C-Data Reference Manual
Matt Wette
September 2024

Introduction

The `cdata` module and its partner `arch-info` provide a way to work with data originating from C libraries. Size and alignment is tracked for all types. Types are classified into the following kinds: base, struct, union, array, pointer, enum and function. The module has been designed with the goals of being easy to understand and easy to use. The procedures `cbase`, `cstruct`, `cunion`, `cpointer`, `carray`, `cenum` and `cfunction` generate *ctype* objects, and the procedure `make-cdata` will generate data objects for ctyped data. The underlying data is 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 `*arch*` parameter.

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 machine symbol `u64le`.

Here is a simple example using structures:

```
(define timeval (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)
                          (cpointer 'void)))))

(define d1 (make-cdata timeval))
(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
```

Basic Usage

This section provides the most-used procedures.

`cbase` *name* [Procedure]

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

`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* [Procedure]

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

`carray` *type n* [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*] [Procedure]

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

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

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

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

Generate a C function pointer type. You must pass the *wrapper* and *unwrapper* procedures that convert a pointer to a procedure, and procedure to pointer, respectively. The optional argument *#:variadic*, if `#t`, indicates the function uses variadic arguments. For this case, (to be documented).

`make-cdata` *type* [*value* [*name*]] [Procedure]

Generate a *cdata* object of type *type* with optional *value* and *name*. To specify name but no value use something like

```
(make-cdata mytype #f "foo")
```

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 ...*] [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 XXX for cdata kinds *base*, *pointer* and (in the future) *function*. Attempting to obtain values for C-type kinds *struct*, *union*, *array* will result in `#f`. If, in those cases, you would like a cdata then use this:

```
(or (cdata-ref data tag ...) (cdata-sel data tag ...))
```

(Or should we just make this the default behavior?)

`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.

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*` *data* `=>` *cdata* [Procedure]
De-reference a pointer. Returns a *cdata* object representing the contents at the address in the underlying bytevector.

`cdata&-ref` *data* [*tag ...*] [Procedure]
Does not work work (yet) for march offset addresses.

`cdata*-ref` *data* [*tag ...*] [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]
Reference a deconstructed cdata object. See *cdata-set!*.

Working with Types

`name-ctype` *name type* `=>` *ctype* [Procedure]
Add a name tag to a ctype.

`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)
```

`unwrap-number` [Procedure]

doc to come

`unwrap-pointer` [Procedure]

doc to come

`unwrap-array` [Procedure]

doc to come

`ctype->ffi` [Procedure]

doc to come

Handling Machine Architectures

Needs love ...

```
> (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)))))
> (ctype-equal? tx64 tr64)
$1 = #t
> (ctype-equal? tr64 tr32)
$1 = #f
> (pretty-print-ctype tx64)
```

```
(cstruct ((a s32le) (b s64le)))
> (pretty-print-ctype tr64)
(cstruct ((a s32le) (b s64le)))
> (pretty-print-ctype tr32)
(cstruct ((a s32le) (b s32le)))
```

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`