PLT Foreign Interface Manual

PLT (scheme@plt-scheme.org)

370
Released May 2007

Copyright notice

Copyright ©1996-2007 PLT

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Library General Public License, Version 2 published by the Free Software Foundation. A copy of the license is included in the appendix entitled “License.”

Contents

1. Introduction

The **foreign.ss** library enables the direct use of C-based APIs within Scheme programs (i.e., without writing any new C code). From the Scheme perspective, functions and data with a C-based API are *foreign*, hence the term *foreign interface*. Furthermore, since most APIs consist mostly of functions, the foreign interface is sometimes called a *foreign function interface*, abbreviated *FFI*.

Although using the FFI requires writing no new C code, it provides very little insulation against the issues that C programmer faces related to safety and memory management. An FFI programmer must be particularly aware of memory management issues for data that spans the Scheme–C divide. Thus, this manual relies in many ways on the information in *Inside PLT MzScheme*, which defines how MzScheme interacts with C APIs in general.

Since using the FFI entails many safety concerns that Scheme programmers can normally ignore, merely importing **foreign.ss** with `(require (lib "foreign.ss"))` does not import all of the FFI functionality. Only safe functionality is immediately imported. For example, `ptr-equal?` can never cause memory corruption or an invalid memory access, so it is immediately available on import.

Use `(unsafe!)` at the top-level of a module that imports **foreign.ss** to make unsafe features accessible.[1] Using this macro should be considered as a declaration that your code is itself unsafe, therefore can lead to serious problems in case of bugs: it is your responsibility to provide a safe interface.

In rare cases, you might want to provide an *unsafe* interface hat builds on the unsafe features of the FFI. In such cases, use the `provide*` macro with `unsafe` bindings, and use `define-unsafer` to provide an `unsafe!`-like macro that will make these bindings available to importers of your library. Providing users with unsafe operations without using this facility should be considered a bug in your code.

For examples of common FFI usage patterns, see the defined interfaces in the **ffi** collection.

[1] For additional safety, the `unsafe!` is itself protected (see §9.4 in *PLT MzScheme: Language Manual*).

2. Loading Foreign Libraries

The FFI is normally used by extracting functions and other objects from shared objects (a.k.a. *shared libraries* or *dynamically loaded libraries*). The `ffi-lib` function loads a shared object.

`(ffi-lib` *`path-or-false`* [*`version`*]`)` PROCEDURE

Returns an `ffi-lib` object. If *`path-or-false`* is a path (or string), the result represents the foreign library, which is opened in an OS-specific way (using `LoadLibrary` under Windows, and `dlopen` under Unix and Mac OS X). The path is not expected to contain the library suffix, which is added according to the current platform. If adding the suffix fails, several other filename variations are tried — retrying without an automatically added suffix, and using a full path of a file if it exists relative to the current directory (since the OS-level library function usually searches, unless the library name is an absolute path). An optional *`version`* string can be supplied, which is appended to the name after any added suffix. If you need any of a few possible versions, use a list of version strings, and `ffi-lib` will try all of them.

If *`path-or-false`* is `#f`, then the resulting `ffi-lib` object represents all libraries loaded in the current process, including libraries previouly opened with `ffi-lib`. In particular, use `#f` to access C-level functionality exported by the run-time system (as described in *Inside PLT MzScheme*).

Note: `ffi-lib` tries to look for the library file in a few places like the PLT libraries (see `get-lib-search-dirs` in the **setup** collection), a relative path, or a system search. However, if `dlopen` cannot open a library, there is no reliable way to know why it failed, so if all path combinations fail, it will raise an error with the result of `dlopen` on the unmodified argument name. For example, if you have a local **foo.so** library that cannot be loaded because of a missing symbol, using `(ffi-lib "foo.so")` will fail with all its search options, most because the library is not found, and once because of the missing symbol, and eventually produce an error message that comes from `dlopen("foo.so")` which will look like the file is not found. In such cases try to specify a full or relative path (containing slashes, e.g., **./foo.so**).

`(ffi-lib? v)` PROCEDURE

Returns `#t` if *`v`* is the result of `ffi-lib`, `#f` otherwise.

`(get-ffi-obj` *`objname lib type`* [*`failure-thunk`*]`)` PROCEDURE

Looks for the given object name *`objname`* (a string, a byte string, or a symbol), in the given *`lib`* library (a library object, string, path, or `#f`). If *`lib`* is not a library object produced by `ffi-lib`, it is converted to one by calling `ffi-lib`. If *`objname`* is found in *`lib`*, it is converted to Scheme using the given *`type`*. Types are described in the next chapter; in particular the `get-ffi-obj` procedure is most often used with function types (see section 3.5).

Keep in mind that *`get-ffi-obj`* is an unsafe procedure; see Chapter 1 for details.

If the object is not found, and *`failure-thunk`* is provided, it is used to produce a return value. For example, a failure thunk can be provided to report a specific error if an object is not found:

```
(define foo
```

```
(get-ffi-obj "foo" foolib (_fun _int -> _int)
  (lambda ()
    (error 'foolib
           "your installed foolib version does not provide \"foo\""))))
```

The default (also when *failure-thunk* is provided as `#f`) is to raise an exception

`(set-ffi-obj! objname lib type new)` PROCEDURE

Looks for the *objname* in *lib* similarly to `get-ffi-obj`, but then it stores the given *new* value into the library, converting it to a C value. This can be used for setting library customization variables that are part of its interface, including Scheme callbacks.

`(make-c-parameter objname lib ctype)` PROCEDURE

Returns a parameter-like procedure that can either reference the specified foreign value, or set it. This is useful in case Scheme code and library code interact through a library value. It can be used with any type, but it is not recommended to use this for foreign functions since each reference through this will construct the low-level interface before the actual call.

`(define-c var lib type)` SYNTAX

This syntax uses `make-c-parameter` above: it defines a *var* syntax that behaves like a Scheme binding, referencing and setting it is achieved through such a C parameter (so the same comments apply). The *var* part is used both for the Scheme binding and for the foreign object's name.

`(ffi-obj-ref objname lib)` FAILURE-THUNK procedure

Returns a pointer object for the required foreign object. This is for rare cases where `make-c-parameter` is insufficient because there is no type to cast the foreign object to (e.g., a vector of numbers).

3. C Types

C types are the main concept of the FFI, either primitive types or user-defined types. The FFI deals with primitive types internally, converting them to and from C types. A user type is define in terms of existing primitive and user types, along with conversion functions to and from the existing types.

`(make-ctype` *ctype scheme-to-C-proc C-to-scheme-proc*`)` PROCEDURE

Creates a new C type object, with the given conversions functions. The conversion functions can be `#f` meaning that there is no conversion for the corresponding direction. If both functions are `#f`, *ctype* is returned.

`(ctype?` *v*`)` PROCEDURE

Returns `#t` if *v* is a C type (primitive or user-defined), `#f` otherwise.

`(ctype-sizeof` *ctype*`)` PROCEDURE
`(ctype-alignof` *ctype*`)` PROCEDURE

Return the size and alignment of a given *ctype* for the current platform.

`(compiler-sizeof` *symbol*`)` PROCEDURE

Possible values for *symbol* are `'int`, `'char`, `'short`, `'long`, `'*`, `'void`, `'float`, `'double`. The result is the size of the correspond type according to the C `sizeof` operator for the current platform. The `compiler-sizeof` operation should be used to gather information about the current platform, such as defining alias type like `_int` to a known type like `_int32`.

3.1 Numeric Types

There are basic integer types at various sizes. These are: `_int8`, `_sint8`, `_uint8`, `_int16`, `_sint16`, `_uint16`, `_int32`, `_sint32`, `_uint32`, `_int64`, `_sint64`, and `_uint64`. The 's' or 'u' prefix specifies a signed or an unsigned integer respectively; the ones with no prefix are signed.

In addition, there are several type 'aliases' (extra bindings for some of the above types):

- `_byte`, `_ubyte` and `_sbyte`: aliases for `_uint8` and `_sint8` (`_byte` is unsigned),
- `_word`, `_uword` and `_sword`: aliases for `_uint16` and `_sint16` (`_word` is unsigned),
- `_short`, `_ushort` and `_sshort`: aliases for the integer type that correspond to the platform's `short` type (`_short` is signed),
- `_int`, `_uint` and `_sint`: aliases for the integer type that correspond to the platform's `int` type (`_int` is signed),

- _long, _ulong and _slong: aliases for the integer type that correspond to the platform's long type (_long is signed),

In cases where speed matters, and you know that the integer is small enough, use the types _fixnum and _ufixnum, which are similar to _long and _ulong but assume that the quantities fit in MzScheme's immediate integers (not bignums). If you need this capability, but you want to be sure that the C level integer is a 32-bit size (as opposed to a long integer on some platforms), then use _fixint and _ufixint.

Finally, there are two floating point types, _float and _double for the corresponding C types, and the type _double* that implicitly coerces any non-complex number to a C double.

3.2 Other Atomic Types

_bool C TYPE

Translates #f to a 0 _int, and any other value to 1.

_void C TYPE

This type indicates a Scheme void return value, and it cannot be used to translate values to C (i.e., this type cannot be used for function inputs).

3.3 String Types

3.3.1 Primitive String Types

The five primitive string types corerspond to cases where a C representation matches MzScheme's representation without encodings.

_bytes C TYPE

A type for Scheme byte strings, which corresponds to C's char* type. In addition to translating byte strings, #f corresponds to the NULL pointer. Note that this is also a custom-type macro; see section 3.5.1 below.

_string/ucs-4 C TYPE

A type for MzScheme's native Unicode strings, which is in UCS-4 format. These correspond to the C mzchar* type used by MzScheme.

_string/utf-16 C TYPE

Unicode strings in UTF-16 format.

_path C TYPE

Simple char* strings, corresponding to MzScheme's paths.

_symbol C TYPE

Simple char* strings as Scheme symbols (encoded in UTF-8). Return values using this type are interned as symbols.

3.3.2 Fixed Auto-Converting String Types

Several addition string types correspond to encoring conversions. The `_string/utf-8`, `_string/locale`, and `_string/latin-1` types all correspond to (character) strings on the Scheme side and `char*` strings on the C side. The brige between the two requires a transformation on the content of the string. As usual, the types treat `#f` as `NULL` and vice-versa.

The `_string*/utf-8`, `_string*/locale`, and `_string*/latin-1` types are similar, but they accept a wider range of values: Scheme byte strings are allowed passed as is, and Scheme paths are converted using `path->bytes`.

3.3.3 Variable Auto-Converting String Type

The `_string/ucs-4` type is rarely useful when interacting with foreign code. Using `_bytes` is somewhat unnatural, since it forces Scheme programmers to use byte strings. Using _string/utf-8, etc. may prematurely commit to a particular encoding of strings as bytes. The `_string` type supports conversion between Scheme strings and `char*` strings using a parameter-determined conversion.

`_string` C TYPE

Expands to a use of the `default-_string-type` parameter. The parameter's value is consulted when `_string` is evaluated, so the parameter should be set before any interface definition that uses `_string`.

`(default-_string-type [`*ctype*`])` PROCEDURE

A parameter that determines the current meanging of `_string`. It is initially set to `_string/*utf-8`. If you change it, do so *before* interfaces are defined.

3.3.4 Other String Types

`_file` C TYPE

Like `_path`, but when values go from Scheme to C, `expand-path` is used on the given value. As an output value, it is identical to `_path`.

`_bytes/eof` C TYPE

Similar to the `_bytes` type, except that a foreign return value of `NULL` is translated to a Scheme `eof` value.

`_string/eof` C TYPE

Similar to the `_string` type, except that a foreign return value of `NULL` is translated to a Scheme `eof` value.

3.4 Pointer Types

`_pointer` C TYPE

Corresponds to Scheme "C pointer" objects. These pointers can have an arbitrary Scheme object attached as a type tag. The tag is ignored by built-in functionality; it is intended to be used by interfaces. See section 5.1 for creating pointer types that use these tags for safety.

_scheme C TYPE

This type can be used with any Scheme object; it corresponds to the Scheme_Object* type of MzScheme's C API (see *Inside PLT MzScheme*). It is useful only for libraries that are aware of MzScheme's C API.

_fpointer C TYPE

Similar to _pointer, except that it should be used with function pointers. Using these pointers avoids one dereferencing, which is the proper way of dealing with function pointers. This type should be used only in rare situations where you need to pass a foreign function pointer to a foreign function; using a _cprocedure type is possible for such situations, but inefficient, as every call will go through Scheme unnecessarily. Otherwise, _cprocedure should be used (it is based on _fpointer).

3.5 Function Types

(_cprocedure *input-types output-type* [*wrapper-proc*]) PROCEDURE

A type constructor that creates a new function type, which is specified by the given *input-types* list and *output-type*. Usually, the _fun syntax (described below) should be used instead, since it manages a wide range of complicated cases.

The resulting type can be used to reference foreign functions (usually ffi-objs, but any pointer object can be referenced with this type), generating a matching foreign callout object. Such objects are new primitive procedure objects that can be used like any other Scheme procedure.

A type created with _cprocedure can also be used for passing Scheme procedures to foreign functions, which will generate a foreign function pointer that calls the given Scheme procedure when it is used. There are no restrictions on the Scheme procedure; in particular, its lexical context is properly preserved.

The optional *wrapper-proc*, if provided, is expected to be a function that can change a callout procedure: when a callout is generated, the wrapper is applied on the newly created primitive procedure, and its result is used as the new function. Thus, *wrapper-proc* is a hook that can perform various argument manipulations before the foreign function is invoked, and return different results (for example, grabbing a value stored in an 'output' pointer and returning multiple values). It can also be used for callbacks, as an additional layer that tweaks arguments from the foreign code before they reach the Scheme procedure, and possibly changes the result values too.

(_fun [*args ::*] *input-type* ⋯ -> *output-type* [-> *output-expr*])

Creates a new function type. This is a convenient syntax for the _cprocedure type constructor that can handle complicated cases of argument handling. In its simplest form, only the *input-type*s and the *output-type* are specified and each one is a simple expression, which creates a straightforward function type.

In its full form, the _fun syntax provides an IDL-like language that can be used to create a wrapper function around the primitive foreign function. These wrappers can implement complex foreign interfaces given simple specifications. First, the full form of each of the types can include an optional label and an expression:

```
type-spec is one of
  type
  (label : type)
  (type = expr)
  (label : type = expr)
```

If an expression is provided, then the resulting function will be a wrapper that calculates the argument for that position

itself, meaning that it does not expect an argument for that position. The expression can use previous arguments if they were labeled. In addition, the result of a function call need not be the value returned from the foreign call: if the optional *output-expr* is specified, or if an expression is provided for the output type, then this specifies an expression that will be used as a return value. This expression can use any of the previous labels, including a label given for the output which can be used to access the actual foreign return value.

In rare cases where complete control over the input arguments is needed, the wrapper's argument list can be specified as *args*, in any form (including a 'rest' argument). Identifiers in this place are related to type labels, so if an argument is there is no need to use an expression, for example:

```
(_fun (n s) :: (s : _string) (n : _int) -> _int)
```

specifies a function that receives an integer and a string, but the foreign function will get the string first.

3.5.1 Custom Function Types

The behavior of the `_fun` type can be customized via custom function types. These are pieces of syntax that can behave as C types and C type constructors, but they can interact with function calls in several ways that are not possible otherwise. When the `_fun` form is expanded, it tries to expand each of the given type expressions, and ones that expand to certain keyword-value lists interact with the generation of the foreign function wrapper. This makes it possible to construct a single wrapper function, avoiding the costs involved in compositions of higher-order functions.

Custom function types are macros that expand to a list that looks like: '`(key: val ...)`', where all of the '`key:`'s are from a short list of known keys. Each key interacts with generated wrapper functions in a different way, which affects how its corresponding argument is treated:

- `type:` specifies the foreign type that should be used, if it is `#f` then this argument does not participate in the foreign call.
- `expr:` specifies an expression to be used for arguments of this type, removing it from wrapper arguments.
- `bind:` specifies a name that is bound to the original argument if it is required later (e.g., `_box` converts its associated value to a C pointer, and later needs to refer back to the original box).
- `1st-arg:` specifies a name that can be used to refer to the first argument of the foreign call (good for common cases where the first argument has a special meaning, e.g., for method calls).
- `prev-arg:` similar to `1st-arg:`, but refers to the previous argument.
- `pre:` a pre-foreign code chunk that is used to change the argument's value.
- `post:` a similar post-foreign code chunk.

The `pre:` and `post:` bindings can be of the form `(id => expr)` to use the existing value. Note that if the `pre:` expression is not `(id => expr)`, then it means that there is no input for this argument to the `_fun`-generated procedure. Also note that if a custom type is used as an output type of a function, then only the `post:` code is used.

All of the special custom types that are described here are defined this way.

Most custom types are meaningful only in a `_fun` context, and will raise a syntax error if used elsewhere. A few such types can be used in non-`_fun` contexts: types which use only `type:`, `pre:`, `post:`, and no others. Such custom types can be used outside a `_fun` by expanding them into a usage of `make-ctype`, using other keywords makes this impossible — it means that the type has specific interaction with a function call.

`(define-fun-syntax identifier transformer)` SYNTAX

The results of expanding custom type macros is taken apart by the `_fun` macro, which will lead to code certificate problems. To solve this, do use `define-fun-syntax` instead of `define-syntax`. It is used in the same way, but will avoid such problems.

`_?` CUSTOM C TYPE

Not a conventional C type, but a marker for expressions that should not be sent to the ffi function. Use this to bind local values in a computation that is part of an ffi wrapper interface, or to specify wrapper arguments that are not sent to the foreign function (e.g., an argument that is used for processing the foreign output).

`(_ptr mode type)` CUSTOM C TYPE

For C pointers, where *mode* indicates input or output pointers (or both). *mode* can be one of the following:

- '`i`', indicating an *input* pointer argument: the wrapper will arrange for the function call to receive a value that can be used with the *type* and to send a pointer to this value to the foreign function. After the call the value is discarded.
- '`o`', indicating an *output* pointer argument: the foreign function expects a pointer to a place where it will save some value, and this value is accessible after the call, to be used by an extra return expression. If `_ptr` is used in this mode, then the generated wrapper does not expect an argument since one will be freshly allocated before the call.
- '`io`' combines the above into an *input/output* pointer argument: the wrapper will get the Scheme value, allocate and set a pointer using this value, and reference the value after the call. The '`_ptr`' can be confusing here: it means that the foreign function expects a pointer, but the generated wrapper uses an actual value. (Note that if this is used with structs, a struct is created when calling the function, and a copy of the return value is made too — inefficient, but ensures that structs are not modified by C code.)

For example, the `_ptr` type can be used in output mode to create a foreign function wrapper that returns more than a single argument. The following type:

```
(_fun (i : (_ptr o _int))
      -> (d : _double)
      -> (values d i))
```

will create a function that calls the foreign function with a fresh integer pointer, and use the value that is placed there as a second return value.

`_box` TYPE Custom C Type

This is similar to a `(_ptr io type)` argument, where the input is expected to be a box holding an appropriate value, which is unboxed on entry and modified accordingly on exit.

`(_list mode type [len])` CUSTOM C TYPE

Similar to `_ptr`, except that it is used for converting lists to/from C vectors. The optional *len* argument is needed for output values where it is used in the post code, and in the pre code of an output mode to allocate the block. In any case it can refer to a previous binding for the length of the list which the C function will most likely require.

_vector CUSTOM C TYPE

Same as _list, except that it uses Scheme vectors instead of lists.

_bytes CUSTOM C TYPE

_bytes can be used by itself as a simple type that uses a byte string as a C pointer. Alternatively, it can be used as a '(_bytes o *len*)' form is for a pointer return value, where the size should be explicitly specified. There is no need for other modes: input or input/output would be just like _bytes since the string carries its size information (there is no real need for the 'o' part of the syntax, but it's there for consistency with the above macros).

_cvector CUSTOM C TYPE

Like _bytes, _cvector can be used as a simple type that corresponds to a pointer that is managed as a safe C vector on the Scheme side — this is described in section 5.2 above. The syntax specified here is an alternative that makes it behave similarly to the _list and _vector custom types, except that this is more efficient since no Scheme list or vector are needed. (It can be used with all three modes.)

3.6 C Struct Types

(make-cstruct-type *ctypes*) PROCEDURE

The primitive type constructor for creating new C struct types. These types are actually new primitive types — they don't have any conversion functions associated. The corresponding Scheme objects that are used for structs are pointers, but when these types are used, the value that the pointer *refers to* is used rather than the pointer itself. This value is basically made of a number of bytes that is known according to the given list of *ctypes* list.

(_list-struct *ctypes* ...[1]) PROCEDURE

A type constructor that builds a struct type using the above make-cstruct-type function and wraps it in a type that marshals a struct as a list of its components. Note that space for structs needs to be allocated; the converter for a _list-struct type immediately allocates and uses a list from the allocated space, so it is inefficient. Use define-cstruct below for a more efficient approach.

(define-cstruct *_id ((field-id ctype) ...)*) SYNTAX

Defines a new C struct type, but unlike _list-struct, the resulting type deals with C structs in binary form rather than marshaling them to Scheme values. It uses a define-struct-like approach, providing accessor functions for raw struct values (which are pointer objects). The new type uses pointer tags to guarantee that only proper struct objects are used. The name must have a form of *_id*. The form and the generated bindings are intentionally similar to define-struct only with type specification for the fields — the identifiers that will be bound as a result are:

- *_id*: the new C type for this struct
- *_id*-pointer: a pointer type that should be used when a pointer to values of this struct are used
- *id*?: a predicate for the new type
- *id*-tag: the tag string object that is used with these values
- make-*id*: a constructor, expects an argument for each type
- *id-field-id*...: an accessor function for each *field-id*

- `set-`*`id-field-id`*`!...`: a mutator function for each *`field-id`*

Objects of this new type are actually cpointers, with a type tag that is a list that contains `"`*`id`*`"`. Since structs are implemented as pointers, they can be used for a `_pointer` input to a foreign function: their address will be used. To make this a little safer, the corresponding cpointer type is defined as *`_id`*`-pointer`. The *`_id`* type should not be used when a pointer is expected, since it will cause the struct to be copied rather than use the pointer value, leading to memory corruption.

If the first field is itself a cstruct type, its tag will be used in addition to the new tag. This feature supports common cases of object inheritance, where a sub-struct is made by having a first field that is its super-struct. Instances of the sub-struct can be considered as instances of the super-struct, since they share the same initial layout. Using the tag of an initial cstruct field means that the same behavior is implemented in Scheme; for example, accessors and mutators of the super-cstruct can be used with the new sub-cstruct. See Section 3.6.1 for an example.

Note that structs are allocated as atomic blocks, which means that the garbage collector ignores their content. Currently, there is no safe way to store pointers to GC-managed objects in structs (even if you keep a reference to avoid collecting the referenced objects, a the 3m variant's GC will invalidate the pointer's value). Thus, only non-pointer values and pointers to memory that is outside the GC's control can be lpaced into struct fields.

`(define-cstruct` *`(_id _super-id) ((field-id ctype) ···)`* SYNTAX

This alternative form of `define-cstruct`, is shorthand for using an initial field named *`super-id`* using *`_super-id`* as its type. Remember that the new struct will use *`_super-id`*'s tag in addition to its own tag, meaning that instances of *`_id`* can be used as instances of *`_super-id`*. Aside from the syntactic sugar, the constructor function will be different when this syntax is used: instead of expecting a first argument which is an instance of *`_super-id`*, it will expect arguments for each of *`_super-id`*'s fields in addition for the new fields. This is, again, in analogy to using a super-struct with `define-struct`.

3.6.1 C Struct Examples

A few examples will help understanding how to use structs. Assuming the following C code:

```
typedef struct { int x; char y; } A;
typedef struct { A a; int z; } B;

A* makeA() {
  A *p = malloc(sizeof(A));
  p->x = 1;
  p->y = 2;
  return p;
}

B* makeB() {
  B *p = malloc(sizeof(B));
  p->a.x = 1;
  p->a.y = 2;
  p->z   = 3;
  return p;
}

char gety(A* a) {
  return a->y;
}
```

First, using the simple _list-struct, you might expect this code to work:

```
(define makeB
  (get-ffi-obj 'makeB "foo.so"
    (_fun -> (_list-struct (_list-struct _int _byte) _int))))
(makeB) ; should return ((1 2) 3)
```

The problem here is that makeB returns a pointer to the struct rather than the struct itself. The following works as expected:

```
(define makeB
  (get-ffi-obj 'makeB "foo.so" (_fun -> _pointer)))
(ptr-ref (makeB) (_list-struct (_list-struct _int _byte) _int))
```

As described above, _list-structs should be used in cases where efficiency is not an issue. We continue using define-cstruct, first define a type for 'A' which makes it possible to use 'makeA':

```
(define-cstruct _A ([x _int] [y _byte]))
(define makeA
  (get-ffi-obj 'makeA "foo.so"
    (_fun -> _A-pointer))) ; using _A is a memory-corrupting bug!
(define a (makeA))
(list a (A-x a) (A-y a))
 ; -> (#<cpointer:A> 1 2)
```

'gety' is also simple to use from Scheme:

```
(define gety
  (get-ffi-obj 'gety "foo.so"
    (_fun _A-pointer -> _byte)))
(gety a)
 ; -> 2
```

We now define another C struct for 'B', and expose 'makeB' using it:

```
(define-cstruct _B ([a _A] [z _int]))
(define makeB
  (get-ffi-obj 'makeB "foo.so"
    (_fun -> _B-pointer)))
(define b (makeB))
```

We can access all values of *b* using a naive approach:

```
(list (A-x (B-a b)) (A-y (B-a b)) (B-z b))
```

but this is inefficient as it allocates and copies an instance of 'A' on every access. Inspecting the tags (cpointer-tag b) we can see that A's tag is included, so we can simply use its accessors and mutators, as well as any function that is defined to take an A pointer:

```
(list (A-x b) (A-y b) (B-z b))
(gety b)
```

Constructing a B instance in Scheme requires allocating a temporary A struct:

```
(define b (make-B (make-A 1 2) 3))
```

To make this more efficient, we switch to the alternative `define-cstruct` syntax, which creates a constructor that expects arguments for both the super fields ands the new ones:

```
(define-cstruct (_B _A) ([z _int]))
(define b (make-B 1 2 3))
```

3.7 Enumerations and Masks

Although the constructors below are describes as procedures,they are implemented as syntax, so that error messages can report a type name where the syntactic context implies one.

(_enum *symbols* [*basetype*]) PROCEDURE

Takes a list of symbols and generates an enumeration type. The enumeration maps between the given *symbols* and integers, counting from 0. The list of *symbols* can also set the values of symbols, if a symbol is followed by `'=` and an integer. For example, the list `'(x y = 10 z)` maps `'x` to `0`, `'y` to `10`, and `'z` to `11`.

The optional *basetype* argument specifies the base type to use, defaulting to `_ufixint`.

(_bitmask *symbols* [*basetype*]) PROCEDURE

Similar to `_enum`, but the resulting mapping translates a list of symbols to a number and back, using a logical or. A single symbol can be given as an input to make things a little more convenient. The default *basetype* is `_uint`, since high bits are often used for flags.

4. Pointer Functions

`(cpointer? v)` PROCEDURE

A predicate for pointer values: returns `#t` for C pointer objects as well as other values that can be used as pointers: `#f` (used as a NULL pointer), byte strings (used as memory blocks), and some additional internal objects (`ffi-obj`s and callbacks, see section 7). The result is `#f` for other values.

`(ptr-ref cptr ctype [['abs] offset])` PROCEDURE

`(ptr-set! cptr ctype [['abs] offset] value)` PROCEDURE

The `pre-ref` procedure return the object referenced by *cptr*, using the given *ctype*. The `ptr-set!` procedure stores the *value* in the memory *cptr* points to, using the given *ctype* for the conversion, and returns void.

In each case, *offset* defaults to 0 (which is the only value that should be used with `ffi-obj` objects, see section 7). If an *offset* index is given, the value is stored at that location, considering the pointer as a vector of *ctype*s — so the actual address is the pointer plus the size of *ctype* multiplied by *offset*. In addition, a *'abs* flag can be used to use the *offset* as counting bytes rather then increments of the specified *ctype*.

Beware that the `ptr-ref` and `ptr-set!` procedure do not keep any meta-information on how pointers are used. It is the programmer's responsibility to use this facility only when appropriate. For example, on a little-endian machine:

```
> (define block (malloc _int 5))
> (ptr-set! block _int 0 196353)
> (map (lambda (i) (ptr-ref block _byte i)) '(0 1 2 3))
(1 255 2 0)
```

In addition, `ptr-ref` and `ptr-set!` cannot detect when offsets are beyond an object's memory bounds; out-of-bounds access can easily lead to a segmentation fault or memory corruption.

`(ptr-equal? cptr1 cptr2)` PROCEDURE

Compares the values of the two pointers. (Note that two different Scheme pointer objects can contain the same pointer.)

`(ptr-add cptr offset-k [ctype])` PROCEDURE

Returns a cpointer that is like *cptr* offset by *offset-k* instances of *ctype*. If *ctype* is not provided, *cptr* is offset by *offset-k* bytes.

The resulting cpointer keeps the base pointer and offset separate. The two pieces are combined at the last minute before any operation on the pointer, such as supplying the pointer to a foreign function. In particular, the pointer and offset are not combined until after all allocation leading up to a foreign-function call; if the called function does not itself call anything that can trigger a garbage collection, it can safey use pointers that are offset into the middle of a GCable object.

(offset-ptr? *cptr*) PROCEDURE

A predicate for cpointers that have an offset, such as pointers that were created using *ptr-add*. Returns #t even if such an offset happens to be 0. Returns #f for other cpointers and non-cpointers.

(ptr-offset *cptr*) PROCEDURE

Returns the offset of a pointer that has an offset. (The resulting offset is always in bytes.)

(set-ptr-offset! *cptr offset-k* [*ctype*]) PROCEDURE

Sets the offset component of an offset pointer. The arguments are used in the same way as ptr-add. Raises an error if it is given a pointer that has no offset.

(ptr-add! *cptr offset-k* [*ctype*]) PROCEDURE

Like ptr-add, but destructively modifies the offset contained in a pointer. (This can also be done using ptr-offset and set-ptr-offset!.)

(cpointer-tag *cptr*) PROCEDURE

Returns the Scheme object that is the tag of the given *cptr* pointer.

(set-cpointer-tag! *cptr tag*) PROCEDURE

Sets the tag of the given *cptr*. The *tag* argument can be any arbitrary value; other pointer operations ignore it. When a cpointer value is printed, its tag is shown if it is a symbol, a byte string, a string. In addition, if the tag is a pair holding one of these in its car, the car is shown (so that the tag can contain other information).

(memmove *cptr* [*offset-k*] *src-cptr* [*src-offset-k*] *count-k* [*ctype*])

Copies to *cptr* from *src-cptr*. The destination pointer can be offset by an optional *offset-k*, which is in bytes if *ctype* is not supplied, or in *ctype* instances when supplied. The source pointer can be similarly offset by *src-offset-k*. The number of bytes copied from source to destination is determined by *count-k*, which is also in bytes if *ctype* is not supplied, or in *ctype* instances when supplied.

(memcpy *cptr* [*offset-k*] *src-cptr* [*src-offset-k*] *count-k* [*count-ctype*])

Like memmove, but the result is undefined if the destination and source overlap.

(memset *cptr* [*offset-k*] *byte count-k* [*count-ctype*])

Similar to memmove, but the destination is uniformly filled with *byte* (i.e., an exact integer between 0 and 255 includive).

4.1 Memory Management

For general information on C-level memory management with MzScheme, see *Inside PLT MzScheme*.

(malloc *bytes-or-type* [*type-or-bytes*] [*cptr*] [*mode*] ['*fail-ok*]) PROCEDURE

Allocates a memory block of a specified size using a specified allocation. The result is a `cpointer` to the allocated memory. The four arguments can appear in any order since they are all different types of Scheme objects; a size specification is required at minimum:

- If a C type *bytes-or-type* is given, its size is used to the block allocation size.
- If an integer *bytes-or-type* is given, it specifies the required size in bytes.
- If both *bytes-or-type* and *type-or-bytes* are given, then the allocated size is for a vector of values (the multiplication of the size of the C type and the integer).
- If a *cptr* pointer is given, its contents is copied to the new block, it is expected to be able to do so.
- A symbol *mode* argument can be given, which specifies what allocation function to use. It should be one of `'nonatomic` (uses `scheme_malloc` from MzScheme's C API), `'atomic` (`scheme_malloc_atomic`), `'stubborn` (`scheme_malloc_stubborn`), `'uncollectable` (`scheme_malloc_uncollectable`), `'eternal` (`scheme_malloc_eternal`), `'interior` (`scheme_malloc_allow_interior`), `'atomic-interior` (`scheme_malloc_atomic_allow_interior`), or `'raw` (uses the operating system's `malloc`, creating a GC-invisible block).
- If an additional `'failok` flag is given, then `scheme_malloc_fail_ok` is used to wrap the call.

If no mode is specified, then `'nonatomic` allocation is used when the type is any pointer-based type, and `'atomic` allocation is used otherwise.

(free *cpointer*) PROCEDURE

Uses the operating system's `free` function for `'raw`-allocated pointers, and for pointers that a foreign library allocated and we should free. Note that this is useful as part of a finalizer (see below) procedure hook (e.g., on the Scheme pointer object, freeing the memory when the pointer object is collected, but beware of aliasing).

(end-stubborn-change *cpointer*) PROCEDURE

Uses `scheme_end_stubborn_change` on the given stubborn-allocated pointer (see *Inside PLT MzScheme*).

(register-finalizer *obj finalizer-proc*) PROCEDURE

Registers a finalizer procedure *finalizer-proc* with the given *obj* which can be any Scheme (GC-able) object. The finalizer is registered with a will executor (see §13.3 in *PLT MzScheme: Language Manual*); it is invoked when *obj* is about to be collected. (This is done by a thread that is in charge of triggering these will executors.)

This is mostly intended to be used with cpointer objects (for freeing unused memory that is not under GC control), but it can be used with any Scheme object — even ones that have nothing to do with foreign code. Note, however, that the finalizer is registered for the *Scheme* object. If you intend to free a pointer object, then you must be careful to not register finalizers for two cpointers that point to the same address. Also, be careful to not make the finalizer a closure that holds on to the object.

For example, suppose that you're dealing with a foreign function that returns a C string that you should free. Here is an attempt at creating a suitable type:

```
(define _bytes/free
  (make-ctype _pointer
              #f ; a Scheme bytes can be used as a pointer
```

```
                (lambda (x)
                  (let ([b (make-byte-string x)])
                    (register-finalizer x free)
                    b))))
```

This is wrong: the finalizer is registered for `x`, which is no longer needed once the byte string is created. Changing this to register the finalizer for `b` correct this problem, but then `free` will be invoked on it instead of on `x`. In an attempt to fix this, we will be careful and print out a message for debugging:

```
(define _bytes/free
  (make-ctype _pointer
              #f ; a Scheme bytes can be used as a pointer
              (lambda (x)
                (let ([b (make-byte-string x)])
                  (register-finalizer b
                    (lambda (_)
                      (printf "Releasing ~s\n" b)
                      (free x)))
                  b))))
```

but we never see any printout — the problem is that the finalizer is a closure that keeps a reference to `b`. To fix this, you should use the input argument to the finalizer. Simply changing the `_` to `b` will solve this problem. (Removing the debugging message also avoids the problem, since the finalization procedure would then not close over `b`.)

`(make-sized-byte-string cptr length)` PROCEDURE

Returns a byte string made of the given pointer and the given length. No copying is done. This can be used as an alternative to make pointer values accessible in Scheme when the size is known.

If `cptr` is an offset pointer created by `ptr-add`, the offset is immediately added to the pointer. Thus, this function cannot be used with `ptr-add` to create a substring of a Scheme byte string, because the offset pointer would be to the middle of a collectable object (which is not allowed).

5. Derived Utilities

5.1 Tagged C Pointer Types

`(cpointer-has-tag? cptr tag)` PROCEDURE

`(cpointer-push-tag! cptr tag)` PROCEDURE

These two functions treat pointer tags as lists of tags. As described in Section 4, a pointer tag does not have any role except for Scheme code that uses it to distinguish pointers — these functions treat the tag value as a list of tags, which makes it possible to construct pointer types that can be treated as other pointer types, mainly for implementing inheritance via upcasts (when a struct contains a super struct as its first element).

`cpointer-has-tag?` checks if the given *`cptr`* has the *`tag`* — a pointer has a tag *t* when its tag is either `eq?` to *t* or a list that contains (`memq`) *t*.

`cpointer-push-tag!` pushes the given *`tag`* value on *`cptr`*'s tags. The main properties of this operation are: (a) pushing any tag will make later calls to *`cpointer-has-tag?`* succeed with this tag, (b) the pushed tag will be used when printing the pointer (until a new value is pushed). (Technically, pushing a tag will simply set it if there is no tag set, otherwise push it on an existing list or an existing value (treated as a single-element list).)

`(_cpointer tag [ptr-type [scheme-to-C-proc C-to-scheme-proc]])` PROCEDURE

`(_cpointer/null tag [ptr-type [scheme-to-C-proc C-to-scheme-proc]])` PROCEDURE

These functions construct a kind of a pointer that gets a specific tag when converted to Scheme, and accept only such tagged pointers when going to C. An optional *`ptr-type`* can be given to be used as the base pointer type, instead of `_pointer`. (See `set-cpointer-tag!` and `cpointer-tag` in section 4 for more details.)

Pointer tags are checked with `cpointer-has-tag?` and changed with `cpointer-push-tag!` which means that other tags are preserved. Specifically, if a base *`ptr-type`* is given and is itself a `_cpointer`, then the new type will handle pointers that have the new tag in addition to *`ptr-type`*'s tag(s). When the tag is a pair, its first value is used for printing, so the most recently pushed tag which corresponds to the inheriting type will be displayed.

Note that tags are compared with `eq?` (or `memq`), which means an interface can hide its value from users (e.g., not provide the `cpointer-tag` accessor), which makes such pointers un-fake-able.

`_cpointer/null` is similar to `_cpointer` except that it tolerates NULL pointers both going to C and back. Note that NULL pointers are represented as `#f` in Scheme, so they are not tagged.

`(define-cpointer-type _name [ptr-type [scheme-to-C-proc C-to-scheme-proc]])` SYNTAX

A macro version of `_cpointer` and `_cpointer/null` above, using the defined name for a tag string, and defining

a predicate too. The name should look like '_foo', the predicate will be 'foo?', and the tag will be "foo". In addition, 'foo-tag' will be bound to the tag. The optional arguments are the same as those of _cpointer. '_foo' will be bound to the _cpointer type, and '_foo/null' to the _cpointer/null type.

5.2 Safe C Vectors

(make-cvector *ctype length*) PROCEDURE

Creates a C vector using the given *ctype* and *length*. This will allocate a memory block for the vector.

(cvector *ctype vals* ···) PROCEDURE

Creates a C vector using the given *ctype*, initialized to the given list of values.

(cvector? *x*) PROCEDURE

Predicate for C vectors.

(cvector-length *cvec*) PROCEDURE

Returns the length of a C vector.

(cvector-type *cvec*) PROCEDURE

Returns the C type object of a C vector.

(cvector-ref *cvec idx*) PROCEDURE

References the *idx*th element of the *cvec* C vector. The result will have the type that the C vector uses.

(cvector-set! *cvec idx val*) PROCEDURE

Sets the *idx*th element of the *cvec* C vector to *val*. *val* should be a value that can be used with the type that the C vector uses.

(cvector->list *cvec*) PROCEDURE

Converts the *cvec* C vector object to a list of values.

(list->cvector *list ctype*) PROCEDURE

Converts the list *list* to a C vector of the given *ctype*.

(make-cvector* *cptr ctype length*) PROCEDURE

Constructs a C vector using an existing pointer object. This operation is not safe, so it is intended to be used in specific situations where the *ctype* and *length* are known.

`_cvector` C TYPE

An input type for C vectors, which uses the pointer to the memory block. Also, see the `_cvector` custom type in section 3.5.1.

5.3 SRFI-4 Vectors

SRFI-4 vectors are similar to the above C vector, except it defines different types of vectors, each with a hard-wired type.

An internal '`make-srfi-4`' macro defines and provides 8 functions for each *TAG* type — a 'make-*TAG*vector' constructor, a '*TAG*vector' constructor (uses its arguments), a '*TAG*`vector?`' predicate, a '*TAG*`vector-length`' function, a '*TAG*`vector-ref`' accessor and a '*TAG*`vector-set!`' setter, and conversions to and from a list, '*TAG*`vector->list`' and '`list->`*TAG*`vector`'. The functions are the same as the corresponding `cvector` functions above, except that there is no type argument.

In addition, there is a `_`*TAG*`vector` type similar to `_cvector` that can be used when interfacing foreign functions. Just like `_cvector`, `_`*TAG*`vector` can be used both as a simple type to pass a pointer value to foreign code, or as a custom type, expecting a mode flag for input, output, or input-output, where output-mode requires specifying the size of the result.

The following homogeneous vector types are defined, for a total of 80 functions.

- `s8vector` using `_int8`,
- `u8vector` using `_uint8`,
- `s16vector` using `_int16`,
- `u16vector` using `_uint16`,
- `s32vector` using `_int32`,
- `u32vector` using `_uint32`,
- `s64vector` using `_int64`,
- `u64vector` using `_uint64`,
- `f32vector` using `_float`,
- `f64vector` using `_double*`.

The implementation of *`u8vector`* is an exception: it is implemented as MzScheme's byte-strings (§3.6 in *PLT MzScheme: Language Manual*).

6. Miscellaneous Support

(regexp-replaces *name substs*) PROCEDURE

A function that is convenient for many interfaces where the foreign library has some naming convention that you want to use in your interface as well. The first *name* argument can be any value that will be used to name the foreign object — a symbol, a string, a byte string etc. This is first converted into a string, and then modified according to the given *substs* list in sequence, where each element in this list is a list of a regular expression and a substitution string. Usually, `regexp-replace*` is used to perform the substitution, except for cases where the regular expression begins with a "^" or ends with a "$" where `regexp-replace` is used.

For example, the following makes it convenient to define Scheme bindings such as `foo-bar` for foreign names like `MyLib_foo_bar`:

```
(define mylib (ffi-lib "mylib"))
(define-syntax defmyobj
  (syntax-rules (:)
    [(_ name : type ...)
     (define name
       (get-ffi-obj (regexp-replaces 'name '((#rx"-" "_") (#rx"^" "MyLib_")))
                    mylib (_fun type ...)))]))
(defmyobj foo-bar : _int -> _int)
```

(list->cblock *list ctype*) PROCEDURE

Allocates a memory block of an appropriate size, and initializes it using values from *list* and the given *ctype*. The *list* must hold values that can all be converted to C values according to the given *ctype*.

(cblock->list *cblock ctype length*) PROCEDURE

Converts C pointers *cblock* to vectors of *ctype*, to Scheme lists. The arguments are the same as in the `list->cblock`. *length* must be specified because there is no way to know where the block ends.

(vector->cblock *vector ctype*) PROCEDURE

Same as the `list->cblock` function, only for Scheme vectors.

(cblock->vector *cblock ctype length*) PROCEDURE

Same as the `cblock->vector` function, only for Scheme vectors.

7. Unexported Primitive Functions

Parts of the **foreign.ss** libray are implemented by the MzScheme built-in `#%foreign` module. The `#%foreign` module is not intended for direct use, but it exports the following procedures. If you find any of these useful, please let us know.

`(ffi-obj` *objname ffi-lib-or-libname*`)` PROCEDURE

Pulls out a foreign object from a library, returning a Scheme value that can be used as a pointer. If a name is provided instead of a library object, `ffi-lib` is used to create a library object.

`(ffi-obj?` *x*`)` PROCEDURE
`(ffi-obj-lib` *ffi-obj*`)` PROCEDURE
`(ffi-obj-name` *ffi-obj*`)` PROCEDURE

A predicate for objects returned by `ffi-obj`, and accessor functions that return its corresponding library object and name. These values can also be used as C pointer objects.

`(ctype-basetype` *ctype*`)` PROCEDURE
`(ctype-scheme->c` *ctype*`)` PROCEDURE
`(ctype-c->scheme` *ctype*`)` PROCEDURE

Accessors for the components of a C type object, made by `make-ctype`. `ctype-basetype` returns `#f` for primitive types (including cstruct types).

`(ffi-call` *ptr in-types out-type*`)` PROCEDURE

The primitive mechanism that creates Scheme "callout" values. The given *ptr* (any pointer value, including `ffi-obj` values) is wrapped in a Scheme-callable primitive function that uses the types to specify how values are marshaled.

`(ffi-callback` *proc in-types out-type*`)` PROCEDURE

The symmetric counterpart of `ffi-call`. It receives a Scheme procedure and creates a callback object, which can also be used as a pointer. This object can be used as a C-callable function, which invokes *proc* using the types to specify how values are marshaled.

`(ffi-callback?` *x*`)` PROCEDURE

A predicate for callback values that are created by `ffi-callback`.

License

GNU Library General Public License

Version 2, June 1991

Copyright (C) 1991 Free Software Foundation, Inc.

675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries

themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

GNU LIBRARY GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called "this License"). Each licensee is addressed as "you".

   A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

   The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

   "Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) The modified work must itself be a software library.
   b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
   c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
   d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.
      (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

   In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

   This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

   If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

   When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

   If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

   Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

   You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

   a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
   b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
   c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
   d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

   For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

   a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
   b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

    If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

    It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

    This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Index