C-Data Reference Manual
Matt Wette
October 2024

Introduction

The *cdata* module and its partner *arch-info* provide a way to work with data originating from C libraries. We hope module is reasonably easy to understand and use. Size and alignment is tracked for all types. Types are classified into the following kinds: base, struct, union, array, pointer, enum and function. The procedures `cbase`, `cstruct`, `cunion`, `cpointer`, `carray`, `cenum` and `cfunction` generate *ctype* objects, and the procedure `make-cdata` will generate data objects based on these. The underlying bits of data are stored in Scheme bytevectors. Access to component data is provided by the `cdata-ref` procedure and mutation is accomplished via the `cdata-set!` procedure. The modules support non-native machine architectures via a global parameter called `*arch*`.

Beyond size and alignment, base type objects carry a symbolic tag to determine the appropriate low level machine type. The low level machine types map directly to bytevector setters and getters. Support for C base types is handled by the `cbase` procedure which converts them to underlying types. For example, on a 64 bit little endian architecture, `(cbase 'uintptr_t)` would generate a type with underlying symbol `u64le`.

Here is a simple example of using *cdata* for structures:

```
(use-modules (system foreign))
(use-modules (system foreign-library))
(use-modules (nyacc foreign cdata))

(define timeval_t (cstruct '((tv_sec long) (tv_usec long))))

(define gettimeofday
  (foreign-library-function
   #f "gettimeofday"
   #:return-type (ctype->ffi (cbase 'int))
   #:arg-types (map ctype->ffi
                    (list (cpointer timeval_t)
                          (cpointer 'void)))))

(define d1 (make-cdata timeval_t))
(gettimeofday (cdata-ref (cdata& d1)) %null-pointer)
(format #t "time: ~s ~s\n"
        (cdata-ref d1 'tv_sec) (cdata-ref d1 'tv_usec))
time: 1719062561 676365
```

In the above `cdata&` generates a cdata pointer to `d1` and `cdata-ref` extracts the Guile value.

Basic Usage

This section provides an introduction to procedures you are likely to want on your first approach.

`cbase` *name* [Procedure]

Given symbolic *name* generate a base ctype. The name can be a symbol like `unsigned-int`, `double`, or can be a *arch-info* machine type symbol like `u64le`.

`cpointer` *type* => <*ctype*> [Procedure]

Generate a C pointer type for *type*. To reference or de-reference cdata object see `cdata&` and `cdata*`. *type* can be the symbol `void` or a symbolic name used as argument to `cbase`.

note: Should we allow *type* to be a promise?

```
(define foo_t (cbase 'int))
(cpointer (delay foo_t))
```

`cstruct` *fields* [*packed*] => *ctype* [Procedure]

Construct a struct ctype with given *fields*. If *packed*, `#f` by default, is `#t`, create a packed structure. *fields* is a list with entries of the form `(name type)` or `(name type lenth)` where `name` is a symbol or `#f` (for anonymous structs and unions), `type` is a `<ctype>` object or a symbol for a base type and `length` is the length of the associated bitfield.

`cunion` *fields* => <*ctype*> [Procedure]

Construct a ctype union type with given *fields*. See *cstruct* for a description of the *fields* argument.

`carray` *type n* => <*ctype*> [Procedure]

Create an array of *type* with *length*. If *length* is zero, the array length is unbounded: it's length can be specified as argument to `make-cdata`.

`cenum` *enum-list* [*packed*] => <*ctype*> [Procedure]

enum-list is a list of name or name-value pairs

```
(cenum '((a 1) b (c 4))
```

If *packed* is `#t` the size will be smallest that can hold it.

`cfunction` *proc->ptr ptr->proc* [*variadic?*] => <*ctype*> [Procedure]

Generate a C function type to be used with `cpointer`. You must pass the *wrapper* and *unwrapper* procedures that convert a procedure to a pointer, and pointer to procedure, respectively. The optional argument *#:variadic*, if `#t`, indicates the function uses variadic arguments. For this case (I need to add documention). Here is an example:

```
(define (f-proc->ptr proc)
  (ffi:procedure->pointer ffi:void proc (list)))
(define (f-ptr->proc fptr)
  (ffi:pointer->procedure ffi:void fptr (list)))
(define ftype (cpointer (cfunction f-proc->ptr f-ptr->proc)))
```

`make-cdata` *type* [*value*] => <*cdata*> [Procedure]

Generate a *cdata* object of type *type* with optional *value*. As a special case, an integer arg to a zero-sized array type will allocate storage for that many items, associating it with an array type of that size.

`cdata-ref` *data* [*tag ...*] => *value* [Procedure]

Return the Scheme (scalar) slot value for selected *tag ...* with respect to the cdata object *data*.

```
(cdata-ref my-struct-value 'a 'b 'c))
```

This procedure returns Guile values for cdata kinds *base*, *pointer* and *procedure*. For other cases, a *cdata* object is returned. If you always want a cdata object, use `cdata-sel`.

`cdata-set!` *data value* [*tag ...*] [Procedure]

Set slot for selcted *tag ...* with respect to cdata *data* to *value*. Example:

```
(cdata-set! my-struct-data 42 'a 'b 'c))
```

If *value* is a `<cdata>` object copy that, if types match.
If *value* can be a procedure used to set a cfunction pointer value.

`cdata&` *data* => *cdata* [Procedure]

Generate a reference (i.e., cpointer) to the contents in the underlying bytevector.

`cdata*` *data* => *cdata* [Procedure]

De-reference a pointer. Returns a *cdata* object representing the contents at the address in the underlying bytevector.

Notes

Digression on Garbage Collection

Before going further we remind you that the underlying datastructure is bytevectors. Now, since bytevectors in Guile are not searched for pointers during garbage collection there is a risk that the objects being referenced might be collected during usage. A systematic method to prevent this is work to go. One might try to use `cdata&` in the following way keep intermediate values from being collected.

```
(let ((val (make-cdata foo_t))
      (ptr (cdata& val)))
  (bar ptr))
```

Going Further

`cdata-sel` *data tag ...* => *cdata* [Procedure]

Return a new `cdata` object representing the associated selection. Note this is different from `cdata-ref`: it always returns a cdata object. For example,

```
> (define t1 (cstruct '((a int) (b double))))
> (define d1 (make-cdata t1))
```

```
> (cdata-set! d1 42 'a)
> (cdata-sel d1 'a)
$1 = #<cdata s32le 0x77bbf8e52260>
> (cdata-ref $1)
$2 = 42
```

`cdata&-ref` *data* [*tag* ...] => *value* [Procedure]
: Shortcut for `(cdata-ref (cdata& data tag ...))` This always returns a Guile *pointer*.

`cdata*-ref` *data* [*tag* ...] => *value* [Procedure]
: Shortcut for `(cdata-ref (cdata* data tag ...))`

`Xcdata-ref` *bv ix ct* -> *value* [Procedure]
: Reference a deconstructed cdata object. See *cdata-ref*.

`Xcdata-set!` *bv ix ct value* [Procedure]
: Set the value of a deconstructed cdata object. See *cdata-set!*.

Working with Types

`name-ctype` *name type* => <*ctype*> [Procedure]
: Create a new named version of the type. The name is useful when the type is printed. This procedure does not mutate: a new type object is created. If a specific type is used by multiple names the names can share the underlying type guts. The following generates two named types.

```
(define raw (cstruct '((a 'int) (b 'double))))
(define foo_t (name-ctype 'foo_t raw))
(define struct-foo (name-ctype 'struct-foo raw))
```

These types are equal:

```
(ctype-equal? foo_t struct-foo) => #t
```

`ctype-equal?` *a b* [Procedure]
: This predicate assesses equality of it's arguments. Two types are considered equal if they have the same size, alignment, kind, and eqivalent kind-specific properties. For base types, the symbolic mtype must be equal; this includes size, integer versus float, and signed versus unsigned. For struct and union kinds, the names and types of all fields must be equal.
TODO: algorithm to prevent infinite search for recursive structs

`ctype-sel` *type ix* [*tag* ...] => ((*ix* . *ct*) (*ix* . *ct*) ...) [Procedure]
: This generate a list of (offset, type) pairs for a type. The result is used to create getters and setter for foreign machine architectures. See *make-cdata-getter* and *make-cdata-setter*.

`make-cdata-getter` *sel* [*offset*] => *lambda* [Procedure]
: Genererate a procedure that given a cdata object will fetch the value at indicated by the *sel*, generated by `ctype-sel`. The procedure takes one argument: `(proc data`

[tag ...]). Pointer dereference tags ('*') are not allowed. The optional *offset* argument (default 0), is used for cross target use: it is the offset of the address in the host context.

`make-cdata-setter` *sel* [*offset*] => *lambda* [Procedure]

Genererate a procedure that given a cdata object will set the value at the offset given the selector, generated by `ctype-sel`. The procedure takes two arguments: `(proc data value [tag ...])`. Pointer dereference tags ('*') are not allowed. The optional *offset* argument (default 0), is used for cross target use: it is the offset of the address in the host context.

Working with C Function Calls

The procedure `ctype->ffi` is a helper for using Guile's *pointer->procedure*.

`ccast` *type data* [*do-check*] => *<cdata>* [Procedure]

need to be able to cast array to pointer

```
(ccast Target* val)
```

`arg->number` *arg* => *number* [Procedure]

Convert an argument to numeric form for a ffi procedure call. This will reference a cdata object or pass a number through.

The above procedure was previously called `unwrap-number`.

`arg->pointer` *arg* [*hint*] => *pointer* [Procedure]

Convert an argument to a Guile pointer for a ffi procedure call. This will reference a cdata object or pass a number through. If the argument is a function, it will attempt to convert that to a pointer via `procedure->pointer` if given the function pointer type *hint*.

The above procedure used to be called `unwrap-pointer`. It is also used to implement the old `unwrap-array`.

`ctype->ffi` *ctype* => *ffi-type* [Procedure]

Generate a argument spec for Guile's ffi interface. Example:

```
(ctype->ffi (cpointer (cbase int))) => '*
```

Operations on CType Kinds

The ctype `kind` field indicates which kind a type is and the `info` field provide kind-specific information for a ctype. The `name` field provides the type name, if provided, or `#f` if not.

Note that the kind procecures, `cstruct`, `cpointer`, ..., create *ctype* objects of different *kinds*. To operate on kind-specific attributes of types, requries one to fetch the `info` field from the ctype. From the `info` field, one can then operate using the fields specific to the kind info.

```
> (define float* (cpointer (cbase 'float)))
> double*
```

```
$1 = #<ctype pointer 0x75f3212cbed0>
> (ctype-kind float*)
$2 = pointer
> (define float*-info (ctype-info float*))
> (cpointer-type float*-info)
$3 = #<ctype f32le 0x75f323f8ec90>
> (cpointer-mtype float*-info)
$4 = u64le
```

The `cpointer-mtype` procedure lets us know that pointers are stored as unsigned 64 bit (little endian) integers.

The info field for base types is special. Since the only kind-specific type information for a base type is the machine type the info field provides that. Consider the following example.

```
> (define foo-t (name-ctype 'foo-t (cbase 'int)))
> (ctype-name foo-t)
$1 = foo-t
> (ctype-kind foo-t)
$2 = base
> (ctype-info foo-t)
$3 = s32le
```

Structs are more involved.

```
> (define bar-s
    (cstruct `((a int) (b float) (#f ,(cstruct '(x int) (y int))))))
> (define bar-s-info (ctype-info bar-s))
> (cstruct-fields bar-s-info)
$4 = (#<<cfield> name: a type: #<ctype s32le 0x75f323f8ecf0> offset: 0>
      #<<cfield> name: b type: #<ctype f32le 0x75f323f8ec90> offset: 4>
      #<<cfield> name: #f type: #<ctype struct 0x75f32181a570> offset: 8>)
> (define x-fld ((cstruct-select bar-s-info) 'x))
> x-fld
$5 = #<<cfield> name: x type: #<ctype s32le 0x75f323f8ecf0> offset: 8>
> (cfield-offset x-fld)
$6 = 8
```

Note that the selection of the `x` component deals with a field which is an anonymous struct. The struct `bar-s` would look like the following in C:

```
struct bar_s {
  int a;
  float b;
  struct {
    int x;
    int y;
  };
};
```

And just for kicks

```
> (define sa
```

```
    (cstruct `((a int) (b double) (#f ,(cstruct '((x short) (y int)))))))
> (define sp
    (cstruct `((a int) (b double) (#f ,(cstruct '((x short) (y int))))) #t))

> (pretty-print-ctype sa)
(cstruct
  ((a s32le #:offset 0)
   (b f64le #:offset 8)
   (#f
    (cstruct
      ((x s16le #:offset 0) (y s32le #:offset 4)))
    #:offset
    16)))
> (pretty-print-ctype sp)
(cstruct
  ((a s32le #:offset 0)
   (b f64le #:offset 4)
   (#f
    (cstruct
      ((x s16le #:offset 0) (y s32le #:offset 4)))
    #:offset
    12)))
```

Note the difference in offsets: `sa` is aligned and `sp` is packed. The offsets reported for anonymous structs can be misleading. To get the right offsets use select:

```
> (define tia (ctype-info sa))
> (define tip (ctype-info sp))
> ((cstruct-select tia) 'y)
$8 = 20
> ((cstruct-select tip) 'y)
$9 = 16
```

Enum Conversions

The enum ctype provides procedures to convert between the numeric and symbolic parts of each enum entry. Currently, the cdata module does not provide enum wrapper and unwrapper routines. However, the FFI Helper will create these. The wrapper, converting a number to a symbol, and unwrapper, converting a symbol to a number, can be generated as the following example demonstrates.

```
> (define color_t (cenum '((RED #xf00) (GREEN #x0f0) (BLUE #x00f))))
> (define color_t-info (ctype-info color_t))
> (define wrap-color_t (cenum-symf color_t-info))
> (define unwrap-color_t (cenum-numf color_t-info))
> (wrap-color_t #xf00)
$1 = RED
> (unwrap-color_t 'GREEN)
$2 = 240
```

Handling Machine Architectures

One of the author's main motivations for writing CData was to be able to work with cross-target machine architectures. This is pretty cool. Just to let you know what's going on, consider the following:

```
> (use-modules (nyacc foreign arch-info))
> (define tx64 (with-arch "x86_64" (cstruct '((a int) (b long)))))
> (define tr64 (with-arch "riscv64" (cstruct '((a int) (b long)))))
> (define tr32 (with-arch "riscv32" (cstruct '((a int) (b long)))))
> (define sp32 (with-arch "sparc" (cstruct '((a int) (b long)))))
> (ctype-equal? tx64 tr64)
$1 = #t
> (ctype-equal? tr64 tr32)
$1 = #f
> (ctype-equal? tr32 ts32)
$1 = #f
> (pretty-print-ctype tx64)
(cstruct ((a s32le #:offset 0) (b s64le #:offset 8)))
> (pretty-print-ctype tr64)
(cstruct ((a s32le #:offset 0) (b s64le #:offset 8)))
> (pretty-print-ctype tr32)
(cstruct ((a s32le #:offset 0) (b s32le #:offset 4)))
> (pretty-print-ctype ts32)
(cstruct ((a s32be #:offset 0) (b s32be #:offset 4)))
```

Rocks, right?

arch-info maps base C types to machine types (e.g., i32le) and alignment for the given machine architecture. To get sizes, it's a simple matter of mapping machine types to sizes.

The arch-info module currently has size and alignment information for the for the following: aarch64, avr, i383, i686, powerpc32, powerpc64, ppc32, ppc64, riscv32, riscv64, sparc32, sparc64, x86_64.

CData Utilities

`pretty-print-ctype` *type* [*port*] [Procedure]

> Converts type to a literal tree and uses Guile's pretty-print function to display it. The default port is the current output port.

`cdata-kind` *data* [Procedure]

> Return the kind of *data*: pointer, base, struct, ...

Miscellaneous

More to come.

Base Types

```
void*
```

```
char short int long float double unsigned-short unsigned unsigned-long
size_t ssize_t ptrdiff_t int8_t uint8_t int16_t uint16_t int32_t
uint32_t int64_t uint64_t signed-char unsigned-char short-int
signed-short signed-short-int signed signed-int long-int signed-long
signed-long-int unsigned-short-int unsigned-int unsigned-long-int
_Bool bool intptr_t uintptr_t wchar_t char16_t char32_t long-double
long-long long-long-int signed-long-long signed-long-long-int
unsigned-long-long unsigned-long-long-int
```

Other Procedures

More to come.

Guile FFI Support

More to come.

`ctype->ffi-type` *type* [Procedure]

Convert a *ctype* to the (integer) code for the associated FFI type.

Copyright

References

1. Guile Manual: `https://www.gnu.org/software/guile/manual`
2. Scheme Bytestructures: `https://github.com/TaylanUB/scheme-bytestructures`