# *UFFI* Reference Guide

**Kevin M. Rosenberg**
**Heart Hospital of New Mexico**

**kevin@rosenberg.net**
**504 Elm Street N.E.**
**Albuquerque**
**New Mexico**
**87102**

***UFFI* Reference Guide**
by Kevin M. Rosenberg

$Id: bookinfo.sgml,v 1.2 2002/03/14 16:53:27 kevin Exp $

File $Date: 2002/03/14 16:53:27 $

# Table of Contents

# Preface

This reference guide describes the usage and features of *UFFI*. The first chapter provides an overview to the design of *UFFI*. Following that chapter is the reference section for all user accessible functions of *UFFI*. The appendix covers the installation and implementation-specifc features of *UFFI*.

# Chapter 1. Introduction

## Purpose

This reference guide describes *UFFI*, a package that provides a cross-implementation interface from Common Lisp to C-language compatible libraries.

## Background

Every Common Lisp implementation has a method for interfacing to C-language compatible libraries. These methods are often termed a *Foreign Function Library Interface* (FFI). Unfortunately, these methods vary widely amongst implementations, thus preventing the writing of a portable FFI to a particular C-library.

*UFFI* gathers a common subset of functionality between Common Lisp implementations. *UFFI* wraps this common subset of functionality with it's own syntax and provides macro translation of uffi functions into the specific syntax of supported Common Lisp implementations.

Developers who use *UFFI* to interface with C libraries will automatically have their code function in each of uffi's supported implementations.

## Supported Implementations

The primary tested and supported platforms for *UFFI* are:

- AllegroCL v6.2 on Debian GNU/Linux FreeBSD 4.5, Solaris v2.8, and Microsoft Windows XP.
- Lispworks v4.2 on Debian GNU/Linux and Microsoft Windows XP.
- CMUCL 18d on Debian GNU/Linux, FreeBSD 4.5, and Solaris 2.8
- SBCL 0.7.8 on Debian GNU/Linux
- SCL 1.1.1 on Debian GNU/Linux
- OpenMCL 0.13 on Debian GNU/Linux for PowerPC

Beta code is included with *UFFI* for

- OpenMCL and MCL with MacOSX

## Design

### Overview

*UFFI* was designed as a cross-implementation compatible *Foreign Function Interface*. Necessarily, only a common subset of functionality can be provided. Likewise, not every optimization for that a specific implementation provides can be supported. Wherever possible, though, implementation-specific optimizations are invoked.

## Priorities

The design of *UFFI* is dictated by the order of these priorities:

- Code using *UFFI* must operate correctly on all supported implementations.
- Take advantage of implementation-specific optimizations. Ideally, there will not a situation where an implementation-specific FFI will be chosen due to lack of optimizations in *UFFI*.
- Provide a simple interface to developers using *UFFI*. This priority is quite a bit lower than the above priorities. This lower priority is manifest by programmers having to pass types in pointer and array dereferencing, needing to use `cstring` wrapper functions, and the use of ensure-char-character and ensure-char-integer functions. My hope is that the developer inconvenience will be outweighed by the generation of optimized code that is cross-implementation compatible.

# Chapter 2. Programming Notes

## Implementation Specific Notes

### AllegroCL

### Lispworks

### CMUCL

## Foreign Object Representation and Access

There are two main approaches used to represent foreign objects: an integer that represents an address in memory, and a object that also includes run-time typing. The advantage of run-time typing is the system can dereference pointers and perform array access without those functions requiring a type at the cost of additional overhead to generate and store the run-time typing. The advantage of integer representation, at least for AllegroCL, is that the compiler can generate inline code to dereference pointers. Further, the overhead of the run-time type information is eliminated. The disadvantage is the program must then supply the type to the functions to dereference objects and array.

## Optimizing Code Using UFFI

### Background

Two implementions have different techniques to optimize (open-code) foreign objects. AllegroCL can open-code foreign object access if pointers are integers and the type of object is specified in the access function. Thus, *UFFI* represents objects in AllegroCL as integers which don't have type information.

CMUCL works best when keeping objects as typed objects. However, it's compiler can open-code object access when the object type is specified in `declare` commands and in `:type` specifiers in `defstruct` and `defclass`.

Lispworks, in converse to AllegroCL and CMUCL does not do any open coding of object access. Lispworks, by default, maintains objects with run-time typing.

## Cross-Implementation Optimization

To fully optimize across platforms, both explicit type information must be passed to dereferencing of pointers and arrays. Though this optimization only helps with AllegroCL, *UFFI* is designed to require this type information be passed the dereference functions. Second, declarations of type should be made in functions, structures, and classes where foreign objects will be help. This will optimize access for Lispworks

Here is an example that should both methods being used for maximum cross-implementation optimization:

```
(uffi:def-type the-struct-type-def the-struct-type)
(let ((a-foreign-struct (allocate-foreign-object 'the-struct-type)))
  (declare 'the-struct-type-def a-foreign-struct)
  (get-slot-value a-foreign-struct 'the-struct-type 'field-name))
```

# I. Declarations

## Overview

Declarations are used to give the compiler optimizing information about foreign types. Currently, only CMUCL supports declarations. On AllegroCL and Lispworks, these expressions declare the type generically as `T`

# def-type

## Name

`def-type` — Defines a Common Lisp type.

Macro

## Syntax

```
def-type name type
```

## Arguments and Values

*name*

A symbol naming the type

*type*

A form that is evaluated that specifies the *UFFI* type.

## Description

Defines a Common Lisp type based on a *UFFI* type.

## Examples

```
(def-type char-ptr '(* :char))
...
(defun foo (ptr)
  (declare (type char-ptr ptr))
  ...
```

## Side Effects

Defines a new ANSI Common Lisp type.

## Affected by

None.

## Exceptional Situations

None.

# II. Primitive Types

## Overview

Primitive types have a single value, these include characters, numbers, and pointers. They are all symbols in the keyword package.

- `:char` - Signed 8-bits. A dereferenced :char pointer returns an character.
- `:unsigned-char` - Unsigned 8-bits. A dereferenced :unsigned-char pointer returns an character.
- `:byte` - Signed 8-bits. A dereferenced :byte pointer returns an integer.
- `:unsigned-byte` - Unsigned 8-bits. A dereferenced :unsigned-byte pointer returns an integer.
- `:short` - Signed 16-bits.
- `:unsigned-short` - Unsigned 16-bits.
- `:int` - Signed 32-bits.
- `:unsigned-int` - Unsigned 32-bits.
- `:long` - Signed 32-bits.
- `:unsigned-long` - Unsigned 32-bits.
- `:float` - 32-bit floating point.
- `:double` - 64-bit floating point.
- `:cstring` - A `NULL` terminated string used for passing and returning characters strings with a `C` function.
- `:void` - The absence of a value. Used to indicate that a function does not return a value.
- `:pointer-void` - Points to a generic object.
- `*` - Used to declare a pointer to an object

# def-constant

## Name

`def-constant` — Binds a symbol to a constant.

Macro

## Syntax

```
def-constant name value &key export
```

## Arguments and Values

*`name`*

A symbol that will be bound to the value.

*`value`*

An evaluated form that is bound the the name.

*`export`*

When `T`, the name is exported from the current package. The default is `NIL`

## Description

This is a thin wrapper around `defconstant`. It evaluates at compile-time and optionally exports the symbol from the package.

## Examples

```
(def-constant pi2 (* 2 pi))
(def-constant exported-pi2 (* 2 pi) :export t)
```

## Side Effects

Creates a new special variable..

### Affected by

None.

### Exceptional Situations

None.

## def-foreign-type

### Name

`def-foreign-type` — Defines a new foreign type.

Macro

### Syntax

```
def-foreign-type name type
```

### Arguments and Values

*`name`*

A symbol naming the new foreign type.

*`value`*

A form that is not evaluated that defines the new foreign type.

### Description

Defines a new foreign type.

### Examples

```
(def-foreign-type my-generic-pointer :pointer-void)
(def-foreign-type a-double-float :double-float)
(def-foreign-type char-ptr (* :char))
```

### Side Effects

Defines a new foreign type.

### Affected by

None.

### Exceptional Situations

None.

## null-char-p

### Name

`null-char-p` — Tests a character for `NULL` value.

Macro

### Syntax

```
null-char-p char => is-null
```

### Arguments and Values

*`char`*

A character or integer.

*`is-null`*

A boolean flag indicating if char is a `NULL` value.

### Description

A predicate testing if a character or integer is `NULL`. This abstracts the difference in implementations where some return a `character` and some return a `integer` whence dereferencing a `C` character pointer.

## Examples

```
(def-array-pointer ca :unsigned-char)
(let ((fs (convert-to-foreign-string "ab")))
   (values (null-char-p (deref-array fs 'ca 0))
           (null-char-p (deref-array fs 'ca 2))))
=> NIL
   T
```

## Side Effects

None.

## Affected by

None.

## Exceptional Situations

None.

# III. Aggregate Types

## Overview

Aggregate types are comprised of one or more primitive types.

# def-enum

## Name

`def-enum` — Defines a `C` enumeration.

Macro

## Syntax

```
def-enum name fields &key separator-string
```

## Arguments and Values

*`name`*

A symbol that names the enumeration.

*`fields`*

A list of field defintions. Each definition can be a symbol or a list of two elements. Symbols get assigned a value of the current counter which starts at `0` and increments by `1` for each subsequent symbol. It the field definition is a list, the first position is the symbol and the second position is the value to assign the the symbol. The current counter gets set to `1+` this value.

*`separator-string`*

A string that governs the creation of constants. The default is `"#"`.

## Description

Declares a `C` enumeration. It generates constants with integer values for the elements of the enumeration. The symbols for the these constant values are created by the `concatenation` of the enumeration name, separator-string, and field symbol. Also creates a foreign type with the name *`name`* of type `:int`.

## Examples

```
(def-enum abc (:a :b :c))
;; Creates constants abc#a (1), abc#b (2), abc#c (3) and defines
;; the foreign type "abc" to be :int

(def-enum efoo (:e1 (:e2 10) :e3) :separator-string "-")
;; Creates constants efoo-e1 (1), efoo-e2 (10), efoo-e3 (11) and defines
;; the foreign type efoo to be :int
```

### Side Effects

Creates a :int foreign type, defines constants.

### Affected by

None.

### Exceptional Situations

None.

## def-struct

### Name

`def-struct` — Defines a `C` structure.

Macro

### Syntax

```
def-struct name &rest fields
```

### Arguments and Values

*`name`*

A symbol that names the structure.

*`fields`*

A variable number of field defintions. Each definition is a list consisting of a symbol naming the field followed by its foreign type.

### Description

Declares a structure. A special type is available as a slot in the field. It is a pointer that points to an instance of the parent structure. It's type is `:pointer-self`.

### Examples

```
(def-struct foo (a :unsigned-int)
                (b (* :char))
                (c (:array :int 10))
                (next :pointer-self))
```

### Side Effects

Creates a foreign type.

### Affected by

None.

### Exceptional Situations

None.

## get-slot-value

### Name

`get-slot-value` — Retrieves a value from a slot of a structure.

Macro

### Syntax

```
get-slot-value obj type field => value
```

### Arguments and Values

*`obj`*

A pointer to foreign structure.

*`type`*

A name of the foreign structure.

*`field`*

A name of the desired field in foreign structure.

*`value`*

The value of the field in the structure.

### Description

Accesses a slot value from a structure.

### Examples

```
(get-slot-value foo-ptr 'foo-structure 'field-name)
```

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## get-slot-pointer

### Name

`get-slot-pointer` — Retrieves a pointer from a slot of a structure.

Macro

### Syntax

`get-slot-pointer` *`obj type field => pointer`*

### Arguments and Values

*obj*

A pointer to foreign structure.

*type*

A name of the foreign structure.

*field*

A name of the desired field in foreign structure.

*pointer*

The value of the field in the structure.

### Description

This is similar to `get-slot-value`. It is used when the value of a slot is a pointer type.

### Examples

```
(get-slot-pointer foo-ptr 'foo-structure 'my-char-ptr)
```

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## def-array-pointer

### Name

`def-array-pointer` — Defines a pointer to a array of type.

Macro

### Syntax

```
def-array-pointer name type
```

### Arguments and Values

*name*

A name of the new foreign type.

*type*

The foreign type of the array elements.

### Description

Defines a type tat is a pointer to an array of type.

### Examples

```
(def-array-pointer byte-array-pointer :unsigned-char)
```

### Side Effects

Defines a new foreign type.

### Affected by

None.

### Exceptional Situations

None.

## deref-array

### Name

`deref-array` — Deference an array.

Macro

## Syntax

```
deref-array array type positon => value
```

## Arguments and Values

*array*

A foreign array.

*type*

The foreign type of the array.

*position*

An integer specifying the position to retrieve from the array.

*value*

The value stored in the position of the array.

## Description

Dereferences (retrieves) the value of an array element.

## Examples

```
(def-array-pointer ca :char)
(let ((fs (convert-to-foreign-string "ab")))
   (values (null-char-p (deref-array fs 'ca 0))
           (null-char-p (deref-array fs 'ca 2))))
=> NIL
   T
```

## Notes

The TYPE argument is ignored for CL implementations other than AllegroCL. If you want to cast a pointer to another type use WITH-CAST-POINTER together with DEREF-POINTER/DEREF-ARRAY.

## Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## def-union

### Name

`def-union` — Defines a foreign union type.

Macro

### Syntax

```
def-union name &rest fields
```

### Arguments and Values

*`name`*

A name of the new union type.

*`fields`*

A list of fields of the union.

### Description

Defines a foreign union type.

### Examples

```
(def-union test-union
  (a-char :char)
  (an-int :int))

(let ((u (allocate-foreign-object 'test-union))
  (setf (get-slot-value u 'test-union 'an-int) (+ 65 (* 66 256)))
  (prog1
    (ensure-char-character (get-slot-value u 'test-union 'a-char))
```

```
    (free-foreign-object u)))
=> #\A
```

## Side Effects

Defines a new foreign type.

## Affected by

None.

## Exceptional Situations

None.

# IV. Objects

## Overview

Objects are entities that can allocated, referred to by pointers, and can be freed.

# allocate-foreign-object

## Name

`allocate-foreign-object` — Allocates an instance of a foreign object.

Macro

## Syntax

```
allocate-foreign-object type &optional size => ptr
```

## Arguments and Values

*`type`*

The type of foreign object to allocate. This parameter is evaluated.

*`size`*

An optional size parameter that is evaluated. If specified, allocates and returns an array of *`type`* that is *`size`* members long. This parameter is evaluated.

*`ptr`*

A pointer to the foreign object.

## Description

Allocates an instance of a foreign object. It returns a pointer to the object.

## Examples

```
(def-struct ab (a :int) (b :double))
(allocate-foreign-object 'ab)
=> #<ptr>
```

## Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## free-foreign-object

### Name

`free-foreign-object` — Frees memory that was allocated for a foreign boject.

Macro

### Syntax

```
free-foreign-object ptr
```

### Arguments and Values

*`ptr`*

A pointer to the allocated foreign object to free.

### Description

Frees the memory used by the allocation of a foreign object.

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

# with-foreign-object

### Name

`with-foreign-object` — Wraps the allocation of a foreign object around a body of code.

Macro

### Syntax

```
with-foreign-object (var type) &body body => form-return
```

### Arguments and Values

*`var`*

The variable name to bind.

*`type`*

The type of foreign object to allocate. This parameter is evaluated.

*`form-return`*

The result of evaluating the *`body`*.

### Description

This function wraps the allocation, binding, and destruction of a foreign object. On CMUCL and Lispworks platforms the object is stack allocated for efficiency. Benchmarks show that AllegroCL performs much better with static allocation.

### Examples

```
(defun gethostname2 ()
  "Returns the hostname"
  (uffi:with-foreign-object (name '(:array :unsigned-char 256))
    (if (zerop (c-gethostname (uffi:char-array-to-pointer name) 256))
(uffi:convert-from-foreign-string name)
(error "gethostname() failed."))))
```

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## size-of-foreign-type

### Name

`size-of-foreign-type` — Returns the number of data bytes used by a foreign object type.

Macro

### Syntax

```
size-of-foreign-type ftype
```

### Arguments and Values

*`ftype`*

A foreign type specifier. This parameter is evaluated.

### Description

Returns the number of data bytes used by a foreign object type. This does not include any Lisp storage overhead.

### Examples

```
(size-of-foreign-object :unsigned-byte)
=> 1
(size-of-foreign-object 'my-100-byte-vector-type)
=> 100
```

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## pointer-address

### Name

`pointer-address` — Returns the address of a pointer.

Macro

### Syntax

```
pointer-address ptr => address
```

### Arguments and Values

*`ptr`*

A pointer to a foreign object.

*`address`*

An integer representing the pointer’s address.

### Description

Returns the address as an integer of a pointer.

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## deref-pointer

### Name

`deref-pointer` — Deferences a pointer.

Macro

### Syntax

```
deref-pointer ptr type => value
```

### Arguments and Values

*`ptr`*

A pointer to a foreign object.

*`type`*

A foreign type of the object being pointed to.

*`value`*

The value of the object where the pointer points.

### Description

Returns the object to which a pointer points.

### Examples

```
(let ((intp (allocate-foreign-object :int)))
  (setf (deref-pointer intp :int) 10)
  (prog1
    (deref-pointer intp :int)
    (free-foreign-object intp)))
=> 10
```

### Notes

The TYPE argument is ignored for CL implementations other than AllegroCL. If you want to cast a pointer to another type use WITH-CAST-POINTER together with DEREF-POINTER/DEREF-ARRAY.

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## ensure-char-character

### Name

`ensure-char-character` — Ensures that a dereferenced `:char` pointer is a character.

Macro

### Syntax

```
ensure-char-character object => char
```

## Arguments and Values

*`object`*

Either a character or a integer specifying a character code.

*`char`*

A character.

## Description

Ensures that an objects obtained by dereferencing `:char` and `:unsigned-char` pointers are a lisp character.

## Examples

```
(let ((fs (convert-to-foreign-string "a")))
  (prog1
    (ensure-char-character (deref-pointer fs :char))
    (free-foreign-object fs)))
=> #\a
```

## Side Effects

None.

## Affected by

None.

## Exceptional Situations

Depending upon the implementation and what *UFFI* expects, this macro may signal an error if the object is not a character or integer.

# ensure-char-integer

## Name

`ensure-char-integer` — Ensures that a dereferenced `:char` pointer is an integer.

Macro

## Syntax

```
ensure-char-integer object => int
```

## Arguments and Values

*`object`*

Either a character or a integer specifying a character code.

*`int`*

An integer.

## Description

Ensures that an object obtained by dereferencing a `:char` pointer is an integer.

## Examples

```
(let ((fs (convert-to-foreign-string "a")))
  (prog1
    (ensure-char-integer (deref-pointer fs :char))
    (free-foreign-object fs)))
=> 96
```

## Side Effects

None.

## Affected by

None.

### Exceptional Situations

Depending upon the implementation and what *UFFI* expects, this macro may signal an error if the object is not a character or integer.

## make-null-pointer

### Name

`make-null-pointer` — Create a `NULL` pointer.

Macro

### Syntax

```
make-null-pointer type => ptr
```

### Arguments and Values

*`type`*

A type of object to which the pointer refers.

*`ptr`*

The `NULL` pointer of type *`type`*.

### Description

Creates a `NULL` pointer of a specified type.

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## null-pointer-p

### Name

`null-pointer-p` — Tests a pointer for `NULL` value.

Macro

### Syntax

```
null-pointer-p ptr => is-null
```

### Arguments and Values

*`ptr`*

A foreign object pointer.

*`is-null`*

The boolean flag.

### Description

A predicate testing if a pointer is has a `NULL` value.

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

# +null-cstring-pointer+

## Name

`+null-cstring-pointer+` — A constant `NULL` cstring pointer.

Constant

## Description

A `NULL` cstring pointer. This can be used for testing if a cstring returned by a function is `NULL`.

# with-cast-pointer

## Name

`with-cast-pointer` — Wraps a body of code with a pointer cast to a new type.

Macro

## Syntax

```
with-cast-pointer (binding-name ptr type) & body body => value
```

## Arguments and Values

*`ptr`*

A pointer to a foreign object.

*`type`*

A foreign type of the object being pointed to.

*`value`*

The value of the object where the pointer points.

## Description

Executes BODY with POINTER cast to be a pointer to type TYPE. If BINDING-NAME is provided the cast pointer will be bound to this name during the execution of BODY. If BINDING-NAME is not provided POINTER must be a name bound to the pointer which should be cast. This name will be bound to the cast pointer during the execution of BODY. This is a no-op in AllegroCL but will wrap BODY in a LET form if BINDING-NAME is provided. This macro is meant to be used in conjunction with DEREF-POINTER or DEREF-ARRAY. In Allegro CL the "cast" will actually take place in DEREF-POINTER or DEREF-ARRAY.

## Examples

```
(with-foreign-object (size :int)
   ;; FOO is a foreign function returning a :POINTER-VOID
   (let ((memory (foo size)))
      (when (mumble)
         ;; at this point we know for some reason that MEMORY points
         ;; to an array of unsigned bytes
         (with-cast-pointer (memory :unsigned-byte)
           (dotimes (i (deref-pointer size :int))
            (do-something-with
              (deref-array memory '(:array :unsigned-byte) i)))))))
```

## Side Effects

None.

## Affected by

None.

## Exceptional Situations

None.

# def-foreign-var

## Name

`def-foreign-var` — Defines a symbol macro to access a variable in foreign code

Macro

## Syntax

```
def-foreign-var name type module
```

## Arguments and Values

*name*

A string or list specificying the symbol macro's name. If it is a string, that names the foreign variable. A Lisp name is created by translating #_ to #\- and by converting to upper-case in case-insensitive Lisp implementations. If it is a list, the first item is a string specifying the foreign variable name and the second it is a symbol stating the Lisp name.

*type*

A foreign type of the foreign variable.

*module*

A string specifying the module (or library) the foreign variable resides in. (Required by Lispworks)

## Description

Defines a symbol macro which can be used to access (get and set) the value of a variable in foreign code.

## Examples

### C code

```
int baz = 3;

typedef struct {
  int x;
  double y;
} foo_struct;

foo_struct the_struct = { 42, 3.2 };

int foo () {
  return baz;
}
```

### Lisp code

```
(uffi:def-struct foo-struct
  (x :int)
  (y :double))
```

```
  (uffi:def-function ("foo" foo)
      ()
    :returning :int
    :module "foo")

  (uffi:def-foreign-var ("baz" *baz*) :int "foo")
  (uffi:def-foreign-var ("the_struct" *the-struct*) foo-struct "foo")


*baz*
  => 3

(incf *baz*)
  => 4

(foo)
  => 4
```

## Side Effects

None.

## Affected by

None.

## Exceptional Situations

None.

# V. Strings

## Overview

*UFFI* has functions to two types of C-compatible strings: *cstring* and *foreign* strings. cstrings are used *only* as parameters to and from functions. In some implementations a cstring is not a foreign type but rather the Lisp string itself. On other platforms a cstring is a newly allocated foreign vector for storing characters. The following is an example of using cstrings to both send and return a value.

```
(uffi:def-function ("getenv" c-getenv)
    ((name :cstring))
  :returning :cstring)

(defun my-getenv (key)
  "Returns an environment variable, or NIL if it does not exist"
  (check-type key string)
  (uffi:with-cstring (key-native key)
    (uffi:convert-from-cstring (c-getenv key-native))))
```

In contrast, foreign strings are always a foreign vector of characters which have memory allocated. Thus, if you need to allocate memory to hold the return value of a string, you must use a foreign string and not a cstring. The following is an example of using a foreign string for a return value.

```
(uffi:def-function ("gethostname" c-gethostname)
    ((name (* :unsigned-char))
     (len :int))
  :returning :int)

(defun gethostname ()
  "Returns the hostname"
  (let* ((name (uffi:allocate-foreign-string 256))
         (result-code (c-gethostname name 256))
         (hostname (when (zerop result-code)
                     (uffi:convert-from-foreign-string name))))
    (uffi:free-foreign-object name)
    (unless (zerop result-code)
      (error "gethostname() failed."))))
```

Foreign functions that return pointers to freshly allocated strings should in general not return cstrings, but foreign strings. (There is no portable way to release such cstrings from Lisp.) The following is an example of handling such a function.

```
(uffi:def-function ("readline" c-readline)
    ((prompt :cstring))
  :returning (* :char))

(defun readline (prompt)
  "Reads a string from console with line-editing."
  (with-cstring (c-prompt prompt)
      (let* ((c-str (c-readline c-prompt))
             (str (convert-from-foreign-string c-str)))
        (uffi:free-foreign-object c-str)
        str)))
```

# convert-from-cstring

## Name

`convert-from-cstring` — Converts a cstring to a Lisp string.

Macro

## Syntax

```
convert-from-cstring cstring => string
```

## Arguments and Values

*`cstring`*

A cstring.

*`string`*

A Lisp string.

## Description

Converts a Lisp string to a `cstring`. This is most often used when processing the results of a foreign function that returns a cstring.

## Side Effects

None.

## Affected by

None.

## Exceptional Situations

None.

# convert-to-cstring

## Name

`convert-to-cstring` — Converts a Lisp string to a cstring.

Macro

## Syntax

```
convert-to-cstring string => cstring
```

## Arguments and Values

*`string`*

A Lisp string.

*`cstring`*

A cstring.

## Description

Converts a Lisp string to a `cstring`. The `cstring` should be freed with `free-cstring`.

## Side Effects

On some implementations, this function allocates memory.

## Affected by

None.

## Exceptional Situations

None.

# free-cstring

## Name

`free-cstring` — Free memory used by cstring.

Macro

## Syntax

```
free-cstring cstring
```

## Arguments and Values

*`cstring`*

A cstring.

## Description

Frees any memory possibly allocated by `convert-to-cstring`. On some implementions, a cstring is just the Lisp string itself.

## Side Effects

None.

## Affected by

None.

## Exceptional Situations

None.

# with-cstring

## Name

`with-cstring` — Binds a newly created cstring.

Macro

## Syntax

```
with-cstring (cstring string) {body}
```

## Arguments and Values

*`cstring`*

A symbol naming the cstring to be created.

*`string`*

A Lisp string that will be translated to a cstring.

*`body`*

The body of where the cstring will be bound.

## Description

Binds a symbol to a cstring created from conversion of a string. Automatically frees the `cstring`.

## Examples

```
(def-function ("getenv" c-getenv)
   ((name :cstring))
   :returning :cstring)

(defun getenv (key)
  "Returns an environment variable, or NIL if it does not exist"
  (check-type key string)
  (with-cstring (key-cstring key)
    (convert-from-cstring (c-getenv key-cstring))))
```

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## convert-from-foreign-string

### Name

`convert-from-foreign-string` — Converts a foreign string into a Lisp string.

Macro

### Syntax

```
convert-from-foreign-string foreign-string &key length null-terminated-p => string
```

### Arguments and Values

*`foreign-string`*

A foreign string.

*`length`*

The length of the foreign string to convert. The default is the length of the string until a `NULL` character is reached.

*`null-terminated-p`*

A boolean flag with a default value of `T` When true, the string is converted until the first `NULL` character is reached.

*`string`*

A Lisp string.

### Description

Returns a Lisp string from a foreign string. Can translated ASCII and binary strings.

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## convert-to-foreign-string

### Name

`convert-to-foreign-string` — Converts a Lisp string to a foreign string.

Macro

### Syntax

```
convert-to-foreign-string string => foreign-string
```

### Arguments and Values

*`string`*

A Lisp string.

*`foreign-string`*

A foreign string.

### Description

Converts a Lisp string to a foreign string. Memory should be freed with `free-foreign-object`.

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## allocate-foreign-string

### Name

`allocate-foreign-string` — Allocates space for a foreign string.

Macro

### Syntax

```
allocate-foreign-string size &key unsigned => foreign-string
```

### Arguments and Values

*`size`*

The size of the space to be allocated in bytes.

*`unsigned`*

A boolean flag with a default value of `T`. When true, marks the pointer as an `:unsigned-char`.

*`foreign-string`*

A foreign string which has undefined contents.

### Description

Allocates space for a foreign string. Memory should be freed with `free-foreign-object`.

## Side Effects

None.

## Affected by

None.

## Exceptional Situations

None.

# VI. Functions & Libraries

# def-function

## Name

`def-function` — Declares a function.

Macro

## Syntax

```
def-function name args &key module returning
```

## Arguments and Values

*`name`*

A string or list specificying the function name. If it is a string, that names the foreign function. A Lisp name is created by translating #_ to #\- and by converting to upper-case in case-insensitive Lisp implementations. If it is a list, the first item is a string specifying the foreign function name and the second it is a symbol stating the Lisp name.

*`args`*

A list of argument declarations. If `NIL`, indicates that the function does not take any arguments.

*`module`*

A string specifying which module (or library) that the foreign function resides. (Required by Lispworks)

*`returning`*

A declaration specifying the result type of the foreign function. If `:void` indicates module does not return any value.

## Description

Declares a foreign function.

## Examples

```
(def-function "gethostname"
  ((name (* :unsigned-char))
   (len :int))
  :returning :int)
```

### Side Effects

None.

### Affected by

None.

### Exceptional Situations

None.

## load-foreign-library

### Name

`load-foreign-library` — Loads a foreign library.

Function

### Syntax

```
load-foreign-library filename &key module supporting-libraries force-load => success
```

### Arguments and Values

*`filename`*

A string or pathname specifying the library location in the filesystem. At least one implementation (Lispworks) can not accept a logical pathname.

*`module`*

A string designating the name of the module to apply to functions in this library. (Required for Lispworks)

*`supporting-libraries`*

A list of strings naming the libraries required to link the foreign library. (Required by CMUCL)

*`force-load`*

Forces the loading of the library if it has been previously loaded.

*success*

A boolean flag, `T` if the library was able to be loaded successfully or if the library has been previously loaded, otherwise `NIL`.

## Description

Loads a foreign library. Applies a module name to functions within the library. Ensures that a library is only loaded once during a session. A library can be reloaded by using the `:force-load` key.

## Examples

```
(load-foreign-library #p"/usr/lib/libmysqlclient.so"
                      :module "mysql"
                      :supporting-libraries '("c"))
  => T
```

## Side Effects

Loads the foreign code into the Lisp system.

## Affected by

Ability to load the file.

## Exceptional Situations

None.

# find-foreign-library

## Name

`find-foreign-library` — Finds a foreign library file.

Function

## Syntax

```
find-foreign-library names directories & drive-letters types => path
```

## Arguments and Values

*names*

A string or list of strings containing the base name of the library file.

*directories*

A string or list of strings containing the directory the library file.

*drive-letters*

A string or list of strings containing the drive letters for the library file.

*types*

A string or list of strings containing the file type of the library file. Default is `NIL`. If `NIL`, will use a default type based on the currently running implementation.

*path*

A path containing the path found, or `NIL` if the library file was not found.

## Description

Finds a foreign library by searching through a number of possible locations. Returns the path of the first found file.

## Examples

```
(find-foreign-library '("libmysqlclient" "libmysql")
    '("/opt/mysql/lib/mysql/" "/usr/local/lib/" "/usr/lib/" "/mysql/lib/opt/")
    :types '("so" "dll")
    :drive-letters '("C" "D" "E"))
=> #P"D:\\mysql\\lib\\opt\\libmysql.dll"
```

## Side Effects

None.

## Affected by

None.

## Exceptional Situations

None.

# Appendix A. Installation

## Download *UFFI*

You need to download the *UFFI* package from its web *home* (http://uffi.med-info.com). You also need to have a copy of ASDF. If you need a copy of ASDF, it is included in the *CCLAN* (http://www.sourceforge.net/projects/cclan) package. You can download the file `defsystem.lisp` from the CVS *tree* (http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/cclan/asdf/asdf.lisp).

## Installation

After downloading and installing ASDF, simply `push` the directory containing *UFFI* into `asdf:*central-registry*` variable. Whenever you want to load the *UFFI* package, use the function `(asdf:oos 'asdf:load-op :uffi)`.

# Glossary

**Foreign Function Interface FFI)**

An interface to a C-compatible library.