arguments may indicate the type of file being loaded.

As described below, **load** will consider the file type to determine whether the file is a foreign file. **load** also has a non-standard keyword argument *foreign* which, when true, tells the system the file is a foreign file. (When *foreign* is `nil`, the type determines whether the file will be treated as a foreign file or not.) See *Using the load function* in *loading.htm* for details of the Allegro CL implementation to **load**.

Note that the Operating System must know where to find foreign files, either those needed to resolve externals in a file or the file itself if it is specified to load with no directory information.

The specifics on the various Operating Systems differ, but the principle is usually the same: some single environment variable or group of variables is set by the user to tell the OS where to look for library files. On most UNIX and UNIX-like (i.e. LINUX and Mac OS X) operating systems the variable is

`LD_LIBRARY_PATH`. On HP-UX, it is `SHLIB_PATH`. Note that once Lisp has started, changes to the value of environment variables will likely not be seen by the running Lisp.

On Windows, the following locations are searched:

1. The directory from which the application loaded.
2. The current directory.
3. On Windows 98/Me: The Windows system directory and on Windows NT/2000: The 32-bit Windows system directory.
4. Windows NT/2000: The 16-bit Windows system directory.
5. The Windows directory.
6. The directories that are listed in the `PATH` environment variable.

An additional keyword argument is provided to **load**, *unreferenced-library-names*. If specified, it should be a list of entry points (strings). The system will check if these entry points exist and signal an error if they do not.

Here is how **load** works.

1. It determines the argument type (whether a stream or a file).
2. If the argument is a *stream*, load from it (as a Lisp or *fasl* file) and return. (You should not open a stream to a Foreign library or shared object file and pass that stream to **load**.)
3. The argument filename is converted to a pathname, called the *argument pathname*. The argument pathname is now put through search list processing, yielding the searched pathname. If the search was successful, the argument pathname is set to the *searched pathname*. (If the search is unsuccessful, **load** will signal an error.)
4. If the *foreign* keyword argument to **load** is specified true, foreign load processing is done, as described [below](#).
5. if *foreign* is `nil` or unspecified, then the file type is considered. If the searched pathname has a type listed in `*load-foreign-types*` (the test is case insensitive on Windows), foreign load processing is done, as described [below](#).
6. If the searched pathname does not have a type listed in `*load-foreign-types*` the file is assumed to be a Lisp source or compiled file and it is loaded as a Lisp file.

1.6.1 Foreign File Processing

Foreign file processing behavior can differ slightly, depending on your version of Lisp.

Versions of Lisp with `:dlfcn` on `*features*` (e.g., SunOS 5.x and later):

1. If `:unreferenced-lib-names` was given, then make sure all entry points are defined and return.
2. If the searched pathname is `nil`, then signal an error, since `dlopen()` will not find the pathname.
3. Use `dlopen()` to map the searched pathname into the address space and return.

Versions of Lisp with `:dlwin` on `*features*` (e.g. Windows):

1. If the searched pathname is `nil`, then signal an error, since `GetModuleHandle()` will not find the pathname.
2. Use `GetModuleHandle()` to map the searched pathname into the address space and return.

Versions of Lisp with `:dlhp` on `*features*` (e.g., HP-UX 11.0):

1. If `:unreferenced-lib-names` was given, then make sure all entry points are defined and return.
2. If the searched pathname is `nil` and there is a directory or host component to this pathname, then signal an error, since `shl_load()` will not find the pathname.
3. Use `shl_load()` to map the searched pathname into the address space and return.

Versions of Lisp with `:dlmac` on `*features*` (e.g., Mac OS X machines):

1. If `:unreferenced-lib-names` was given, then make sure all entry points are defined and return.
2. If the searched pathname is `nil`, then signal an error, since `NSLoadModule` will not find the pathname.
3. Use `NSLoadModule` to map the searched pathname into the address space and return.

1.6.2 Externals must be resolved when the `.so/.sl/.dylib/.dll` file is created

Suppose one library file contains a reference to a function defined in another file. For example, suppose we are on a Solaris machine and `t_double.so`, contains a call to `foo()` which is defined in `foo.so`. When we try to load `t_double.so`, we get the following error (on Solaris, see below for behavior on other platforms):

```
USER(24): :ld t_double.so
; Foreign loading /net/rubix/usr/tech/dm/acl/t_double.so.
Error: loading library "/net/rubix/usr/tech/dm/acl/t_double.so"
      caused the following error:

ld.so.1: /net/sole/scm3/temp-images/lisp:
relocation error: symbol not found: foo: referenced in
/net/rubix/usr/tech/dm/acl/t_double.so
```

Below, we point out that loading `foo.so` (before or after the attempt to load `t_double.so`) will not resolve the unsatisfied external. The point, which we emphasize here, is any unsatisfied external must be resolved when the `so/sl/.dylib/dll` file is created (e.g. by using `-l[libname]` arguments passed to `ld` on Solaris). Even though the error is signaled when Lisp attempts to load the `.so` file, the problem cannot be resolved within Lisp. A new `.so` file, created with the correct arguments passed to `ld`, must be created and that new `.so` file must be loaded into Lisp.

This problem is not as bad as it might be. Many standard shared libraries are linked automatically and calls to

routines in them will be resolved when the `.so` file is loaded into Lisp. The important point, to state it again, is if you do get an unsatisfied external error, you must recreate the `.so` file and then load the modified `.so` file into Lisp. The example [below](#) shows how a `.so` file might be recreated.

Here is the behavior in this case on various platforms. On the following platforms, the load will fail with an error message that lists an unresolved external symbol:

- **Solaris:** Use the command `ldd your_library.so` in a shell to generate a list of library dependencies.
- **HP-UX:** Use the command `chatr your_library.sl` in a shell to generate a list of library dependencies.

On the following platforms, the load will fail without an adequate explanation. Unresolved externals is always a possibility for failure.

- **Dec Unix:** Use the command `odump -Dl your_library.so` in a shell to generate a list of library dependencies.
- **AIX:** The message may say "No such file or directory". Use the command `dump -H your_library.so` in a shell to generate a list of library dependencies.
- **Windows NT/2000:** You can use the NT application **Quickview** to examine which DLL's the DLL you are trying to load depends on.
- **Windows 98/Me:** We neither provide nor point to tools to give you further information. We are aware of that such tools might be available on the Web (one user suggested looking at the information on the <http://www.dependencywalker.com/> site). But if you did not build the DLL, it might be best to contact the DLL's author for further information.

1.6.3 One library file cannot depend on an already loaded library file

Each file must be complete in itself: all necessary routines must either be defined in the file or must be in a library specified when the file is created. On Windows, Dec Unix, and AIX, by default, the linker prevents building a shared library with unresolved external symbols.

Consider the example on a Solaris machine: `t_double()` (defined in `t_double.so`) calls `foo()` (defined in `foo.so`). Even if you have already loaded `foo.so`, trying to load `t_double.so` will fail, as follows:

```
USER(23): :ld foo.so
; Foreign loading /net/rubix/usr/tech/dm/acl/foo.so.
USER(24): :ld t_double.so
; Foreign loading /net/rubix/usr/tech/dm/acl/t_double.so.
  Error: loading library "/net/rubix/usr/tech/dm/acl/t_double.so"
        caused the following error:
```

```
ld.so.1: /net/sole/scm3/temp-images/lisp:
relocation error: symbol not found: foo: referenced in
/net/rubix/usr/tech/dm/acl/t_double.so
```

```
Restart actions (select using :continue):
0: retry the load of t_double.so
[1] USER(25):
```

There are two solutions to this problem. You can combine all necessary files (`t_double.so` and `foo.so` in our case) into a single file. On Solaris, this could be done with the following command starting with the `.o` files used to make the `.so` files:

```
% ld -G -o combine.so foo.o t_double.o
```

Alternatively, you can specify the file holding the needed routines (`foo.so` in our example) as a library for the file needing them (`t_double.so` in our example). On Solaris again, a command like:

```
% ld -G -o t_double.so t_double.o -R /net/rubix/usr/dm/acl -lfoo
```

1.6.4 Entry points are updated automatically

Suppose you load a library file which defines `bar()` and then use `def-foreign-call` to define `bar`, which calls `bar()`. Then you load another `.so` file which defines `bar()`. What happens?

Lisp will automatically modify `bar` so that it calls the `bar()` defined in the newly loaded file (and it prints a warning that it is doing so).

Let us make this clear with an example on a Solaris machine. Consider the two files `bar1.c` and `bar2.c`:

```
/* bar1.c */
void bar()
{
    printf("This is BAR 11111 in bar1.c!\n");
    fflush(stdout);
}
```

```
/* bar2.c */
void bar()
{
    printf("This is BAR 22222 in bar2.c!\n");
    fflush(stdout);
}
```

`bar1.so` defines `bar()` and `bar2.so` also defines `bar()`. We first load `bar1.so` and use `def-foreign-call` to define `bar`, and then load `bar2.so`.

```

USER(37): :ld bar1.so
; Foreign loading /net/rubix/usr/tech/dm/acl/bar1.so.
USER(38): (ff:def-foreign-call bar nil :returning :void)
T
USER(39): (bar)
This is BAR 11111 in bar1.c!
nil
USER(40): :ld bar2.so
; Foreign loading /net/rubix/usr/tech/dm/acl/bar2.so.
Warning: definition of "bar"
         moved from /net/rubix/usr/tech/dm/acl/bar1.so
         to /net/rubix/usr/tech/dm/acl/bar2.so.
USER(41): (bar)
This is BAR 22222 in bar2.c!
nil
USER(43):

```

1.6.5 Duplicate entry points are never a problem

Since one library file cannot depend on another unless specified when it was built, there is never a problem with duplicate entry points. It is important, however, to be sure that you know which foreign routine is being called.

1.6.6 If you modify a so/sl/dylib/dll file that has been loaded, you must load the modified file!

Failure to do so may cause Lisp to fail unrecoverably.

Suppose on Solaris you have **bar()** defined in *bar1.so* and you load *bar1.so* into Lisp. You also define the function **bar** with **def-foreign-call**. Then, you decide to modify *bar1.c* and *bar1.so*. Once the new *bar1.so* has been created, you must load it into Lisp before calling **bar**. Otherwise, Lisp may fail.

The problem is that the function **bar** knows it should look for the definition of **bar()** in *bar1.so* and it knows where to look in that file. If *bar1.so* is modified in any way, the place where Lisp will look for the definition of **bar()** is likely wrong. Lisp, however, does not know that and will look at that location anyway, taking whatever it finds to be valid code. The result can be disastrous, since typically Lisp hangs unrecoverably. In that case, Lisp may have to be killed from a shell and restarted.

1.7 Foreign code cannot easily be included at installation time

In earlier versions of Allegro CL on Unix, it was possible and relatively easy to build an executable image that included foreign code. Because of changes in the way images are built and the fact that the executable and image files are different (which is a new feature on Unix platforms), it is no longer easy to build foreign code into an image. The only way is to write a **main()** which does the necessary linking. See *main.htm*.

1.8 Foreign functions and multiprocessing

There are two models of multiprocessing used by Allegro CL (as described in *multiprocessing.htm*): the **:os-threads** model and the non **:os-threads** model. (**:os-threads** appears on the **features** list of implementations that use it).

In the non **:os-threads** model, foreign code is not interruptable and Lisp code never runs until foreign code returned control to Lisp, typically by completing but sometimes when the foreign code called back to Lisp. In the **:os-threads** model, Lisp code on one process may run while foreign code in another process may be waiting or be interrupted. This means that certain implicit features of foreign code may no longer hold. Among them is the most significant:

Lisp pointers passed to foreign code are not guaranteed to be valid by default. In earlier releases, because Lisp code never ran while foreign code was running, a Lisp pointer passed to foreign code was guaranteed to be valid until the foreign code completed or called back to Lisp. In the **:os-threads** model, because Lisp code and foreign code can run in different OS processes, Lisp code may be run concurrently with foreign code. A garbage collection may move Lisp objects rendering the pointer in the foreign code invalid. We recommend storing all values used by both Lisp and foreign code in foreign (not garbage-collected) space or dynamically allocated on the stack. It is also possible to block Lisp code from running until foreign code completes.

See the *release-heap* keyword argument to **def-foreign-call**. It prevents or permits other threads from running along with the foreign code.

A note on calling into Lisp from threads started by foreign code

Note that in all implementations of Allegro CL it is possible for a foreign function called from Lisp to explicitly start computation in additional threads, as supported by the OS (by having foreign code make the appropriate system calls to start the threads). One could imagine any of these new threads using the foreign function interface to invoke a Lisp function defined as foreign-callable (see **defun-foreign-callable**). In an Allegro CL with the non **:os-threads** model of multiprocessing, doing this is almost certain to have disastrous consequences. It is wholly unsupported but there is no protection in the foreign function interface to prevent it from happening. The only legitimate calls to a foreign-callable function will occur in the Lisp's own thread of control, as call-backs from foreign code that was itself called from Lisp.

In an **:os-threads** Allegro CL, however, it is legitimate for a thread started outside Lisp to call into Lisp via any

foreign-callable function. Some extra work has to be done to create a Lisp process to represent that thread within the lisp world. That extra work is performed by a "customs agent" process that is started by a call to **start-customs**. See the documentation for that function for details.

1.9 Creating Shared Objects that refer to Allegro CL Functionality

When creating foreign code to load dynamically into Lisp, it is sometimes necessary to refer to variables within Allegro CL's runtime such as `nilval` or `UnboundValue`, or else to call a function directly from C such as `lisp_call_address()`. In cases like these it is necessary to link in the Allegro CL shared-library (either `.dll`, `.so`, or `.sl`, depending on the architecture) with the shared-library that is being built. The actual name of the Allegro CL shared-library is available using the function `get-shared-library-name` and can be added to the link line to resolve symbols within it that are referenced in the foreign code. The situation on Mac OS X is different. See [Section 1.9.1 Linking to Allegro CL shared library on Mac OS X](#).

It is also possible to avoid delay linking the library until runtime. The advantage is that the library location can easily be determined at runtime, while at shared-library creation time finding the Allegro CL shared-library in a robust fashion (without worrying about `LD_LIBRARY_PATH` or equivalent and without hardwiring the location) can be difficult. See [Section 1.9.2 Delaying linking the Allegro CL shared-library until runtime](#) for details.

For example, on a Sparc,

```
% cat foo.c

#include "lisp.h"

LispVal
get_nil ()
{
    return nilval;
}

% cc -c -K pic -DAcl32Bit -I[ipath] foo.c
```

[ipath] is the location of *lisp.h* (usually *[Allegro directory]/misc/*). Note that on platforms that support 64 bit Lisps (HP's and HP Alphas, Sparcs, Mac OS X, etc.), the flag `-DAcl64Bit` must be used instead of `-DAcl32Bit` when compiling for the 64-bit Lisp.

Now, before linking, we need to find out where the Allegro CL shared-library is, so in the Lisp (for example, your own name and path might be different):

```
user(4): (get-shared-library-name)
"libacl80.so"
```

```
user(5): (translate-logical-pathname "sys:")
#p"/usr/acl80/"
user(6):
```

Then back in the shell and build the shared library (this example is from a Solaris machine):

```
ld -G -o foo.so foo.o /usr/acl80/libacl80.so
```

Note that other libraries, including system libraries, might need to be linked in order to complete the `ld`. Then, back in the Lisp, the load can be done:

```
user(6): :ld ./foo.so
; Foreign loading ./foo.so
user(7): (ff:def-foreign-call get_nil () :returning :lisp)
get_nil
user(8): (get_nil)
nil
user(9):
```

1.9.1 Linking to Allegro CL shared library on Mac OS X

Allegro CL does the `NSLoadModule` of `libacl8*.dylib` (the Allegro CL shared library) with the options `NSLINKMODULE_OPTION_BINDNOW | NSLINKMODULE_OPTION_RETURN_ON_ERROR`. The first of these options is just like the `RTLD_NOW` option of `dlopen`, and the second causes a zero return-value from a failed call to `NSLinkModule` instead of calling some error handlers (whose default action would be to kill the program). Missing from this set of options is the `NSLINKMODULE_OPTION_PRIVATE`, the lack of which means that all symbols that are exported from the module are made global and thus accessible in the running program. Unfortunately this means that there can be no multiply defined externals in either the program or the files it loads.

The file `/usr/lib/bundle1.o` that gets linked into a bundle file has an entry point called `"dyld_stub_binding_helper"`. Whenever any unresolved symbol is used in the loaded bundle file, this helper calls on the dynamic loader (`dyld`) to find the symbol in the running image. Thus the entry points are looked up in a truly dynamic manner, and linking against the Allegro CL library is unnecessary.

But it is also impossible to link against the Allegro CL shared-library, because being a bundle type object, it is self-contained, and the operating system doesn't allow relinking against this kind of shared library.

Thus the shared-libraries that can be dynamically loaded into a program can be thought of as separate entities which communicate via the dynamic loader.

1.9.2 Delaying linking the Allegro CL shared-library until runtime

When your foreign function needs to call into Lisp (for example, to call `lisp_call_address` or `lisp_value`), it may be complicated when the shared library created from your foreign function is created to link to the Allegro CL shared-library (`aclxxx.dll` or `libaclxx.{sl,so,dylib}`). It is possible to pass the Allegro CL shared-library handle to another shared-library so that it doesn't have to explicitly call entry-points within the Allegro CL shared-library. As a result, you need not specify the Allegro CL shared-library location when your shared library is created.

The runtime information is provided by the functions `get-shared-library-handle` and `get-executable-handle`.

Consider the following example, from a Solaris. A shared-library is created to call `lisp_call_address` without referring to that function in the link phase (`lisp_call_address`' address is looked up at runtime). Instead, information about the location is determined at runtime and passed to the foreign functions using the `init_mylib` function (see the `def-foreign-call` form in the comments at the end of the example).

```
#ifdef HP_CC
extern "C" {
#if !defined(Acl64Bit)
#define hp32 1
#endif
#endif

#ifdef WINDOWS
# define Dllexport _declspec(dllexport)
#else
# define Dllexport
#endif

#define CALLNAME "lisp_call_address"

#ifdef WINDOWS
void *
find_ff_symbol(void* handle, char *name)
{
    return GetProcAddress(handle, name);
}
#endif

#if defined(__APPLE__) && defined(__ppc__)
#include <mach-o/dyld.h>

#undef CALLNAME
#define CALLNAME "_lisp_call_address"

void *
find_ff_symbol(void *handle, char *name)
```

```

{
    NSSymbol sym;
    sym = NSLookupSymbolInModule(handle, name);
    if (sym != NULL) {
        return NSAddressOfSymbol(sym);
    }
    return NULL;
}

#endif

#if defined(hp32)
#include <dl.h>
void *
find_ff_symbol(void *handle, char *symbol)
{
    int value;
    shl_t handlecopy = (shl_t) handle;

    if (shl_findsym(&handlecopy, symbol, 0, &value) == 0) {
        return (void *)value;
    } else {
        return (void *)0;
    }
}

#endif

#if !defined(WINDOWS) && !defined(__APPLE__) && !defined(hp32)
#include <dlfcn.h>

void *
find_ff_symbol(void *handle, char *name)
{
    return dlsym(handle, name);
}
#endif

void *(*get_lisp_call_address)(int);

int Dlllexport
init_mylib(void *handle)
{
    get_lisp_call_address = (void *(*)(int))find_ff_symbol(handle,
CALLNAME);
    if(get_lisp_call_address) {
        return 1;
    }
}

```

```

    }
    return 0;
}

/*
 * In lisp, initialize by defining and calling init_mylib as a foreign
 * function:
 *
 * (ff:def-foreign-call init_mylib ((handle :foreign-address))
 *   :returning :int)
 * (init_mylib (excl:get-shared-library-handle))
 *
 * After init, usage of get_lisp_call_address() in inline C code is
 * the same as lisp_call_address():
 *
 * ...
 * int (*lispfunc)(...) = (int(*)())get_lisp_call_address(index);
 * lispfunc(...);
 * ...
 * where ... denotes arguments and their types in proper C convention.
 */

/* body of mylib code goes here */

#ifdef HP_CC
};
#endif

```

1.10 Releasing the heap when calling foreign functions

When you are running multiprocessing on a platform using the `:os-threads` model (see *multiprocessing.htm*), you have to worry about foreign functions in different threads modifying values in the Lisp heap in a thread unsafe way. (This is not a problem when running on a platform that does not use the `:os-threads` model because no lightweight Lisp process will run while foreign code is being executed.)

Therefore, on platforms using the `:os-threads` model, in order for any thread in a process to execute lisp code, it must have exclusive control of the data and control structures that define the lisp execution environment.

These resources are collectively called "the Heap" and access to them is controlled by OS synchronization primitives. Initially, the single thread that initializes the lisp environment has possession of the Heap. If multiple threads are running lisp code, Allegro arranges that they will each have access to the heap at appropriate times. A thread that runs lisp code for a long time may be preempted at various points so that control of the heap can be given to another thread. A thread that makes a call to a foreign function has the option of "releasing the heap" for the duration of the call. This allows another thread to take control of the heap while the first thread performs an action that may require significantly more real time than cpu time. A foreign call that is cpu-bound, however, would be better off not releasing the heap, especially if the call took a small amount of processor time and/or occurred frequently. See the discussion of the *release-heap* argument to **def-foreign-call** for information on how to indicate the heap can be released in a foreign call.

2.0 The 4.3.x UNIX model for foreign functions

All existing 4.3 exported symbols become deprecated, and preserved only for compatibility. There is a compatibility module for 4.3.x style foreign functions code. The module is called *ffcompat* and can be loaded with

```
(require :ffcompat)
```

3.0 The foreign function interface specification

The remainder of this document describes the foreign function interface in Allegro CL. Except as noted, the interface is the same on all platforms.

The description of foreign types is in the document *ftype.htm*.

3.1 Foreign function interface function summary

The following table shows the functions used in the foreign function interface. The **Notes** in the following table provide brief and by no means complete descriptions of the object being described. Please follow the link to the description page for the complete description including warnings and subtleties.

Name	Arguments	Notes

def-foreign-call	<i>name-and-options arglist &key option ...</i>	This macro defines a foreign function as the symbol-function of a Lisp symbol. See Section 3.2 def-foreign-call below.
def-foreign-variable	<i>name-and-options &key type convention</i>	This macro defines a foreign variable to Lisp. See Section 3.3 def-foreign-variable below.
def-foreign-type	<i>name-and-options def ...</i>	This macro defines a foreign type to Lisp and can make a connection between foreign and Lisp types, allowing proper conversion. See <i>ftype.htm</i> .
defun-foreign-callable	<i>name arglist &body body</i>	This macro defines a Lisp function which can be directly called by a foreign function. See Section 8.0 Using Lisp functions and values from C below.
register-foreign-callable	<i>name &optional index-or-reuse convert-to-c-types</i>	Registers the location of a Lisp function to allow access by foreign code. See Section 8.2 Calling Lisp functions from C: lisp_call_address() and lisp_call() below.
unregister-foreign-callable	<i>index</i>	Unregisters a Lisp function registered with register-foreign-callable . See Section 8.2 Calling Lisp functions from C: lisp_call_address() and lisp_call() below.
register-lisp-value	<i>value &optional index</i>	Register a Lisp value to allow access by foreign code. See Section 8.1 Accessing Lisp values from C: lisp_value() below.
unregister-lisp-value	<i>index</i>	Unregisters a Lisp value registered with register-lisp-value . See Section 8.1 Accessing Lisp values from C: lisp_value() below.
lisp-value	<i>index</i>	Simulates a call to lisp_value() . See Section 8.1 Accessing Lisp values from C: lisp_value() below.

convert-foreign-name	<i>name &key language</i>	Default function used to convert a Lisp name to a foreign entry point (prepending and appending characters such as underscores as needed. See Section 3.2.1 def-foreign-call syntax below.
get-entry-point	<i>name</i>	Returns the entry point for <i>name</i> , if there is one. Returns <code>nil</code> if there is not.
list-all-foreign-libraries	<i>&key return-structs &allow-other-keys</i>	Returns a list of all foreign libraries (so/sl/dylib/dll files) that are loaded into the current Lisp image.
unload-foreign-library	<i>filename</i>	Unmaps (i.e. unloads) the so/sl/dylib/dll files previously loaded with load specified by <i>filename</i> .
char*-to-string [Obsolete, use native-to-string instead]	<i>address &optional string make-string-p</i>	Takes the address of a C string and returns an equivalent Lisp string.
native-to-string	<i>address &key string make-string? external-format</i>	Takes the address of a C string and returns an equivalent Lisp string. This function has other uses in International Allegro CL (see <i>iacl.htm</i>). See Section 5.0 Passing strings between Lisp and C below.
string-to-char* [Obsolete, use string-to-native instead]	<i>string &optional address</i>	Takes a Lisp string and writes a C string to malloc space.
string-to-native	<i>string &key :address :external-format</i>	Takes a Lisp string and writes a C string to malloc space. This function has other uses in International Allegro CL (see <i>iacl.htm</i>). See Section 5.0 Passing strings between Lisp and C below.

with-native-string	<i>(string-var string-exp &key (start 0) end native-length-var external-format) &body body</i>	This macro provides an efficient, portable, and non-garbage (from the Lisp garbage collector's point of view) way of converting lisp-strings to addresses acceptable for foreign functions expecting native string arguments. This macro has other uses in International Allegro CL (see <i>iacl.htm</i>).
foreign-strlen	<i>address</i>	Returns the length of the C string located at address. See Section 5.0 Passing strings between Lisp and C below.

3.1.1 A note on foreign addresses

Locations of foreign objects (usually objects stored outside the Lisp) are typically specified by addresses. A true address is a machine integer. Lisp integers are not machine integers (because certain bits have special meaning in fixnums and bignum have a header) but a true machine integer can be extracted easily enough so long as the system knows to do it. Foreign functions defined to Lisp with **def-foreign-call** may take foreign type objects, or vectors or foreign-addresses (integers, vectors, or foreign-pointer objects) as arguments.

A *foreign type* is either one of the built-in types (like `:int`) or one of a system of foreign structures defined with **def-foreign-type**.

A *foreign-pointer* is an instance of the class `foreign-pointer` (see also **make-foreign-pointer**) that has a special slot (with accessor **foreign-pointer-address**) intended to point to raw data.

A *foreign adress* is either a Lisp integer, a Lisp vector, or a foreign-pointer instance.

Note that foreign-types and foreign-pointers have nothing particular to do with each other. (A foreign-pointer is really a way to identify an integer as a foreign pointer -- rather than any old value -- to allow better argument and type checking.)

The second required argument to **def-foreign-call** is the argument list for the foreign function. The elements of that list are lists of argument names (symbols) followed by the foreign type to which it should be converted (and perhaps followed by additional values). A symbol alone is the same as the list `(symbol :int)`.

A Lisp value passed as an argument to a foreign function will be converted appropriately according to the specification in the argument list.

Thus, when a foreign type is specified as an argument to a foreign call, the value passed will be converted from Lisp to C by specific rules. For example, a `:int` type expects a Lisp integer and converts it to a machine

integer. A `:struct` is adjusted to point to its first element, etc.

When the argument is specified `:foreign-address`, the system will accept as a value (1) a Lisp integer (converted to a machine-integer), (2) a Lisp vector (adjusted to point to its first element), or (3) a foreign-pointer (converted by extracting the **foreign-pointer-address** from the object).

3.2 def-foreign-call

The macro **def-foreign-call** associates a foreign function with a Lisp symbol, allowing it to be called as a Lisp function would be called.

The problem in making such an association is that Lisp types and foreign types are usually different, and so as part of the definition of a foreign function, the system must be told how Lisp objects passed as arguments should be converted to foreign types and how a returned value should be converted to a Lisp type. Further information can be provided as well: should the call be a candidate for inlining? When the multiprocessing model is **:os-threads**, should other processes be able to modify the Lisp heap while the foreign function is running? And so on.

The definition of **def-foreign-call** is on its own description page, found by following the link. We discuss the syntax of a call next and then make some other comments and provide examples.

3.2.1 def-foreign-call syntax

```
def-foreign-call name-and-options arglist &key kwopt ... MACRO
```

This macro is used to describe a function written in a foreign language (usually C). The action of this macro is to define a Lisp function which when called will pass control to the foreign function, making appropriate data transformations on arguments to and the return value from the foreign function.

```
name-and-options -> name-symbol ;; default conversion to external name
                  -> (lisp-name-symbol external-name)

external-name    -> [convert-function] [external-name-string] ;; default
convert function is

                                                    ;; convert-
foreign-name

arglist          -> ()           ;; Implies default argument processing
                  ;; (like old defforeign :arguments t spec)
                  -> (:void)    ;; Explicitly looking for no arguments
```

-> (arg ...)

arg

-> name

-> (name complex-type-spec)

;; If name is nil, a dummy name will be supplied.

type-spec

-> foreign-type

-> (complex-type-spec)

complex-type-spec-> foreign-type [lisp-type [user-conversion]]

user-conversion -> symbol *;; Not yet documented*

lisp-type

-> any Lisp type

;; This constrains the runtime arg to this

;; Lisp type. When trusting declarations we

;; assume the Lisp type and optimize accordingly.

;; If declarations are appropriate the compiler

;; may generate checks to validate the lisp type.

kwopt

-> :RETURNING type-spec

;; Note that if the foreign type mentioned

;; in the :RETURNING clause is expressed as

;; a list (i.e. begins with a parenthesis) then the

;; second form of type-spec -- (complex-type-spec)

;; -- must be used. Otherwise, the first

;; component of the foreign type is

;; treated as the first component of a

;; complex-type-spec and a spurious

;; error is signaled. For example, if

;; the type is (Foreigntype), that is*

;; a pointer to a struct of type

;; Foreigntype, then the correct specification

;; is `:returning ((Foreigntype))'*

-> :CONVENTION { :C | :STDCALL | :FORTRAN}

;; On NT, C (cdecl) and stdcall are now equivalent.

;; :FASTCALL is not supported.

-> :ARG-CHECKING { NIL | T }

-> :CALL-DIRECT { NIL | T }

-> :METHOD-INDEX index *;; Ordinal index of C++ member*

;; method; vtbl is first argument.

-> :CALLBACK { NIL | T } *;; Callback currently forced to*

T.

-> :RELEASE-HEAP { :NEVER | :ALWAYS | :WHEN-OK }

-> :ERROR-VALUE { NIL | :ERRNO | :OS-SPECIFIC } *;; When*

;; non-NIL, return error value as

```
;; second return value from foreign call.
```

3.2.2 def-foreign-call: the :returning keyword argument

The `:returning` keyword argument specifies what the foreign function will return when it completes. This value, suitably modified to a Lisp value, will be returned by the Lisp function associated with the foreign function.

The value specified for `:returning` (1) a foreign type (defined by **def-foreign-type**). (2) a list of a foreign type and a Lisp type (and an optional third element which is not used but may be in a later release); example: `(:double single-float)`. (3) `((* :char))`, or `((* :char) string)`, etc.; this causes **native-to-string** to be called automatically after the return. (4) the value `:lisp`, meaning a Lisp object will be returned and no conversion should be done. This is a dangerous option since if the returned value is not actually a Lisp object, a gc failure may occur. (5) The value `:foreign-address`, which will be interpreted as an unsigned integer and converted to a positive Lisp integer. (6) The value `:void`, meaning nothing will be returned and the Lisp function should return `nil`. (There are some additional deprecated options: see the description of **def-foreign-call**.)

The default for `:returning` is the foreign type `:int`, which is a value of type (1) in the list above. The integer value is converted upon return to Lisp type integer (a fixnum with a possible overflow to a bignum). This default value is not appropriate when the foreign function is returning a long, an unsigned long, or a pointer of some sort. However, the fact that it is not appropriate is masked on 32-bit architectures by the fact that, as it happens, an `:int` is effectively equivalent to those values.

On 64-bit architectures, however, this is not true so `:int` cannot serve for a long, an unsigned long, or a pointer of some sort. Therefore, when returning a pointer, the type should be `:unsigned-long` and when returning another integer value, the type should be `:int` or `:long` or `:unsigned-long`, as appropriate.

3.2.3 def-foreign-call examples

Here are some examples of **def-foreign-call** usage:

```
(def-foreign-call add2 (x y))
```

Call a function, probably named "add2" in C, whose first arg is named "x" and is an integer in Lisp and which is converted to an int for passing to C. If the integer is larger than can be held in a C int, it is truncated. As with the first arg, the second arg named "y" is an integer converted to a C int. The return value is interpreted as a C int type, and is converted to a Lisp integer (which may either be a fixnum or consed as a bignum).

```
(def-foreign-call t_double ((x :double)
```

```

        (y :double single-float)
        (z :int fixnum))
:returning :double))

```

Call a function, probably named "t_double" in C, whose first arg is a double-float both in Lisp and in C, and whose second arg is a single-float in Lisp but is converted to double float for passing into C (this is the calling convention used by some non ANSI C compilers and by others when the arguments are not prototyped), and the third argument is a fixnum Lisp passed as an integer to C. The function returns and boxes a double-float value to Lisp.

```

(def-foreign-call (t-float dash-to-underscore) ((x :double)
                                                (y (:float :no-proto))
                                                (z :int fixnum)
                                                (w (* :char) string))
:returning #- (or (and sun4 (not svr4)) sun3q) :float
            #+ (or (and sun4 (not svr4)) sun3q) (:double single-float)

```

```

(def-foreign-call (t-float "t_float") ((x :double)
                                       (y (:float :no-proto))
                                       (z :int fixnum)
                                       (w (* :char) string))
:returning #- (or (and sun4 (not svr4)) sun3q) :float
            #+ (or (and sun4 (not svr4)) sun3q) (:double single-float)

```

These two examples do the same thing: call a function, named "t_float" in C (assuming in the first case proper conversion by **dash-to-underscore**, which must already be defined and should downcase the symbol name and replace dashes with underscores), whose first arg is a double-float both in Lisp and in C. Like the previous example, the second arg is a float in Lisp, and is converted to double float for passing into C. The third arg named "z" is a fixnum passed as an int, and "w" is a (null-terminated) Lisp string, whose first-character-address is passed to C (beware, the string may move if a gc is allowed). Depending on the architecture, the C function will return either a double (from older C compilers) or a float, each interpreted and boxed as a Lisp single-float value.

We give both examples to show how a lisp name (the symbol `t-float`) is converted to a foreign name ("`t_float`"). You can either specify a function that takes a symbol and returns the correct string (so `(dash-to-underscore 't-float)` returns "`t_float`") or you can simply specify the correct string. Note again that **dash-to-underscore** must be already defined when the `def-foreign-call` form is evaluated.

```

(def-foreign-call c_array ((str (* (* :char)) (simple-array simple-string
(*)))
                          (n :int fixnum))
:returning :char)

```

Call a function whose C name is probably `c_array`, whose "str" argument is an array of strings, properly converted (by copying from the Lisp parts). The second arg "n" is a Lisp fixnum shifted to make a C int. The C

function returns a char which is made into a Lisp character.

3.3 def-foreign-variable

The **def-foreign-variable** macro is used to describe a variable exported by a foreign language (usually C). The action of this macro is to define a Lisp **symbol-macro** which when evaluated will obtain the address of the foreign variable and either retrieve or set the value of the variable based on the access-type.

3.3.1 def-foreign-variable syntax

def-foreign-variable name-and-options &key kwopt ... MACRO

This macro is used to describe a variable exported by a foreign language (usually C). The action of this macro is to define a Lisp symbol-macro which when evaluated will obtain the address of the foreign variable and either retrieve or set the value of the variable based on the access-type.

```
name-and-options -> name-symbol ;; default conversion to external name
                  -> (lisp-name-symbol external-name)

external-name    -> [convert-function] [external-name-string]

kwopt            -> :type access-type
                  -> :CONVENTION { :C | :FORTRAN}
```

3.3.2 def-foreign-variable examples

Here are some examples of **def-foreign-variable** usage:

```
user(1): (ff:def-foreign-variable sigblockdebug)
sigblockdebug
user(2): sigblockdebug
0
user(3): (setq sigblockdebug 1)
1
user(4): (ff:def-foreign-variable (print-lso-relocation
"print_lso_relocation"))
print-lso-relocation
```

```

user(5): print-lso-relocation
0
user(6): (setq print-lso-relocation 1)
1
user(7):

```

In these examples, variables from the Allegro CL internal runtime are set up to allow Lisp to access the variables in a lisp-y way. Since these variables are not documented, it would take a C debugger to examine and verify that the values of the C variables had indeed been changed.

WARNING: Do not attempt to use this macro on the C "errno" variable, nor on any other variable that might be bound on a per-thread basis; doing so might cause (incorrect) information to be returned for the wrong thread.

4.0 Conventions for passing arguments

Arguments to function calls can be passed in two ways, by value and by address. When an argument is passed by value, a copy of the value is placed somewhere (typically on the stack) where the function can access it. When an argument is passed by address, a pointer to its actual location is given to the function. Arguments in C are usually (but not always) passed by value, while arguments in Fortran are normally passed by address.

4.1 Modifying arguments called by address: use arrays

A function that receives an argument called by value can modify that value. There are two problems with such modifications:

- Lisp may not see the modification when you want it to (because Lisp makes a copy of the object and passes the address of the copy -- the copy is indeed modified but the original is not).
- Lisp may see the modification when you do not expect it to. If you pass a floating point value, it may be modified and Lisp may see the modification even though that is not intended.

We deal with each situation in turn. The solution is the same in both cases: pass the address of an array.

4.2 Lisp may not see a modification of an argument passed by address

When an argument is passed by address in C or Fortran code and the called function changes the value of the argument, the argument will stay changed even after control returns from the called function. The actual stored

value of the argument will have been permanently modified. This is expected behavior and is generally what is intended and desired.

Users therefore should be warned that in many cases (arrays being the main exception) when Lisp code calls a foreign function that modifies one of the arguments passed by address, the Lisp value of that argument will be unchanged. The reason is that Lisp represents objects differently from C or Fortran and so often cannot pass the actual address of the Lisp object to the foreign code, since the foreign code would not correctly interpret the value pointed to. Instead, Lisp makes a copy of the Lisp object, changing the representation appropriately, and passes the address of the copy. Although this copied value is modified by the foreign code, Lisp ignores the copied value after the function returns, looking only at the unmodified Lisp object.

To repeat the above warning: Fortran functions do not always affect the value of a Lisp object when this is passed to a Fortran function. The following example illustrates this behavior. Say we have the Fortran file *fnames.f* containing the function **itimes2()**:

```
function itimes2(x)
integer x
x = 2*x
itimes2 = x
return
end
```

This function appears to double the (C or Fortran) integer value stored in the location specified by *x*. We compile and load the file into Lisp and then run the function. Here are the results:

```
USER(21): (load "fnames.so")
t
USER(22): (ff:def-foreign-call itimes2 (fixnum)
          :convention :fortran :returning :void)
;; send in a fixnum, but pass by address
t
USER(23): (setq x 19)
19
USER(24): (itimes2 x)
38 ;; gives 38 as expected
USER(25): x
19 ;; but x is unchanged.
```

The problem is that a Lisp fixnum is not the same as a Fortran (or C) integer, and thus the foreign-function interface must convert it. It copies the Lisp fixnum, converts it to a Fortran integer, and then passes the address of the converted copy to the Fortran function. The expected Fortran behavior can be achieved by passing an array instead of a fixnum.

```
USER(30): (ff:def-foreign-call itimes2 ((simple-array fixnum (1)))
          :convention :fortran :returning :void)
t
```

```

USER(31): (setq x (make-array 1 :element-type 'fixnum
                             :initial-element 19))
#(19)
USER(32): (itimes2 x)
38 ;; gives 38 as expected
USER(33): x ;; as does x
#(38)

```

4.3 Lisp unexpectedly sees a change to an argument passed by address

Again, when an argument is passed by address in C or Fortran code and the called function changes the value of the argument, the argument will stay changed even after control returns from the called function. The actual stored value of the argument will have been permanently modified.

This can cause trouble when, for example, you pass a floating point value to a FORTRAN routine that modifies the value. Consider the following FORTRAN subroutine:

```

subroutine dtest(x)
double precision x

x = x + 1.0d0
return
end

```

It increases the value of its argument by 1.0d0. We compile this function, load it into Lisp, and define it as a foreign function with **def-foreign-call**:

```

(ff:def-foreign-call dtest ((x :double double-float))
                          :convention :fortran
                          :returning :void)

```

Now we run it, though the behavior you see may be different from what is reported here. In some cases the constant value is changed (as shown). In other cases, Lisp simply hangs unrecoverably. In any case, things are too broken to continue (computation which believes 0.0d0 equals 1.0d0 is unlikely to be useful).

```

USER(3): (dtest 0.0d0) ;; Lisp may hang at this point rather than
                      ;; returning control to the Listener.

nil
USER(4): 0.0d0
1.0d0

```

Whoa! What happened to 0.0d0? Well, here is what happened. 0.0d0 is a Lisp object of type double-float. As such, it has a type code and a pointer to its actual value, 0.0d0. When `dtest` is called with 0.0d0 as an argument, the location of the value is passed. FORTRAN then modifies that location. The trouble is that Lisp still thinks that location contains 0.0d0 and so when Lisp reads 0.0d0, it gets the value in the location and finds 1.0d0 instead.

In fact, Lisp reuses floating-point values whenever it can. Since there is no way within Lisp to modify a value, there should be no problem with using the same value over and over and doing so will save time and space.

It could be argued that Lisp should protect itself by making a copy of the floating-point value and passing that to FORTRAN. However, that would put a cost on many foreign calls where protection was not necessary. Instead, there is a simple workaround for users who wish to call by address code that modifies floats: specify the float as a length 1 vector and pass the vector:

```
(ff:def-foreign-call dtest ((x (:array :double)))
                          :convention :fortran
                          :returning :void)

USER(8): (setq x1 (make-array 1 :element-type 'double-float
                             :initial-element 0.0d0))

#(0.0d0)
USER(9): (dtest x1)
nil
USER(10): x1
#(1.0d0)
USER(11): 0.0d0
0.0d0
```

Fortran cannot distinguish between the address of the start of an array and the address of a single value, so passing the address of the double-float array `x1` is correct and works as we want.

4.4 Passing fixnums, bignums, and integers

Another difficulty arising out of differing Lisp and non-Lisp representations of values is illustrated by the example `itimes2()` [above](#). The argument passed to the foreign function was a fixnum, not an integer. Integers can be bignums or fixnums. C or Fortran integers may be larger than all possible Lisp fixnums and smaller than most but not all bignums. If a fixnum is passed to foreign code, it is always correctly represented, but a bignum can be represented only if it is small enough. The foreign-function interface will truncate any bignum that does not fit into the foreign integer representation without warning. Users can avoid this by not using integer as a value for *arglist*, the second required argument to **def-foreign-call** and thus not passing bignums, except when the argument value was generated by foreign code. The return value from foreign code (the value of the *returning* keyword argument) defaults to type `:int`, and since some foreign integers are too big to be fixnums, they may be bignums. But, since they came from foreign code, they will be correctly represented as foreign

integers when passed back to foreign code. Only in the case where you are sure the value can be represented as a machine integer do we recommend integer as a value for *arglist* (more precisely, as an element in the list which is the value).

4.5 Another example using arrays to pass values

An example illustrates the use of arrays. Say there is a compiled C shared object file *myreverse.so*:

```
int myreverse(n, x)
double *x; /* pointer to array of doubles */
int n; /* array length */
{
    int i;
    double d;
    for (i=0; i <= n/2; i++) {
        d = x[i];
        x[i] = x[n-1-i];
        x[n-1-i] = d;
    }
    return n;
}
```

In Lisp you might define (after loading *myreverse.so*) this function as follows:

```
USER(40): (ff:def-foreign-call myreverse ((i :int fixnum) (x (:array :
double))))
t
USER(41): (setq x (make-array 3 :element-type 'double-float
                           :initial-contents '(1.0d0 2.0d0 3.0d0)))
#(1.0d0 2.0d0 3.0d0)
USER(42): (myreverse (length x) x)
3
USER(43): x
#(3.0d0 2.0d0 1.0d0)
```

5.0 Passing strings between Lisp and C

Allegro Common Lisp provides a few functions to help converting strings from Lisp to C and back again. In this section, we discuss the issues and provide some examples.

Starting in release 6.0, specifying `((* :char))`, or `((* :char) string)`, etc. as the value of the `returning` to **def-foreign-call** causes **native-to-string** to be called automatically after the return. This is a safe change as far as compatibility goes, because pre-6.0 versions would error on such a specification. The alternative specification described here still works, and should be used if any external-format other than `:default` is desired. Thus in the example just below, you can specify `'((* :char))` instead of `:int` as the value of `:returning` and you will see the string rather than the address of the string returned. Of you cannot apply (and do not need to apply) **foreign-strlen** to the result.

Note that ``C strings'` are conceptual only. A ``C string'` is really not a type, but a usage of a character pointer and its storage. A Lisp string is an actual type and can be distinguished from other pointers. There are other uses for a C character pointer besides strings, but strings are most often passed in interface functions. By default, the **def-foreign-call** interface expands a `((* :char))` to `((* :char) string)`, which causes a Lisp string to be either created (on return) or checked for (when passed as an argument). If such checking or creation is not desired, (as would be the case where the Lisp value would be just an integer or an array of a different type) specify the actual type of the value of the argument, e.g

```
(def-foreign-call foo ((x (* :char) integer))
  :returning ((* char) integer))
```

`;; or`

```
(def-foreign-call foo ((x (* :char)
  (simple-array (unsigned-byte 8) (*))))))
```

If you have an address of a C string, you can pass it to **native-to-string** and a Lisp string with the same contents will be returned. (You can specify a string to receive the C string as a keyword argument to **native-to-string**, as described in the full description of that function).

The function **foreign-strlen** takes an address of a C string and returns its length.

```
;; This example calls a foreign function, GET-MESSAGE, that
;; returns a pointer to a string.
;; The C language file getmessage.c contains:
```

```
#include <stdio.h>
```

```
char *mess="this is a test";
char *getmessage()
{
  return mess;
}
```

```
;; Compile the C language file and load it into Lisp.
```

```
;; GETMESSAGE returns an integer pointer to the string.
```

```
USER(12): (ff:def-foreign-call getmessage (:void))
```

```
    :returning :int)
```

```
T
```

```
USER(13): (getmessage)
```

```
8790592
```

```
;; EXCL:NATIVE-TO-STRING converts the integer returned by getmessage
;; to a string.
```

```
USER(13): (excl:native-to-string (getmessage))
```

```
"this is a test"
```

```
;; Use FOREIGN-STRLEN to find the length
;; of the string returned by GETMESSAGE.
```

```
USER(14): (ff:foreign-strlen (getmessage))
```

```
14
```

With the **def-foreign-call** foreign function definer, one can specify string as one of the arguments. That will cause Lisp strings to automatically be converted to C style char* arrays at function call time.

To copy a Lisp string to a C style char * string outside of using **def-foreign-call** is to use the function **string-to-native**.

Our example passes a string to C, which calculates its length and returns it.

```
:: UNIX example
```

```
;;
```

```
;; This example calls a foreign function, PUTMESSAGE, that
;; returns a pointer to a string. Note that this is a simple
;; example. PUTMESSAGE could be def-foreign-call'ed to take a
;; SIMPLE-ARRAY as an argument.
```

```
;;
```

```
;; The C language file put-message.c contains:
```

```
#include <stdio.h>
```

```
/* putmessage expects a pointer to a string as an argument.*/
```

```
/* putmessage prints that string to stdout.*/
```

```
putmessage(s)
```

```
char *s;
```

```
{
    puts(s);
    fflush(stdout);
}
```

```
;; Compile the C language file and load into Lisp.
```

```
;; PUTMESSAGE takes a pointer to a string (an integer) as
;; an argument
```

```

USER(20): (ff:def-foreign-call putmessage (integer)
          :returning :void)

T
;; Create a string in lisp.
USER(21): (setf lisp-message "This is a message from lisp")
          "This is a message from lisp"

;; Run PUTMESSAGE with a lisp string as an argument.
USER(22): (putmessage (excl:string-to-native lisp-message))
          This is a message from lisp

NIL

```

Here is the example for Windows:

```

;; Windows example
;;
;; Allegro CL makes available to C programs the function
;; aclprintf() which operates just like printf and the result
;; is printed to the Allegro Common Lisp Console.
;; To show the console window right click on the
;; Lisp icon on the system tray and choose Show Console.

```

```

#|

```

```

----- the file put-message.c

```

```

extern void aclprintf(char *, ...);

```

```

void _declspec(dllexport) putmessage(char *str)
{
    aclprintf("message from lisp: '%s'\n", str);
}

```

```

|#

```

```

;; Compile the C language file and load into Lisp.

;; PUTMESSAGE takes a pointer to a string (an integer) as
;; an argument

```

```

user(2): (ff:def-foreign-call (putmessage "putmessage")
        ((str (* :char))) :returning :void)

```

```
putmessage
user(3): (putmessage "a lisp string")
nil
```

;; On the console window you see printed:

```
message from lisp: 'a lisp string'
```

You may want to pass strings from Lisp to C and from C to Lisp. Passing strings from Lisp to C is pretty easy. A Lisp string will be converted correctly. Passing an array of strings is more complex. Examples of both are shown in [Passing strings from Lisp to C](#) and [Special case: passing an array of strings from Lisp to C](#).

5.1 Passing strings from Lisp to C

Lisp will correctly convert Lisp strings when passing them to C and therefore passing a string from Lisp to C is quite easy. Consider the following example.

We define a C function **stringl()** which takes a string as an argument and returns its length.

```
;;; C code for UNIX:

# include <stdio.h>
# include <string.h>

int stringl(char *s)
{
    return strlen(s);
}

;; C code for Windows:

#ifdef _WIN32
#define DllExport __declspec(dllexport)
#else
#define DllExport
#endif

# include <stdio.h>
# include <string.h>

DllExport int stringl (char *s)
{
    return strlen(s);
```

}

We compile this function and load the resulting `.so/.sl/.dll/.dylib` file into Lisp. We then call **def-foreign-call** as follows and then call the C function:

```
user(50): (ff:def-foreign-call stringl ((string (* :char)))
          :strings-convert t
          :returning :int)
t
user(51): (stringl "hello")
5
user(52):
```

5.2 Special Case: Passing an array of strings from Lisp to C

Passing an array of strings from Lisp to C is somewhat more complex, as the next example shows. A common usage in C is typified by the following program fragment:

```
#define null 0
char *z[] = {"strings1", "string2", null};
...
    handle_strings(z);
...
handle_strings(argv)
char **argv;
{
    while( *argv != null ){
        handler_for_string(*argv);
        argv = argv + 1;
    }
}
```

Similar usage is also common with the array size included:

```
char *z[] = {"strings1", "string2", "string3"};
...
    handle_strings(3,z);
...
handle_strings(argc, argv)
    char **argv;
    int argc;
{
```

```
...
}
```

The variable `argv` is an array with each element pointing to a C string in both cases. (Note, however, that in the first case a NULL pointer terminates the array.) One may like to call `handle_strings()` from Lisp (after doing a **def-foreign-call**) by something like the following:

```
(handle_strings
  (make-array 3
    :initial-contents
    '("string1" "string2" 0)))
```

or perhaps

```
(handle_strings 3 (make-array 3
  :initial-contents
  '("string1"
    "string2"
    "string3")))
```

depending on the definition of `handle_strings()` above. However, the foreign-function interface does not normally convert the individual elements of a Lisp array.

One can convert an array of Lisp strings to a foreign object acceptable as a C `char**` argument by using a function such as that below. Note that as written it does fresh allocations on each call, so a user may wish to tailor it as desired. In particular, a call to **string-to-native** returns a value which must be passed to **acfree** in order to be reclaimed.

```
;; Take a lisp vector of lisp strings, and return an equivalent
;; foreign array of C strings. Useful for C functions expecting
;; 'char **' arguments.
;;
(defun lisp-string-array-to-c-string-array (a)
  (let ((r (ff:allocate-fobject (list ':array (* :char) (length a)))))
    (dotimes (i (length a))
      (setf (ff:fslot-value-typed '(:array (* :char)) nil r i)
            (string-to-native (aref a i))))
    r))
```

The array-of-strings (`argv`) argument to the foreign function can then be declared as follows:

```
(ff:def-foreign-call handle_strings ((argc :int) (argv (* (* :char)))))
```

The foreign function can then be called as follows:

```
(handle_strings 3 (lisp-string-array-to-c-string-array
                  (make-array 3
                              :initial-contents '("one" "two"
"three"))))
```

Note that before the current Allegro CL deftype facilities were available, foreign-function definers **def-foreign-call** and its (now obsolete) predecessor **defforeign** were designed to handle arrays of strings specially by use of the argument type `(simple-array simple-string (*))`. This argument type usage is no longer recommended, but it is documented here to describe backward compatibility.

While this is not implemented as a distinct data type in Allegro CL, the foreign-function interface will recognize this declaration and convert the array appropriately for C. This is a slow function call as the interface must allocate space to do the conversion. To get the desired behavior (e.g. for the second of the above two possibilities for **handle_strings()**) you should use:

```
(ff:def-foreign-call handle_strings
                    ((integer :int)
                     (string (:array (* :char))
                              (simple-array simple-string))))
```

Note that if you do not declare arguments - e.g. if you use:

```
(ff:def-foreign-call handle_strings nil)
```

The array will not be converted correctly on the call to **handle_strings()**. Note that this is not typical; the interface normally converts arguments according to their Lisp data type whether or not they are declared.

If you do make this declaration and pass in an arbitrary Lisp array, all bets are off. Only 0 and array elements of type `simple-string` are guaranteed to be correctly converted.

6.0 Handling signals in foreign code

The Lisp image catches all signals from the operating system. In particular, when an asynchronous interrupt (e.g. SIGINT on Unix) occurs, the signal handler in Lisp sets a flag and then returns. This flag is checked inside Lisp functions only when it is safe to do so. If you are executing in foreign code, any signals that are received will not be processed until some point after you return to Lisp. So if your C code gets into an infinite loop, you won't be able to interrupt it cleanly (you will be able to interrupt -- see *startup.htm*). This also implies that a foreign function is not interruptible by the Lisp scheduler in implementations that use the non `:os-threads` model of multiprocessing, see *multiprocessing.htm*.

If you need to be able to catch signals in foreign code, you must establish your own signal handlers when you enter the foreign code and restore the Lisp signal handlers before you return to Lisp. Perhaps the easiest way to

do this is to `wrap` your foreign code in a function that takes care of these tasks. This wrapper function calls your real function, and it returns the value returned by the real function.

Here is an example of such a function, which catches interrupts (Signets). This example uses the `signal()` function for simplicity. (Some versions of Unix supply other, more advanced, functions.)

```
#include <setjmp.h>
jmp_buf catch;
int (*old_handler)();

int
wrapper(arg1, ..., argn)
...
{
    auto int return_value;
    extern int new_handler(),
    real_function();

    if ((return_value = setjmp(catch)) != 0)
        return return_value;
    old_handler = signal(sigint, new_handler);
    return_value = real_function(arg1, ..., argn);
    signal(sigint, old_handler);
    return return_value;
}

int
new_handler()
{
    signal(sigint, old_handler);
    longjmp(catch, -1);
}
```

The wrapper function first calls **setjmp()** to establish a C stack-frame environment for a subsequent **longjmp()** to return to. (This is the C equivalent of Lisp `catch` and `throw`.) The **setjmp()** function returns zero when the catch is established; it returns the value of the second argument to **longjmp()** otherwise. If the **setjmp()** was the target of a **longjmp()** (from within the interrupt handler), we return the value returned by the **longjmp()** (here -1 to signal an abnormal return).

The wrapper then installs the new `SIGINT` handler, saving the address of the old one. Once the interrupt handler is established, the real function is called and we save the return value (here an integer). Next we restore the Lisp signal handler, then return the value returned by the real function to Lisp.

If an interrupt occurs while this C code is executing, **new_handler()** gains control. It restores the Lisp signal handler, and then jumps to the established catch: control returns to the point of the `setjmp()`, which this time returns -1. Foreign code execution is interrupted, and we cleanly return to Lisp.

Note that the Lisp callback functions `lisp_value()` and `lisp_call_address()` (described in [Section 8.1 Accessing Lisp values from C: `lisp_value\(\)`](#) and [Section 8.2 Calling Lisp functions from C: `lisp_call_address\(\)` and `lisp_call\(\)`](#)) should never be called from a foreign signal handler. If your signal handler does call `lisp_value()` or `lisp_call_address()`, failures may occur should the Lisp need to garbage collect while in the signal handler. The reason for this failure is that the system stack may not have been set up to indicate a call from Lisp to C. Also, if the signal is delivered while the garbage collector is running, then the entire Lisp heap may be inconsistent and accesses to the Lisp heap may result in attempting to follow pointers to nonexistent data.

7.0 Input/output in foreign code

Input and output from foreign code may require special consideration to obtain reasonable behavior.

Because foreign output operations will be interspersed with Lisp output operations, it is necessary to flush output in foreign code before returning to Lisp if it is desirable to maintain synchronous behavior. For example, if a C function writes information to the standard output, it may not be displayed contemporaneously unless `fflush()` is used before returning to Lisp.

When performing input and output from Fortran, it may be necessary to set up the Fortran I/O environment before performing any operations. For 4.*n* BSD Unix systems, using the standard AT&T Fortran compiler or one of its derivatives, the following steps are necessary to perform I/O successfully.

Suppose you want to call the following simple subroutine from Allegro CL:

```
subroutine fiotest
write(6, '("this is some fortran output.")')
call flush(6)
return
end
```

Because the above program contains an input/output statement, certain subroutines from the Fortran I/O library must be loaded. (Note that the call to subroutine `fflush()` is implementation-dependent.) These subroutines initialize the I/O units by 'preconnecting' unit 5 to standard input, unit 6 to standard output, and unit 0 to standard error. If this initialization is not done, various errors can occur. For example, some versions of the Fortran library routines will execute an `abort()` if I/O has not been initialized, causing Lisp to dump core. Other versions merely ignore requests for output. Some versions will create disk files named `fort.[n]` where `[n]` is the Fortran unit number. The routine `f_init()` of the Fortran I/O library will perform the proper initialization. This initialization need only be done once for every Lisp session.

Assuming the compiled version of the above program is contained in the file `fiotest.so`, on a UNIX machine, here is a transcript of how to load the program into Allegro CL:

```

;; After loading the Fortran subroutine and
;; the F77 and I77
;; libraries into the Lisp
;; we use FF:DEF-FOREIGN-CALL to create a Lisp function
;; F_INIT that points to the function _f_init() in
;; the Fortran library:
USER(61): (ff:def-foreign-call f_init nil :returning :void)
t
;; We define our Fortran test subroutine to Lisp.

USER(62): (ff:def-foreign-call fiotest nil :convention :fortran
          :returning :void)

;; We initialize the Fortran I/O system, . . .
USER(63): (f_init)
nil
;; . . . then call our Fortran subroutine.
USER(64): (fiotest)
This is some fortran output.
nil

```

8.0 Using Lisp functions and values from C

This section describes the Allegro CL facility that permits C functions to call Lisp functions and access Lisp values. The C functions must have been loaded into Lisp, and must have been called from Lisp.

Because some Lisp objects may move in memory when a garbage collection occurs, calling out to Lisp must be used with great care on the part of the C programmer. As an example, if an array is passed to a C function which calls out to a Lisp function and a garbage collection occurs, then after the C function returns, the pointer to the array will point to nothing; the array data will have moved somewhere else. So if a C function accesses a Lisp value and calls out to Lisp, then it is recommended that the Lisp value be registered and accessed as described next.

Another alternative is to store the value in a location where they are not moved. Allegro CL supports static arrays which are guaranteed never to move once they have been allocated (they are allocated in foreign rather than Lisp space). Static arrays are discussed in *gc.htm*. However, static arrays are not completely general. This section covers cases where you are interested in accessing objects other than arrays or in types of arrays not supported by static arrays.

To give a particular example, let us say:

1. You register a Lisp object (e.g. an array). (this is detailed [below](#).)

2. You use **def-foreign-call** to define a C function with return type `:lisp`.
3. In the C program you retrieve and use the registered Lisp value.
4. You call out to a Lisp function and a garbage collection occurs - the Lisp value in C is no longer valid.
5. The C function returns the Lisp value it retrieved earlier.
6. Then on the next garbage collection Lisp dies because of an illegal object reference: the Lisp value returned by the C function no longer points to valid data.

The problem only occurs if a garbage collection happens during the call to the Lisp function. What you should do is to make sure that any Lisp value you return to Lisp or work with within a C function is retrieved only after there is no possibility of calling out to a Lisp function where a garbage collection may occur. To fix the example above so it is safe, you should add another step after step 4:

- 4a. Retrieve the registered Lisp value again.

Other scenarios can be played out. For example, if C code changes array data using an invalid array pointer, Lisp will never see the changes, and Lisp's data space may be corrupted by the indirection through a bad pointer.

For purposes of allowing call-backs from foreign code, Allegro CL maintains two tables of Lisp objects: one is the function table and the other is the value table. The Lisp program can register functions or values by requesting that they be stored in the respective table. The size of these tables will grow dynamically as needed (in contrast to earlier releases where the function table could not be increased in size). The use of these tables is explained in the following two sections.

8.1 Accessing Lisp values from C: `lisp_value()`

Before accessing a Lisp value from C, it should be registered first. When a Lisp value is registered, an index is returned as a 'handle' on the Lisp object. A C function is provided that will return a pointer to the Lisp object given its index. This is preferable to passing addresses of Lisp objects to C functions, since Lisp objects will generally move with each garbage collection. If a garbage collection is triggered from C code - by calling back from C to Lisp - the addresses of Lisp objects originally passed to the C function from Lisp may become invalid. And since one will have lost one's only handle on these Lisp objects, their addresses, there will be no way to find their new addresses. If instead one were to pass the registration indices of these Lisp objects, one could readily find their new addresses using these indices following a call-back to Lisp.

Note that passing the addresses of Lisp objects to C functions is not recommended only in those cases where a garbage-collection may be triggered by the C code. Passing values of Lisp objects (converted to C values by virtue of declarations made with **def-foreign-call**) is not discouraged. However, some Lisp data types are passed to C as pointers (for example, double-float data). Such data types should be registered and passed to C by their indices if the C code might cause a garbage collection.

The function **register-lisp-value** registers Lisp values in the value table. The function **unregister-lisp-value** clears the registration.

Once a value is registered, the C program can obtain the value from the value table with the C function:

```
long lisp_value(index)
    int index;
```

where `index` is the index of the registered value in the value table in Lisp. This C function will always return the current value at `index` even after a garbage collection has occurred. The result value from `lisp_value()` will be a C pointer to a Lisp object. Macros are provided to help C analyze the Lisp object and convert it to something meaningful. These macros are found in the C include file `lisp.h`, usually distributed in the `home/misc/` directory with Allegro CL.

Note that when one passes Lisp values to foreign functions that have been declared using **def-foreign-call**, most Lisp data types are converted to corresponding C data types automatically. When one obtains Lisp values by calling **lisp_value()**, the conversion must be performed explicitly by the foreign code.

The Lisp function **lisp-value** may be useful for debugging code. It simulates the C function **lisp_value()**, but it may be called from within Lisp at any time.

8.2 Calling Lisp functions from C: `lisp_call_address()` and `lisp_call()`

A Lisp function that is to be called from C code must satisfy two requirements: (1) it must be registered, and (2) it must be defined using the special macro **defun-foreign-callable**.

When a Lisp function is registered, an index is returned as a 'handle' on the Lisp function. A C function is provided that accesses the table and returns the address associated with the index. That address is a valid C function pointer.

Registering a function and going through a table to find the function address is necessary for two reasons: (1) Lisp functions (and their code vectors) will generally move with each garbage collection, and (2) Lisp functions observe different calling conventions than C functions. Conceptually, when a Lisp function is registered, a small 'wrapper' is created that can be called from C and which will set up the arguments correctly for the Lisp function. This wrapper calls the Lisp function with a single argument, a descriptor that points to the arguments stacked by C. The **defun-foreign-callable** macro provides a convenient mechanism for associating formal Lisp arguments with the C arguments.

The function **register-foreign-callable** registers Lisp functions in the function table. The function **unregister-foreign-callable** clears the entry from the table.

The C program can get a pointer to the registered Lisp function by using the C function **lisp_call_address()**. The form is

```
void *lisp_call_address(int index)
```

where `index` is the index of the registered function in the function table.

A C program can call a Lisp function using the syntax `(*f)(arg1, arg2, ..., argn)`, where `f` is the integer returned by `lisp_call_address()`. It is important to realize the `address' of a Lisp function returned by `lisp_call_address()` is not the same as the address of the Lisp function object associated with the Lisp symbol. It is not possible to call a Lisp function directly from C, one must always use the `wrapper' provided by **register-foreign-callable**.

See the various appendices ([Appendix A Foreign Functions on Windows](#), [Appendix B Building shared libraries on Solaris](#), [Appendix C Building shared libraries on HP-UX 11.0](#), [Appendix D Building shared libraries on Compaq Tru64 4.0 or later](#), [Appendix E Building shared libraries on AIX 4.2 or later](#), [Appendix F Building shared libraries on Linux](#), [Appendix G Building shared libraries on FreeBSD](#), [Appendix H Building shared libraries on Mac OS X](#)) for information on creating Shared Object/Library/DLL etc. files suitable for including compiled foreign code in Lisp and also see [Section 1.9 Creating Shared Objects that refer to Allegro CL Functionality](#).

For example, say we have loaded the compiled C file:

```
void c_calls_lisp(fun, index)
    long (*fun)();
{
    void (*lisp_call)(), *lisp_call_address();
    (*fun)();
    /* direct call to lisp function */
    lisp_call = lisp_call_address(index);
    (*lisp_call)();
    /* call to lisp function using index */
}
```

and had the following session in Lisp:

```
USER(70): (setq called 0)
0
USER(71): (defun-foreign-callable lisp-talks ()
           (format t "This is lisp called for ~
                    the ~:r time.~%"
                   (setq called (1+ called))))
lisp-talks
USER(72): (progn
           (multiple-value-setq (ptr index prev-ptr)
                                (register-foreign-callable 'lisp-talks))
           (list ptr index prev-ptr))
(1404302 0 nil)
;; ptr is 1404302, index 0 in
;; function table, previous function none
```

```

USER(73): (ff:def-foreign-call c_calls_lisp (integer fixnum) :returning :
void)
t
USER(74): (c_calls_lisp ptr index)
This is Lisp called for the first time.
This is Lisp called for the second time.

```

Note: using `lisp_call_address()` is recommended over passing the address from Lisp to C because that address may become invalid after a Lisp image is dumped with `dumplisp`. `lisp_call_address()` is very efficient and calling it just before calling the Lisp function from C ensures that the function pointer is valid.

The C representation and the Lisp representation of data types are not necessarily the same. When a C function calls a Lisp function, the Lisp function needs to have its arguments declared so that it 'knows' what the C arguments were and how to convert them. This declaration scheme is implemented in a macro **defun-foreign-callable**.

We give an example: say that we define the following C function, compile it and load it into Lisp:

```

void add(x, y, index)
int x, y, index;
{
    void (*lisp_call)(), *lisp_call_address();
    lisp_call = lisp_call_address(index);
    (*lisp_call)(x, y);
}

```

Then the following Lisp session could take place:

```

USER(80): (ff:def-foreign-call add (integer integer fixnum)
          :returning :void)
t
USER(81): (defun-foreign-callable add-two-c-args
          ((x :signed-long) (y :signed-long))
          (setq xy (+ x y)))
;; set a global variable
;; to the sum of x and y
add-two-c-args
USER(82): (setq index (cadr (multiple-value-list
                             (register-foreign-callable
                              'add-two-c-args))))
1
USER(83): (add 4 5 index) ;; call to the foreign function
nil
USER(84): xy
;; test the value of the
;; global variable xy

```

Note that in the example above, we get exactly the same result by omitting the type declarations for the function `add-two-c-args` - i.e. we could have defined it as:

```
(defun-c-callable add-two-c-args (x y)
  (setq xy (+ x y)))
```

since the default is to assume the arguments are signed-longs.

8.3 Calling foreign callables from Lisp

A foreign-callable (defined via **defun-foreign-callable** or the deprecated **defun-c-callable**) is meant to be called by a foreign function. The arguments are converted explicitly by the foreign-callable from foreign argument types to Lisp types, and the return value is possibly converted from Lisp to an appropriate foreign type, if specified. However, there may be times when such code needs to be debugged, and the user wishes to call the foreign-callable directly from Lisp for such purposes.

The function called **lisp-call**, which existed in some earlier releases, has been removed, and instead, the user can simply call the foreign-callable directly from Lisp. The foreign-callable now uses a dual-entry-point technique, which allows the Lisp call to be caught and the arguments pre-converted before making the "real" call to the body of the foreign-callable.

It is important to understand the pre-conversion technique used in a foreign-callable Lisp call; it depends not on the declared arguments in the foreign-callable, but in the actual arguments passed to the foreign-callable from Lisp. Lisp calls to foreign callables convert all arguments in the following way:

fixnum	arithmetic shifted to create a 30-bit integer
bignum	reassembled into 32-bit integer, properly sign-extending but truncating any LS Bits beyond 32.
single-float	unboxed to float
double-float	unboxed to double
character	converted to char (within an int) type
other	not converted

A Lisp call to a foreign-callable will never reconvert its return value, because it always returns a Lisp value (for foreign-callables that have been registered to convert their return values, it is actually the callback mechanism that performs the conversion, and not the foreign-callable itself).

Note that these conversions are not as extensive as the general conversions done in foreign calls; this mechanism is intended as only a quick debug tool. For more extensive callback testing, define an actual foreign function

with explicit argument types which calls back to Lisp.

Appendix A: Foreign Functions on Windows

This section contains notes on foreign functions for the UNIX Allegro CL programmer porting to Windows. We recommend using Microsoft Visual C++ (MSVC++).

The steps to using foreign functions on Windows are:

1. Make a *.dll* of your C files, including:
 - a. declare the correct calling convention for functions externally visible to Lisp,
 - b. prevent C++ name mangling,
 - c. import the symbols you want to use from other *.dlls*,
 - d. export the symbols you want to use from Allegro CL (so they are visible to **get-entry-point**), and
 - e. compile and link the C files properly.
2. Load the *.dll* into Allegro CL.
3. Define the Lisp side of the foreign function linkage using **def-foreign-call** and **defun-foreign-callable**.

DLL file cannot be changed whole it is loaded into a program

Please note that you cannot create a new *.dll* with the same name as a *.dll* file already loaded into Lisp (or loaded into another program). If you have a *.dll* file loaded into lisp, use **unload-foreign-library** to unload it before recreating it and loading it again.

Appendix A.1 Making a .dll

To access C++ functions from Lisp you must ensure C++ name mangling does not occur. The easiest way to do this is to use a file extension of *.c* instead of *.cpp*. If you must use the *.cpp* file extension, then use the `extern "C"` linkage specification, like this:

```
extern "C"
int foo (int a) {
    return (a + 1);
}
```

Import the symbols you need from other libraries by specifying the *.lib* file to the linker. There are two important entry points in *acl[version].dll* (*[version]* changes with each release of Allegro) which users of the foreign function interface might need: `lisp_call_address` and `lisp_value`. To use these entry points from your *.dll*, you must import the symbols using the linker using *acl[version].lib* provided in the Allegro directory.

For example, to compile the example in [Section 8.2 Calling Lisp functions from C: `lisp_call_address\(\)` and `lisp_call\(\)`](#) above, you would need to specify `acl[version].lib` on your `cl` command line, like this (assuming the function `c_calls_lisp` is in the file `foo.c`):

```
cl -D_MT -MD -LD -Fefoo.dll foo.c
  [Allegro CL directory]\acl[version].lib
user32.lib gdi32.lib
kernel32.lib comctl32.lib comdlg32.lib
winmm.lib advapi32.lib msvcrt.lib
```

After evaluating `(load "foo.dll")` in Lisp, the rest of the session in [Section 8.2 Calling Lisp functions from C: `lisp_call_address\(\)` and `lisp_call\(\)`](#) is the same.

The general issue of cross linking is discussed in [Section 1.9 Creating Shared Objects that refer to Allegro CL Functionality](#) above.

Export the symbols you want to be visible from Lisp by using a linker `.def` file, or by using the `_declspec (dllexport)` declaration:

```
extern "C" _declspec(dllexport)
int foo (int a) {
    return (a + 1);
}
```

Lastly, compile and link your C code into a `.dll`:

- use the `-D_MT` C compiler option to compile your C code to insure the compilation will produce multi-threaded safe C code,
- use the `-MD` linker option to link your object files to insure you link with the multi-threaded safe C runtime libraries,
- use the `-LD` linker option to produce a `dll` instead of an `exe`, and
- link with libraries `user32.lib`, `gdi32.lib`, `kernel32.lib`, `comctl32.lib`, `comdlg32.lib`, `winmm.lib`, `advapi32.lib`, and `msvcrt.lib` (you may not use external symbols from each of these libraries, but it does no harm to link against them all--it is the complete list of Windows libraries that you might need).

Appendix A.2 Making a Fortran .dll

Open Watcom open source project (<http://www.openwatcom.org/>) FORTRAN needs the following steps to work with Allegro CL. Watcom allows for several calling sequences, but only one style is compatible with the argument and return value passing that Allegro CL uses to be compatible with Windows C++. To achieve this, the `/sc` option must be used to specify stack-based argument passing, and the following pragma must be used (starting in column 1):

```
*$pragma aux floatfunc value [8087]
```

for each *floatfunc* in the source that returns a float value. Our experience is that these pragmas may cause warning messages on other architectures, but will otherwise work without causing errors.

Also, a file with a *.lnk* extension should be created, which specifies the link options; advanced Fortran users may want to specify link options on the command line, however. For our example, we will describe a file called *bar*.

lnk:

```
system nt_dll initinstance terminstance
import calltolisp 'call_lisp.dll'
export floatfunc
export funca
export funcb
file bar
```

Note that exports are necessary for each function that will be accessed from the Lisp side. Also, it is possible to call-back into Lisp from Fortran, though it must be done indirectly through a C function in a specified *.dll* file.

Once the source files are set up, the following commands can be run:

```
wfc386 /bd /ms /sc /fpi87 bar.f
wlink @bar
wlib bar +bar.dll
```

Appendix A.3 The Lisp side of foreign functions

Compatibility

As on UNIX, foreign functions are defined to Lisp using **def-foreign-call**. The older (and now obsolete) **defforeign** interface works. When a foreign function defined with **defforeign** is called directly, it acts (with respect to threads) as if it was declared with the `:release-heap :never` option. (*release-heap* is a keyword argument to **def-foreign-call** but not to **defforeign**.) Use the `:call-direct` argument to **def-foreign-call**, as well as other necessary arguments, to allow direct C calling to be compiled inline by the Lisp compiler.

Windows usually expects callbacks to be declared `:stdcall`. You should check your Windows documentation carefully to verify the required calling convention. From the Lisp side, you will need to use a declaration to **defun-foreign-callable**:

```
(declare (:convention { :c | :stdcall | :method })
         (:unwind value))
```

The `(:unwind value)` declaration says that throwing past this boundary is to be performed by returning *value* (**not** evaluated) to the C caller after arranging when control eventually returns to the Lisp-that-called-C-before-this-callback, the throw will be continued from that point. This effect does not require any special action on the part of the Lisp-to-C calling code, except that it had to have been built with a C link (no leaf calls).

Absence of an `:unwind` declaration is equivalent to having `(declare (:unwind 0))`.

Either `:c` or `:stdcall` can now be used interchangeably as the value of the *convention* on **def-foreign-call** definitions. The stack will be properly updated, regardless of the style the foreign function expects to be called with. Callbacks however, must be declared with the proper convention.

Appendix A.4 A complete example

```
C:\TMP>type foo.c
#ifdef _WIN32
#define DllExport __declspec(dllexport)
#else
#define DllExport
#endif
```

```
DllExport int test_fun(int foo)
{
    return foo + 101;
}
```

Compile *foo.c* into *foo.dll*:

```
C:\TMP>cl -D_MT -MD -nologo -LD -Zi -W3 -Fefoo.dll foo.c user32.lib
gdi32.lib kernel32.lib comctl32.lib comdlg32.lib winmm.lib
advapi32.lib msvcrt.lib
```

```
foo.c
    Creating library foo.lib and object foo.exp
```

Then, in Lisp:

```
user(1): (load "foo.dll")
; Foreign loading foo.dll.
t
user(2): (ff:def-foreign-call (test "test_fun") (a))
t
user(3): (test 10)
111
```

```
user(4):
```

or, using the old **ff:defforeign** interface:

```
user(1): (load "foo.dll")
; Foreign loading foo.dll.
t
user(2): (ff:defforeign 'test
         :arguments '(integer)
         :entry-point "test_fun"
         :return-type :integer)
t
user(3): (test 10)
111
user(4):
```

Appendix B: Building shared libraries on Solaris

On Solaris, you must produce *.so* files which are loadable into Allegro CL. The *-K pic* flag is optional (only on Solaris), and creates slightly different modes of sharing when used. As an example:

```
% cc -c -K pic foo.c
% ld -G -o foo.so foo.o
```

Fortran is similar:

```
% f77 -c bar.f
% ld -G -o bar.so bar.o
```

It is often useful to use the *-z defs* option to find which references are not yet resolved. If an undefined symbol is detected that should be present in the Lisp (such as `lisp_call_address`) then that shared-library must be included in the command line. For example, for a call to `lisp_call_address()` in *libacl80.so*:

```
% ld -G -o foo.so foo.o libacl80.so
```

The general issue of cross linking is discussed in [Section 1.9 Creating Shared Objects that refer to Allegro CL Functionality](#) above.

lisp.h is an include file that describes the format of Allegro CL Lisp objects. Because the Sparc has two ports, 32-bit and 64-bit, one of the flags `-DAcl32Bit` or `-DAcl64Bit` must be passed into the C compiler if *lisp.h* is used.

Further, for the 64-bit version, you must specify `-xarch=v9` on the `cc` line for sparc 64-bit and `-xarch=amd64` for amd64/x86-64/emt32. You must also link with 64-bit libraries, which is usually done with `-L/usr/lib/sparcv9`.

Appendix C: Building shared libraries on HP-UX 11.0

The shared objects loadable by Allegro CL are `.sl` files. You make these by using `cc` and `ld` in this way (the instructions are different for 32-bit and 64-bit platforms):

32-bit

```
% cc +z -Ae +DA1.1 -c foo.c
% ld -b -o foo.sl foo.o
```

You can also use `make_shared` (in the `bin/` subdirectory of the Allegro CL distribution) in place of `ld`:

```
% bin/make_shared -o foo.sl foo.o
```

and then load the shared library into Allegro CL:

```
(load "foo.sl")
```

For Fortran, the commands are:

```
% f77 +z +DA1.1 -c bar.f
% ld -b -o bar.sl bar.o
```

64-bit

```
% cc +Z +DA2.0W -c foo.c
```

or

```
% aCC +Z +DA2.0W -c foo.c
```

Note the capital `Z`'s in the above calls. The call to `ld` is then:

```
% ld -b +s -o foo.so foo.o
```

You can also use `make_shared` (in the `bin/` subdirectory of the Allegro CL distribution) in place of `ld`:

```
% bin/make_shared -o foo.so foo.o
```

and then load the shared library into Allegro CL:

```
(load "foo.so")
```

Fortran (f90) is not yet supported on hp 64-bit.

See also the discussion of cross linking in [Section 1.9 Creating Shared Objects that refer to Allegro CL Functionality](#) above.

Note about compiling when including lisp.h

lisp.h is an include file that describes the format of Allegro CL Lisp objects. Because the HP has two ports, 32-bit and 64-bit, one of the flags `-DAcl32Bit` or `-DAcl64Bit` must be passed into the C compiler if *lisp.h* is used.

Appendix D: Building shared libraries on Compaq Tru64 4.0 or later

For 32-bit Lisps, you must use the proper *taso* options to the C compiler. For 64-bit Lisps, do not specify a *taso* option. For both 32-bit and 64-bit Lisps, you should also use the **make_shared** script supplied in the *bin* directory of the Allegro CL distribution, to produce *.so* files which are loadable into Allegro CL. You may wish to put *[Allegro directory]/bin* in your path, but **make_shared** does not itself call additional programs in that directory. For example:

```
;; 32-bit Lisp
```

```
% cc -c -G 0 -taso -xtaso -xtaso_short foo.c
% bin/make_shared -o foo.so foo.o
```

and for Fortran:

```
% f77 -c -taso bar.f
% bin/make_shared -o bar.so bar.o -lm
```

```
;; 64-bit Lisp
```

```
% cc -c -G 0 foo.c
% bin/make_shared -o foo.so foo.o
```

and for Fortran:

```
% f77 -c bar.f
% bin/make_shared -o bar.so bar.o -lm
```

See also the discussion of cross linking in [Section 1.9 Creating Shared Objects that refer to Allegro CL Functionality](#) above.

Note about compiling when including `lisp.h`

`lisp.h` is an include file that describes the format of Allegro CL Lisp objects. Because the Alpha has two ports, 32-bit and 64-bit, one of the flags `-DAc132Bit` or `-DAc164Bit` must be passed into the C compiler if `lisp.h` is used.

Appendix E: Building shared libraries on AIX 4.2 or later

You must use the `make_shared` script supplied in the `bin` directory of the Allegro CL distribution to produce `.so` files which are loadable into Allegro CL.

Because `make_shared` itself calls other things in `bin/`, `[Allegro directory]/bin` must be in your path, that is it must be in the list that is the value of the `PATH` environment variable.

32-bit

```
% cc -c -D_BSD -D_NO_PROTO -D_NONSTD_TYPES -D_MBI=void foo.c
% bin/make_shared -o foo.so foo.o
```

and for Fortran:

```
% xlf -c bar.f
% bin/make_shared -o bar.so bar.o -lm
```

Note about compiling when including `lisp.h` in a 32-bit image

`lisp.h` is an include file that describes the format of Allegro CL Lisp objects. In general, when there are both 32-bit and 64-bit Lisps, one of the flags `-DAc132Bit` or `-DAc164Bit` should be passed into the C compiler if `lisp.h` is used. Use `-DAc132Bit` for the 32-bit Lisp and `-DAc164Bit` for the 64-bit.

See also the discussion of cross linking in [Section 1.9 Creating Shared Objects that refer to Allegro CL Functionality](#) above.

64-bit

The instructions for building a shared library for a 64-bit Lisp are similar to those given just above, but the additional argument `-q64` should be added to the `cc` or `xlf` commands when producing shared libraries for the 64-bit version (the `make_shared` calls are unchanged):

```
% cc -q64 -c -D_BSD -D_NO_PROTO -D_NONSTD_TYPES -D_MBI=void foo.c
% bin/make_shared -o foo.so foo.o
```

and for Fortran:

```
% xlf -q64 -c bar.f
% bin/make_shared -o bar.so bar.o -lm
```

Note about compiling when including `lisp.h` in a 64-bit image

`lisp.h` is an include file that describes the format of Allegro CL Lisp objects. Because there are two AIX ports, 32-bit and 64-bit, one of the flags `-DAcl32Bit` or `-DAcl64Bit` must be passed into the C compiler if `lisp.h` is used. Use `-DAcl64Bit` for the 64-bit Lisp.

Again, see also the discussion of cross linking in [Section 1.9 Creating Shared Objects that refer to Allegro CL Functionality](#) above.

Appendix F: Building shared libraries on Linux

On Linux (either on machines with Intel or AMD processors or machines with PowerPC processors), you must produce `.so` files which are loadable into Allegro CL. Compile with the `-fPIC` flag. As an example:

```
% cc -c -fPIC foo.c -o foo.o
% ld -shared -o foo.so foo.o
```

Fortran is similar:

```
% f77 -c bar.f
% ld -shared -o bar.so bar.o
```

The general issue of cross linking is discussed in [Section 1.9 Creating Shared Objects that refer to Allegro CL Functionality](#) above.

Further, for the 64-bit version, you must specify `-m64` on the `cc` line.

Appendix G: Building shared libraries on FreeBSD

On FreeBSD, you must produce `.so` files which are loadable into Allegro CL. Compile with the `-fPIC -DPIC` flags. As an example:

```
% cc -c -fPIC -DPIC foo.c -o foo.o
% ld -Bshareable -Bdynamic -o foo.so foo.o
```

Fortran is similar:

```
% f77 -c bar.f
% ld -Bshareable -Bdynamic -o bar.so bar.o
```

The general issue of cross linking is discussed in [Section 1.9 Creating Shared Objects that refer to Allegro CL Functionality](#) above.

Appendix H: Building shared libraries on Mac OS X

On Mac OS X, you must produce a specific type of `.dylib` file which can be loaded into Allegro CL. These are called bundle files on Mac OS X, and are fully packaged shared libraries. Unfortunately, they may not be reused as input to `ld()` once they have been created, in contrast with shared-objects on other architectures. However, they may contain undefined symbols, which may be resolved lazily when the shared-objects are loaded. These may include symbols in `libacl*.dylib` such as `lisp_value` and `lisp_call_address`, without having to link against the library. Compile with the `-dynamic` flag. As an example:

```
% cc -dynamic -c foo.c -o foo.o
% ld -bundle /usr/lib/bundle1.o -flat_namespace -undefined suppress -o foo.
dylib foo.o
```

Fortran interfacing is not supported at this time.

The general issue of cross linking is discussed in [Section 1.9.1 Linking to Allegro CL shared library on Mac OS X](#).

`lisp.h` is an include file that describes the format of Allegro CL Lisp objects. Because the Mac OS X has two ports, 32-bit and 64-bit, one of the flags `-DAcl32Bit` or `-DAcl64Bit` must be passed into the C compiler if `lisp.h` is used.

Further, for the 64-bit version, you must specify `-arch ppc64` on the `cc` line.