CFFI-SYS Interface Specification

Copyright © 2005-2006, James Bielman <jamesjb at jamesjb.com>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Table of Contents

1 Introduction

CFFI, the Common Foreign Function Interface, purports to be a portable foreign function interface for Common Lisp.

This specification defines a set of low-level primitives that must be defined for each Lisp implementation supported by CFFI. These operators are defined in the `CFFI-SYS` package.

The `CFFI` package uses the `CFFI-SYS` interface to implement an extensible foreign type system with support for typedefs, structures, and unions, a declarative interface for defining foreign function calls, and automatic conversion of foreign function arguments to/from Lisp types.

Please note the following conventions that apply to everything in `CFFI-SYS`:

- Functions in `CFFI-SYS` that are low-level versions of functions exported from the `CFFI` package begin with a leading percent-sign (eg. `%mem-ref`).
- Where "foreign type" is mentioned as the kind of an argument, the meaning is restricted to that subset of all foreign types defined in Chapter 2 [Built-In Foreign Types], page 2. Support for higher-level types is always defined in terms of those lower-level types in `CFFI` proper.

2 Built-In Foreign Types

`:char` [Foreign Type]
`:unsigned-char` [Foreign Type]
`:short` [Foreign Type]
`:unsigned-short` [Foreign Type]
`:int` [Foreign Type]
`:unsigned-int` [Foreign Type]
`:long` [Foreign Type]
`:unsigned-long` [Foreign Type]
`:long-long` [Foreign Type]
`:unsigned-long-long` [Foreign Type]

These types correspond to the native C integer types according to the ABI of the system the Lisp implementation is compiled against.

`:int8` [Foreign Type]
`:uint8` [Foreign Type]
`:int16` [Foreign Type]
`:uint16` [Foreign Type]
`:int32` [Foreign Type]
`:uint32` [Foreign Type]
`:int64` [Foreign Type]
`:uint64` [Foreign Type]

Foreign integer types of specific sizes, corresponding to the C types defined in `stdint.h`.

`:size` [Foreign Type]
`:ssize` [Foreign Type]
`:ptrdiff` [Foreign Type]
`:time` [Foreign Type]

Foreign integer types corresponding to the standard C types (without the `_t` suffix).

Implementor's note: I'm sure there are more of these that could be useful, let's add any types that can't be defined portably to this list as necessary.

`:float` [Foreign Type]
`:double` [Foreign Type]

The `:float` type represents a C `float` and a Lisp `single-float`. `:double` represents a C `double` and a Lisp `double-float`.

`:pointer` [Foreign Type]

A foreign pointer to an object of any type, corresponding to `void *`.

`:void` [Foreign Type]

No type at all. Only valid as the return type of a function.

3 Operations on Built-in Foreign Types

`%foreign-type-size` *type* ⇒ *size* [Function]

Return the *size*, in bytes, of objects having foreign type *type*. An error is signalled if *type* is not a known built-in foreign type.

`%foreign-type-alignment` *type* ⇒ *alignment* [Function]

Return the default alignment in bytes for structure members of foreign type *type*. An error is signalled if *type* is not a known built-in foreign type.

Implementor's note: Maybe this should take an optional keyword argument specifying an alternate alignment system, eg. :mac68k for 68000-compatible alignment on Darwin.

4 Basic Pointer Operations

`pointerp` *ptr* ⇒ *boolean* [Function]
: Return true if *ptr* is a foreign pointer.

`null-pointer` ⇒ *pointer* [Function]
: Return a null foreign pointer.

`null-pointer-p` *ptr* ⇒ *boolean* [Function]
: Return true if *ptr* is a null foreign pointer.

`make-pointer` *address* ⇒ *pointer* [Function]
: Return a pointer corresponding to the numeric integer *address*.

`inc-pointer` *ptr offset* ⇒ *pointer* [Function]
: Return the result of numerically incrementing *ptr* by *offset*.

5 Foreign Memory Allocation

`foreign-alloc` *size* ⇒ *pointer* [Function]

Allocate *size* bytes of foreign-addressable memory and return a *pointer* to the allocated block. An implementation-specific error is signalled if the memory cannot be allocated.

`foreign-free` *ptr* ⇒ *unspecified* [Function]

Free a pointer *ptr* allocated by `foreign-alloc`. The results are undefined if *ptr* is used after being freed.

`with-foreign-pointer` (*var size* **&optional** *size-var*) **&body** *body* [Macro]

Bind *var* to a pointer to *size* bytes of foreign-accessible memory during *body*. Both *ptr* and the memory block it points to have dynamic extent and may be stack allocated if supported by the implementation. If *size-var* is supplied, it will be bound to *size* during *body*.

6 Memory Access

`%mem-ref` *ptr type* **&optional** *offset* [Accessor]

Dereference a pointer *offset* bytes from *ptr* to an object for reading (or writing when used with `setf`) of built-in type *type*.

Example

```
;; An impractical example, since time returns the time as well,
;; but it demonstrates %MEM-REF. Better (simple) examples wanted!
(with-foreign-pointer (p (foreign-type-size :time))
  (foreign-funcall "time" :pointer p :time)
  (%mem-ref p :time))
```

7 Foreign Function Calling

`%foreign-funcall` *name* {*arg-type arg*}* **&optional** *result-type* ⇒ *object* [Macro]

`%foreign-funcall-pointer` *ptr* {*arg-type arg*}* **&optional** *result-type* ⇒ *object* [Macro]

Invoke a foreign function called *name* in the foreign source code.

Each *arg-type* is a foreign type specifier, followed by *arg*, Lisp data to be converted to foreign data of type *arg-type*. *result-type* is the foreign type of the function's return value, and is assumed to be `:void` if not supplied.

`%foreign-funcall-pointer` takes a pointer *ptr* to the function, as returned by `foreign-symbol-pointer`, rather than a string *name*.

`%foreign-funcall-varargs` *name* ({*fixed-type arg*}*) {*vararg-type arg*}* **&optional** *result-type* ⇒ *object* [Macro]

`%foreign-funcall-varargs-pointer` *ptr* ({*fixed-type arg*}*) {*vararg-type arg*}* **&optional** *result-type* ⇒ *object* [Macro]

Invoke a foreign variadic function called *name* in the foreign source code.

Each *fixed-type* and *vararg-type* is a foreign type specifier, followed by *arg*, Lisp data to be converted to foreign data of type *arg-type*. *result-type* is the foreign type of the function's return value, and is assumed to be `:void` if not supplied.

`%foreign-funcall-pointer-varargs` takes a pointer *ptr* to the variadic function, as returned by `foreign-symbol-pointer`, rather than a string *name*.

Both functions have default implementation which call `%foreign-funcall` and `%foreign-funcall-pointer` appropriety.

Examples

```
;; Calling a standard C library function:
(%foreign-funcall "sqrtf" :float 16.0 :float) ⇒ 4.0

;; Dynamic allocation of a buffer and passing to a function:
(with-foreign-ptr (buf 255 buf-size)
  (%foreign-funcall "gethostname" :pointer buf :size buf-size :int)
  ;; Convert buf to a Lisp string using MAKE-STRING and %MEM-REF or
  ;; a portable CFFI function such as CFFI:FOREIGN-STRING-TO-LISP.
  )
```

8 Loading Foreign Libraries

`%load-foreign-library` *name* ⇒ *unspecified* [Function]

Load the foreign shared library *name*.

Implementor's note: There is a lot of behavior to decide here. Currently I lean toward not requiring NAME to be a full path to the library so we can search the system library directories (maybe even get LD_LIBRARY_PATH from the environment) as necessary.

9 Foreign Globals

`foreign-symbol-pointer` *name* ⇒ *pointer* [Function]

Return a pointer to a foreign symbol *name*.

Symbol Index