# The ins and outs of foreign interface design

## The ins and outs of foreign interface design

*Author:* *Jörg Höhle*
*Version:* *0.4.3*
*Date:* *00.2.2002*

## Foreign Language Interface Design

### Restrictions

Although portability is aimed, the examples are limited to five implementations of Common Lisp whose documentation I studied: Allegro CL (from Franz Inc.), Lispworks (Xanalysis), CormanLisp, CMUCL and CLISP. I welcome additions for other implementations (e.g. ECLS).

The Ada language community distinguishes what they call *thin bindings* from *thick bindings*. TODO

TODO restructure completely: The “operational profile” I expect is that the examples should be easy to find and will be what people will look at more than once, while the background text and my usual ranting will be read at most once. So the background should float around the examples, not the opposite.

### The examples used throughout this article

**`strtol`**

```
int strtol(const char* text, char **rest);
```

TODO what happens with spaces at beginning of text?

What makes `strtol` interesting?

- Rest is used as an TODO It has an out parameter, which it uses to return several values.
- The out parameter is somehow optional, as it can be initialized with NULL instead of pointing to a location where to store the result, in which case there is no way for the caller to tell how many characters were used to build the number.
- The rest string is part of the original string (cf. Section Returning Pointers into strings – don't! 0).

The functionality is uninteresting from a Common Lisp point of view, as `PARSE-INTEGER &KEY JUNK-ALLOWED` supersedes it completely and portably. The Lisp function also returns two values: the number parsed and how much of the string was read (as an index instead of a pointer).

**`gethostname`**

```
int gethostname(char* buf, int len);
```

What makes `gethostname` interesting?

- It's the prototype of a C function that returns a string, or seen at a lower level, that fills a string buffer.

- We'll discuss string representation issues (esp. Unicode) here.

**`zlib::compress`**

The zlib compression library from http://www.gzip.org/zlib/manual.html provides (among many others) a function to compress a region of memory:

```
int compress(char* dest, ulong* destlen, const char* source, ulong
sourcelen);
```

What makes `compress` interesting?

- `destlen` is used for in- and output.
- Both in- and output buffers are of variable (unknown during compilation) length.

## Use of in and out parameter declarations TODO bad name

### How the AllegroCL FFI radically differs from other FFI

The following C declaration translates immediately and trivially to a foreign function definition in Allegro Common Lisp:

```
long strtol(const char* text, char **rest);
(def-call-out strtol TODO)
```

Other Common Lisp implementations (LispWorks, CMUCL, CLISP, CormanLisp) require additional information before a foreign function definition can be written down.

What other information may be needed then?

- What some call *mode*. in and/or out modes are distinguished. This is used in conjunction with pointers. It cannot be seen from a C declaration whether a given pointer points to initialized memory that is going to be read or to memory where a result is to be placed. TODO Blabla COM IDL know in/out in another section.
- Maybe the equivalent of `const`
- hints for storage allocation?
- Other we'll see later

For example, here is one for CMUCL (in CMUCL terminology, *alien* is used for what other implementations call *foreign*): (TODO? better use CLISP because rest string will be correctly returned without too complex declarations?)

```
(def-alien- strtol TODO)
```

This defines a function which takes as single argument a string and returns two values: the number read and the remaining string. Using this function feels rather different from using the AllegroCL function which takes two arguments and returns a single value, exactly like the C one.

TODO Note how CMUCL's alien-funcall extern-alien construct knows no modes either, returns a single value and is at the same level as AllegroCL.

What are the respective advantages of these differing approaches?

- Translation from C declarations to foreign function definitions is trivial in AllegroCL and can thus be completely automated.

- However, the burden of using the function correctly and passing the correct arguments *entirely* relies on the shoulders of the programmer. This makes him/her feel like a C programmer. This is typical of thin bindings.
- The thick binding characteristics of the other approaches may TODO
- The other approaches are more elaborate and attempt to capture a high level of information. TODO flexibility loss?

**TODO multiple-value&failure here?**

**Lows and Highs of Abstraction**

TODO What I often observe when abstraction raises (or information is condensed) is an apparent TODO perceived loss of flexibility. I will discuss later on whether this is real or not. I explain what I mean:

For instance, in CLISP, it is not possible to use a single foreign function definition to capture both uses of `strtol` (remember, obtain a pointer to the rest string or pass `NULL`):

Suppose you're not interested in the rest pointer. Define

```
(def-c-call-out strtol
  (:arguments (text c-string)
              (rest c-ptr-null ffi:int)); always to be called with NIL
  (:return-type long))
```

And call with:

```
(strtol "123 bla" NIL)
```

Compare this to the previous definition for CLISP. There's one additional input that must be NIL and a single return value.

So what happened? I think that the CLISP FFI fails to cover this situation. This is what can generally happen when trying to be more abstract or provide higher level interfaces. There's no such risk using AllegroCL, since their level is as low as what the C programmer is used to. No flexibility is lost, there's no scenario that fails to be covered.

The higher comfort restricts flexibility.

TODO! Lispworks (TODO or Corman?) provides an interesting feature to overcome this situation. Although a parameter mode can be declared out, it can still be preset. In effect one can call the function using the above declaration, yet have `rest` set to `NULL`.

`TODO (LW call with rest as NULL)`

**Returning Pointers into strings – don't!**

Let's have a closer look at what happens when using the initial function definition for `strtol` in CLISP with a call like `(strtol "123 bla…bla")`. There is no means to declare that the rest pointer being returned is a pointer into the original string. Therefore it will allocate a *new* string in Lisp to contain the rest. This is extremely wasteful! We don't want that. TODO thin bindings again?

Suppose for example that you are repeatedly parsing numbers from a huge text, like a `.csv` file of comma separated values read into memory: this would be extremely wasteful! It would lead to an implementation in Lisp that behaves awfully. Compare

this to the implementation in C where only pointers would be manipulated and the text be kept in memory once — which makes sense. How can we achieve the same using the foreign function?
First, notice that this is not a problem with Lisp, but with the attempt of foreign language integration. More specifically, it shows up as a problem when trying to call functions that embed C idioms from any other language with radically different idioms. I once had exactly the same problem when calling from within the REXX language a function host (a regexp package) written in C. In our case, using `PARSE-INTEGER` and its `:start` keyword instead of `strtol` would lead to a native Lisp solution as efficient as possible.

- The culprit is the C idiom of returning a pointer into a part of a data structure, i.e. the rest string: C people favour pointer arithmetic instead of using indices on arrays like people from many other languages do.
- Knowing this, if our goal is to design a library purported to be usable from other languages as well, we better avoid returning such pointers.
- What we can do for know in order to stay sane is to either add a wrapper function in C around `strtol` which would return an index as an integer (`*rest-text`), or to write this wrapper in Lisp.

My experience: I find the C wrapper easier to write than the Lisp one. It's because we face a C idiom, so it's easy to work around in C and it makes me feel like I'm doing a good thing to the C interface, whereas writing the wrapper in Lisp would make me feel like alienating or corrupting Lisp by importing a braindamage.

*Getting your hands very dirty – manage memory and addresses*
Sometimes, you don't want or cannot write a wrapper in C. For example, you obtain a shared library and just wish to link it in. You don't even have a C compiler to create a tiny wrapper module. It comes at a cost: you'll need to do all the C programmer has to care about, manage memory yourself etc. You cannot let the Lisp implementation do all of it for you.

Here's how to write a wrapper for `strtol` in CMUCL:

```
(defun strtol (text)
  (with-alien
    (alien-funcall ...
      (values number
       (- (addr rest) (addr text)) ))))
```

You'll notice that CMUCL still has powerful primitives that will let you do away with manual memory management.

This is not the case when using Lispworks or CormanLisp. TODO sketch how

As far as I can tell, it's not possible to write the wrapper in Lisp with CLISP because there's no access to the primitives (like address-of, alloc-foreign). Given that the current (1995-2002) FFI of CLISP doesn't allow loading shared modules or libraries but requires an internal C compilation step (hidden by the module link script), the situation is not as bad as it sounds to the knowledgeable C programmer: just go and find a C programmer.

### In and/or out mode parameter declarations

## What happens under the woods

What follows is a description about which steps either the implementation or program goes through.

The varying abstraction levels mentioned earlier just shift more of the work described from the programmer to the implementation.

### Check types

### Maintain temporary structure eventually needed by the following step

### Obtain immediates (numbers or characters) or pointers to be used as parameters to the function.

For pointers this means either:

- Pass a pointer to the internal Lisp object, or
- Construct a temporary structure on the heap or on the stack, initialized from the Lisp object and pass a pointer to this structure. For example, initialize or copy an array and its contents TODO-clearer

There are striking differences between the various implementations in this area:

AllegroCL will happily pass pointers to internal objects (esp. arrays). Furthermore, in AllegroCL, Lisp strings are internally terminated by a zero byte so they are usable as C strings directly. In implementations where this is not the case, like (TODO), a C string must first be created from the non zero terminated Lisp string.

CLISP, at the other edge, will never pass any pointer to a Lisp object (or to part of it) and will always work with a temporary copy on the stack.

### Convert string representations (handle UNICODE)

CLISP and AllegroCL (TODO an Lispworks) provide means to control character set conversions between strings in Lisp and what the external world uses. E.g. TODO uses 8bits internally, whereas MS-Windows mostly uses Unicode. OTOH, CLISP uses Unicode internally whereas most operating systems it runs on use 8 bit characters (and country specific extensions in e.g. Asia).

Such a conversion has an impact on the length of strings (or buffers). This matters especially when the length is passed as a separate argument to a foreign function. E.g. `(length "äöüß")` returns 4 whereas this string in the Unicode UTF-8 encoding would be 8 octets long. That's why the issize() annotation in IDL (cf. TODO) is useful again since it may conserve the relationship.

CMUCL strings are based on 8-bit characters (not necessarily ISO-Latin-1, I once used it in Korea) so no particular conversion happens.

*CLISP*

`custom:*foreign-encoding*` (aka `ffi:*foreign-encoding*`) controls how Lisp strings are converted back and forth around the foreign function call (and also around callbacks).

`ext:convert-string-from-bytes` and `ext:convert-string-to-bytes` enable the programmer to do the conversion manually. TODO example from my mail

*AllegroCL*
ff: TODO controls

### Call the function

*Late binding* means that the address of the function may not be known yet. The function name must be resolved to an address. How this is done heavily depends on the operating system. The declaration of the function in Lisp may therefore also differ among implementations from a single vendor. TODO LispWorks MS-Windows and UNIX.

Late binding is independent on the ability to dynamically load shared libraries. It merely defines the time at which a function name is mapped to an address. Resolution could occur at library opening time, at function definition time, at first call, at every call, whenever.

What's also open is how resolution can happen when a Lisp image containing foreign functions is restarted. Anything could happen: crashes (because the previously used address is not valid anymore), error messages (when the implementation has a mean to discover invalid addresses), or smooth (silent, transparent, on the fly) address updates (at the time the image is restarted or at the time a foreign function is called?). This topic is again out of the scope of this article.

### If needed (out parameters) copy back from the temporary structure to the original Lisp objects, or create Lisp objects as necessary

A long integer result may be too large to hold in a `fixnum` (which is an immediate value) and so will require a `bignum` object to be created. E.g. these days, a `fixnum` is often 24-30 bits wide, whereas a C (or CORBA) `long` is 32 or 64 bits wide.

This step is why it can be essential to know (i.e. to tell the implementation) whether arrays will be modified by the called function or not.

This is why *not* indicating whether this is needed can only work in implementations like AllegroCL which will never duplicate content prior to the call and happily pass pointers to Lisp objects (esp. arrays): As the callee will work on the original object, whether it is modified doesn't bother the implementation. It only bothers the programmer when this happens inadvertently. This implies program crashes at the time of the next garbage collection if the foreign function writes past array bounds, as doing so will usually corrupt the Lisp heap.

It is open to debate whether passing pointers to temporary objects on the stack only is less likely to crash the Lisp system if the foreign function writes past boundaries. Doing so will corrupt data on the stack that may as well cause the FFI (or even the called function itself) to behave incorrectly and crash later on.

### If needed free temporary structures.

TODO move contents to manual memory management

Several implementations (CMUCL, TODO) provide `WITH-` style constructs that reliably clean up (i.e. free alien resources) when their scope exists, e.g. `WITH-ALIEN` in CMUCL. LispWorks can even free foreign resources allocated while in the dynamic scope of a particular macro, which makes this construct work more like memory pool that can be freed at once.

*LispWorks*
"Any object created with `allocate-dynamic-foreign-object` within body will automatically be deallocated once the scope of the `with-dynamic-foreign-objects` function [well, macro] has been left."

I wonder whether this works reliably in the presence of callbacks. When a callback calls another foreign function, thus creating a nesting, what's going to be freed when?

Thus, the following are equivalent:

```
(with-dynamic-foreign-objects
  ((binding
TODO same with empty binding (cf their example)
```

**Return the result(s).**

Some implementations may preallocate storage for Lisp objects (if the size is known) prior to the call to ensure that the prior steps will succeed and no return value gets lost.

Since some implementations (e.g. AllegroCL) always return a single value, while others return multiple values depending on the number of out parameters, you should never make such a function (or rather the symbol that names the function) part of the external interface of your package. My advice is to wrap it up and provide an external view on the package that is close to Lisp idioms.

So, the following transformations occur:

- GETHOSTNAME: `int gethostname(buf,len)` in Lisp would take no arguments and return a string as single value.
- COMPRESS could take two forms: `(COMPRESS vector)` which yields a vector with same element type (which could be restricted among ([UNSIGNED-]BYTE 8), 16, 32, 64), and `(COMPRESS-INTO result vector)`, modeled after `MAP-INTO` which would compress into an existing vector, and it would be an error for it not to be long enough.

## A word about IDL and COM/CORBA/whatever

Recommended reading: Haskell/Direct

These IDLs provide a convenient declarative approach to the problem of passing back and forth arrays of various sizes. Let's have a look:

```
gethostname([out,issize(len)] cstring name,
            [] int len); TODO in/out??
```

The `issize` declaration relates two parameters: an array and its size (in number of TODO bytes or elements).

The situation seems clear for arrays: what's passed is the number of elements. C Strings are also arrays, however I'm always uncertain whether a given function expects or returns the terminating `\0` character to be counted as well. TODO little survey. TODO argv/argc differs

If the CLISP FFI were extended to support such a declaration, it would allow for an easy and straightforward foreign function definition for both `gethostname` and `compress`. However what would also be needed is a mean to allow the indication of

the array size to occur at run-time[*]. Currently, the array size is part of the function signature that is fixed at the time `def-call-out` is evaluated.

### Why C-style interfaces are preponderant

Some may argue it's because of its historical weight in fashionable operating systems. It's IMHO part of the truth. Others may argue they are not, COM or CORBA interfaces may be felt as preponderant, or even Java's RMI.

What are the typical aspects of a function?

- It's signature: types of parameters and return value(s).
- Parameter modes
- Partial data descriptions (like array element type and especially length)
- Possible exceptions
- Extent (aka. lifetime) of objects passed through and back

Gift and curse at the same time, from a modern programming language point of view is the lack of exceptions in C. ... TODO

## Variable sized buffers

The Haskell and COM (TODO and others?) IDL catch a common idiom

*CLISP*

CLISP provides various levels of support for the situation at hand. It all depends on properties of the variable sized array.

- `c-string` are always variable size by nature.
- `c-array-max` is usable with 0, \0 or NULL terminated arrays of known maximal size.
- For other situations, there's more of a work-around hack than support, which I describe in the following.

Currently, the only way to handle variable sized arrays in function calls is a trick using CAST. Very unfortunately, CAST is only allowed on foreign variables. Therefore we have to create such a variable and set it prior to the call. This is rather limiting and quite dangerous in the presence of callbacks.

```
(ffi:def-c-var compress-indirect (c-ptr TODO))
(eval-when (compile)
  (FFI:c-lines "int (*compress_indirect)() =NULL;~%"))
(defun compress ()
  (setf compress-indirect (cast (function (char n) TODO
```

Since I don't want to create a single file with just the variable definition, and because compiling a .lisp file containing `ffi:def-x` definitions, I use `FFI:C-LINES` to augment the C file being created while compiling the Lisp file with my definition:

```
(ffi:c-lines "~&char *clisp_ffi_indirector = NULL;~%")
```

`FFI:C-LINES` is a macro expanding to an `EVAL-WHEN (COMPILE)` form[†].

[*] If it were known at compile-time, there would be no need to express the relationship between the two parameters, except maybe for consistency.

[†] Proper use of EVAL-WHEN is a topic for another cookbook entry.

## Success and failure with out parameters

There's a still a problem that was not yet addressed that occurs with out parameters. One can only expect out parameters to be set to some sane value if the function succeeds. However, none of the IDL or foreign function interfaces I know allow to express this relationship.

Note that this is "only" a problem in the context where the function uses some return value to indicate success or failure. If it were to raise an exception, no such problem would exist. But as we've seen before, exceptions across language boundaries are not in the scope of this document and they require (too?) heavy-weight mechanisms (like CORBA) to be able to deal with.

Consider again `gethostname` in the original definition for CormanLisp, CMUCL, CLISP or Lispworks. They all return two values: an integer (the C function return value) indicating success or failure and a string, obtained from the contents of the memory buffer that was passed to the function. The problem is that the contents of this memory is not defined in case the function returns failure! The memory must only be dereferenced in case of success.

This seems to indicate that in the presence of such functions, we need to either

- find a way to relate success and out parameters, so as to only construct out values when successful, or
- separate calling the function from returning meaningful values.

The former would be a declarative approach, but it is presently in none of the FFI of the implementations discussed in this article. It would be doable (similar to the `issize` approach). There's a risk that the solution that will be chosen may not cover all possible scenarios, like some out parameter being only set in case of failure (e.g. an error message), while another would only be set in case of success).

The latter means more lines of code and careful thought. Observe how the growing amount of TODO-denkbare declarations slowly becomes a mini language about properties of the interface and its functions. Is this a reasonable approach?

### Domain Specific Languages for Interfacing with Foreign Functions

What seems to be repeatedly needed is a way to reliably and consistently express the following:

- Prepare the call of the function – whatever that means.
- Call the function.
- Dereference output values only when its correct to do so, e.g. the function returned success so all output parameters are set.

The CLISP FFI has a bias towards dereferencing too early, i.e. immediately after the function call, without a possibility of intervention as of whether its correct or not.

TODO example from my mail (gethostname?)

Dereferencing slightly after the call means that in between, one must manipulate references or pointers. It also implies that memory management for intermediate data structures cannot be encapsulated and thus hidden inside a single call (like CLISP's `FFI::FOREIGN-CALL-OUT` does) but will be spread across a few calls and become somehow visible to the programmer.

*DSL and loss of flexibility?*
It may seem that a declarative approach leads to solutions that are less powerful than the "everything under programmer control" approach. It is obviously true that the declarative approach catches some scenarios that it can solve extremely well, but fails to cover all scenarios that the authors did not foresee. TODO lack of flexibility. I argue that it is not as limiting as it sounds.

It depends on what comes first:

- *n* varying historical ways to solve a problem and an attempt to unify all these afterwards, or
- a "good" language or framework, well explained, with good examples, which shows how to do things comfortably. TODO explain the initial little group and later larger user base image

I have often observed that when the second comes first, almost everybody will stick to the beaten path, provided that they don't feel too limited or until they hit the barriers. Therefore, I believe there is some value in attempting to derive a model, language, pattern, framework or whatever that provides more semantic and a denser information to the user and hides ugly and cumbersome implementation details.

TODO rephrase even more positively and less abstract: don’t reinvent wheel (which is neither safe nor reliable) TODO striking example

### A last word on thin and thick bindings

… or possibly the acknowledgement that there's no such last word.

TODO? varying citations on this topic.

While I tend to advocate the creation of thick bindings, I realize that while creating an FFI for the Amiga computer with CLISP in 1994, I implemented a lowest-level minimal interface: . TODO really a contradiction? Or just flexibility of functions.

- memory management was not part of my package (unlike Lispworks `with-foreign-objects`, CMUCL `with-alien` etc.), since these functions could be implemented using the FFI itself.
- It would not dereference foreign structures and convert them into Lisp objects (while the CLISP FFI excels at this). The programmer would do that using the primitives `MEM-READ`, `MEM-WRITE` etc. and therefore would have to know the dreaded C structure offsets.
- Yet it is able to *invoke any function of any standard shared library* on the Amiga, and extract meaningful results.
- It has a notion of strings and input-output buffers, but in hindsight, these appear as shortcuts to common functionality. TODO what means?

## Summary of rules

- Don't export a foreign function from a package, except the most trivial ones. This will hinder portability due to all the different implementations. Wrap the functionality instead and provide a Lisp-like interface.

- Don't return pointers into substructures, especially not into strings. Return indices instead. Indices are independent of memory locations and pointers. They are portable across languages and borders.

## Other ideas

Variable number of arguments (e.g. ioctl): split depending on usage.

## Stolen Code

TODO include

From Cookbook, cl-pdf: very practical: examples that work so I have less work

*AllegroCL*

```
(def-foreign-call (c-get-hostname "gethostname")
         ((name (* :char) (simple-array 'character (*)))
          (len :int integer))
  :returning :int)
(defun get-hostname ()
  (let* ((name (make-array 256 :element-type 'character))
         (result (c-get-hostname name 256)))
    (if (zerop result)
            (let ((pos (position #\null name)))
              (subseq name 0 pos))
          (error "gethostname() failed."))))
```

## Food for another article

Explain how to link or open shared libraries.

Explain how in CLISP, it's quite easy yet completely unknown how to turn extensions into a separate module. Explain modules advantages (separate compilation: no need to recompile CLISP when module interface changes, among others) and disadvantages (gettext supported only in core CLISP part?). As an example, show how DIRKEY (LDAP access) can and ought to be turned into a module.

## Dictionary

Implementation: the Lisp implementation, e.g. AllegroCL, CMUCL, CLISP, CormanLisp, Lispworks.

Programmer: the person using an implementation to call a foreign function.

## Acknowledgements

TODO who checked the AllegroCL/CMUCL/Lispworks/Corman specific examples.

## References

- H/Direct: A Binary Foreign Language Interface for Haskell, Simon Peyton-Jones, See section Foreign-language integration of URL http://research.microsoft.com/Users/simonpj/Papers/papers.html
- AllegroCL FFI documentation
- Lispworks FLI documentation fli-4.2.pdf

- CormanLisp documentation .pdf
- CLISP FFI documentation http://clisp.cons.org/impnotes.html#dffi
- CMUCL User's Manual about Alien Objects http://cvs2.cons.org/ftp-area/cmucl/doc/cmu-user/aliens.html